Electronic Notes in Theoretical Computer Science 175 (2007) 39–57 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Configurations of Web Services

# Marco Antonio Barbosa [1](#_bookmark0)

*DI-CCTC – Universidade do Minho Braga, Portugal*

# Lu´ıs Soares Barbosa[2](#_bookmark0)

*DI-CCTC – Universidade do Minho Braga, Portugal*

Abstract

The quest for sound foundations for the orchestration of web services is still open. To a great extent its relevance comes from the possibility of defining formal semantics for new language standards (like BPEL4WS or WS-CDL) in this emerging and challenging technology. As a step in that direction, this paper resorts to a notion of *configuration*, developed by the authors in the context of a Reo-like exogenous coordination model for software components, to formally express service orchestration. The latter is regarded as involving both the architectural assembly of independent services and the description of their interactions.

*Keywords:* Web services, configuration, coordination.

# Introduction

As the most popular technology in the emerging paradigm of service-oriented com- putation, web services are re-shaping the Web from a document-centered to a service-centered environment. The impact of such a move, both in the world’s economy and in our everyday life, is just beginning to loom.

Technically the definition of what a web service is offers no special difficulty. According to the *World Wide Web Consortium*, it is just 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 application using XML based messages via Internet- based protocols. A sightly more refined specification would abstract from concrete representations of data and messages, defining the underlying notion of a *service*

1 Email: [marco.antonio@di.uminho.pt](mailto:marco.antonio@di.uminho.pt)

2 Email: [lsb@di.uminho.pt](mailto:lsb@di.uminho.pt)

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2007.03.004

as a state-distributed and platform-independent computational entity which can be defined, published, classified, discovered and dynamically assembled for developing massively distributed, interoperable, evolvable systems and applications. Services, typically running in different platforms and owned by different organizations, inter- act and cooperate to achieve some complex goals. Therefore, suitable formal models for interaction and cooperation become essential to represent and reason about web service composition. Such is the theme of this paper.

In practice, web service composition is described in terms of either *choreography* or *orchestration* languages. The former specifies the conversation rules which govern interactions between all the services involved in a particular application, whereas the latter provides means to program a specific service, called the orchestrator, responsible for some form of external coordination of the services. These two ap- proaches have been separately developed by industrial consortia and international organizations such as W3C and OASIS. In particular, WS-CDL and BPEL4WS spec- ifications represent the most credited languages for the Web Services technology which deal with choreography and orchestration respectively.

In such a context, the starting point of this paper is the striking similarity between the orchestration service, mentioned above, and what is called *the glue code* in classical coordination approaches [[24](#_bookmark45),[23](#_bookmark44)]. The corresponding research question is:

* Taking, as an underlying assumption, that web services do not interact directly, can a ’general-purpose’ coordination model be used to specify their orchestration in typical applications?

In particular, the paper tackles this question in the context of a variant to Reo exogenous coordination model [[5](#_bookmark24)], developed by the authors and documented in [[10](#_bookmark31),[9](#_bookmark30)].

This model is based on a notion of *software connector* which regulates the flow of data by relating data items present to its input and output ports. 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.

This is however clearly insufficient to count as an *interface* for a web service. The latter should also include a description of what is commonly called the service *workflow patterns*. I.e., a specification of which, when and under what conditions ports become activated (*i.e.*, ready to deliver or consume a datum). This raises a second question to be addressed in this paper:

* How can the model be extended with service interfaces exhibiting some form of behavioural specification to model the intended workflow, or *use*, pattern?

Clearly, to be useful such descriptions have to be *compositional*, in the sense that the overall behaviour of a web service application should be computed from the behaviour of individual services and that of the *connectors* forming the orchestration layer.

The paper resorts to process algebra to build such specifications. Therefore, both service interfaces and software connectors become equipped with a *use* pattern given as an expression in a process algebra. The idea is, in itself, not new. For example, reference [[28](#_bookmark46)] uses a process language to describe the message exchange between web services, and to *reason* about them. A similar work, but now in the choreography side, is reported in [[13](#_bookmark34)].

The challenging issue is composition. Actually it comes with no surprise that the interaction discipline which governs web services integration is distinct from the one underlying the global composition of web services and the *glue code*. 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 [[20](#_bookmark41)]). This leads us to another piece of previous work (documented in [[7](#_bookmark25),[8](#_bookmark29)]) on the development of *generic* process algebra. I.e., process algebras in which parallel composition is parametric on an interaction discipline suitably encoded. One may then have *different* interaction models governing different aspects of a specification. Such is the genesis of the orchestration model proposed in this paper. Expressive power and the possibility of computing the overall behaviour of a particular web

service based application are, in our opinion, its merits.

The paper results are presented in two main sections: section [2](#_bookmark1) which introduces *behavioural interfaces* for web services and section [3](#_bookmark9) which discusses their exoge- nous orchestration through external *glueing code*. Finally, section [4](#_bookmark19) presents some conclusions and directions for future work.

# Behavioural Interfaces

* 1. *Deﬁning Interfaces*

As mentioned before, in exogenous coordination models, like [[5](#_bookmark24)] or [[9](#_bookmark30)], components are black box entities accessed by purely syntactic interfaces. The role of an interface is restricted to keeping track of port names and, possibly, of admissible types for data items flowing through them [3](#_bookmark2) . For a web-service, however, the specification of the corresponding workflow pattern is as important as the description of the available actions or of the orchestration structure. This leads to the following definition:

Definition 2.1 A web-service *S* interface is specified by a *port signature*, *sig*(*S*) over D, given by a port name and a polarity annotation (either in(put) or out(put)), and a *use pattern*, *use*(*S*), given by a process term, as detailed below, over port names.

