Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 105 (2004) 73–94
www.elsevier.com/locate/entcs

Formalizing Web Service Choreographies [1]

Antonio Brogi[a], Carlos Canal[b], Ernesto Pimentel[b]
Antonio Vallecillo[b]

[a] *Dept. of Computer Science, University of Pisa, Italy.*
Email: brogi@di.unipi.it

[b] *Dept. of Computer Science, University of Málaga, Spain.*
Emails: canal@lcc.uma.es, ernesto@lcc.uma.es, av@lcc.uma.es

Abstract

Current Web service choreography proposals, such as BPEL4WS, BPSS, WSFL, WSCDL or WSCI, provide notations for describing the message flows in Web service collaborations. However, such proposals remain at the descriptive level, without providing any kind of reasoning mechanisms or tool support for checking the compatibility of Web services based on the proposed notations. In this paper we present the formalization of one of these Web service choreography proposals (WSCI), and discuss the benefits that can be obtained by such formalization. In particular, we show how to check whether two or more Web services are *compatible* to interoperate or not, and, if not, whether the specification of *adaptors* that mediate between them can be automatically generated —hence enabling the communication of (a priori) incompatible Web services.

Keywords: Web services, choreography, WSCI, formal methods, process algebra, interoperability, adaptation.

1 Introduction

Current Web service descriptions allow operations to define the contents and direction of messages (incoming or outgoing), but they do not describe the behaviour of services involving multiple individual operations — i.e., they do not provide enough details of what the service is expected to do. In this

[1] This work has been partly supported by the project NAPOLI funded by the Italian Ministry of Instruction, University and Research (MIUR), and the projects TIC2002-4309-C02-02 and TIC2001-2705-C03-02 funded by the Spanish Ministry of Science and Technology (MCYT).

doi:10.1016/j.entcs.2004.05.007

sense, the Web Service Description Language (WSDL [10]) may be adequate for simple information retrieval in a stateless message exchange, such as a stock quote. There is no standard way, though, of correlating the messages that are exchanged by the Web service with the context in which the Web service operates. This is an important function for building useful service collaboration, and this is the area addressed by current Web service choreography proposals, such as BPEL4WS, BPSS, WSFL, WSCDL, WSCI, etc.

One of these proposals, WSCI (Web Service Choreography Interface [11]), builds upon current Web service technologies to enable the users of a service — such as another Web service, or an application — to understand how to interact with it meaningfully. WSCI also enables developers and architects to describe and compose a global view of the dynamics of the message exchange, allowing to create greater collaborations of services.

Using WSCI we can raise the level of expressiveness currently provided by Web service descriptions, capturing not only *static* information about the signature and direction of the operations supported by a given Web service, but also the *dynamic* information (i.e. the behaviour or *protocol* of the service [9]), describing the partial order in which messages are expected to be exchanged during the collaborations in which the Web service may engage in.

Being able to express this kind of information on top of the WDSL descriptions is a big step forward. However, once we count with this information, the obvious question is about the benefits it can bring along. For instance, what kind of properties can be inferred from the WSCI descriptions? How such properties can be proved? In case two Web services are not compatible to interoperate, can we remedy such situation by adapting them somehow?

Our present goal is to make use of the fair amount of work currently available on the behavioural descriptions of objects and components at the protocol level (e.g., [1,2]), in order to apply many of such theoretical results to the practical field of the Web services. In particular, in this paper we show how the WSCI descriptions can be formalized using a process algebra approach (in particular CCS [5]), and then be in a position to check whether two or more Web services are *compatible* to interoperate or not, and if not, whether the specification of *adaptors* that mediate between them can be automatically generated. In this context, compatibility can be described as the ability of two Web services to work properly together, i.e., that all exchanged messages between them are understood by each other, and that their communication is deadlock-free [12].

The structure of this document is as follows. After this introduction, Section 2 provides a brief introduction to Web services and WSCI. Then, Section 3 discusses how to formalize WSCI. Using this formalization, Section 4 describes how to reason about the compatibility and replaceability of Web services, and the sort of properties that can be proved. Next, Section 5 discusses the possible automated generation of the adaptors, which enable, under some circumstances, the communication of a priori incompatible Web services. Finally, Section 6 draws some conclusions and outlines some further research activities.

2 Web Services and WSCI

A Web service, as defined by the *World Wide Web Consortium* (W3C), is a software application identified by a uniform resource identifier (URI), whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts, and that supports direct interactions with other software applications using XML based messages via Internet-based protocols. Although in theory independent from the Internet technology used, Web services normally use HTTP and SOAP for exchanging messages, WSDL for describing their supported and required operations, and UDDI for being registered and discovered.

On top of WSDL, WSCI is a proposal of an XML-based language used to describe the flow of messages exchanged by a Web service. It allows the description of the observable behaviour of a Web service in a message exchange. WSCI describes a one-sided interface for a single Web service, i.e., the message exchange is described from the point of view of each Web service.

WSCI addresses Web service choreography from two primary levels. At the first level, WSCI builds up on the WSDL `<portType>` capabilities to describe the flow of messages exchanged by a Web service. The `<interface>` construct introduced by WSCI permits the description of the externally "observable behaviour" of a Web service, facilitating the expression of sequential and logical dependencies of exchanges at different operations in WSDL `<portType>`. In addition, the WSCI `<interface>` allows the specification of correlations of related exchanges, grouping of exchanges into transactions, and allows also exception handling and process definition.

