Formalizing WSBPEL Business Processes Using Process Algebra

Javier Cámara [a], Carlos Canal [b], Javier Cubo [a]
Antonio Vallecillo [b]

[a] *CITIC, Andalusian ICT Centre, Málaga, Spain.*
Emails: jcamara@citic.es, jcubo@citic.es

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

Abstract

Industry standards for Web Service composition, such as WSBPEL, provide the notation and additional control mechanisms for the execution of business processes in Web Service collaborations. However, these standards do not provide support for checking interesting properties related to Web Service and process behaviour. In an attempt to fill this gap, we describe a formalization of WSBPEL business processes, that adds protocol information to the specifications of interacting Web Services, and uses a process algebra to model their dynamic behaviour — thus enabling their formal analysis and the inference of relevant properties of the systems being built.

Key words: Web Services, orchestration, WSBPEL, formal methods, process algebra, interoperability, adaptation.

1 Introduction

Web Services workflow systems emerge as the natural evolution of workflow systems and business processes in organizations. This is due to the evolution that business operations have experimented in the last few years. Nowadays a process may not necessarily be based only on a single or even a group of internal applications, as in traditional workflow systems. Examples are resource management, production control, or any kind of collaborative process in general.

Organizational workflow tends to be more interdepartmental and involve different partners. Each one of these entities owns its own processes, which can be more or less heterogeneous and complex. In order to build applications that can give support to these processes, new systems which overcome the limitations of traditional workflow systems have been developed. These are usually denominated Business Process Management Systems (BPM). BPMs

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

are capable of coordinating long "conversations" between the entities involved in the process, managing different aspects such as execution threads, or error handling.

Many implementations of these BPMs have been developed and tested during the last few years. However, none of them has obtained wide acceptance. This situation has recently changed with the arrival of the Web Services Business Process Execution Language, or WSBPEL [5] (formerly known as BPEL4WS), supported by major industrial partners and totally based on Web Services. This standard allows the description of the interactions between the different entities involved in a business process.

WSBPEL uses an XML-based description language which basically identifies partners, interactions, and the global process coordinating them. The coordination model that WSBPEL uses is referenced as Web Service "orchestration". In contraposition, we have Web Services "choreography", used in other standards such as the W3C's Web Service Choreography Description Language, or WS-CDL [13]. While in a choreography interaction occurs between any pair of partners arbitrarily, in orchestration all interactions have a single partner and the coordination process as endpoints. A classical example of a Web Service orchestration is the booking process of an Internet travel agency, where there are different partners such as airlines, car rental services, and hotels. With WSBPEL we can describe, for instance, how to coordinate (orchestrate) these partners in order to produce a travel plan, by defining interaction rules and specifying how to handle any potential incidence throughout the process.

Unfortunately, elaborating the description of service orchestrations is still a very demanding task in terms of effort and domain knowledge, making the development of integrated Web Services a time consuming and expensive task. The need to achieve a higher degree of automation in the orchestration of Web Services has generated an important research effort by the Web Services community in order to address this issue. The software industry is also devoting more resources to solve interoperability problems, through organizations like W3C or WS-I [14]. These organizations promote the development and deployment of applications and services able to interact among them in a simple and efficient way through the Internet, independently of their underlying platforms or languages.

We believe WSBPEL is a medium-term realistic approach towards automation in Web Service Composition. Adding protocol information to interacting services, and using additional adaptation techniques based in formalization through a process algebra, will establish solid foundations for seamless integration taking advantage of previous research made in this field [11].

Being able to express with WSBPEL dynamic behaviour is an important achievement, but once we have that information, what can we do with it? Which kind of properties can be inferred from the WSBPEL descriptions? How can we prove those properties? In the case that the related Web Services

are not compatible, can we solve this situation by adapting them somehow?

In this work we show how the core constructs of the WSBPEL descriptions can be formalized using a standard process algebra notation (namely, CCS [8]). This will enable us to check if a group of Web Services are compatible for interoperation in the context of a business process. If not, we can check if it is possible to produce an specification from which an adaptor that mediates among them can be automatically generated [12]. In addition, system descriptions elaborated with this basic formalization of the core WSBPEL constructs can be subject to formal analysis (e.g., model checking) using some of the available tools which support standard process algebra notation.

This paper is organized as follows. Section 2 presents a brief description of the WSBPEL language syntax. Section 3 describes the modelling of the core WSBPEL language constructs using CCS. Section 4 discusses several issues about business process behavioural analysis. Finally, Section 5 draws some conclusions and outlines some future work.

2 Web Services Process Execution Language (WSBPEL)

WSBPEL is an XML-based specification language used to describe business processes which manage (orchestrate) the interaction of separate Web Services. To get an idea of how a WSBPEL process looks like, we show below a very simple process description that selects the best insurance offer among several proposals.

WSBPEL deals with Web Services orchestration at two different levels. Firstly, WSBPEL builds on top of the WSDL `<portType>` descriptions in order to provide a canalization for the messages exchanged by Web Services. WSBPEL defines `<partnerLinkType>` elements—that can be separate artifacts independent of the service's WSDL document (see Figure 1). Alternatively, the partner link type definition can be placed within the WSDL document defining *portTypes* from which the different roles are defined.