3 In the sequel, however, we assume a unique, general data domain, denoted by D, as the type of all data values flowing in a web service based application.

The relevant question concerns what sort of formalism should be used for the specification of *use patterns*? Transition systems [[19](#_bookmark40),[29](#_bookmark47)], regular-expressions [[25](#_bookmark48),[26](#_bookmark49),[31](#_bookmark51)] or process algebras [[18](#_bookmark39),[3](#_bookmark26)] are part of the huge diversity of formal struc- tures 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 such disciplines have to be used, at the same time, to capture different aspects of web services orchestration. For example the discipline governing to composition of *software connectors* between them (to build the overall *glue code*) differs from the one used to capture the in- teraction between the latter and the relevant web 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. Our previous work on a coalgebraic reconstruction of classical process calculi, documented in [[7](#_bookmark25),[8](#_bookmark29)], provides the necessary ingredients. This work is briefly reviewed in the following sub-section, which paves the way to the discussion of its application in the context of web service orchestration.

* 1. *Generic Process Algebra*

References [[7](#_bookmark25),[8](#_bookmark29)] introduced a denotational approach to the *design* of process algebras in which processes are identified with inhabitants of a final coalgebra [[17](#_bookmark38)] 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 whole calculus is based [4](#_bookmark3) .

coinductive extension. I.e., the existence, for each arbitrary coalgebra ⟨*U, p* : Technically, this amounts to the systematic use of the universal property of *U* −→ P(*Act* × *U* )⟩, of a unique morphism [(*p*)] to the final coalgebra *ω* : *ν* −→ P(*Act* × *ν* ) satisfying

*k* = [(*p*)] ⇔ *ω* · *k* = P(id × *k*) · *p* (1) where P is the finite powerset functor [5](#_bookmark4) . Therefore, processes being the inhabi- tants of the final coalgebra, expression P(*Act* × *ν*) stands for a set of pairs each

one representing a transition and a corresponding continuation process. Such [(*p*)]

represents the behaviour generated by *p* and comes equipped with a bunch of laws useful in calculation.

4 Combined with the *pointfree* ‘calculational’ style entailed by category theory, this leads to a generic way of reasoning about processes in which, in particular, proofs by bisimulation, which classically involve the explicit construction of such a relation [[20](#_bookmark41)], are replaced by *equational reasoning*. 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* [[12](#_bookmark33)] and the foundational work of T. Hagino [[15](#_bookmark36)].

5 The definition generalizes, of course, to an arbitrary coalgebra.

Process combinators are defined either in a direct way (if they are consumed by transitions) or by coinductive extension (if permanent). Examples in the first group are the *inactive* process 0, whose set of observations is empty, and non deterministic choice +, whose observations are the union of the possible observation upon its arguments [6](#_bookmark5) Clearly, *preﬁx* (*α.p*) is another example. 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 web 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* : *ν* × *ν* −→ *ν* 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, = [(*α* )],

where [7](#_bookmark6)

*α* = *ν* × *ν*  Δ (*ν* × *ν*) × (*ν* × *ν*) (*ω*×id)×(id×*ω*) (P (*Act* × *ν*) × *ν*) × (*ν* × P(*Act* × *ν*))

*τr* ×*τl* P( *Act* × (*ν* × *ν*)) × P(*Act* × (*ν* × *ν*)) ∪ P( *Act* × (*ν* × *ν*))

*Synchronous product* models the simultaneous execution of its two arguments.

moment, represent such interaction by a function *θ* : *Act* −→ *Act* × *Act*. Formally, In each step, processes interact through the actions they realize. Let us, for the

⊗ = [(*α*⊗)] where

*α* = *ν* × *ν* (*ω*×*ω*) P (*Act* × *ν*) × P(*Act* × *ν*) sel·*δr*  P (*Act* × (*ν* × *ν*))

⊗

where sel filters out all synchronisation failures (*i.e.*, cases in which *aθb* = 0, see

below) and *δr* is given by

*δr* ⟨*c*1*, c*2⟩ = {⟨*a*' *θ a,* ⟨*p, p*'⟩⟩| ⟨*a, p*⟩∈ *c*1 ∧ ⟨*a*'*, p*'⟩∈ *c*2}

The fundamental point to note is that the definition is parametric on *θ*, which

as an Abelian positive monoid ⟨*Act*; *θ,* 1⟩ with a zero element 0. The intuition is encodes an *interaction discipline*. Technically, an *interaction discipline* is modeled interaction: for all *a* ∈ *Act*, *aθ*0 = 0. On the other hand, being a positive monoid that *θ* determines the interaction discipline whereas 0 represents the absence of captures action co-occurrence, in which case *θ* is defined as *aθb* = ⟨*a, b*⟩, for all entails *aθa*' = 1 iff *a* = *a*' = 1. A typical example of an interaction structure

*a, b* ∈ *Act*. Another example is provided by the action complement match used in

CCS [[21](#_bookmark42)]. In the sequel we shall introduce a number of specifications for *θ* suitable

to express web service orchestration.