At the second level, WSCI defines the `<model>` construct, which allows composition of two or more WSCI `<interface>` definitions (of the respective Web services) into a collaborative process involving the participants represented by the Web services. WSCI calls this the "*global model*". A WSCI global model provides the ability to "link" operations in different interfaces and to specify the direction of message flow between the linked operations, into a collective message exchange among interacting Web services, providing a global, message-oriented view of the overall process based on Web services.

```
<wsdl:portType name="BStoTraveller">
    <wsdl:operation name="Book">
        <wsdl:input message = "bookingRequest" />
        <wsdl:output message = "bookingAck" />
    </wsdl:operation>
    <wsdl:operation name="Confirmation">
        <wsdl:output message = "bookingConfirmation" />
    </wsdl:operation>
    <wsdl:operation name="Refusal">
        <wsdl:output message = "bookingRefusal" />
    </wsdl:operation>
</wsdl:portType>
...
<interface name = "BookingService">
    <process name = "BookTrip" instantiation = "message" >
        <sequence>
            <action name = "ReceiveBooking"
                operation = "BStoTraveller/Book"
            </action>
         <switch>
           <case>
             <condition>placesAvailable</condition>
               <action name = "SendConfirmation"
                  operation = "BStoTraveller/Confirmation"
               </action>
           </case>
           <default>
               <action name = "SendRefusal"
                  operation = "BStoTraveller/Refusal"
               </action>
           </default>
         </switch>
        </sequence>
    </process>
</interface>
```

Fig. 1. Example of WSCI description for a simple booking service.

The following subsections describe the main concepts defined in the WSCI specification. It is important to note that WSCI builds on top of WSDL, so all data, messages and operations types are defined using the WSDL mechanisms — WSCI just adds to them the choreography descriptions.

2.1 WSCI interfaces

WSCI describes the behaviour of a Web service in terms of choreographed *activities*. Activities may be atomic or complex. Atomic activities are also known as *actions*. They constitute the basic unit of behaviour of a Web service, such as sending or receiving a message, or waiting for a specified amount of time. Actions dealing with messages are bound to WSDL operations, as defined in a WSDL `<portType>`.

Complex activities are recursively composed of other activities. Ultimately, complex activities are built on actions. Each complex activity defines a specific kind of choreography for the actions it is composed of. WSCI supports the definition of sequential and parallel executions of activities, conditional executions, and loops (the activities are repeatedly executed based on the evaluation of a conditional expression).

A *process* is a portion of behaviour that is labelled with a name. Processes are the basic WSCI units of behaviour reuse. Processes can be instanced by calling or spawning them from actions or from other processes, or by the receipt of one or more of the messages that are defined as triggering the process.

Exception handling behaviour can be declared in any context, and associates *exceptions* with the set of activities that the service will perform in response to such exceptional behaviour (that can be the receipt of a given message, the occurrence of a fault, or the occurrence of a timeout). The occurrence of an exception causes the current context to terminate right after the activities associated with the exception have been performed. At this point, the behaviour defined in the parent context is resumed.

In order to illustrate these concepts, Figure 1 shows an example inspired by [11] that describes a fragment of the WSCI description of a Web service implementing a booking service functionality in an airline ticketing system. The WSDL `<portType>` element describes the Web service operations, while the WSCI interface describes their choreography. In this case, it defines a process consisting of three actions. Each action is mapped to the corresponding WSDL operation. Notice that for sake of simplicity we have omitted the possible parameters present in actions, which would notably increase the length of the example.

2.2 WSCI global model

As previously described, a WSCI global model permits a global view of the overall message exchange among the set of Web services involved in a conversation. It imports all the WSCI descriptions of all communicating participants, and links the names of individual operations in each service. More precisely, it provides a set of connections (or mappings) between pairs of individual operations of communicating participants.

An example of a WSCI global model, describing an `AirlineTicketing` system composed of three Web services (the `BookingService` in Figure 1, and two hypothetical `Traveller` and `Airline` components) is shown in Figure 2. The model specifies how the operations in the interfaces of the services are connected (e.g. `Book` in the `BookingService` with `BookFlight` in the `Traveller`) in order to orchestrate the choreography of the composite system. However,

```
<model name= "AirlineTicketing">
   <interface ref="bos:BookingService" />
   <interface ref="tra:Traveller" />
   <interface ref="air:Airline" />
    ....
  <connect operations = "bos:BStoTraveller/Book
                         tra:TravellerToBS/BookFlight" />
   ....
</model>
```

Fig. 2. Example of WSCI global model for an airline ticketing system.

these operations are supposed to be mirror images of each other, which is a very strong assumption. For instance, the global model can associate operations that have been named differently in two communicating parties, but it supposes that the arguments of these operations match perfectly. Therefore, WSCI only provides very simplistic mechanisms at the global model level for connection and adaptation.

3 Formalizing WSCI

The nature and intrinsic features of WSCI suggest the use of a process algebra for formalizing it. For instance, the π-calculus would be a good candidate, but since mobility is not required to formalize WSCI (all channels are known beforehand, and there is no such things as object or web service "factories"), CCS [5] will be sufficient for our purposes. Anyway, the encodings proposed in this paper can be easily translated into any other standard process algebra. Moreover, the choice of CCS (rather than π-calculus) reduces the complexity of the verification of compatibility of behaviour and the adaptation of mismatching behaviour, as we shall discuss in the next sections.

