Science of Computer Programming 74 (2009) 671–687

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A perspective on service orchestration

Marco A. Barbosa *, Luis S. Barbosa

CCTC, Universidade do Minho, 4700-320 Braga, Portugal

ARTICLE INFO

Article history:
Received 29 June 2007
Received in revised form 15 May 2008
Accepted 15 September 2008
Available online 26 February 2009

Keywords:
Services
Software connectors
Exogenous coordination
Reo

ABSTRACT

Service-oriented computing is an emerging paradigm with increasing impact on the way modern software systems are designed and developed. Services are autonomous, loosely coupled and heterogeneous computational entities able to cooperate to achieve common goals. This paper introduces a model for service orchestration, which combines a *exogenous coordination* model, with services' interfaces annotated with behavioural patterns specified in a process algebra which is *parametric on the interaction discipline*. The coordination model is a variant of Reo for which a new semantic model is proposed.

1. Introduction: The problem and its context

Web services are re-shaping the Web from a document-centered to a service-centered environment. Simultaneously, they are challenging our understanding of how applications develop, and even of the nature of software itself (regarded more as a *service* to be contracted than as a *product* to acquire). The impact of such a move, both in the world's economy and in our everyday life, is just beginning to loom.

It is widely recognised that the emergence of *service-oriented computing* [42,26], of which Web services [7] are the most popular manifestation, entails the need for new rigorous frameworks to model and reason about composition and interaction in a way which goes beyond object-oriented or component-based techniques [9,23,35,30]. Unlike traditional composition, service composition is not based on the physical integration of components. Services are not libraries to be compiled and linked as part of applications. They are dynamic entities, running in different platforms and owned by different organizations, interacting through public interfaces which can invoke and be invoked by other services and cooperate to achieve common complex goals. Typically services involved in an application remain loosely coupled, often unaware of each other.

The specific characteristics of interaction in service-oriented computing suggests the suitability of exogenous coordination models [10,39] to describe service orchestration. The starting point of such models lies on a clear separation between computation and coordination [41], therefore enforcing *anonymous* communication and externally controlled component interconnection. The coordination perspective is (partially) implemented, for example, in JavaSpaces on top of Jini [40] and fundamental to a number of approaches to componentware which identify communication by generic channels as the basic interaction mechanism — see, e.g., Reo [11] or Piccola [50,39].

Exogenous coordination models are always based on some notion of a *software connector* (see, e.g., [25,10,18,44,52]) which regulates the dataflow, by relating data items crossing its input and output ports, and enforces synchronization constraints. Typically the coordinated entities are regarded as black-boxes, characterized by a set of ports through which data values are sent or received. Ports have a polarity (either *input* or *output*) and, maybe, a type to classify the admissible values, but in general such models make few assumptions on components.

* Corresponding author.
E-mail addresses: marco@di.uminho.pt (M.A. Barbosa), lsb@di.uminho.pt (L.S. Barbosa).

0167-6423/$ – see front matter © 2009 Published by Elsevier B.V.
doi:10.1016/j.scico.2008.09.019

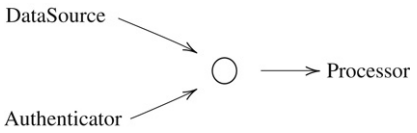


Fig. 1. An example of a *configuration*.

This is, however, clearly insufficient to count as an *interface* for, for example, a Web service. Typically, the latter includes a description of what is commonly called the *business protocol* or *behavioural pattern*, i.e., a specification of which, when and under what conditions ports become activated (i.e., ready to deliver or consume a datum). To be useful such specifications have to be *compositional*, in the sense that the overall behaviour of an application should be computed from the behaviour of individual services and that of the *connectors* forming the orchestration layer.

In such a context, this paper proposes an approach to service orchestration resorting both to an exogeneous coordination model and the use of behavioural patterns for the specification of services. Behavioural patterns are also used to describe the semantics of software connectors used for coordination, so that the emergent behaviour of an orchestration can be computed compositionally. These patterns are described here as expressions in a *process algebra* [29,38,49], a well-known family of compositional approaches to the description of interactive behaviours [37,3].

The idea, in itself, is not new. For example, Reference [48] uses a process language to describe the message exchange between web services, and to reason about them. Similar work, but now in the choreography side, is reported in, e.g., [22] or [56]. The challenging issue is composition. Actually, it comes with no surprise that different interaction disciplines govern service aggregation, on the one hand, connector composition, on the other, and, finally, the integration of both. Typical process algebras, however, have a specific interaction discipline which is fixed once and for all (e.g., the action/coaction synchronization which characterizes Ccs [36]).

This lead us to build on our own previous work (documented in [15,16,45]) on the development of *generic* process algebra. I.e., process algebras in which parallel composition is parametric on an interaction discipline suitably encoded in the 'gene' of some process combinators. The word 'gene', which be taken as a metaphor, has a precise technical meaning in the coalgebraic framework adopted, as explained in Section 3. Therefore, *different* interaction models become available to govern different aspects of a specification.

In such a context, the main contribution of this paper is

- An integrated approach to the application of an exogeneous coordination model for service orchestration, taking into account the behavioural specification of both connectors and services, and enabling the calculation of the emergent behaviour.

Having chosen Reo [11] as such a model, the paper also provides

- A new semantics for Reo connectors which is full compliant to the operational description of the model in [11]. In particular, the semantics of *lossy channels* and their application in, e.g., the construction of an *exclusive router*, is correctly captured. This is still an open issue for the most popular semantics for Reo based in timed data steams [13] or constraint automata [14].
- An approach to connector design based on a set of five combinators (*parallel*, *interleaving*, *hook*, *left join* and *right join*) which, in our opinion, provides a more structural alternative to the one based on graph manipulation used in the Reo literature.

The paper is structured as follows: Following an overview of the whole approach in Sections 2 and 3 provides a generic way to build process algebras and its application to the specification of services' business protocols (to be referred here as *use*, or behavioural, patterns). A model for service orchestration is discussed in Sections 4, concerned with connectors and their composition into a coordination layer, and 5 where connectors and services are put together to cooperate in a loosely-coupled way. Finally, Section 6 concludes and gives a few pointers to current work.

2. Interfaces, connectors and configurations

The purpose of this section is to give an overview of the strategy proposed in the paper for service orchestration. Formal definitions will appear in subsequent sections. For illustrative purposes consider the situation depicted in Fig. 1 involving three Web services: one produces data items, another emits authentication certificates for them and, finally, the third collects both data and certificates and processes them in a certain way.

According to the *World Wide Web Consortium*, a Web service is a software application identified by a uniform resource identifier (URI), whose interfaces and binding can be defined, described, and discovered by Xml artifacts, and that supports direct interactions with other software applications using Xml based messages via Internet-based protocols. Less biased definitions abstract from concrete representations of data and messages. In [31] a Web service is a *self-contained, modular applications that can be described, published, located, and invoked over a network, generally the Web*. And in [7] it is characterised as *a program accessible over the Web with a stable interface, published with additional descriptive information on some*

service directory. At an even more abstract level, the underlying notion of a *service* emerges as a platform-independent computational entity which can be defined, published, classified, discovered and dynamically assembled for developing distributed, interoperable, evolvable systems and applications.

In general, services make themselves available by publishing an *interface* which describes a number of operations that other services may invoke according to given patterns, known as *conversations*, whose collection forms what is called the service *business protocol*. In the case of Web services, interfaces are typically described in WSDL [55], which resemble a classical IDL ('interface description language') enriched with contextual information, such as the service address or the transport protocol used for access. The definition of admissible behaviours and the roles associated to them, usually resorts to a *coreography* language — e.g. WS-CDL [54]. It should also be noted that, unlike conventional middleware, a service can proactively initiate an interaction, a fact that blurs the classical distinction between clients and servers.

As in [11] or [18], we assume services are black box entities, accessed by purely syntactic interfaces. Moreover, such black boxes encapsulate active entities which are responsible for producing or consuming data items through their boundaries. The primary role of an interface is to keep track of port names and, possibly, of admissible types for data items flowing through them. As mentioned before, the model proposed in this paper extends interfaces with a protocol specification over port activations.

Definition 1. Let $\mathbb{D}$ be a data domain understood as a general type for messages. An interface for service S is specified by a *port signature*, $sig(S)$ over $\mathbb{D}$, given by a port name and a polarity annotation (either in(put) or out(put)), and a behavioural pattern, $use(S)$, representing its use or business protocol, given by a process term, as later detailed, over port activations.

What sort of formalism should be used for the specification of *use patterns*? Transition systems [34,51], regular-expressions [43,44,53] or process algebras [33,6] are part of the huge diversity of formal structures typically used to represent behaviour, which has also been explored in the formalization of web services. Process algebra, in particular, provides an expressive setting for representing behavioural patterns and establish/verify their properties in a compositional way. Some flexibility, however, is required with respect to the underlying interaction discipline. Actually, different disciplines have to be used, at the same time, to capture different aspects of services orchestration. For example the discipline governing the composition of *software connectors* between them (to build the application overall *glue code*) differs from the one used to capture the interaction between the latter and the relevant services' interfaces. In any case, one needs a way of specifying the relevant interaction discipline while guaranteeing that behaviour combinators used are *parametric* on it. Meeting this goal entails the need for a *generic* way to *design* process algebras, a topic addressed in Section 3.

For the moment, let us suppose each service in Fig. 1 has a particular protocol attached and moreover that there exists an overall restriction specifying that data from DataSource and Authenticator are received by the processing service in strict alternation (i.e., each data item is followed by the corresponding certificate).

The latter restriction clearly belongs to the orchestration layer (the *glue code*): in principle the two data sources do not even need to be aware of it. Enforcing such restrictions is the role of connectors which mediate services' interconnection. Connectors have ports through which the exchange of messages takes place. The activation of their ports also obey particular patterns, whose specification resort to the same formalism used for business protocols. When interfacing with services both sides impose constraints on how the conversation has to proceed. Our purpose is to be able to infer the global behaviour of an application from such constraints.