6 Formally, recalling that final coalgebra *ω* gives, for each process denotation, the set of its observations, one would write *ω* · 0 = ∅ and *ω* · += ∪ · (*ω* × *ω*), respectively. In a pointwise notation the latter equation becomes *ω*(*p* + *q*)= *ω*(*p*) ∪ *ω*(*q*).

7 Morphisms *τr* : P(*Act* × *X*) × *C* −→ P(*Act* × (*X* × *C*)) and *τl* : *C* ×P(*Act* × *X*) −→ P(*Act* × (*C* × *X*)) stand for, respectively, the right and left *strength* associated to functor P(*Act* × Id).

*chronous product* ⊗. Such a combination is performed at the *genes* level: | = [(*α*|)], *Parallel composition* combines the effects of both *interleaving* and *syn-* where

*α* = *ν* × *ν* Δ (*ν* × *ν*) × (*ν* × *ν*) (*α* ×*α*⊗)

|

P(*Act* × (*ν* × *ν*)) × P(*Act* × (*ν* × *ν*)) ∪ P (*Act* × (*ν* × *ν*))

* 1. *Use Patterns and Interaction*

Once defined a *parametric* semantics for parallel composition, we may return to the definition of *use patterns* for web services.

Definition 2.2 Let P be the set of port identifiers and *S* stand for (the specification over *Act* PP, given by the following grammar: of) a web service. Its use pattern, denoted by *use*(*S*) is given by a process expression

*P* ::= 0 | *α.P* | *P* + *P* | *P* ⊗ *P* | *P* *P* | *P* ; *P* | *P* | *P* |

*σP* | fix (*x* = *P* )

where *α* is an element of *Act* (*i.e.*, a set of port identifiers) and *σ* is a substitution.

Notice that choosing *Act* as a *set* of port identifiers allows for the synchronous activation of several ports in a single computational step. The semantics of such expressions is fairly standard, but for the parametrization of all forms of parallel

composition (*i.e.*, ⊗ and |) by an interaction discipline as discussed above. The reader is referred to [[27](#_bookmark50)] for the full details. Combinators 0, *.*, +, |, ⊗ and , were already introduced in the previous sub-section. Renaming is given by term

substitution. The fix (*X* = *P* ) is a fixed point construction, which, as usual, can abbreviated in an explicit recursive definition.

ment to be a terminating process. Symbol § represents a successfully terminating Sequential composition, as in Csp [[16](#_bookmark37)], is given by ‘;’ and requires its first argu- mally, § a=bv C*.*0. process, *i.e.*, a process that engages in the success event, C, and then stops. For-

The approach proposed in this paper precludes direct interaction between web

services — all interaction being mediated by a specific connector. Therefore, if two web 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 2.3 The joint behaviour of a collection {*Si*| *i* ∈ *n*} of web services is given by

*use*(*S*1) | *...* | *use*(*Sn*)

where the interaction discipline is fixed by *θ* = ∪ , *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 application of Milner’s *expansion law* [8](#_bookmark10) , while obeying to the interaction discipline given by *θ*. The following example illus- trates this construction.

Example 2.4 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) = 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) = fix (*x*' = *cd.x*')*,* where, *cd* a=bv

{*c, d*}

According to definition [2.3](#_bookmark8), the joint behaviour of *S*1 and *S*2 is

*use*(*S*1) | *use*(*S*2) = 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.*,

Clearly, expansion leads to

*use*(*S*3) = fix (*y* = *e.f.y*)

*use*(*S*2) | *use*(*S*3) = fix (*x* = *cd.x* + *e.f.x* + *cde.f.x* + *cde.cdf.x*

+ *e.cdf.x* + *cd.e.f.x* + *e.cd.f.x* + *e.f.cd.x*)

# Configurations

The fundamental notion proposed in this paper as a basis for the orchestration of web services is that of a *conﬁguration*. As explained in the Introduction, this captures the intuition that web services cooperate through specific *connectors* which abstracts the idea of an intermediate *glue code* to handle interaction. Having already defined a notion of web service *interface*, which records all what may be assumed to be known by the web service user, we shall now complete the picture by defining

* what *connectors* are and how they compose;
* the way web services’ *interfaces* and *connectors* interact in a configuration.

These points are tackled in the following sub-sections. As one would expect, the two forms of composition (of connectors with themselves and with web services’ interfaces) follow *different* interaction disciplines, captured by specific definitions of *θ*.

8 This law, which states that a process is always equivalent to the non deterministic choice of its derivatives, is a fundamental result in interleaving models for concurrency.

* 1. *Connectors*

Connectors are *glueing devices* between services which ensure the flow of data and the meet of synchronization constraints. Their specification builds on top of our previous work on component interconnection [[9](#_bookmark30)], which is extended here with an explicit annotation of activation, or *use*, patterns for their *ports*.

Ports are *interface points* through which messages flow. Each port has an *inter- action polarity* (either *input* or *output* ), but, in general, connectors are blind with respect to the data values flowing through them. Another particular characteristic is the ability to construct complex connectors out of simpler ones using a set of *combinators*.

Let C be a connector with *m* input and *n* output ports. Assume, again, D as a generic type of data values and P as a set of (unique) *port identiﬁers* Formally, the behaviour of a connector may be given by

D*n* ←− D*m* which records the flow of data, and a process expression port*.*[[C]] which Definition 3.1 The specification of a connector C is given by a relation data*.*[[C]] : gives the pattern of port activation.