Although we refer to [5] for a detailed description of CCS, we will give here a brief introduction to its syntax[2]. A process P in CCS will be given by:

$$P ::= 0 \mid \alpha.P \mid P + P \mid P \parallel P \mid A(\tilde{x})$$
$$\alpha ::= a?(x) \mid a!(x) \mid \tau$$

where a is a channel name, x is a data value, $\tilde{x}$ is a sequence of values, and 0 denotes the empty process. Every process can be prefixed by an atomic action α, or composed (either in parallel '$\parallel$' or by means of the choice '+' operator) with other processes. Atomic actions are given by the internal (or silent) action τ, input actions (a message x is received from a channel a) and output actions (a message is sent through a channel). For any process identifier A there must be a unique defining equation $A(\tilde{x}) = P$. Then, $A(\tilde{y})$ behaves

[2] Some CCS operators (e.g. *restriction* or *conditinal*) are not necessary for the discussion below. Thus, we will present here only a subset of CCS.

like $P\{\tilde{y}/\tilde{x}\}$. Defining equations provide recursion, since P may contain any process identifier, even A itself.

The operational semantics of CCS is defined by a transition system where standard rules model parallel and choice operators, and synchronization is produced by the parallel composition of two complementary actions, following the transition rule:

$$\frac{P \xrightarrow{a!(x)} P' \quad Q \xrightarrow{a?(y)} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'\{x/y\}}$$

where labelled transitions $\xrightarrow{a!x()}$ and $\xrightarrow{a?(y)}$ are provided by prefix actions. The resulting communication is represented by a τ transition.

In spite of its simplicity, CCS presents a high expressive power, capable of capturing WSCI. The formalization of WSCI in CCS can be described in three steps:

(i) We first consider an "untagged" version[3] of WSCI — shown in Figure 1 — which is isomorphic to WSCI, but without the verbosity of the XML tags.

(ii) Then, we show how the untagged version of WSCI can be translated into CCS. This kind of translation is normal practice in process algebra, and indeed it is quite similar to the translation of the procedural language $\mathcal{M}$ into CCS originally described by Milner in [5] to show how non-trivial programming languages can be quite simply encoded in CCS.

(iii) Finally, the connections between operations in the WSCI global model are naturally translated into CCS by putting the corresponding processes in parallel, and linking their channels accordingly to what is specified in the global model.

Instead of providing the full formal definition of the WSCI to CCS translator, we will illustrate the translation process by selecting some WSCI constructs (following the syntax defined in Figure 3), and by showing the resulting CCS processes. We think this gives the reader a clearer and easier view of the approach followed.

A number of the WSCI constructs correspond trivially to some of the CCS operators. This is the case of WSCI atomic actions for sending and receiving messages. Each WSDL message is represented by a CCS chan-

[3] The version of WSCI considered here does not include some constructs such as correlations and transactions, and hence the corresponding XML elements `<correlation>`, `<correlate>`, `<transaction>`, `<compensation>` and `<compensate>` have been omitted. WSCI elements' `<documentation>` attribute, as well as action `roles`, have been ignored too, since they are not required at this level of abstraction.

```
interfaceDef  ::= interface name = { processDef+ }
processDef    ::= process name = { [contextDef] activityDef+ }
contextDef    ::= context { processDef* [exceptionDef] }
activityDef   ::= actionDef
                | sequence [name] { [contextDef] activityDef+ }
                | all [name] { [contextDef] activityDef* }
                | switch [name] { case+[default] }
                | choice [name] {onMessage* | onTimeout* | onFault*}
                | foreach [name] { list } do { [contextDef] activityDef+ }
                | while [name] { boolExpr } do { [contextDef] activityDef+ }
                | until [name] { boolExpr } do { [contextDef] activityDef+ }
                | empty [name]
                | fault [name] { faultName }
                | call [name] { processName }
                | spawn [name] { processName }
actionDef     ::= operation [name] operation ; [ call [name] { processName } ; ]
operation     ::= oneWay | requestResp | notification | solicitResp
oneWay        ::= in msg
requestResp   ::= in msg ;  out msg
notification  ::= out msg
solicitResp   ::= out msg ;  in msg
msg           ::= wsdlPortType/wsdlOperation/msgName
case          ::= case { boolExpr } do { [contextDef] activityDef+ }
default       ::= default { [contextDef] activityDef+ }
exceptionDef  ::= exception {onMessage* | onTimeout* | onFault*}
onMessage     ::= onMessage { (oneWay | requestResp) [contextDef] activityDef* }
onTimeout     ::= onTimeout { timeout } do { [contextDef] activityDef+ }
onFault       ::= onFault { name } do { [contextDef] activityDef+ }
timeout       ::= atDateTime timeInstant
                | duration timeDuration
                | fromStartOf name duration timeDuration
                | fromEndOf name duration timeDuration
```

Fig. 3. Grammar for the untagged version of WSCI.

nel, whose name is formed by the `portType`, operation, and message names (e.g.,“`port/op/msg`”). Hence, WSCI atomic actions correspond to input and output actions on the corresponding channels. The values transmitted through channels correspond to the actual contents of the messages. Empty values are used when we are not interested in the actual contents of the messages being exchanged. For instance:

$$[\![\,\texttt{in port/op/msg}\,]\!] = \texttt{port/op/msg}\ ?()$$

$$[\![\,\texttt{out port/op/msg}\,]\!] = \texttt{port/op/msg}\ !()$$

Obviously, the empty action corresponds to the silent transition:

$$[\![\,\texttt{empty}\,]\!] = \tau$$

Similarly, the WSCI `all` construct can be directly translated by considering the parallel composition of CCS processes. Thus, the translation of an activity as:

```
A = all P1 P2 ... Pn
```

will be given by a process $[\![\,\texttt{A}\,]\!]$ defined as:

$$[\![\,\texttt{A}\,]\!] = [\![\,\texttt{P1}\,]\!] \parallel [\![\,\texttt{P2}\,]\!] \parallel \cdots \parallel [\![\,\texttt{Pn}\,]\!]$$