As detailed in Section 4, connectors are specified by a relation, which captures the flow of data, and a behavioural pattern. Their construction is compositional: new connectors are built out of old through five specific connectives: parallel *aggregation*, *interleaving*, left and right port *join* and *hook*, the latter corresponding to a feedback mechanism.

Connectors provide the essential mechanism for service composition. This guarantees loosely coupled cooperation among services and entails a number of simplifications. For example there is no need to syntactically distinguish between different forms of interaction, e.g., between *one-way* or *request-response* interactions. Such patterns are enforced at the coordination level. A collection of services, specified by their interfaces, interconnected by a particular, eventually rather complex connector, forms a *configuration*, as depicted in Fig. 1. The remaining of this paper is devoted to make all this precise.

3. Specifying behaviour

Recent approaches to process calculi semantics are resorting to representations of *labelled transition systems* as coalgebras [47] for (some combinations of) the powerset functor. Such coalgebraic characterizations not only provide a generic setting for fundamental constructions (e.g., bisimulation regarded as equality in the final coalgebra), but also make it easier to generalize typical transition systems concepts to broader classes of dynamic systems (e.g., probabilistic automata [32,5] or hybrid systems [28]).

This section provides an introduction to our own coalgebraic approach to the design of generic process algebra. References [15,16] introduce a denotational approach to the *design* of process algebras in which processes are identified with inhabitants of a final coalgebra and their combinators defined by coinductive extension (of 'one-step' behaviour generator functions). The *universality* of such constructions entails both definitional and proof principles on top of which the development of the process calculi is based. This leads to a generic way of reasoning about processes in which, in particular,

proofs by bisimulation, which classicaly involve the explicit construction of such a relation [36], are replaced by *equational reasoning*.[1]

In this approach, transition systems over a state space U and a set A of labels, classicaly specified as *binary relations*

$$_{\alpha}\longleftarrow : A \times U \longleftarrow U \tag{1}$$

are given by coalgebras

$$\alpha : \mathcal{P}\,(A \times U) \longleftarrow U \tag{2}$$

for $\mathcal{P}\,(A \times \mathsf{Id})$, where $\mathcal{P}$ and Id denote, respectively, the (finite) powerset and the identity functor. It is well-known that set-valued functions, such as coalgebra (2) are models of binary relations and, conversely, any such relation is uniquely transposed into a set-valued function. The existence and uniqueness of such a transformation leads to the identification of a *transpose* operator Λ [20] characterized by an universal property which, for this particular case, reads

$$\alpha = \Lambda\ _{\alpha}\longleftarrow \quad \equiv \quad _{\alpha}\longleftarrow = \in \cdot \alpha \tag{3}$$

where $\in$ denotes set membership and $\cdot$ is relational composition. Moreover, whenever $\mathcal{P}$ in (2) is restricted to the *finite* powerset, to enforce the existence of a final universe, equivalence (3) establishes again a bijective correspondence between the resulting coalgebras and *image finite* relations.

We are interested in final coalgebras because, technically, our approach amounts to the systematic use of the universal property of coinductive extension both to define process combinators and to reason about them. I.e., the existence, for each arbitrary coalgebra $\langle U, p : \mathcal{P}\,(A \times U) \longleftarrow U\rangle$, of a unique morphism $[\![p]\!]$ to the final coalgebra $\omega : \mathcal{P}\,(A \times \nu) \longleftarrow \nu$ satisfying

$$k = [\![p]\!] \Leftrightarrow \omega \cdot k = \mathcal{P}\,(\mathsf{id} \times k) \cdot p \tag{4}$$

Therefore, processes being the inhabitants of the final coalgebra ν, elements of $\mathcal{P}\,(A \times \nu)$ are sets of pairs, each one representing a transition label and the corresponding continuation process. Morphism $[\![p]\!]$ represents the behaviour generated by p (which is appropriately called its *gene*) and comes equipped with a bunch of laws useful in calculation.

3.1. Combinators

Process combinators are defined either in a direct way (if they are consumed by transitions) or by coinductive extension (if persistent). Examples in the first group are the *inactive* process **0**, whose set of observations is empty, non deterministic choice, $+$, whose observations are the union of the possible observation upon its arguments, and *prefix*, $(a.p)$. Formally,

$$\omega \cdot \mathbf{0} = \emptyset \tag{5}$$

$$\omega \cdot + = \cup \cdot (\omega \times \omega) \tag{6}$$

$$\omega \cdot a. = \mathsf{sing} \cdot \mathsf{label}_a \tag{7}$$

In the familiar pointwise notation the last equality reads

$$\omega(a.p) = \{\langle a, p\rangle\}$$

The second group contains all combinators recursively defined. Although this is not the place for a detailed account, we shall briefly review the specification of both *parallel composition* and *synchronous product*, not only because these combinators are used in the paper to join independent services, but also because they make concrete the notion of *parametrization* by an interaction discipline discussed above. However, to do this, we need first to introduce the *interleaving* combinator.

Interleaving ($\boxplus : \nu \longleftarrow \nu \times \nu$) represents an interaction-free form of parallel composition. Observations over the interleaving of two processes correspond to all possible interleavings of observations of their arguments. Thus, $\boxplus = [\![(\alpha_{\boxplus})]\!]$, where 'gene' $\alpha_{\boxplus} : \mathcal{P}\,(A \times (\nu \times \nu)) \longleftarrow \nu \times \nu$ is given by

$$\alpha_{\boxplus} = \{\langle a, \langle u', v\rangle\rangle |\ \langle a, u'\rangle \in \omega\, u\} \cup \{\langle a, \langle u, v'\rangle\rangle |\ \langle a, v'\rangle \in \omega\, v\}$$

Synchronous product models the simultaneous execution of its two arguments. In each step, processes interact through the actions they realize. Let us, for the moment, represent such interaction by a function $\theta : A \times A \longleftarrow A$. Than, $\otimes = [\![(\alpha_{\otimes})]\!]$ where

$$\alpha_{\otimes} = \{\langle a\,\theta\,b, \langle u', v'\rangle\rangle |\ \langle a, u'\rangle \in \omega\, u \ \wedge\ \langle b, v'\rangle \in \omega\, v \ \wedge\ (a\,\theta\,b) \neq 0\}$$

The fundamental point to note is that the definition is parametric on θ, which encodes an *interaction discipline*. Technically, an *interaction discipline* is modeled as an Abelian positive monoid $\langle A; \theta, 1\rangle$ with a zero element 0. The intuition is that θ

[1] In the *dual* world of functional programming the role of such 'universals' is the basis of a whole discipline of algorithm derivation and transformation, which can be traced back to the so-called *Bird-Meertens formalism* [21] and the foundational work of T. Hagino [27].

determines the interaction discipline whereas 0 represents the absence of interaction: for all $a \in A$, $a\theta 0 = 0$. On the other hand, being a positive monoid entails $a\theta a' = 1$ iff $a = a' = 1$. A typical example of an interaction structure captures action co-occurrence, in which case θ is defined as $a\theta b = \langle a, b\rangle$, for all $a, b \in A$. Another example is provided by the matching of complementary actions in Ccs [36].

Later in this paper we shall introduce a number of specifications for θ to capture the different sorts of interaction involved in service orchestration. Note, in particular, that actions organised into the behavioural patterns of services or connectors may have a number of different meanings: not only representing *port activation* but also, for example, the *absence of data requests* at a particular port. Actually, the structure imposed upon actions is richer than one is used to consider in classical process algebra.

Parallel composition combines the effects of both *interleaving* $\boxplus$ and *synchronous product* $\otimes$. Such a combination is performed at the *genes* level: $\boxtimes = [(\alpha_{\boxtimes})]$, where $\alpha_{\boxtimes}$ can be defined in a pointfree style as

$$\alpha_{\boxtimes} = \nu \times \nu \xrightarrow{\triangle} (\nu \times \nu) \times (\nu \times \nu) \xrightarrow{(\alpha_{\boxplus} \times \alpha_{\otimes})} \mathcal{P}(A \times (\nu \times \nu)) \times \mathcal{P}(A \times (\nu \times \nu)) \xrightarrow{\cup} \mathcal{P}(A \times (\nu \times \nu))$$

where $\triangle$ is the diagonal function, which produces two copies of its argument, and $\cup$ denotes set union.

3.2. Behaviour laws

The coalgebraic setting provides for free a notion of bisimulation for functor $\mathcal{P}(A \times \mathsf{Id})$ which boils down to equality in the final coalgebra where (denotations of) processes live. The universal characterisation of finality in equivalence (4), entails a proof style in which equational reasoning replaces the explicit construction of bisimulations. This is used in [15] to prove properties of process combinators. Reference [45], in particular, introduces and characterises the theory of a number of non -standard combinators dealing with process interruption and recovery. The extension of such a framework to prove equalities based on observational equivalence is done in [46].