Let us illustrate this definition with a number of examples.

* + 1. *Synchronous channel.*

The *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.

data*.*[[ • • ] ] = IdD *and* port*.*[[ • • ] ] = fix (*x* = *ab.x*)

Its semantics is simply the identity relation on data domain D and its behaviour is

captured by the simultaneous activation of its two ports.

* + 1. *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

data*.*[[ • • ] ] ⊆ IdD *and* port*.*[[ • • ] ] = 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.

* + 1. *Filter channel.*

to a given predicate *φ* : 2 ←− D. Noting that any predicate *φ* can be seen as a This is a channel in which some messages are discarded in a controlled way, according

relation R*φ* : D ←− D such that *d*R*φd*' iff *d* = *d*' ∧ (*φ d*), define

data*.*[[ • *φ*  • ] ] = R *and* port*.*[[ • *φ*  • ] ] = fix (*x* = *ab.x*)

*φ*

* + 1. *Drain.*

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

data*.*[[ • V • ]] = D × D *and* port*.*[[ • V • ]] = fix (*x* = *ab.x*)

and,

data*.*[[ • o • ]] = D × D *and* port*.*[[ • o • ]] = fix (*x* = *a.x* + *b.x*)

* + 1. *Fifo*1*.*

This is a channel with a buffer of a single position.

data*.*[[ • • ] ] = IdD *and* port*.*[[ • • ] ] = fix (*x* = *a.b.x*)

* 1. *Combining Software Connectors*

let *t*#*a*, for *t* ∈ D*n* and *a* ∈ P, denote the component of data tuple *t* correspond- Connectors can be combined to build more complex *glueing code*. In the sequel, been deleted, *i.e.*, *t*|*a* = (*tn,* ··· *, t*#*a*+1*, t*#*a*−1*,* ··· *, t*0). Consider, now, the following ing to port *a*. Define *t*|*a* as a tuple identical to *t* from which component *t*#*a* has combinators:

* + 1. *Aggregation.*

This combinator places its arguments side-by-side, with no direct interaction be- tween them. Then,

data*.*[[C1 ☒ C2]] = data*.*[[C1]] × data*.*[[C2]]

and

port*.*[[C1 ☒ C2]] = port*.*[[C1]] ⊗ port*.*[[C2]]

with *θ* = ∪.

* + 1. *Hook.*

This combinator encodes a *feedback* mechanism, drawing a direct connection be- tween an output and an input port. Formally, port*.*[[C t*j* **]** is obtained from port*.*[[C]] by deleting references to ports *i* and *j*. To be well-formed it is required that *i* and

*i*

*j* appear in different factors of some form of parallel composition ( , ⊗, or |).

If data*.*[[C]] : D*n* ←− D*m*, the combinators’ effect on data is modelled by relation

data*.*[[C t*j*]] : D*n*−1 ←− D*m*−1

*i*

specified by

*t*' (data*.*[[C t*j*]]) *t*|*i* iff *t*' (data*.*[[C]]) *t* ∧ *t*'

= *t*#*i*

|*j i* #*j*

* + 1. *Join.*

Its effect is to plug ports with identical polarity. The aggregation of *output* ports is done by a *right join* (C *i > z*), where 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 in a nondeterministic way.

*j*

On the other hand, aggregation of *input* ports resorts to a *left join* (*z <i* C).

*j*

This behaves like a *broadcaster* sending synchronously messages from *z* to both *i*

and *j*. Formally, at a behavioural level, both operators effect is that of a renaming operation

port*.*[[(C *i > z*)]] = port*.*[[(*z <i* C)]] = {*z* ← *i, z* ← *j*}port*.*[[C]]

*j j*

Their difference is specified at the data level, as follows. Again, let

data*.*[[C]] : D*n* ←− D*m*. Then,

*j*

*Right join:*

given by

data*.*[[C *i > z*]] : D*n*−1 ←− D*m*

*r* (data*.*[[C *i > z*]]) *t* iff *t*' (data*.*[[C]]) *t* ∧ *r*|*z* = *t*' ∧ (*r*#*z* = *t*' ∨ *r*#*z* = *t*' )

*j* |*i,j* #*i* #*j*

*Left join:*

given by

data*.*[[*z <i* C]] : D*n* ←− D*m*−1

*t*' (data*.*[[*z <i* C]]) *r* iff *t*' (data*.*[[C]]) *t* ∧ *r*|*z* = *t*|*i,j* ∧ *r*#*z* = *t*#*i* = *t*#*j*

*j*

*j*

* 1. *Conﬁgurations*

A *conﬁguration* is simply a collection of web services, characterized by their inter- faces, interconnected through an *orchestrator*, *i.e.*, a connector network built from elementary connectors using the combinators mentioned above. Formally,

Definition 3.2 A configuration involving a collection *S* = {*Si*| *i* ∈ *n*} of web services is a tuple

⟨*U,* C*, σ*⟩ (2)

where *U* = *use*(*S*1) | *use*(*S*2) | ··· | *use*(*Sn*) is the (joint) use pattern for *S*, C is a connector and *σ* a mapping of ports in *S* to ports in C.