However, the `sequence` construct is not simulated in such a simple way. In fact, an activity definition like:

```
Seq = sequence P1 P2 ...  Pn
```

would be translated to the process $[\![\,\texttt{Seq}\,]\!](sq_0, sq_n) \parallel sq_0!().\ 0$, with:

$$\begin{aligned}[\![\,\texttt{Seq}\,]\!](sq_0, sq_n) = & [\![\,\texttt{P1}\,]\!](sq_0, sq_1)\\ \parallel\ & [\![\,\texttt{P2}\,]\!](sq_1, sq_2)\\ & \ldots\\ \parallel\ & [\![\,\texttt{Pn}\,]\!](sq_{n-1}, sq_n)\end{aligned}$$

where

$$[\![\,P\,]\!](begin, end) = begin?()\ .\,[\![\,P\,]\!](end)$$

and $[\![\,P\,]\!](end)$ is recursively defined on the process structure, in such a way that after proceeding with all actions in P, a signal is sent on the channel end ($end!()$). Therefore, actions in process $[\![\,P_i\,]\!]$ will only proceed when a signal is sent on channel sq_{i-1} by the process $[\![\,P_{i-1}\,]\!]$ ($i > 0$), and this is made only when all the actions in the latter process have been executed. The first process P_1 is immediately activated by the signal $sq_0!()$ in the initial call.

Loops are encoded accordingly, combining the choice operator (‘+’) and recursion in CCS as described in [5].

Setting a `timeout` is modelled by spawning a CCS agent `Timer` that waits for the timeout to expire, and then either sends back a notification through a common channel (*timeout*), or accepts an input action to cancel the timeout:

$$\begin{aligned} Timer(timeout, cancel, end) = {} & \tau.\ timeout!().\ 0 \\ & + cancel?().\ end!().\ 0 \end{aligned}$$

where τ represents an internal (silent) action of the *Timer* agent, and *end* is a channel on which a signal is sent when the timer is cancelled.

Faults are exception mechanisms declared, raised, and handled within the same Web service. In WSCI, faults are raised by means of a `fault` construct, which also indicates a fault name. Such name will be used to "catch" the fault in an `exception` construct. Faults are modelled by channels, with the same name as the fault. Causing a fault is modelled by performing an output through the corresponding channel. Similarly, timing exceptions are also managed by the `exception` action, but in this case, the way of proceeding depends on what kind of timed property has to be fulfilled. For instance, a context `Exc` composed by a process `P` and a timeout clause with an exception handler `Q` as follows:

```
Exc = context {
         process P
         exception {
            onTimeout t do Q
         }
      }
```

will be translated to:

$$\begin{aligned} [\![\,\texttt{Exc}\,]\!](begin, end) = {} & [\![\,\texttt{P}\,]\!](begin, raised, cTO) \\ & \parallel\ Timer(eoTO, cTO, end) \\ & \parallel\ eoTO?().raised!().catch!().0 \\ & \parallel\ [\![\,\texttt{Q}\,]\!](catch, end) \end{aligned}$$

where invoking `Exc` corresponds to the process $[\![\,\texttt{Exc}\,]\!](begin, end) \parallel begin!().\ 0$.

It is worth noting that the translation to CCS abstracts away some details irrelevant to the choreography itself like the duration of the timeout. That is, the resulting process $[\![\,\texttt{Exc}\,]\!]$ will behave as $[\![\,\texttt{P}\,]\!]$ or $[\![\,\texttt{Q}\,]\!]$, depending on the *Timer* behaviour, but does not take into account the argument `t` of the `onTimeout` clause. Also observe that the translation of a process using an exception handler introduces an extra argument for aborting its execution when the exception is *raised*.

The WSCI conditional construct `switch`, in which one out of several sets of activities is executed based on the evaluation of conditions is modelled using the choice operator ('+') in CCS. For instance, a `switch` construct like:

$$
\begin{aligned}
[\![\,\texttt{BookTrip}\,]\!] = {} & \texttt{BStoTraveller/Book/bookingRequest}?().\\
& \texttt{BStoTraveller/Book/bookingAck}!().\\
& (\ \tau.\ \texttt{BStoTraveller/Confirmation/bookingConfirmation}!().\ 0\\
& +\ \tau.\ \texttt{BStoTraveller/Refusal/bookingRefusal}!().\ 0\\
&)
\end{aligned}
$$

Fig. 4. Choreography of the booking service process `BookTrip` translated into CCS.

```
Sw  =  switch bExp
          case exp1 do P1
          case exp2 do P2
          ...
          case expN do Pn
```

will be translated to:

$$
\begin{aligned}
[\![\,\texttt{Sw}\,]\!] = {} & \tau.\,[\![\,\texttt{P1}\,]\!]\\
& + \tau.\,[\![\,\texttt{P2}\,]\!]\\
& \quad \cdots\\
& + \tau.\,[\![\,\texttt{Pn}\,]\!]
\end{aligned}
$$

where, once again, the translation process abstracts from some computational details, which are irrelevant from the modeling point of view. Thus, the resulting process $[\![\,\texttt{Sw}\,]\!]$ will non-deterministically proceed by one of the switch branches; each one prefixed by a τ action, since the choice will be made according to an internal decision of the process `Sw` (in this case, the evaluation of certain boolean expression). Thus, for example, the process `BookTrip` declared in the interface of Figure 1 is translated to the CCS term shown in Figure 4.