For the purpose of this paper it is sufficient to recall that both $\boxplus$, $\boxtimes$ and $\otimes$ form Abelian monoids for whatever *interaction discipline* one might consider. Similarly, sum (i.e., choice) is also an Abelian idempotent monoid. Also useful in the sequel is the well-known expansion law stating a process is bisimilar to the sum of its derivations, i.e.,

$$p \sim \sum_{p' \xleftarrow{a} p} a.p' \tag{8}$$

This law, a cornerstone in interleaving models of concurrency [36], is proved in our framework as follows (notice final coalgebra ω is an isomorphism):

$$\begin{aligned}
& \omega\left(\sum_{p' \xleftarrow{a} p} a.p'\right) \\
= \quad & \{\text{definition of } \xleftarrow{a}\} \\
& \omega\left(\sum_{\langle a,p'\rangle \in \omega\, p} a.p'\right) \\
= \quad & \{\text{equation (6)}\} \\
& \bigcup \mathcal{P}\,\omega\,\{a.p' \mid \langle a, p'\rangle \in \omega\, p\} \\
= \quad & \{\text{definition of } \mathcal{P}\} \\
& \bigcup \{\omega\,(a.p') \mid \langle a, p'\rangle \in \omega\, p\} \\
= \quad & \{\text{equation (7)}\} \\
& \bigcup \{\{\langle a, p'\rangle\} \mid \langle a, p'\rangle \in \omega\, p\} \\
= \quad & \{\cup \text{ reduction}\} \\
& \{\langle a, p'\rangle \mid \langle a, p'\rangle \in \omega\, p\} \\
= \quad & \{\text{functions}\} \\
& \omega\, p
\end{aligned}$$

4. The coordination layer

The fundamental notion proposed in this paper, as a basis for service orchestration, is that of a *configuration*. As explained in the Introduction, this captures the intuition that services cooperate through specific *connectors* which abstracts the idea of an intermediate *glue code* to handle interaction. This section concentrates on the coordination layer addressing

- what *connectors* are and how they compose.

Our framework is a variant of the Reo model for which a specific semantics is proposed. Section 5 will, later, complete the picture by defining

- how the services layer is specified
- and how services' *interfaces* and *connectors* interact in configurations.

4.1. Connectors

The orchestration layer is formed by software connectors, regarded as *glueing devices* between services to ensure the flow of data and the meet of synchronization constraints. As in Reo, a connector is a set of interaction *ends*, or *ports* through which messages flow. Each port has an *interaction polarity* (qualifying as a *source* or a *sink* end), but, in general, connectors are blind with respect to the data values flowing through them. Their semantics builds on top of our previous work on component interconnection [18], which is extended here with an explicit annotation of behavioural patterns for their *ports*.

Let $\mathbb{C}$ be a connector with m input and n output ports. Assume $\mathbb{M}$ is a generic type of messages and $\mathbb{P}$ a set of (unique) *port identifiers*. In a number of cases it is necessary to consider a default value to represent *absence* of messages, for example to describe a transition in a connector's state in which a particular port is not involved. Therefore, the type of data flowing through connectors is

$$\mathbb{D} \triangleq \mathbb{M} + \mathbf{1} \tag{9}$$

where $\mathbf{1}$ is the singleton set whose unique element is represented, by convention, as $\perp$.

Elementary connectors are stateless, but to introduce asynchrony, e.g., through a buffered channel, internal states might be considered. Let $\mathbb{U}$ stand for a generic type of state spaces. For example $\mathbb{U}$, may be defined as a sequence of data ($\mathbb{U} = \mathbb{D}^*$) or, as in the specification of a one-fifo buffer below, simply as $\mathbb{U} = \mathbb{D}$. Default value $\perp$ stands, at this level, for absence of stored information in the connector's memory. Formally, the behaviour of a connector is defined as follows

Definition 2. The specification of a connector $\mathbb{C}$ is given by a relation $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \times \mathbb{U} \longleftarrow \mathbb{D}^m \times \mathbb{U}$ which records the flow of data, and a process expression $\mathsf{port}.[\![\mathbb{C}]\!]$ which specifies the behavioural pattern for port activation.

As expected, behavioural patterns are given by terms in a process algebra according to the following grammar:

$$P ::= \mathbf{0} \mid a.P \mid P + P \mid P \otimes P \mid P \boxplus P \mid P \boxtimes P \mid \sigma\, P \mid \mathsf{fix}\ (x = P)$$

where $a \in A$ and σ is a substitution, i.e., an action renaming. Combinators $\mathbf{0}$, ., $+$, $\boxtimes$, $\otimes$ and $\boxplus$, were introduced in the previous section. Notation $\mathsf{fix}\ (x = P)$ stands for a fixed point construction, which, as usual, can be abbreviated into an explicit recursive definition.

In a first approximation set A, of actions, above is taken as $A = \mathcal{P}\,(\mathbb{P} \cup \{\tau\})$, i.e., as sets of connectors' ends plus a special symbol, τ, to represent any unobservable action. The introduction of τ is technically entailed by the semantics of the *hook* combinator, as explained below. Regarded as an action, a port identifier a asserts the activation of the corresponding port, i.e., the fact that a datum crosses its boundaries. Note that choosing A as a *set* of port identifiers allows for the synchronous activation of several ports in a single computational step. This is enough for the semantics of a number of elementary connectors, as follows.

4.1.1. Synchronous channel

A *synchronous channel* has two ports of opposite polarity. This connector forces input and output to become mutually blocking, in the sense that any of them must wait for the other to be completed.

$$\mathsf{data}.[\![\ a \longrightarrow b\ ]\!] = \mathsf{Id}_{\mathbb{D}\times\mathbb{U}} \quad \text{and} \quad \mathsf{port}.[\![\ a \longrightarrow b\ ]\!] = \mathsf{fix}\ (x = ab.x)$$

Here, as well as in the next three cases, state information is irrelevant. Therefore, $\mathbb{U} = \mathbf{1}$. Its semantics is simply the identity relation on data domain $\mathbb{D}$ and its behaviour is captured by the simultaneous activation of its two ports.

4.1.2. Unreliable channel

Any coreflexive relation, that is any subset of the identity, provides channels which can loose information, thus modelling unreliable communications. Therefore, we define, an *unreliable channel* as

$$\mathsf{data}.[\![\ a \xrightarrow{\diamond} b\ ]\!] \subseteq \mathsf{Id}_{\mathbb{D}\times\mathbb{U}} \quad \text{and} \quad \mathsf{port}.[\![\ a \xrightarrow{\diamond} b\ ]\!] = \mathsf{fix}\ (x = ab.x + a.x)$$

The behaviour is given by a choice between a successful communication, represented by the simultaneous activation of the ports or, by a failure, represented by the single activation of the input port.

4.1.3. Filter channel

This is a channel in which some messages are discarded in a controlled way, according to a given predicate $\phi : \mathbb{B} \longleftarrow \mathbb{D}$. Therefore, all messages failing to verify ϕ are lost. Regarding predicate ϕ as a relation $\mathsf{R}_\phi : \mathbb{D} \times \mathbb{U} \longleftarrow \mathbb{D} \times \mathbb{U}$ such that $(d, \bot)\mathsf{R}_\phi(d', \bot)$ iff $d = d' \wedge (\phi\, d)$, define

$$\mathsf{data}.[\![\ a \xrightarrow{\phi} b\ ]\!] = \mathsf{R}_\phi \quad \text{and} \quad \mathsf{port}.[\![\ a \xrightarrow{\phi} b\ ]\!] = \mathsf{fix}\ (x = ab.x + a.x)$$

4.1.4. Drain

A drain has two source, but no sink, ends. Therefore, it looses any data item crossing its boundaries. A drain is *synchronous* if both write operations are requested to succeed at the same time (which implies that each write attempt remains pending until another write occurs in the other end). It is *asynchronous* if, on the other hand, write operations in the two ports do not coincide. The formal definitions are, respectively,

$$\mathsf{data}.[\![\ a \longmapsto b\ ]\!] = (\mathbb{D} \times \mathbb{U}) \times (\mathbb{D} \times \mathbb{U}) \quad \text{and} \quad \mathsf{port}.[\![\ a \longmapsto b\ ]\!] = \mathsf{fix}\ (x = ab.x)$$

and

$$\mathsf{data}.[\![\ a \overset{\triangledown}{\longmapsto} b\ ]\!] = (\mathbb{D} \times \mathbb{U}) \times (\mathbb{D} \times \mathbb{U}) \quad \text{and} \quad \mathsf{port}.[\![\ a \overset{\triangledown}{\longmapsto} b\ ]\!] = \mathsf{fix}\ (x = a.x + b.x)$$

4.1.5. Fifo$_1$

This is a channel with a buffer of a single position. Thus $\mathbb{U} = \mathbb{D}$

$$\mathsf{data}.[\![\ a \longmapsto\!\square\!\rightarrow b\ ]\!] = \mathsf{R}_\square \quad \text{and} \quad \mathsf{port}.[\![\ a \longmapsto\!\square\!\rightarrow b\ ]\!] = \mathsf{fix}\ (x = a.b.x)$$

where $\mathsf{R}_\square$ is given by the following clauses, for all $d, u \in \mathbb{D}$,

$$(\bot, d)\ \mathsf{R}_\square\ (d, \bot) \tag{10}$$

$$(u, \bot)\ \mathsf{R}_\square\ (\bot, u) \tag{11}$$

Clause (10) corresponds to the effect of an input at port a, whereas clause (11) captures output at port b, which requires the presence of a datum in the internal state. Notice that clause (10) precludes input whenever the buffer is full. An *eager* alternative overwrites the buffer's memory:

$$(\bot, d)\ \mathsf{R}_\square\ (d, u) \tag{12}$$

Similarly, clause (11) defines b as what is often called a *get* port: data is read and removed from the connector. Alternatively, a single *read* port can be specified by

$$(u, u)\ \mathsf{R}_\square\ (\bot, u) \tag{13}$$

In a number of practical situations service orchestration depends not only on port activation, but also on the absence of service requests at particular ports in a configuration. A typical example is provided by one of the basic channels considered in Reo: the *lossy channel*, which acts as a synchronous one if both an input and an output request are pending on its source and sink ends, respectively, but looses any data item on input on the absence of an output request in the other end. Notice this behaviour is distinct from that of the *unreliable* channel, which looses data non deterministically.