The relevant point concerning configurations is the semantics of the interaction between the *connector’s port behaviour* and the *joint use patterns* of the involved

web services. This is captured by a synchronous product ⊗ for a quite peculiar *θ*, which is expected to capture the following requirements:

* Interaction is achieved by the simultaneous activation of identically named ports [9](#_bookmark13) .
* There is no interaction if the connector intends to activate ports which are not linked to the ones offered by the web services’ side. For example if a port *a* of a service *S* is connected to the input end of a synchronous channel whose output end is disconnected, no information can flow and port *a* will never be activated.
* The dual situation is allowed, *i.e.*, if the web services’ side offer activation of all ports plugged to the ones offered by the connectors’ side, their intersection is the resulting interaction.
* Moreover, and finally, activation of unplugged web services’ ports is always pos- sible.

Formally, this is captured in the following definition.

Definition 3.3 The behaviour *bh*(Γ) of a configuration Γ = ⟨*U,* C*, σ*⟩ is given by

*bh*(Γ) = *σU* ⊗ port*.*[[C]] (3)

where *θ* underlying the ⊗ connective is given by

⎧⎨

*c* ∩ (*c*' ∪ free) ⇐ *c*' ⊆ *c*

*c θ c*' =

⎩ 0 ⇐ *otherwise*

(4)

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

*σ*.

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

* 1. *Examples*

Example 3.4 Our first example is taken from [[14](#_bookmark35)]. Suppose an organization offers a “Holiday Reservation Service” (HRS) that allows customers to organize holiday

9 Often this will force the introduction of suitable port renamings.

travels. A possible configuration *HR* is given by

*HR* = ⟨*WHR, SB, σHS*⟩

where

*WHR* = *use*(*HRS*) | *use*(*HORS*) | *use*(*FRS*) | *use*(*CRS*)

*σHS* = {*a* ← *A, b* ← *B, c* ← *C, d* ← *D, e* ← *E, f* ← *F, g* ← *G*}

It is depicted in Fig [1](#_bookmark14).

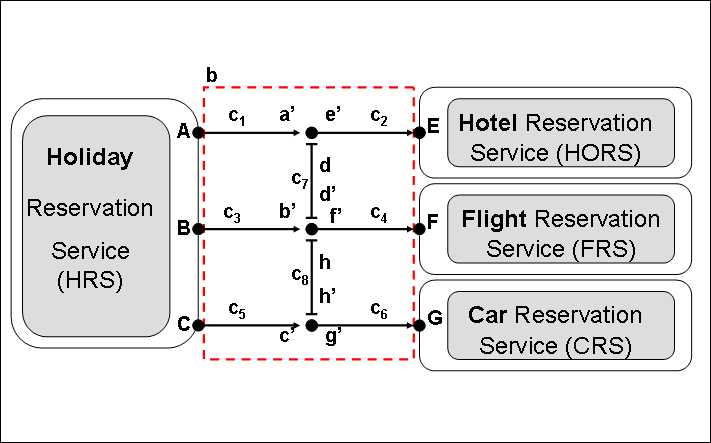


Fig. 1. Holiday Reservation

It is assumed that organizing a holiday requires making reservations for a hotel, for a flight and for a car. Different organizations offer services to deal with each part of the job: hence the HORS, FRS and CRS services. Before asking the cus- tomer to pay, the HRS services needs to *commit* a transaction containing each of the reservations. A holiday reservation should only succeed when all other three reservations succeed.

The *commit* requirement is modeled by a particular external *glue code*: a *bar- rier synchronization* connector consisting of six synchronous channels and two syn- chronous drain channels, organized together as in Fig. [1](#_bookmark14).

Let us now compute the overall behaviour of configuration *HR*. Suppose that, after port renaming by *σHS* , the usage pattern of each web service is as follows:

*use*(*HRS*) = fix (*x* = *a.x* + *b.x* + *c.x* + *abc.x*) *use*(*HORS*) = fix (*x* = *e.x*)

*use*(*FRS*) = fix (*x* = *f.x*) *use*(*CRS*) = fix (*x* = *g.x*)

Its joint behaviour is given by their |-composition, with *θ* = ∪.

On the other hand, the behaviour of connector *SB* is obtained by composition of six elementary connectors as depicted in Fig. [1](#_bookmark14).

port*.*[[*SB*]] = fix *x* = *abcefg.x* (5)

This is computed starting from the behaviours of the elementary connectors

port*.*[[*c*1]] = fix (*x* = *aa*'*.x*)*,* port*.*[[*c*2]] = fix (*x* = *e*'*e.x*)*,*

port*.*[[*c*3]] = fix (*x* = *bb*'*.x*)*,* port*.*[[*c*4]] = fix (*x* = *f* '*f.x*)*,*

port*.*[[*c*5]] = fix (*x* = *cc*'*.x*)*,* port*.*[[*c*6]] = fix (*x* = *g*'*g.x*)*,*

port*.*[[*c*7]] = fix (*x* = *dd*'*.x*)*,* port*.*[[*c*8]] = fix (*x* = *hh*'*.x*)

as follows

*Cn*

'

= port*.*[[(*n <* (*c*

*e*

☒ *c* ))]] = fix (*x* = {*n* ← *e*'*,n* ← *d*} *e*'*edd*'*.x*)

1 *d* 2 7

= fix (*x* = *ed*'*.x*)