Finally, the WSCI `choice` construct, that waits for given messages, faults or timeouts to happen, can also be modelled with this choice operator. For instance, a `choice` construct like:

```
Ch  =  choice
          onMessage in msg P
          ...
          onFault failure do Q
```

will be translated to:

$$
\begin{aligned}
[\![\,\texttt{Ch}\,]\!] = {} & msg?().\,[\![\,\texttt{P}\,]\!]\\
& + \cdots\\
& + failure?().\,[\![\,\texttt{Q}\,]\!]
\end{aligned}
$$

4 Reasoning about Web service compatibility

The main advantage that we can get from using formal notations to describe Web service choreography is the ability to reason about Web services behaviour. Once we have shown how Web service choreographies written in WSCI can be translated into a process algebra, in particular CCS, in this section we discuss the sort of checks that can be carried out, the moments in which those tests can be done, and the mechanisms required for that purposes.

4.1 Web service compatibility

In previous works [2,3] we developed a formal notion of behavioural compatibility for software architectures and CORBA components, respectively. In this Section we will discuss how such notion can be directly applied to Web services.

We will consider that a software system, formed by the composition of several entities specified in a process algebra, is compatible when it terminates without requiring any interaction with its environment, i.e. when it always performs a finite number of silent actions τ leading to the inaction 0. However, the majority of client/server systems are not terminating, since usually servers provide their services running on an infinite loop. Therefore, we must extend this definition in order to accommodate also infinite sequences of silent actions. Thus, we give a negative definition, saying that a system *fails* when, considered as isolated from its environment, it may perform a finite sequence of silent actions leading to a process which is structurally different from the inaction, and that cannot perform any further action by itself. Now, we can say that a system is *compatible*, or that it *succeeds*, when it does not fail. Then, for analysing compatibility we proceed recursively by matching the input and output actions that the different system components may perform at a given point, until we arrive either to inaction, or to a repeated state in the analysis.

The distinction between global and local choices plays a significant role in compatibility analysis. Suppose that a certain service performs one out of several outputs depending on an internal condition (e.g. the value of some internal variable, as in the `switch` construct in the choreography of the booking service process `BookTrip` in Figure 1). In process algebra terminology this is considered as a *local* choice, and it is represented in CCS by a combination of the choice operator ('+') with `tau` actions, as shown in the CCS encoding of `BookTrip` in Figure 4. Every local choice must be taken into account when analysing absence of deadlocks, since the rest of the system cannot foresee what the result of the choice will be, and consequently must be able to follow this decision —otherwise the system would deadlock.

On the contrary, when a service is waiting for the occurrence of one out of several input messages (as in a `choice` construction in WSCI), this is considered as a *global* choice, since the selection of a particular branch will only take place if another component in the system presents the complementary action. If not, simply the global decision will not occur. Hence, only global decisions which are common (in the form of complementary actions) to two or more system elements must be taken into account for determining deadlock-freedom.

In order to illustrate such notion of compatibility, let us consider again the piece of CCS specification of the booking service choreography in Figure 4, and let us suppose that the choreography corresponding to a potential client of the service is as follows:

```
Traveller = TravellerToBS/Book/bookingRequest!().
            TravellerToBS/Book/bookingAck?().
            TravellerToBS/Book/bookingConfirmation?(). 0
```

In this simple example, it is evident that both choreographies are not compatible: the local choice in the service between actions `bookingConfirmation` and `bookingRefusal` is not supported by the client, that just assumes the former as the only possible answer of the service.

In summary, using this kind of compatibility analysis we are in a position to check whether two (or more) Web services can be successfully composed or not. This analysis can be performed either statically at design time, when a Web service is constructed by composing several other services, or dynamically at connection time, when a client wants to check its behavioural compatibility against a given Web service that it is about to use.

4.2 Web service replaceability

Replaceability refers to the ability of a software entity to substitute another, in such a way that the change is transparent to external clients [8]. Replaceability and compatibility are the two flip sides of the *interoperability* coin. This issue is not difficult to solve at the WSDL level, it is just a matter of checking that the interface of a new Web service contains all the operations of the service to be replaced. However, the situation is different at the behavioural level:

- In the first place, we also need to check that the services required by the substitute when implementing the old service's methods are a subset of those required by the old one. Otherwise, we may need to add some additional services to the system when replacing the old service with the new one.
- And second, the behaviour (i.e., relative order among incoming and outgoing

messages) of the new version of the service should be consistent with that of the old one.

The first issue can be easily managed, since required operations are explicitly declared in the WSDL interface of the services. With regard to the second point, process algebras offer the standard axiomatization of bisimilarity, which supports the replacement of processes. However, bisimulation is too strict for our purposes since it forces the behaviour of both processes to be undistinguishable. Effective replacement of a Web service often implies that it must be adapted or specialized to accommodate it to new requirements [8]. For this reason, we make use of a specific mechanism for behavioural subtyping of processes (less restrictive than bisimilarity) defined in [2], which allows to decide whether a given software entity with behavioural description $\mathtt{WS}_1$ can be replaced by another one with behavioural description $\mathtt{WS}_2$, while keeping clients unaware of the change.

The detailed description of this relation of behavioural subtyping is beyond the scope of this paper; we are here more interested in showing its applicability than its technical aspects. However, we may roughly say that a certain behaviour $\mathtt{WS}_2$ is a subtype of another one $\mathtt{WS}_1$ if (*i*) $\mathtt{WS}_2$ preserves the semantics of behaviour of $\mathtt{WS}_1$ (i.e., if any global choice offered by $\mathtt{WS}_1$ is also offered by $\mathtt{WS}_2$); (*ii*) $\mathtt{WS}_2$ does not extend $\mathtt{WS}_1$ (i.e., if any action present in $\mathtt{WS}_2$ is also present in $\mathtt{WS}_1$); and (*iii*) $\mathtt{WS}_2$ terminates when $\mathtt{WS}_1$ does also terminate. If these conditions are fulfilled we can ensure that $\mathtt{WS}_2$ may replace $\mathtt{WS}_1$ in any context, that is, when combined with any client.