To handle these cases we enrich the specification of the set of actions A to include *negative port activations*, or more rigorously stated, *absence of port requests*, denoted, for each port p, by $\tilde{p}$. Technically, actions are given by datatype

$$A = \mathcal{P}\,(\mathbb{P} \cup \{\tau\}) \times \mathcal{P}\,\mathbb{P} \tag{14}$$

subject to the following invariant

$$\mathsf{disjoint}\ \langle pos, neg\rangle = pos \cap neg = \emptyset \tag{15}$$

whose values are represented according the following abbreviation

$$\langle\{a, b, c\}, \{d, f\}\rangle \overset{\text{abv}}{=} abc\tilde{d}\tilde{f}$$

Therefore, the specification of Reo's lossy channel becomes

$$\mathsf{data}.[\![\ \bullet \longmapsto\!\cdots\!\succ \bullet\ ]\!] \subseteq \mathsf{Id}_{\mathbb{D}} \ \textit{and}\ \ \mathsf{port}.[\![\ \bullet \longmapsto\!\cdots\!\succ \bullet\ ]\!] = \mathsf{fix}\ (x = ab.x + a\tilde{b}.x)$$

We will confirm later that this definition captures exactly the operational intuition underlying the lossy channel.

4.2. New connectors from old

Complex connectors are built out of simpler ones through a set of *combinators*, as antecipated in Section 2.

In the sequel, let $t_{\#a}$, for $t \in \mathbb{D}^n \times \mathbb{U}$ and $a \in \mathcal{P}$, denote the component of data tuple t corresponding to port a. Define $t_{|a}$ as a tuple identical to t from which component $t_{\#a}$ has been deleted, i.e., $t_{|a} = (t_n, \ldots, t_{\#a+1}, t_{\#a-1}, \ldots, t_0, u)$.

4.2.1. Aggregation

There are two combinators, denoted by $\boxtimes$ and $\boxplus$, whose effect is to place their arguments side-by-side, with no direct interaction between them. They distinguish one of the other by the way the arguments's behavioural patterns are combined: through *parallel* composition or *interleaving*, respectively. Formally,

$$\mathsf{port}.[\![\mathbb{C}_1 \boxtimes \mathbb{C}_2]\!] = \mathsf{port}.[\![\mathbb{C}_1]\!] \boxtimes \mathsf{port}.[\![\mathbb{C}_2]\!]$$
$$\mathsf{port}.[\![\mathbb{C}_1 \boxplus \mathbb{C}_2]\!] = \mathsf{port}.[\![\mathbb{C}_1]\!] \boxplus \mathsf{port}.[\![\mathbb{C}_2]\!]$$

taking, in the first case, $\theta = \cup$.

At data level both combinators behave as a relational product upon some re-arranging to separate state from data information. Such housekeeping task is done by Set-isomorphism $\mathsf{m} : (A \times B) \times (C \times D) \cong (A \times C) \times (B \times D)$. Formally,

$$\mathsf{data}.[\![\mathbb{C}_1 \boxtimes \mathbb{C}_2]\!] = \mathsf{data}.[\![\mathbb{C}_1 \boxplus \mathbb{C}_2]\!] = \mathsf{m} \cdot (\mathsf{data}.[\![\mathbb{C}_1]\!] \times \mathsf{data}.[\![\mathbb{C}_2]\!]) \cdot \mathsf{m}$$

which, going pointwise and denoting by R and S the arguments' data relations, amounts to

$$((\vec{d'}, \vec{e'}), (u', v'))\ R \boxtimes S\ ((\vec{d}, \vec{e}), (u, v)) \quad \equiv \quad (\vec{d'}, u')\ R\ (\vec{d}, u) \ \wedge\ (\vec{e'}, v')\ S\ (\vec{e}, v)$$

4.2.2. Hook

This combinator encodes a *feedback* mechanism, drawing a direct connection between an output and an input port. This has a double consequence: the connected ports must be activated simultaneously and become externally non observable. Formally, such conditions must be expressed in $\mathsf{port}.[\![\mathbb{C} \upharpoonleft_i^j]\!]$. Therefore, a new process combinator $\mathsf{hide}\, c$, parametric on a set $c \subseteq Act - \{0\}$, is defined which first prunes the behaviour of $\mathbb{C}$ by removing all computations exhibiting occurrences of non empty strict subsets of c, because ports in c must be activated simultaneously. The combinator then hides all references to c in the remaining computations, either by removing them when occurring in a strictly larger context or by mapping them to an unobservable action τ when occurring isolated. Following the method described in Section 3, this is defined as $\mathsf{hide}\, c = [\![(\alpha_{\mathsf{hide}\, c})]\!]$ where

$$\alpha_{\mathsf{hide}\, c} = \nu \xrightarrow{\omega} \mathcal{P}\,(Act \times \nu) \xrightarrow{\mathsf{h}_c} \mathcal{P}\,(Act \times \nu)$$
$$\mathsf{h}_c\ s = \{\langle a \setminus c, u\rangle |\ \langle a, u\rangle \in s \ \wedge\ ((a \cap c \neq \emptyset) \ \Rightarrow\ c \subset a)\} \cup \{\langle \tau, u\rangle |\ \langle c, u\rangle \in s\}$$

Thus

$$\mathsf{port}.[\![\mathbb{C} \upharpoonleft_i^j]\!] = \mathsf{hide}\ \{i, j\}\ \mathsf{port}.[\![\mathbb{C}]\!]$$

If $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$, the effect of *hook* on the data flow relation is modelled by relation

$$\mathsf{data}.[\![\mathbb{C} \upharpoonleft_i^j]\!] : \mathbb{D}^{n-1} \longleftarrow \mathbb{D}^{m-1}$$

$$t'_{|j}\ (\mathsf{data}.[\![\mathbb{C} \upharpoonleft_i^j]\!])\ t_{|i} \quad \text{iff} \quad t'\ (\mathsf{data}.[\![\mathbb{C}]\!])\ t \ \wedge\ t'_{\#j} = t_{\#i}$$

Example 1. Consider connectors $\mathbb{C}$ and $\mathbb{F}$, both with an input and an output port, named a, a' in the first case, and b, b' in the second. Let us analyse composition $(\mathbb{C} \boxtimes \mathbb{F}) \upharpoonleft_{a'}^{b}$. At the data level, one gets

$$\begin{array}{ll} & (y, (u', v'))\ \mathsf{data}.[\![(\mathbb{C} \boxtimes \mathbb{F}) \upharpoonleft_{a'}^{b}]\!]\ (x, (u, v)) \\ = & \quad \{\ \text{unfolding definitions}\ \} \\ & \exists_z\ .\ \ ((z, y), (u', v'))\ \mathsf{data}.[\![\mathbb{C} \boxtimes \mathbb{F}]\!]\ ((x, z), (u, v)) \\ = & \quad \{\ \text{unfolding definitions}\ \} \\ & \exists_z\ .\ \ (z, u')\ \mathsf{data}.[\![\mathbb{C}]\!]\ (x, u) \ \wedge\ (y, v')\ \mathsf{data}.[\![\mathbb{F}]\!]\ (z, v) \end{array}$$

which shows that the *hook combinator* encodes a form of relational composition which is *partial* in the sense that only part of the output is fed back as new input. For the behavioural component, consider $\mathbb{C}$ and $\mathbb{F}$ as synchronous channels. Then,

$$\mathsf{port}.[\![(\mathbb{C} \boxtimes \mathbb{F}) \upharpoonleft_{a'}^{b}]\!] = \mathsf{fix}\ (x = ab'.x)$$

$$\mathbb{M} \triangleq (a \longrightarrow a' \boxplus b \longrightarrow b')\,{}^{a'}_{b'} > w \quad = \quad \begin{matrix} a \searrow \\ & w \\ b \nearrow \end{matrix}$$

Fig. 2. A *merger*.

because the other two terms in the expansion fix $(x = aa'.x + bb'.x + aa'bb'.x)$ contain strict subsets of $c = \{a', b\}$. Note that the synchronous channel always acts as the identity for *hook*. The reader may also easily confirm that, for $\mathbb{C}$ and $\mathbb{F}$ defined as $Fifo_1$ channels, one gets

$$\mathsf{port}.[\![(\mathbb{C} \boxtimes \mathbb{F}) \curvearrowleft^{b}_{a'}]\!] = \mathsf{fix}\ (x = a.\tau.(a.b'.x + b'.a.\tau.x))$$

Ignoring non observable actions, corresponds to fix $(x = a.(a.b'.x + b'.a.x))$, which is the behavioural pattern of a two position buffer.

4.2.3. Join

The last combinator to be considered here is called *join* and its effect is to plug ports with identical polarity. The aggregation of *output* ports is done by a *right join* $(\mathbb{C}\ {}^{i}_{j} > z)$, where $\mathbb{C}$ is a connector, i and j are ports and z is a fresh name used to identify the new port. Port z receives asynchronously messages sent by either i or j. When messages are sent at same time the combinator chooses one of them non deterministically.

On the other hand, aggregation of *input* ports resorts to a *left join* $(z <^{i}_{j} \mathbb{C})$. This behaves like a *broadcaster* sending synchronously messages from z to both i and j. Formally, for $\mathsf{data}.[\![\mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^m$, define

Right join:

The data flow relation $\mathsf{data}.[\![\mathbb{C}\ {}^{i}_{j} > z]\!] : \mathbb{D}^{n-1} \longleftarrow \mathbb{D}^m$ for this operator is given by

$$r\ (\mathsf{data}.[\![\mathbb{C}\ {}^{i}_{j} > z]\!])\ t \quad \text{iff} \quad t'\ (\mathsf{data}.[\![\mathbb{C}]\!])\ t \ \wedge\ r_{|z} = t'_{|i,j} \ \wedge\ (r_{\#z} = t'_{\#i} \ \vee\ r_{\#z} = t'_{\#j})$$

At the behavioural level, its effect is that of a renaming operation

$$\mathsf{port}.[\![(\mathbb{C}\ {}^{i}_{j} > z)]\!] = \{z \leftarrow i, z \leftarrow j\}\ \mathsf{port}.[\![\mathbb{C}]\!]$$

Example 2. The *merger* connector depicted in Fig. 2 is obtained by a right join of the sink ends of two interleaved synchronous channels. Its behavioural pattern is

$$\mathsf{port}.[\![\mathbb{M}]\!] = \mathsf{fix}\ (x = aw.x + bw.x)$$

as the reader may easily compute.