*Cn*2 = port*.*[[((*c*1 ☒ *Cn*1) t*n* )]] = fix (*x* = *aed*'*.x*)

*a*'

*Cn* = port*.*[[(*m <g*' (*c* ☒ *c* ))]] = fix (*x* = *mgh.x*)

3 *h*' 6 8

*Cn*4 = port*.*[[((*Cn*3 ☒ *c*5) t*m*)]] = fix (*x* = *cgh.x*)

*c*'

*Cn* = port*.*[[(*z <d*' (*Cn* ☒ *c* ))]] = fix (*x* = *aef z.x*)

5 *f* ' 2 4

*Cn*6 = port*.*[[((*Cn*5 ☒ *c*3) t*z* )]] = fix (*x* = *abef.x*)

*b*'

and

port*.*[[*SB*]] = port*.*[[((*Cn*6 ☒ *Cn*4) t*z* )]] = fix (*x* = *abcef g.x*)

The result of the ⊗ composition of *WHR* and port*.*[[*SB* **]** is the behaviour of configu- ration *HR*. There is no need, however, to compute the complete expansion of the

*h*

parallel composition in *WHR* expression, which is

fix (*x* = *a.x* + ··· + *e.x* + *f.x* + *g.x*+

*ae.x* + ··· + *be.x* + ··· + *ce.x* + ··· + *abce.x* + ··· +

*aef.x* + ··· + *bef.x* + ··· + *cef.x* + ··· + *abcef.x* + ··· +

*aef g.x* + ··· + *bef g.x* + ··· + *cef g.x* + ··· + *abcef g.x* + ··· +

*ef.x* + *eg.x* + *fg.x* + *ef g.x*)

because, according to interaction discipline ([4](#_bookmark12)), the only successful case of compo- sition with port*.*[[*SB* **]** corresponds to the underlined alternative in the expression above. Clearly, the *θ*-composition of *abcef g* with *abcef g* (from the connector side)

is *abcef g*, while for all other cases it results in the empty set ∅. Therefore, and finally,

*bh*(*HR*) = fix (*x* = *abcef g.x*) (6)

Example 3.5 As a second example consider an elementary *banking system* com- posed by an *AT M* 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. [2](#_bookmark15), is specified as

*BS* = ⟨*WBS, DBC, σBS*⟩

where

*WBS* = *use*(*AT M* ) | *use*(*Bank*) | *use*(*DBRep*)

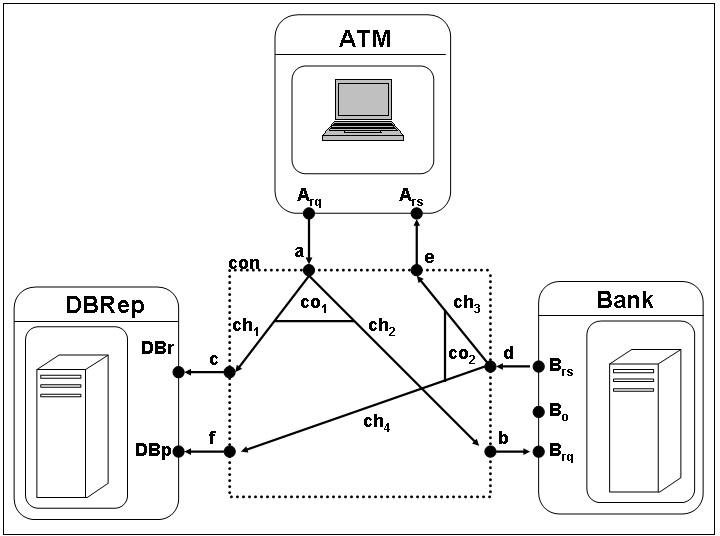
*σHS* = {*a* ← *Arq,e* ← *Ars,c* ← *DBr,f* ← *DBp,d* ← *Brs,b* ← *Brq*}

Fig. 2. Bank System

Consider the following use patterns of each web service after port renaming by

*σBS*:

*use*(*AT M* ) = fix (*x* = *a.e.x*) *use*(*Bank*) = fix (*y* = *b.d.y*)

*use*(*DBRep*) = fix (*z* = *c.z* + *f.z*)

Connector *DBC* behaves like a double broadcaster (hence its name). Its be- haviour allows for both the simultaneous or independent activation of each broad- cast (*co*1 or *co*2) as shown by the following computation:

port*.*[[*ch*1]] = fix (*x* = *b*'*b.x*)*,* port*.*[[*ch*2]] = fix (*x* = *c*'*c.x*)

'

*b*

port*.*[[*co* ]] = port*.*[[(*a <* (*ch*

☒ *ch* ))]] = fix (*x* = *abc.x*)

1 *c*' 1 2

port*.*[[*ch*3]] = fix (*x* = *e*'*e.x*)*,* port*.*[[*ch*4]] = fix (*x* = *f* '*f.x*)

port*.*[[*co* ]] = port*.*[[(*d <e*' (*ch* ☒ *ch* ))]] = fix (*x* = *def.x*)

2 *f* ' 3 4

port*.*[[*DBC*]] = port*.*[[(*co*1 ☒ *co*2))]] = fix (*x* = *abc.x* + *def.x* + *abcdef.x*)