Since condition (*ii*) seems to be too restrictive, we have also introduced a relation of *extension* of behaviour, derived from that of subtyping, that also allows the new Web service to extend the old one by adding new operations that do not interfere with the behaviour already present of the old service, therefore ensuring safe substitution.

Let us consider once again the BookTrip CCS process in Figure 4, and suppose that we want to replace it by a new version, whose behavioural description is as follows:

```
ImprovedBookTrip =
  BStoTraveller/Book/bookingRequest?().
  BStoTraveller/Book/bookingAck!().
  BStoTraveller/Confirmation/bookingConfirmation!(). 0
```

This behaviour states that the now the (rather simplistic) Web service always confirms the booking requests, being the original refusal branch cut off.

Using the definitions of replaceability in [2,3], it is easy to conclude that the ImprovedBookTrip above is a valid substitute for the original service, and that any process that was compatible with the latter will be also compatible with the improved version (hence, keeping clients unaware of the substitution). Furthermore, if we check the new `ImprovedBookTrip` against the `Traveller` described in Section 4.1, we will find that they are compatible, which illustrates the intuitive notion that a service whose behaviour is a subtype of that of another, is compatible with more clients than its supertype.

5 Web service adaptation

Available Web service description languages like WSDL specify the functionality offered (and required) by Web services in a similar way to what is done with IDLs in component and object platforms (e.g., CORBA, COM, J2EE). Such kind of interface descriptions are important for software adaptation, since they highlight possible signature mismatches (i.e., differences in names and parameters of operations and messages). However, solving all signature problems does not guarantee that the entities described will be suitable for interoperation, since mismatches may also occur at the behavioural level. As we have seen in the preceding section, formalizing a choreography description language such as WSCI allows us to detect behavioural mismatch by compatibility analysis. However, what happens if we find that the behaviour of two entities is not compatible (as it happens with `BookTrip` and `Traveller` in our example)? Is it possible to build any extra element that *adapts* their interfaces, compensating their differences? In this section, we will focus on the problem of *adapting* mismatching behaviour that Web services may exhibit.

In a previous work [1] we developed a formal methodology for the automatic adaptation of software components. The main characteristic of that methodology is the use of a simple notation for expressing the specification of adaptors, which are intended to feature the interoperation between two entities. Given a specification of that kind, the generation of an adaptor can be fully automated by exhaustively trying to build a component that satisfies the specification. The separation between adaptor specification and derivation permits the automation of the error-prone, time-consuming task of generating a detailed implementation of a correct adaptor, thus simplifying the task of the (human) software developer.

Adaptation is a difficult problem which involves a large amount of domain knowledge and may require complex reasoning. In the first place, it does not simply amount to unifying names of messages and operations. Such kind of adaptation, typical of IDL-based platforms like CORBA, can solve only name mismatches of identical behaviour. Instead, we are interested in adapting less trivial mismatches where, for instance, reordering and remembering of messages is required.

We will specify an adaptor by means of a *mapping* specification that establishes a number of rules relating actions and data of two software entities. For instance, consider another potential client of the booking service, whose behaviour is given by the following specification:

```
Client = ClientToBS/BookFlight/request!().
         ClientToBS/BookFlight/reply?(answer). 0
```

where the parameter `answer` contains a data value either representing a confirmation or a refusal from the booking service. It is obvious that such a client is not compatible with the behaviour specified in `BookTrip`, not only because they use different names for operations, messages, and parameters, but also because their respective behaviour differ (in particular, the `Client` protocol contains less actions and alternatives).

Our approach aims at providing a notation for specifying the required adaptation between two components in a general and abstract way. The adaptor specification consists of a set of correspondences between actions and parameters of the two entities. For instance, the adaptor specification expressing the intended adaptation for the `BookTrip` process and the `Client` process above can be simply given by:

```
S = { request!()           <> bookingRequest?(), bookingAck!();
      reply?("Confirmed")  <> bookingConfirmation!();
      reply?("Refused")    <> bookingRefusal!() }
```

where the messages in the left hand side refer to the `Client` process, while those in the right refer to the `BookTrip` process. The intended meaning of the first rule is that whenever the client outputs a `request` message, the booking service must eventually input one `bookingRequest` message, and output one `bookingAck` message. Similarly, the second rule states that whenever the booking service outputs a `bookingConfirmation`, the client will eventually input it as a `reply` message with a special value as parameter for indicating confirmation[4]. The third rule states that if the booking service outputs a `bookingRefusal`, this will be again transmitted to the client by means of a `reply` input message, now with a `Refused` parameter value.

The mapping `S` provides the minimal specification of an adaptor that will play the role of a "component-in-the-middle" between the booking Web service and its client, mediating their interactions. It is important to note that the adaptor specification defined by a mapping abstracts away many details of the

[4] Notice that since we have omitted parameters in the WSCI description of the `BookTrip` in Figure 1 in order to reduce its verbosity, this kind of constant parameters are the only ones we can use in the client's part of the example.

components' behaviour. The burden of dealing with such details is left to the automatic process of adaptor construction.

Mappings can be used to specify different important cases of adaptation, as shown for instance in the previous example. They are not only able to express one-to-one translations of message names, but also one-to-many correspondences between messages (as in the first rule of `S`), many-to-many correspondences, non-deterministic correspondences (as in the second and third rules of `S`, where the `reply` message is either matched by a service confirmation or a refusal), or even no-correspondences (e.g., when a message in the protocol of one of the entities has no corresponding message in the protocol of its counterpart). The adaptor derivation process will then be in charge of building an adaptor capable of dealing with all the possible specified situations.