Left join:

The behaviour of a left join is a little more complex: before renaming, all computations of $\mathbb{C}$ in which ports i and j are activated independently of each other must be removed. Again this is specified by a new process combinator force c which forces the joint activation of a set c of ports. Formally, $\mathsf{force}\ c = [\![(\alpha_{\mathsf{force}\ c})]\!]$ where

$$\alpha_{\mathsf{force}\ c} = \nu \xrightarrow{\omega} \mathcal{P}\ (Act \times \nu) \xrightarrow{\mathsf{f}_c} \mathcal{P}\ (Act \times \nu)$$

$$\mathsf{f}_c\ s = \{\langle a, u\rangle \in s \mid a \cap c \ \subseteq\ \{\emptyset, c\}\}$$

Thus

$$\mathsf{port}.[\![(z <^{i}_{j} \mathbb{C})]\!] = \{z \leftarrow i, z \leftarrow j\}\ \mathsf{force}\ \{i, j\}\ \mathsf{port}.[\![\mathbb{C}]\!]$$

On the other hand, the data flow specification $\mathsf{data}.[\![z <^{i}_{j} \mathbb{C}]\!] : \mathbb{D}^n \longleftarrow \mathbb{D}^{m-1}$ is given by

$$t'\ (\mathsf{data}.[\![z <^{i}_{j} \mathbb{C}]\!])\ r \quad \text{iff} \quad t'\ (\mathsf{data}.[\![\mathbb{C}]\!])\ t \ \wedge\ r_{|z} = t_{|i,j} \ \wedge\ r_{\#z} = t_{\#i} = t_{\#j}$$

Example 3. A simple, but useful, illustration of this combinator is the *broadcaster* connector depicted in Fig. 3. It is obtained by a left join of the source ports of two synchronous channels put in parallel. Its behavioural pattern is computed as follows:

$$\begin{aligned}
&\mathsf{port}.[\![\mathsf{B}]\!] \\
\triangleq\ &\{\text{ defined as}\} \\
&a <^{c'}_{b'} (\ b' \longrightarrow b \ \boxtimes\ c' \longrightarrow c\ ) \\
\sim\ &\{\text{ definition of left join and synchronous channel}\} \\
&\{a \leftarrow c', a \leftarrow b'\}\ \mathsf{force}\ \{c', b'\}\ (\mathsf{fix}\ (x = c'c.x) \boxtimes \mathsf{fix}\ (x = b'b.x)) \\
\sim\ &\{\text{ parallel composition and expansion law (8)}\} \\
&\{a \leftarrow c', a \leftarrow b'\}\ \mathsf{force}\ \{c', b'\}\ (\mathsf{fix}\ (x = c'c.x + b'b.x + c'cb'b.x)) \\
\sim\ &\{\text{ definition of force and substituion}\} \\
&\mathsf{fix}\ (x = acb.x)
\end{aligned}$$

$$\mathbb{B} \triangleq a <^{c'}_{b'} (b' \longrightarrow b \boxtimes c' \longrightarrow c) \quad = \quad \begin{array}{l} a \overset{\nearrow}{\searrow} \begin{array}{l} c \\ b \end{array} \end{array}$$

Fig. 3. A *broadcaster*.

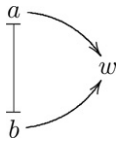


Fig. 4. Imp — An '*impossible*' connector.

4.3. Towards a connector calculus

Notions of connector equivalence and refinement can be defined, entailing the basis for a connector calculus to reason about the coordination layer of applications. For connectors with identical signatures, refinement corresponds, at the data level, to relational inclusion, as one would expect. In this subsection, however, our attention will be focussed on the behavioural side.

Having defined behaviours, in Section 3, coalgebraically, we get for free the notion of bisimulation associated to functor $\mathcal{P}(A \times \mathsf{Id})$:

Definition 3. A relation S on processes is a simulation iff

$$pSq \Rightarrow \forall_{\langle c,p'\rangle \in \omega\, p} \,.\, \exists_{\langle c',q'\rangle \in \omega\, q} \,.\, c = c' \wedge p'Sq' \tag{16}$$

A bisimulation is a simulation whose relational converse is also a simulation. As usual, we denote by $\precsim$ and $\sim$ the similarity and bisimilarity relation, respectively, corresponding to the greatest simulation and bisimulation.

Clearly, by (16), one gets

$$\mathsf{port}.[\![\bullet \longrightarrow \bullet]\!] \precsim \mathsf{port}.[\![\bullet \cdots\!\!> \bullet]\!]$$

or

$$\mathsf{port}.[\![a \longrightarrow b \boxtimes b \longrightarrow c]\!] \sim \mathsf{port}.[\![a \longrightarrow c]\!]$$

A richer, weaker, inequational calculus is derived from the fact that actions in A form a semilattice

$$\langle \mathcal{P}(\mathbb{P} \cup \{\tau\}) \times \mathcal{P}\,\mathbb{P}, \subseteq \times \subseteq, \langle \emptyset, \emptyset \rangle \rangle$$

by relaxing (16) to require $c(\subseteq \times \subseteq)c'$ instead of action equality $c = c'$. Under this new similarity relation, asynchrony appears as a refinement of synchrony as in, e.g.,

$$\mathsf{port}.[\![\bullet \longmapsto\!\!\!\!\square\!\!\!\!> \bullet]\!] \precsim \mathsf{port}.[\![\bullet \longrightarrow \bullet]\!]$$

Although the development of a connector calculus will not be pursued here, it is worthwhile to prove the following fact on the '*impossible*' (or deadlocked) connector depicted in Fig. 4, which indicates our semantics is compliant with the operational description of Reo in [11]:

$$\mathsf{port}.[\![\mathsf{Imp}]\!] \sim \mathbf{0} \tag{17}$$

Actually, Imp is built as

$$\mathsf{Imp} \triangleq a <^{a}_{d} b <^{b}_{e} (d \longmapsto e \boxtimes (a \longrightarrow a' \boxplus b \longrightarrow b') \overset{a'}{}_{b'} > w))$$

Therefore

$$\begin{array}{ll} & \mathsf{port}.[\![\mathsf{Imp}]\!] \\ \sim & \{\text{definiton of a left join, } \sigma = \{a \leftarrow d, b \leftarrow e\}\} \\ & \sigma\ \mathsf{force}\,\{a,b,d,e\}\,(\mathsf{fix}\,(x = de.x) \boxtimes \mathsf{fix}\,(x = aw.x + bw.x)) \\ \sim & \{\text{expansion law (8)}\} \\ & \sigma\ \mathsf{force}\,\{a,b,d,e\}\,(\mathsf{fix}\,(x = deaw.x + debw.x + de.x + aw.x + bw.x)) \\ \sim & \{\text{definition of force}\} \\ & \mathbf{0} \end{array}$$

It is also instructive to compute the semantics of XR, the *exclusive router* connector depicted in Fig. 5, which is not suitably captured by classical Reo semantics [13,14]. The intended behaviour for this connector is to transmit either in b or c, but not in both, whatever receives in a.

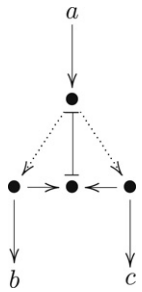
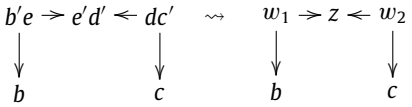
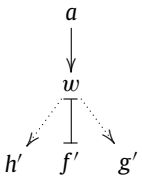


Fig. 5. XR — The *exclusive router* connector.

One component of XR, depicted in the lower part of the diagram, is the *right join* by e' and d' mapping to new port z, of two *broadcasters* obtained by left joining synchronous channels. The assembly process of XR_1 is represented as



The computed behavioural pattern is fix $(x = bzw_1.x + czw_2.x)$. The other component, XR_2 is a left join of two lossy channels and a drain, mapping their source ports, h, g and f, to w, sequentially composed with a synchronous channel from a to a', i.e.,



Its behavioural pattern is computed as follows:

$$
\begin{aligned}
& \mathsf{port}.[\![\mathsf{XR}_2]\!] \\
\sim \quad & \{\text{definiton of } \mathsf{XR}_2\} \\
& \mathsf{port}.[\![(\ a \longrightarrow a \ \boxtimes w <^{f}_{r} r <^{h}_{g} (\ h \cdots\!\!\succ h' \ \boxtimes \ f \longmapsto f' \ \boxtimes \ g \cdots\!\!\succ g'\ ))\ {}^{w}\!\!\upharpoonleft_{a'}]\!] \\
\sim \quad & \{\text{definitions of channels, left join and hook; expansion law (8)}\} \\
& \mathsf{hide}\{a', w\}\ (\mathsf{fix}\ (x = aa'.x) \ \boxtimes \\
& \qquad\qquad \mathsf{fix}\ (x = wh'f'g'.x + w\widetilde{h'}f'g'.x + wh'f'\widetilde{g'}.x + wf'\widetilde{h'g'}.x)) \\
\sim \quad & \{\text{definition of hide}\} \\
& \mathsf{fix}\ (x = ah'f'g'.x + a\widetilde{h'}f'g'.x + ah'f'\widetilde{g'}.x + af'\widetilde{h'g'}.x)
\end{aligned}
$$

Finally, the connector XR is assembled as

$$\mathsf{XR} \triangleq (\mathsf{XR}_1 \boxtimes \mathsf{XR}_2)\ {}^{w_1,w_2,f'}\!\!\upharpoonleft_{h',g',w} \tag{18}$$

leading, as expected, to

$$
\begin{aligned}
& \mathsf{port}.[\![\mathsf{XR}]\!] \\
\sim \quad & \{\text{by (18)}\} \\
& \mathsf{hide}\{h', g', w, w_1, w_2, f'\}\ (\mathsf{port}.[\![\mathsf{XR}_1]\!] \boxtimes \mathsf{port}.[\![\mathsf{XR}_2]\!]) \\
\sim \quad & \{\text{computed above}\} \\
& \mathsf{hide}\{h', g', w, w_1, w_2, f'\}\ (\mathsf{fix}\ (x = bzw_1.x + czw_2.x) \ \boxtimes \\
& \qquad\qquad \mathsf{fix}\ (x = ah'f'g'.x + a\widetilde{h'}f'g'.x + ah'f'\widetilde{g'}.x + af'\widetilde{h'g'}.x)) \\
\sim \quad & \{\text{expansion law (8) and definition of hide}\} \\
& \mathsf{fix}\ (x = ab.x + ac.x)
\end{aligned}
$$

5. Services and configurations

5.1. Services

This section discusses how the behaviour of services is specified and how they interact with the software connectors introduced in the previous section. As the reader may expect, a service specification is given by process term over the collection of (input and output) *ports* where the active entities inside the service consume or make available data items. Such process term represents the service *bussiness protocol*, its externally perceived behaviour referred to, in the sequel, as the service *use pattern*.