Again, to determine *bh*(*BS*) one needs to expand *WBS* [10](#_bookmark20) . According to ([4](#_bookmark12)), we need only to look for summands prefixed by sets of ports which are super-sets of

prefix sets in port*.*[[*DBC* **]**. For the first level of expansion alternative *abc.*(*e.x*| *d.y* |*z*)

consider the expansion of term (*e.x* | *d.y* | *z*): the only alternative worth to consider is the only one to *θ*-compose with *abc* in port*.*[[*DBC* **]**, resulting in *abc* again. Then, (*i.e.*, which does not lead to 0 on *θ*-composition) is *edf.*(*x* | *y* | *z*), the resulting

interaction being *edf* . From this point on the same expansion pattern repeats. This

means that *bh*(*BS*) becomes:

*bh*(*BS*) = fix (*x* = *abc.edf.x*) (7)

Notice how the particular use patterns in the web services act as a *constraint* over the admissible behaviour of connector *DBC*.

This example may be also used to check how definition ([4](#_bookmark12)) deals with the pres- ence of unplugged ports, such us port *Bo* in service *Bank*. Consider, then, the following two alternatives for the use pattern of service *Bank*:

*use*(*Bank*) = fix (*y* = *bBo.d.y*) (8)

*use*(*Bank*) = fix (*y* = *b.d.y* + *Bo.y*) (9)

tion of ports *b* and *Bo*, leads to term *abBoc.*(*e.x* | *d.y* | *z*) which, as free = {*Bo*}, In the expansion of *WBS*, expression ([8](#_bookmark17)), which captures the simultaneous activa- entails

*bh*(*BS*) = fix (*x* = *abcBo.edf.x*) (10)

Alternative ([9](#_bookmark18)) specifies that ports *b* and *Bo* are activated in alternative: no term with both *b* and *Bo* will appear in the expansion and, therefore, *bh*(*BS*) remains as given by equation ([7](#_bookmark16)).

# Conclusions and Future Work

Service-oriented computing is an emerging paradigm for distributed computing 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 some complex 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.

Having proposed formal models for both *behavioural interfaces* and *conﬁgura- tions*, as a base for representing web services’ orchestration, this paper may be a step in that direction. The approach combines two ingredients in which the authors have been working for some time now: *exogenous coordination* and a methodology for the *design of process algebras parametric on the interaction discipline*.

10 A number of tools for process algebras, as. *e.g.*, Cwb [[22](#_bookmark43)], provide support for application of expansion laws.

Lots of questions, however, remain open. Let us enumerate the ones in which we are currently involved.

Negative Information. In a number of practical situations service orchestration also depends on what may be called *negative information*. One of the basic channels considered in Reo [[5](#_bookmark24)] is the *lossy channel* which acts as a synchronous one if both an input and an output request are pending on its two ports, but will loose the data item on the input on the absence of an output request on the other port. Notice this behaviour is distinct from that of the *unreliable* channel discussed above, which looses data non deterministically.

To handle these cases we enrich the specification of *Act* in definition [2.2](#_bookmark7) to include *negative port activations*, or more rigorously stated, *absence of port requests*, denoted, for each port *p*, by *p*˜. Therefore, the specification of Reo’s lossy channel becomes possible as

data*.*[[ • ··· • ] ] ⊆ IdD *and* port*.*[[ • ··· • ] ] = fix (*x* = *ab.x* + *a*˜*b.x*) The following step is to modify *θ* in ([4](#_bookmark12)), definition [3.3](#_bookmark11), so that

*c θ c*' = 0 ⇐ ∃*p*∈P *. p* ∈ *c* ∧ *p*˜ ∈ *c*' (11)

Then the joint behaviour of a configuration orchestrated by a lossy channel and

involving *use* pattern *U* = fix (*x* = *ab.x*) is computed as follows

*U* ⊗ port*.*[[ • ··· • ] ] = fix (*x* = (*abθab*)*.x* + (*abθa*˜*b*)*.x*) = fix (*x* = *ab.x*) whereas

*U* ⊗ port*.*[[ • ··· • ] ] = fix (*x* = (*aθab*)*.x* + (*aθa*˜*b*)*.x*) = fix (*x* = *a.x* + *a.x*)

= fix (*x* = *a.x*)

for *U* = fix (*x* = *a.x*), *i.e.*, in the absence of an output request from one of the services.

Workflow Patterns. The notion of behavioural interface discussed in section 2 is close to that of *workflow patterns* [[1](#_bookmark22)] whose role in the design of service-oriented systems is well recognized. Their formalization is still an ’hot’ research topic (see, *e.g.*, [[2](#_bookmark23),[30](#_bookmark52),[4](#_bookmark27)], among many others). We believe all such patterns can be encoded in a slight extension of the formalism used here to specify behavioural interfaces. Figure [3](#_bookmark21) illustrates the specification of two common patterns, part of a systematic classification effort currently under development.

For the general case, however, this encoding requires both the introduction of two port attributes to keep track of the number of port requests and activations,

and a conditional constructor (*φ* → *P, P* '), where *φ* is a boolean condition on those attributes.

Pattern 2 (Parallel split)

Description. A point in the workflow process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing activities to be executed simultaneously or in any order.

Specification.

*use*(WS2) = *P*1 | *...* | *Pn*

Pattern 3 (Synchronization)

Description. A point in the workflow process where multiple parallel activities converge into one single thread of control, thus synchronizing multiple threads.