The description of the algorithm for adaptor derivation can be found in [1]. Roughly, we may say that the goal of the algorithm is to build an entity `A` such that:

(i) The system formed by the composition of the adaptor and the entities being adapted is successful (i.e., all its traces lead to success), and

(ii) The adaptor satisfies the specification `S`, i.e., all the message correspondences and data dependencies specified in `S` are respected in any trace of the system.

The algorithm incrementally builds the adaptor `A` by trying to eliminate progressively all the possible deadlocks that may occur in the evolution of the system. Informally, while the derivation tree of the system contains a deadlock, the algorithm extends the adaptor by sending (resp., receiving) the message that will match the reception (resp., sending) that is blocking one of the entities being adapted.

Since there may be more than one possible message to match at a given point, the algorithm non-deterministically chooses one of them, and spawns an instance of itself for each possible choice. If there is no action that can be triggered, the algorithm (instance) fails.

Each algorithm instance terminates when the derivation tree of the system does not contain deadlocks. At this point, the generated adaptor may or may not satisfy the specification `S` (i.e., all the correspondences between messages have been respected, or not). If so, the algorithm returns the completed adaptor. It not, the instance fails. The overall algorithm fails if all its instances fail. Otherwise it returns one of the adaptors found.

The algorithm for adaptor derivation described above non-deterministically returns one of the adaptors that satisfy the given specification. While the definition of a suitable pre-order on adaptors may lead to refining the algorithm so as to derive only "minimal" adaptors, two issues arise: (1) it is not so obvious which notion of minimality to employ in this case (e.g., "size" of the adaptors vs. number of allowed interactions); (2) looking for "minimal" adaptors would increase the complexity of the algorithm, as an enumeration of the minimal results would be required.

Now, let us explain briefly how the algorithm proceeds for our example, adapting the booking service and the client accordingly to the mapping specification `S`. Initially, the adaptor is represented by the empty process (`A = 0`), and the system composed by the adaptor `A` and the processes `BookingService` and `Client` is checked for deadlock. We can find two deadlocks, corresponding to the initial actions of each of the components (client's output `request!` and agent's input `bookingRequest`). Being an output action, the first one is selected, and the adaptor is expanded with the corresponding action in order to remove the deadlock:

```
A = ClientToBS/BookFlight/request?().
```

The resulting system is checked for deadlock, and once again we find two of them: the client is blocked on input action `reply?`, while the booking service is still deadlocked in `bookingRequest?`. From these, the latter is selected, since accordingly to the first rule in the mapping `S` the adaptor has now performed the input actions requested for matching it. Hence, the adaptor is expanded to:

```
A = ClientToBS/BookFlight/request?().
    BStoTraveller/Book/bookingRequest!().
```

Then, the construction of the adaptor goes on matching the actions of the booking service, leading to (two steps in one):

```
A = ClientToBS/BookFlight/request?().
    BStoTraveller/Book/bookingRequest!().
    BStoTraveller/Book/bookingAck?().
    BStoTraveller/Confirmation/bookingConfirmation?().
```

At this point the correspondence between actions indicated in the first rule of `S` is fulfilled, and we are in a position for removing client's deadlock on action `reply?`, fulfilling also the second rule in `S`:

```
A = ClientToBS/BookFlight/request?().
    BStoTraveller/Book/bookingRequest!().
    BStoTraveller/Book/bookingAck?().
    BStoTraveller/Confirmation/bookingConfirmation?().
    ClientToBS/BookFlight/reply!("Confirmed"). 0
```

Both the client and the booking service process end at this point, but the derivation of the adaptor is not complete yet. There is still a deadlock in the evolution of the system: after performing an internal `tau` action, the booking service may deadlock on action `bookingRefusal!`. Hence, we have to expand the adaptor with the corresponding input action, as an alternative to `bookingConfirmation?`, leading to:

```
A = ClientToBS/BookFlight/request?().
    BStoTraveller/Book/bookingRequest!().
    BStoTraveller/Book/bookingAck?().
      ( BStoTraveller/Confirmation/bookingConfirmation?().
        ClientToBS/BookFlight/reply!("Confirmed"). 0
      + BStoTraveller/Refusal/bookingRefusal?().
```

Finally, the last deadlock in the system is removed matching client's action `reply?` accordingly to what is indicated in the third rule in `S`, and the adaptor derivation process end with success, returning the full adaptor:

```
A = ClientToBS/BookFlight/request?().
    BStoTraveller/Book/bookingRequest!().
    BStoTraveller/Book/bookingAck?().
      ( BStoTraveller/Confirmation/bookingConfirmation?().
        ClientToBS/BookFlight/reply!("Confirmed"). 0
      + BStoTraveller/Refusal/bookingRefusal?().
        ClientToBS/BookFlight/reply("Refused"). 0
      )
```

that satisfies both the specification given by the above mapping `S`, and the behavioural descriptions of the two entities being adapted. Hence, the adaptor `A` will allow the `BookTrip` and the `Client` processes to interoperate successfully despite their mismatches both in message names and behaviour.

Please note that in the case of WSCI, our adaptor specifications are much more expressive than WSCI "global model" descriptions (that only provide one-to-one correspondences). Our adaptor specifications permit the definition of richer correspondences, which also provides some indications on how the WSCI global model could be extended in this sense.

6 Concluding remarks

In this paper we have briefly shown how WSCI Web service choreographies can be formalized using a process algebra approach (CCS), and the benefits that can be obtained from such formalization, namely the definition of compatibility and replaceability tests between Web services, and the automatic generation of adaptors that can bridge the differences between a priori incompatible Web services. Thus we have shown how existing formal methods can be successfully applied in the context of Web services, providing useful and practical advantages.