Use patterns, of course, are defined in the same process language used for specifying connectors. The set of actions A, however, is restricted to sets of ports (therefore excluding 'negative' port activations and unobservable actions). Formally, let $\mathcal{P}$ be the set of port identifiers and S (the specification of) a service. Its use pattern, $use(S)$ is a process term over $A = \mathcal{P}\,\mathbb{P}$.

The approach to service orchestration proposed in this paper precludes direct interaction between services — all interaction being mediated by a specific connector. Therefore, if two services are active in a particular application, their joint behaviour will allow the realization of both use patterns either simultaneously or in an independent way. Formally,

Definition 4. The joint behaviour of a collection $\{S_i |\ i \in n\}$ of web services is given by

$$use(S_1)\ \boxtimes\ \ldots\ \boxtimes\ use(S_n)$$

where the interaction discipline is fixed by $\theta\ =\ \cup$, i.e., the synchronisation of actions in α and β corresponds to the simultaneous realization of all of them.

This joint behaviour is computed by the expansion law (8), always respecting the interaction discipline given by θ. The following example illustrates this construction.

Example 4. Consider a service S_1 with two ports a and b whose use pattern is restricted to the activation of either a or b, forbidding their simultaneous occurrence. The expected behaviour is captured by

$$use(S_1) = \mathsf{fix}\ (x\ =\ a.x + b.x)$$

Now consider another service, S_2, with ports c and d whose behaviour is given by the co-occurrence of actions in both ports. Therefore,

$$use(S_2) = \mathsf{fix}\ (x'\ =\ cd.x'), \quad \text{where } cd\ \stackrel{\text{abv}}{=}\ \{c, d\}$$

According to Definition 4, the joint behaviour of S_1 and S_2 is

$$use(S_1)\ \boxtimes\ use(S_2)\ = \mathsf{fix}\ (x\ =\ acd.x + bcd.x + a.x + b.x + cd.x)$$

As a final example, consider still another service S_3, with ports e and f activated in strict order, i.e.,

$$use(S_3) = \mathsf{fix}\ (y\ =\ e.f.y)$$

Clearly, expansion leads to $use(S_2) \boxtimes use(S_3)\ = P$, where

$$P\ = \mathsf{fix}\ (x\ =\ cd.x + e.Q + cde.Q)$$
$$Q\ = \mathsf{fix}\ (x\ =\ cd.x + f.P + cdf.P)$$

5.2. The 'whole picture'

A *configuration*, like the one depicted in Fig. 1 is simply a collection of services, characterized by their interfaces, interconnected through an *orchestrator*, i.e., a connector built from elementary connectors using the combinators introduced in the previous section. Actually, we have now all the ingredients to replace the empty circle in that figure by a suitable connector which enforces strict alternation of data sources. Such is the purpose of the *alternate merger* depicted in Fig. 6. Formally, $\mathbb{AM}$ is defined as

$$b <_f^{d'} a <_c^{d}\ (\ c \longrightarrow c'\ \boxtimes\ d \longmapsto d'\ \boxtimes\ f \longrightarrow\!\!\!\!\square\ f')_{f'}^{c'} > w$$

Following the method illustrated in the previous section, its behavioural pattern is easily computed from this expression:

$$\mathsf{port}.[\![\mathbb{AM}]\!] = \mathsf{fix}\ (x\ =\ abw.w.x) \tag{19}$$

Formally,

Definition 5. A configuration involving a collection $S = \{S_i |\ i \in n\}$ of services is a tuple

$$\langle U, \mathbb{C}, \sigma \rangle \tag{20}$$

where $U = use(S_1) \boxtimes use(S_2) \boxtimes \cdots \boxtimes use(S_n)$ is the (joint) use pattern for S, $\mathbb{C}$ is a connector and σ a mapping of ports in S to ports in $\mathbb{C}$.

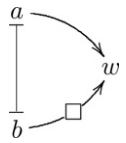


Fig. 6. An *alternate merger*.

The role of renaming σ in the definition above is to syntactically enforce a link between a service port and a connector end. Clearly, σ respects polarities: output (respectively, input) service ports can only be connected to connectors source (respectively, sink) ends. Interaction is achieved by the simultaneous activation of identically named ports.

Actually, the relevant point concerning configurations is the semantics of interaction between the *connector's behavioural pattern* and the *joint use patterns* of the involved services. This is captured by a synchronous product $\otimes$ for a quite peculiar θ, which is expected to capture the requirements below. Recall that, at each point in the execution of a configuration, θ 'decides' the result of the interaction combining a set of ports offered by the services' side and the sets of positive and negative connector's ends. Therefore,

- There is no interaction if the connector requires absence of port requests in an end linked to a port activated by a service.
- Similarly, there is no interaction if the connector's side offers free ports (i.e., ports that are not connected to services).
- The dual situation is allowed, i.e., if the services' side offer activation of all ports plugged to the ones offered by the connector, their intersection is the resulting interaction.
- Finally, free ports on the service side (i.e., ports that are not connected to a connector's end) are not affected by θ: their activation depends only on the service they belong to.

Formally, this is captured in the following definition.

Definition 6. The behaviour $bh(\Gamma)$ of a configuration $\Gamma = \langle U, \mathbb{C}, \sigma \rangle$ is given by

$$bh(\Gamma) = \sigma\, U \;\otimes\; \mathsf{port}.[\![\mathbb{C}]\!] \tag{21}$$

where θ underlying the $\otimes$ connective is given by

$$s\ \theta\ \langle p, np \rangle = \begin{cases} (s \cup \{\tau\}) \cap (p \cup \mathsf{free}) & \Leftarrow\ s \cap np = \emptyset \wedge p \subseteq (s \cup \{\tau\}) \\ s \cap \mathsf{free} & \Leftarrow\ otherwise \end{cases} \tag{22}$$

where free denotes the set of unplugged ports in U, i.e., not in the domain of mapping σ.

Note that in the above definition θ relates different types of actions, its signature being $\theta : (\mathcal{P}\,\mathbb{P} \cup \{\tau\}) \longleftarrow \mathcal{P}\,\mathbb{P} \times (\mathcal{P}\,(\mathbb{P} \cup \{\tau\}) \times \mathcal{P}\,\mathbb{P})$. This means the behaviour of a configuration is expressed in terms of services' ports and the unobservable action τ (which, as explained on introducing the *hook* combinator, can not be ignored). It also means that the resulting synchronous product is not commutative, which however does not restrict the expressive power of this approach.

5.3. Examples

In the sequel the use of configurations, and the computation of their behaviours, is illustrated by two examples:

Example 5. Consider an elementary *banking system* composed by an *ATM* machine, a *Bank*, and a *DBRep* service whose purpose is to backup all the messages flowing through the connector. Therefore, all messages are replicated before being stored. Configuration *BS*, depicted in Fig. 7, is specified as

$$BS = \langle WBS,\ \mathbb{CON},\ \sigma_{BS} \rangle$$

where

$$\begin{aligned} WBS\ &= use(ATM) \boxtimes use(Bank) \boxtimes use(DBRep) \\ \sigma_{HS}\ &= \{a \leftarrow A_{rq}, e \leftarrow A_{rs}, c \leftarrow DB_r, f \leftarrow DB_p, d \leftarrow B_{rs}, b \leftarrow B_{rq}\} \end{aligned}$$

Consider the following use patterns of each web service after port renaming by σ_{BS}:

$$\begin{aligned} use(ATM) &= \mathsf{fix}\ (x = a.e.x) \\ use(Bank) &= \mathsf{fix}\ (y = b.d.y) \\ use(DBRep) &= \mathsf{fix}\ (z = c.z + f.z + cf.z) \end{aligned}$$

Connector $\mathbb{CON}$ behaves like a double broadcaster (hence its name). Its behaviour allows for both the simultaneous or independent activation of each broadcast (co_1 or co_2) as shown by the following computation:

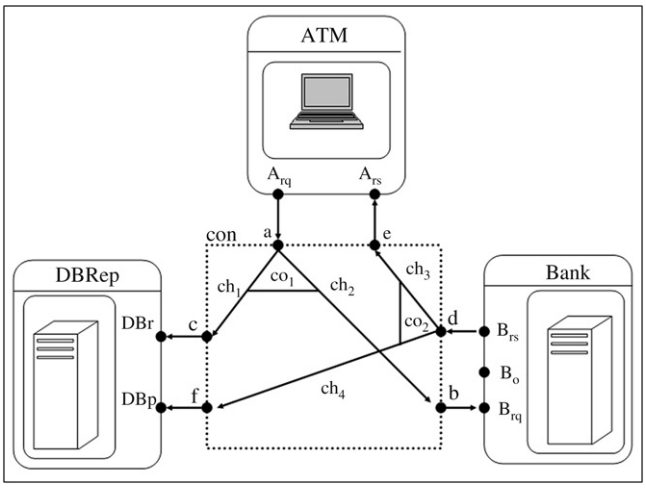


Fig. 7. Bank system.