Specification.

*use*(WS3) = (*a*1*.a*2*.... .an.*§⊗ *b*1*.b*2*.... .bn.*§); *P*

Fig. 3. Two workflow patterns and their encoding as use patterns

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. Tentative solutions in *e.g.*, Reo [[6](#_bookmark28)] or our own contribution documented in [[11](#_bookmark32)], are still of an operational nature.

Language Semantics. Whether formal models, like the one discussed in this pa- per, can be of use in providing precise semantic foundations of emerging languages for web services composition (*e.g.*, BPEL4WS, XLANG or WS-CDL, among others) re- mains a challenge we intend to face in future work. Its relevance, from the point of view of software engineering, cannot be underestimated.

# Acknowledgement

This research was carried on in the context of the PURe Project supported

by Fct, the Portuguese Foundation for Science and Technology, under contract

POSI/CHS/44304/2002.

# References

1. W. M. P. V. D. Aalst, A. H. M. T. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns.

*Distrib. Parallel Databases*, 14(1):5–51, 2003.

1. N. R. Adam, V. Atluri, and W.-K. Huang. Modeling and analysis of workflows using petri nets. *J. Intell. Inf. Syst.*, 10(2):131–158, 1998.
2. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, 1997.
3. R. Amici, F. Corradini, and E. Merelli. A process algebra view of coordination models with a case study in computational system biology. In L. Bocchi and P. Ciancarini, editors, *Proceedings of the* *First International Workshop on Petri Nets and Coordination (PNC04), Satellite Event of the 25th International Conference on Application and Theory of Petri Nets, Bologna, Italy, June 21, 2004*, pages 33–47, 2004.
4. F. Arbab. Reo: a channel–based coordination model for component composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, 2004.
5. F. Arbab and F. Mavadatt. Coordination through channel composition. In *Proc. Coordination* *Languages and Models*. Springer Lect. Notes Comp. Sci. (2315), 2002.
6. L. S. Barbosa. Process calculi *`a la* Bird-Meertens. In M. L. Andrea Corradini and U. Montanari, editors, *CMCS’01*, volume 44.4, pages 47–66, Genova, April 2001. Elect. Notes in Theor. Comp. Sci., Elsevier.
7. L. S. Barbosa and J. N. Oliveira. Coinductive interpreters for process calculi. In *Proc. of FLOPS’02*, pages 183–197, Aizu, Japan, September 2002. Springer Lect. Notes Comp. Sci. (2441).
8. M. Barbosa and L. Barbosa. Specifying software connectors. In K. Araki and Z. Liu, editors, *Proc.* *First International Colloquim on Theoretical Aspects of Computing (ICTAC’04), Guiyang, China*, pages 53–68. Springer Lect. Notes Comp. Sci. (3407), 2004.
9. M. A. Barbosa and L. S. Barbosa. A relational model for component interconnection. In *Journal of Universal Computer Science*, volume 10, pages 808–823, 2004.
10. M. A. Barbosa and 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, June 2006. Elsevier.
11. R. Bird and O. Moor. *The Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997.
12. A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web service choreographies. *Electr. Notes Theor. Comput. Sci.*, 105:73–94, 2004.
13. N. K. Diakov and F. Arbab. Compositional construction of web services using reo. In *WSMAI*, pages 49–58, 2004.
14. T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poign´e, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, pages 140–157. Springer Lect. Notes Comp. Sci. (283), 1987.
15. C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.
16. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–159, 1997.
17. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.
18. J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *WICSA1: Proc. of the TC2 First Working IFIP Conf. on Software Architecture (WICSA1)*, pages 35–50. Kluwer, B.V., 1999.
19. R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.
20. R. Milner. *Communicating and Mobile Processes: the π-Calculus*. Cambridge University Press, 1999.
21. F. Moller and P. Stevens. The edinburgh concurrency workbench (version 7). User’s manual, LFCS, Edinburgh University, 1996.
22. O. Nierstrasz and F. Achermann. A calculus for modeling software components. In F. S. de Boer,

M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO’02)*, pages 339–360. Springer Lect. Notes Comp. Sci. (2852), 2003.

1. G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers — The Engineering of Large Systems*, volume 46, pages 329–400. 1998.
2. F. Plasil and D. Mikusik. Inheriting synchronization protocols via sound enrichment rules. In *JMLC ’97: Proc. of the Joint Modular Languages Conference on Modular Programming Languages*, pages 267–281, London, UK, 1997. Springer-Verlag.
3. F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.
4. P. R. Ribeiro, M. A. Barbosa, and L. S. Barbosa. Generic process algebra: A programming challenge. In *Proc. 10th Brazilian Symposium on Programming Languages*, Itatiaia, Brasil, 2006.
5. G. Salau¨n, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *ICWS ’04: Proc. of IEEE Inter. Conf. on Web Services (ICWS’04)*, page 43, Washington, DC, USA, 2004. IEEE Computer Society.
6. P. Selinger. Categorical structure of asynchrony. In *MFPS’98 (invited talk), New Orleans*. ENTCS, volume 20, Elsevier, March 1999.
7. H. Smith and P. Fingar. Workflow is just a pi process, 2003.
8. S. Visnovsky. *Modeling Software Components Using Behavior Protocols*. Doctoral thesis, Charles University, Czech Replubic, 2003.