In addition, the formal support provided by the process algebra used to represent choreographies provide us with a tool for expressing other safety and liveness properties (apart from those already mentioned of compatibility and replaceability). In fact, any property expressed as a CCS term can be considered as a choreographic specification, and therefore, checking that property on a certain Web service, would consist in analysing the compatibility among both choreographies. On the other hand, having a simple formal description technique (like CCS) to describe Web service protocols will allow us the application of model-checking techniques to construct (or extend existing validation tools, as made in [6] with Promela.

There are currently two major approaches for describing Web services choreographies, depending on whether they describe either (a) the *common view* of the choreography of the system (built only on the individual WSDL descriptions of the constituent Web services), or (b) the choreographies expected by each *individual* Web service, which are then joined together using a (simple) "global model" that describes how such independently defined choreographies relate.

BPEL4WS, WSFL and WSCDL are notations that use the *common view* approach, whilst WSCI is an example of the *individual view* approach. *Common view* notations are in general more adaptable to each particular situation and system, but are not as amenable to Web service reuse as *individual view* descriptions are. Although the Web service community is currently divided trying to decide which is the best approach,[5] we argue that they can be considered as complementary tactics, rather than rivals.

Our proposal has been applied to WSCI, but in principle it is applicable to other choreography notations either following the individual or the common view approaches. It is our opinion that we cannot afford losing the benefits that both approaches provide, specially when they can be combined together to gain all the mutual benefits. Web service providers should supply both the WSDL and WSCI (or similar) descriptions of the Web services they offer, indicating not only the names of the supported operations, but also the expected protocol that the client should follow (as in WSCI). On the other hand, system designers should of course be able to use *common view* notations for describing the choreography of their applications from a global perspective (as in BPEL4WS or WSCDL).

[5] see `http://www.mywebservices.org/index.php/article/view/1178/` for an interesting comparison between both approaches

The way to marry both *views* can be achieved by using the results that we have discussed in this paper, simply checking that the "projections" of the global choreography of a system (defined by using a common view approach) over its constituent Web services can be *replaced* (in our sense) by their individual protocol descriptions (defined using an *individual view* approach). In this way, both approaches could easily co-exist.

Apart from the previous work of the authors [3,2,1], there is a large amount of proposals in the literature dealing with composition, interoperation and adaptation issues in the field of Component-Based Software Engineering (CBSE), and in protocol verification in general. Some of these works have been also applied to Web service choreographies. In cite [4], building on previous work in the field of Software Architecture by the same authors, a model-based approach is proposed for verifying Web service composition, using Message Sequence Charts (MSCs) and BPEL4WS. In [7], and from a semantic Web point of view, a first-order logical language and Petri Nets are proposed for checking the composition of Web services. In [6], model-checking using Promela and SPIN is proposed for analysing the composability of choreographies written in WSFL. All these works deal with the (either manual or automated) simulation and analysis of Web service composites, been able to detect mismatch between their choreographies, but to our knowledge, ours is the first approach that proposes a way for overcoming mismatch between choreographies by means of the automatic generation of adaptors.

There are at least two immediate extensions to the work we have presented here. First, we intend to integrate the translation from WSCI to CCS with the existing tools we have already developed. And second, we intend to make effective use of the tools currently available for CCS to reason about the Web specifications, e.g., model check them. Finally, the translation into CCS presented here must be extended in order to consider full WSCI; in particular, dealing with constructs such as correlations, transactions, properties and others, that have been omitted in this work.

References

[1] Bracciali, A., A. Brogi and C. Canal, *A formal approach to component adaptation*, Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering (in press) (2004), a preliminary version of this paper was published in *COORDINATION 2002: Fifth International Conference on Coordination Models and Languages*, LNCS 2315, pages 88–95. Springer, 2002.

[2] Canal, C., L. Fuentes, E. Pimentel, J. M. Troya and A. Vallecillo, *Adding roles to CORBA objects*, IEEE Transactions on Software Engineering **29** (2003), pp. 242–260.

[3] Canal, C., E. Pimentel and J. M. Troya, *Compatibility and inheritance in software architectures*, Science of Computer Programming **41** (2001), pp. 105–138.

[4] Foster, H., S. Uchitel, J. Kramer and J. Magee, *Model-based verification of web service compositions*, in: *Proc. of Automated Software Engineering (ASE'2003)* (2003).

[5] Milner, R., "Communication and Concurrency," Prentice Hall, 1989.

[6] Nakajima, S., *Model-checking verification for reliable web service*, in: *Proc. of OOPSLA'02 Workshop on Object-Oriented Web Services*, Seattle (USA), 2002.

[7] Narayanan, S. and S. McIlraith, *Simulation, verification and automated composition of Web Services*, in: *Proc. of the Eleventh International World Wide Web Conference (WWW'2002)*, pp. 77–88.

[8] Nierstrasz, O., *Regular types for active objects*, in: O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, Prentice-Hall, 1995 pp. 99–121.

[9] Vallecillo, A., J. Hernández and J. M. Troya, *New issues in object interoperability*, in: *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, number 1964 in Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, 2000 pp. 256–269.

[10] W3C, "Web Service Description Language (WSDL) 1.1," World Wide Web Consortium (2001), available at `http://www.w3.org/TR/wsdl`.

[11] W3C, "Web Service Choreography Interface (WSCI) 1.0," World Wide Web Consortium (2002), available at `http://www.w3.org/TR/wsci`.

[12] Yellin, D. M. and R. E. Strom, *Protocol specifications and components adaptors*, ACM Transactions on Programming Languages and Systems **19** (1997), pp. 292–333.