$$\mathsf{port}.[\![ch_1]\!] = \mathsf{fix}\,(x = b'b.x),\ \ \mathsf{port}.[\![ch_2]\!] = \mathsf{fix}\,(x = c'c.x)$$
$$\mathsf{port}.[\![co_1]\!] = \mathsf{port}.[\![(a <^{b'}_{c'} (ch_1 \boxtimes ch_2))]\!] = \mathsf{fix}\,(x = abc.x)$$
$$\mathsf{port}.[\![ch_3]\!] = \mathsf{fix}\,(x = e'e.x),\ \ \mathsf{port}.[\![ch_4]\!] = \mathsf{fix}\,(x = f'f.x)$$
$$\mathsf{port}.[\![co_2]\!] = \mathsf{port}.[\![(d <^{e'}_{f'} (ch_3 \boxtimes ch_4))]\!] = \mathsf{fix}\,(x = def.x)$$
$$\mathsf{port}.[\![\mathbb{CON}]\!] = \mathsf{port}.[\![(co_1 \boxtimes co_2)]\!] = \mathsf{fix}\,(x = abc.x + def.x + abcdef.x)$$

To determine $bh(BS)$ one needs to expand *WBS*. According to (22), we need only to look for summands prefixed by sets of ports which are super-sets of prefix sets in $\mathsf{port}.[\![\mathbb{CON}]\!]$. For the first level of expansion alternative $abc.(e.x \boxtimes d.y \boxtimes z)$ is the only one to θ-compose with abc in $\mathsf{port}.[\![\mathbb{CON}]\!]$, resulting in abc again. Then, consider the expansion of term $(e.x \boxtimes d.y \boxtimes z)$: the only alternative worth to consider (i.e., which does not lead to 0 on θ-composition) is $edf.(x \boxtimes y \boxtimes z)$, the resulting interaction being edf. From this point on the same expansion pattern repeats. This means that $bh(BS)$ becomes:

$$bh(BS) = \mathsf{fix}\,(x = abc.edf.x) \tag{23}$$

Notice how the particular use patterns in the web services act as a *constraint* over the admissible behaviour of connector $\mathbb{CON}$.

This example may be also used to check how definition (22) deals with the presence of unplugged ports, such us port B_o in service *Bank*. Consider, then, the following two alternatives for the use pattern of service *Bank*:

$$use(Bank) = \mathsf{fix}\,(y = bB_o.d.y) \tag{24}$$
$$use(Bank) = \mathsf{fix}\,(y = b.d.y + B_o.y) \tag{25}$$

In the expansion of *WBS*, expression (24), which captures the simultaneous activation of ports b and B_o, leads to term $abB_oc.(e.x \boxtimes d.y \boxtimes z)$ which, as $\mathsf{free} = \{B_o\}$, entails

$$bh(BS) = \mathsf{fix}\,(x = abcB_o.edf.x) \tag{26}$$

Alternative (25) specifies that ports b and B_o are activated in alternative: no term with both b and B_o will appear in the expansion and, therefore, $bh(BS)$ remains as given by Eq. (23).

Example 6. For a second example suppose a configuration with two instances of a service modelling a *stack* and responding on ports PUSH and POP. A third service intends to build a (electronic version of a paper) *folder* by providing ports to *turn a page left* (TL), *turn a page right* (TR) and *insert a new page* (IN) in the (right side of the) folder. The problem is to orchestrate the three services in such a way that each service *stack* will manage one of the page piles of the folder.

In [17] a solution is given, in the context of a component-based framework, two components modelling stacks are composed by specific operators to implement an interface for a folder. We sketch here an exogenous coordination solution in which the three services do not interact directly, the emergent behaviour of their orchestration being enabled by a suitable connector $\mathbb{SF}$. The configuration is depicted in Fig. 8. It is a small exercise to compute the behavioural pattern associated to $\mathbb{SF}$:

$$\mathsf{port}.[\![\mathbb{SF}]\!] = \mathsf{fix}\,(x = abc.x + rdt.x + it.x + acbrdt.x + acbit.x)$$

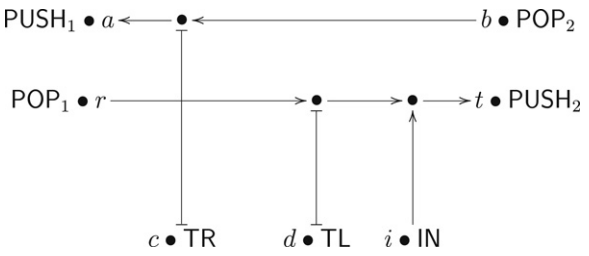


Fig. 8. A *folder* from two stacks.

Assuming the three services involved can activate each of their ports at any time, the global use pattern of the service layer becomes

$$use(Stack) \boxtimes use(FInterface) \boxtimes use(Stack) =$$
$$\mathsf{fix}\ (x = \mathsf{PUSH}_1.x + \mathsf{POP}_1.x + \mathsf{TL}.x + \mathsf{TR}.x + \mathsf{IN}.x + \mathsf{PUSH}_2.x + \mathsf{POP}_2.x)$$

Finally, the behaviour of the configuration is computed according to Definition 5, yielding

$$\mathsf{fix}\ (x = \{\mathsf{TL}, \mathsf{PUSH}_1, \mathsf{POP}_2\}.x + \{\mathsf{TR}, \mathsf{PUSH}_2, \mathsf{POP}_1\}.x + \{\mathsf{IN}, \mathsf{PUSH}_2\}.x)$$

where the port mapping is the one represented in Fig. 8. The emergent behaviour of this configuration includes the simultaneous activation of TL, PUSH_1 and POP_2, 'implementing' a *turn left* action in the folder interface by a synchronisation between a *pop* in the right stack and a *push* in the left one. And dually for a *turn right*. Also notice port PUSH_2 is used either for achieving a *turn right* in the folder or for inserting a new page. None of the three services involved interact directly with one another and even do not need to be aware of each other existence.

6. Conclusions and future work

Service-oriented computing is an emerging paradigm with increasing impact on the way modern software systems are designed and developed. Services are autonomous and heterogeneous computational entities which cooperate, following a loose coupling discipline, to achieve common goals. Web services are one of the most prominent technologies in this paradigm. As an emerging technology, however, it still lacks not only sound semantical *models* but also suitable *calculi* to reason about and transform service-oriented designs.

The approach introduced in this paper for interfaces with *behavioural annotations* and *configurations*, as a basis to represent services' orchestration, may be a step in that direction. It combines two ingredients in which the authors have been working for some time now: models for *exogenous coordination* and a methodology for the *design of process algebras parametric on the interaction discipline*. The latter provides a flexibility which seems to be crucial for the application of process algebras to orchestration problems. The paper also introduces a new semantics for a variant of Reo which is able to deal with a few problems which remain unsolved in classical Reo semantics [13,14]. A full assessment of these semantics and a proper comparison with the so-called *colouring* semantics [24], which also solves, although in a rather complex framework, the above-mentioned problems, is a topic of our current work.

Lots of questions, however, remain open. Maybe the most relevant one concerns *mobility*. It is not clear how the model discussed in this paper can be extended to cope with mobility issues, and, in particular, with dynamic reconfiguration of web services networks. The question is, in fact, more general: we still know very little about the semantics of mobility in the context of exogenous coordination models. Tentative solutions in e.g., Reo [12] or our own contribution documented in [19], are still of an operational nature.

Another interesting topic concerns what is known in the literature as *workflow patterns* [2]. Although their role in the design of service-oriented systems is well recognized, the corresponding formalization is still a 'hot' research topic (see, e.g., [4,8], among many others). We are currently working on their encoding in a slight extension of the formalism used here to specify services' business protocols. In a broader perspective, one may ask whether formal models for service orchestration, like the one discussed in this paper, can be of use in providing precise semantic foundations of emerging languages for web services composition and choreography, as, for example, Ws-Bpel [1] or Ws-Cdl [54]. Finally, it should be mentioned that the effective application of the approach proposed here to real-world problems may require some form of tool support, namely for the systematic application of the expansion law.

References

[1] Business process execution language for web services (version 1.1). www.ibm.com/developerworks/library/specification/ws-bpel/, 2007.
[2] W.M.P.V.D. Aalst, A.H.M.T. Hofstede, B. Kiepuszewski, A.P. Barros, Workflow patterns, Distributed and Parallel Databases 14 (1) (2003) 5–51.
[3] S. Abramsky, S. Gay, R. Nagarajan, Interaction categories and the foundation of typed concurrent programming, in: M. Broy (Ed.), Deductive Program Design: Proc. of the 1994 Marktoberdorf Summer School, in: NATO ASI Series F, Springer Verlag, 1994.

[4] N.R. Adam, V. Atluri, W.-K. Huang, Modeling and analysis of workflows using petri nets, Journal of Intelligent Information Systems 10 (2) (1998) 131–158.
[5] L. Alfaro, T. Henziger, R. Jhala, Compositional methods for probabilistic systems, in: Proc. the 12th International Conference on Concurrency Theory, in: Lect. Notes Comp. Sci., vol. 2154, Springer, 2001, pp. 351–365.
[6] R. Allen, D. Garlan, A formal basis for architectural connection, ACM TOSEM 6 (3) (1997) 213–249.
[7] G. Alonso, F. Casati, H. Kuno, V. Machiraju, Web services — Concepts, architectures and applications, in: Data-centric Systems and Applications, Springer-Verlag, 2004.
[8] R. Amici, F. Corradini, E. Merelli, A process algebra view of coordination models with a case study in computational system biology, in: L. Bocchi, P. Ciancarini (Eds.), Proc. of the First Inter. Workshop on Petri Nets and Coordination, PNC04, Bologna, Italy, 2004, pp. 33–47.
[9] L.F. Andrade, J.L. Fiadeiro, Composition contracts for service interaction, Journal of Universal Computer Science 10 (4) (2004) 751–761.
[10] F. Arbab, Abstract behaviour types: A foundation model for components and their composition, in: F.S. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Proc. First International Symposium on Formal Methods for Components and Objects, FMCO'02, in: Lect. Notes Comp. Sci., vol. 2852, Springer, 2003, pp. 33–70.
[11] F. Arbab, Reo: A channel–based coordination model for component composition, Mathematical Structures in Computer Science 14 (3) (2004) 329–366.
[12] F. Arbab, F. Mavadatt, Coordination through channel composition, in: Proc. Coordination Languages and Models, in: Lect. Notes Comp. Sci., vol. 2315, Springer, 2002.
[13] F. Arbab, J.J.M.M. Rutten, A coinductive calculus of component connectors, in: M. Wirsing, D. Pattinson, R. Hennicker (Eds.), Recent Trends in Algebraic Development Techniques, 16th Inter. Workshop, WADT 2002, Revised Selected Papers, in: Lect. Notes Comp. Sci., vol. 2755, Springer, 2003, pp. 34–55.
[14] C. Baier, M. Sirjani, F. Arbab, J.J.M.M. Rutten, Modeling component connectors in reo by constraint automata, Science of Computer Programming 61 (2) (2006) 75–113.
[15] L.S. Barbosa, Process calculi phà la Bird-Meertens, in: M.L. Andrea Corradini, U. Montanari (Eds.), CMCS'01, Genova, in: Elect. Notes in Theor. Comp. Sci, vol. 44.4, Elsevier, Aprol, 2001, pp. 47–66.
[16] L.S. Barbosa, J.N. Oliveira, Coinductive interpreters for process calculi, in: Proc. of FLOPS'02, in: Lect. Notes Comp. Sci., vol. 2441, Springer, 2002, pp. 183–197.
[17] L.S. Barbosa, J.N. Oliveira, State-based components made generic, in: H.P. Gumm (Ed.), CMCS'03, in: Elect. Notes in Theor. Comp. Sci., vol. 82.1, Elsevier, 2003.
[18] M. Barbosa, L. Barbosa, Specifying software connectors, in: K. Araki, Z. Liu (Eds.), Proc. First International Colloquim on Theoretical Aspects of Computing, ICTAC'04, Guiyang, China, in: Lect. Notes Comp. Sci., vol. 3407, Springer, 2004, pp. 53–68.
[19] M.A. Barbosa, L.S. Barbosa, An orchestrator for dynamic interconnection of software components, in: Proc. 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, MTCoord'06, Bologna, Italy, in: Elect. Notes in Theor. Comp. Sci., vol. 181, Elsevier, 2007, pp. 49–61.
[20] R. Bird, O. Moor, The Algebra of Programming, in: Series in Computer Science, Prentice-Hall International, 1997.
[21] R. S. Bird, L. Meertens, Two exercises found in a book on algorithmics, in: L. Meertens (Ed.), Program Specification and Transformation, North-Holland, 1987, pp. 451–458.
[22] A. Brogi, C. Canal, E. Pimentel, A. Vallecillo, Formalizing web service choreographies, Electronic Notes in Theoretical Computer Science 105 (2004) 73–94.
[23] N. Busi, R. Gorrieri, C. Guidi, R. Luchi, G. Zavattaro, Choreography and orchestration: A synergic approach for systems design, in: B. Benatallah, F. Casati, P. Traverso (Eds.), Proc. ICSOC 2005 Thrid Inter. Conf. on Service-Oriented Computing, 2005, pp. 228–240.
[24] D. Clarke, D. Costa, F. Arbab, Connector colouring I: Synchronisation and context dependency, Science of Computer Programming 66 (3) (2007) 205–225.
[25] J. Fiadeiro, A. Lopes, Semantics of architectural connectors, in: Proc. of TAPSOFT'97, in: Lect. Notes Comp. Sci., vol. 1214, Springer, 1997, pp. 505–519.
[26] J.L. Fiadeiro, Software services: Scientific challenge or industrial hype?, in: K. Araki, Z. Liu (Eds.), Proc. First International Colloquim on Theoretical Aspects of Computing, ICTAC'04, Guiyang, China, in: Lect. Notes Comp. Sci., vol. 3407, Springer, 2004, pp. 1–13.
[27] T. Hagino, A typed lambda calculus with categorical type constructors, in: D.H. Pitt, A. Poigné, D.E. Rydeheard (Eds.), Category Theory and Computer Science, in: Lect. Notes Comp. Sci., vol. 283, Springer, 1987, pp. 140–157.
[28] T. Henziger, Hybrid automata with finite bisimulations, in: Proc. the 22th Inter. Colloquium on Automata, Languages, and Programming, ICALP, in: Lect. Notes Comp. Sci., vol. 944, Springer, 1995, pp. 324–335.
[29] C.A.R. Hoare, Communicating Sequential Processes, in: Series in Computer Science, Prentice-Hall International, 1985.
[30] Y. Hongli, Z. Xiangpeng, C. Chao, Q. Zongyan, Exploring the connection of choreography and orchestration with exception handling and finalization/compensation, in: B. Magnusson (Ed.), Proc. of FORTE'07, in: Lect. Notes Comp. Sci., vol. 4574, Springer, 2007, pp. 81–96.
[31] IBM, Web services architecture overview: The next stage of evolution for e-business. http://www.ibm.com/developerworks/web/library/w-ovr/, 2003.
[32] K. Larsen, A. Skou, Bisimulation through probabilistic testing, Information and Computation (94) (1991) 1–28.
[33] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, in: Proceedings of the 5th European Software Engineering Conference, Springer-Verlag, London, UK, 1995, pp. 137–153.
[34] J. Magee, J. Kramer, D. Giannakopoulou, Behaviour analysis of software architectures, in: WICSA1: Proc. of the TC2 First Working IFIP Conf. on Software Architecture, WICSA1, Kluwer, B.V., 1999, pp. 35–50.
[35] M. Mazzara, S. Govoni, A case study of web services orchestration, in: Proc. of Coordination'05, in: Lect. Notes Comp. Sci., vol. 3454, Springer, 2005.
[36] R. Milner, Communication and Concurrency, in: Series in Computer Science, Prentice-Hall International, 1989.
[37] R. Milner, Elements of interaction (Turing Award Lecture), Communications of the ACM 36 (1) (1993) 78–89.
[38] R. Milner, Communicating and Mobile Processes: The π-Calculus, Cambridge University Press, 1999.
[39] O. Nierstrasz, F. Achermann, A calculus for modeling software components, in: F.S. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Proc. First International Symposium on Formal Methods for Components and Objects, FMCO'02, in: Lect. Notes Comp. Sci., vol. 2852, Springer, 2003, pp. 339–360.
[40] S. Oaks, H. Wong, Jini in a Nutshell, O'Reilly and Associates, 2000.
[41] G. Papadopoulos, F. Arbab, Coordination models and languages, in: Advances in Computers — The Engineering of Large Systems, vol. 46, 1998, pp. 329–400.
[42] M.P. Papazoglou, D. Georgakopoulos, Service-oriented computing, Communications of the ACM 46 (10) (2003).
[43] F. Plasil, D. Mikusik, Inheriting synchronization protocols via sound enrichment rules, in: JMLC '97: Proc. of the Joint Modular Languages Conference on Modular Programming Languages, Springer-Verlag, London, UK, 1997, pp. 267–281.
[44] F. Plasil, S. Visnovsky, Behavior protocols for software components, IEEE Transactions on Software Engineering 28 (11) (2002) 1056–1076.
[45] P. Ribeiro, M.A. Barbosa, L.S. Barbosa, Generic process algebra: A programming challenge, Journal of Universal Computer Science 12 (7) (2006) 922–937.
[46] P. Ribeiro, P. Barbosa, S. Wang, An exercise on transition systems, Electronic Notes in Theoretical Computer Science 207 (2007) 89–106.
[47] J. Rutten, Universal coalgebra: A theory of systems, Theoretical Computer Science 249 (1) (2000) 3–80 (Revised version of CWI Techn. Rep. CS-R9652, 1996).
[48] G. Salaün, L. Bordeaux, M. Schaerf, Describing and reasoning on web services using process algebra, in: ICWS '04: Proc. of IEEE Inter. Conf. on Web Services, ICWS'04, IEEE Computer Society, Washington, DC, USA, 2004, p. 43.
[49] D. Sangiorgi, D. Walker, The π-Calculus: A Theory of Mobile Processes, Cambridge University Press, 2001.
[50] J.-G. Schneider, O. Nierstrasz, Components, scripts, glue, in: L. Barroca, J. Hall, P. Hall (Eds.), Software Architectures — Advances and Applications, Springer-Verlag, 1999, pp. 13–25.
[51] P. Selinger, Categorical structure of asynchrony, in: MFPS'98 (invited talk), New Orleans, in: ENTCS, vol. 20, Elsevier, March, 1999.

[52] S. Stephen Kell, Rethinking software connectors, in: SYANCO'07: Inter. on Synthesis and Analysis of Component Connectors, ACM, New York, NY, USA, 2007, pp. 1–12.
[53] S. Visnovsky, Modeling software components using behavior protocols, Doctoral Thesis, Charles University, Czech Replubic, 2003.
[54] W2C, Web services choreography description language (version 1.0). www.w3.org/TR/ws-cdl-10/, 2005.
[55] W2C, Web services description language (version 2.0). www.w3.org/TR/wsdl20/, 2007.
[56] Q. Zongyan, Z. Xiangpeng, C. Chao, Y. Hongli, Towards the theoretical foundation of choreography, in: P. Patel-Schneider, P. Shenoy (Eds.), Proceedings of the 16th Int Conf. on World Wide Web, ACM, 2007, pp. 973–982.