At the second level, WSBPEL defines the business process specification which includes the following sections.

Partner links, which identify relationships between the business process and the rest of the partners (Web Services) by referencing the partner link type definitions previously described. They basically provide WSDL `<portType>` definitions for process/Web-Service interactions. In our example (Figure 2), we first declare the partner links to the WSBPEL process, and three insurance Web Services (named `currentInsurance`, `insuranceA` and `insuranceB`).

Variables, which can carry data in messages, and define the state of each instance of the process. These may contain partner links, that is, abstract references to other processes. Thereby, they may be used to dynamically connect structures in some specific situations (although this feature is not supported in the current standard specification). Returning to our example, shown in Figure 3, we declare variables for the insurance request (`InsuranceRequest`),

```
...
 <portType name="InsuranceSelectionPT">
  <operation name="SelectInsurance">
    <input message="InsuranceResponseMessage"/>
    <output message="InsuranceRequestMessage"/>
  </operation>
 </portType>
...
 <plnk:partnerLinkType name="selectionLT">
   <plnk:role name="insuranceSelectionService">
     <plnk:portType name="InsuranceSelectionPT"/>
   </plnk:role>
 </plnk:partnerLinkType>
</definitions>
...
```

Fig. 1. WSDL portType and WSBPEL partnerLinkType definitions.

```
<process name="insuranceSelectionProcess">
   <partnerLinks>
      <partnerLink name="client"
                   partnerLinkType="com:selectionLT"
                   myRole="insuranceSelectionService"/>
      <partnerLink name="currentInsurance"
                   partnerLinkType="ins:insuranceLT"
                   myRole="driverInfoRequester"
                   partnerRole="insuranceService"/>
      <partnerLink name="insuranceA"
                   partnerLinkType="ins:insuranceLT"
                   myRole="insuranceRequester"
                   partnerRole="insuranceService"/>
      <partnerLink name="insuranceB" .../>
   </partnerLinks>
...
```

Fig. 2. WSBPEL partner link definition.

insurance A and B responses (`InsuranceAResponse`, `InsuranceBResponse`), and for the final selection (`InsuranceSelectionResponse`).

Activities, which describe the behaviour of a business process by specifying step by step the actions the process performs. These activities can be either basic or structured (sequential and parallel compositions, guarded choice, iterations, and multiple case). In our example (see Figure 4), we first wait for the initial request message from the client (`<receive>`). Then we invoke the three insurance Web Services (`<invoke>`) in parallel using the `flow` construct. The current insurance Web Service returns bonification information so that the insurance Web Services can return their insurance offer accordingly. Then we select the lower amount (`switch`/`case`) and return the result to the client (the caller of the WSBPEL process) using a `<reply>` activity.

In this brief description of WSBPEL we have explicitly omitted `Correlations`.

```
<variables>
   <!-- input for BPEL process -->
   <variable name="InsuranceRequest"
             messageType="InsuranceRequestMessage"/>
   <!-- output from current insurance -->
   <variable name="CurrentInsuranceResponse"
             messageType="DriverInsuranceInfoMessage"/>
   <!-- output from insurance A -->
   <variable name="InsuranceAResponse"
             messageType="InsuranceResponseMessage"/>
   <!-- output from insurance B --> <.../>
   <!-- output from BPEL process --> <.../>
</variables>
```

Fig. 3. WSBPEL variable definition.

They identify interactions relevant for a given process instance, and are used to dispatch messages correctly among different sessions. Since session control is not directly related to the behaviour of the process, issues related to this part of the WSBPEL specification fall out of the scope of this paper. The version of WSBPEL considered here does not include other important constructs such as compensation handlers, for instance. In this paper we focus just on the core constructs of WSBPEL, leaving the rest of the WSBPEL specifications for further work.

3 Formalizing Business Processes

3.1 CCS

The nature and features of WSBPEL suggest the use of a process algebra to formalize it, in a similar way as it was done for WSCI in [2]. This approach will also allow the definition of formal methodologies for the automatic derivation of adaptors in case of protocol mismatches, as described in [1].

Among the different available notations, we have chosen CCS [8] because it is expressive enough for the kind of constructs we deal with in this paper, and because it reduces the complexity (compared to other process algebras such as the π-calculus) of the verification of compatibility and the adaptation of mismatching behaviour. Anyway, the encodings proposed here are easily translatable into any standard process algebra. In addition, CCS is widely supported by tools, whilst the π-calculus lacks such a complete tool support.

The basics of the CCS are quite simple. A process P is 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 the name of a channel, x is a data value, $\tilde{x}$ is a sequence of values, and the special process 0 denotes inaction. Each process can be prefixed by an atomic action α, or composed (either using the parallel '$\parallel$' operator

```
...
<sequence>
<!-- Receive the initial request from client -->
<receive partnerLink="client" portType="InsuranceSelectionPT"
         operation="SelectInsurance" variable="InsuranceRequest"
         createInstance="yes" />
<!-- Concurrent invocations to Current Insurance, A and B -->
<flow>
 <links> <link name="current-to-A" /> <link name="current-to-B" /> </links>
 <!-- Invoke Current Insurance web service -->
 <invoke partnerLink="currentInsurance" portType="GetDriverInsuranceInfoPT"
         operation="GetDriverInsuranceInfo"
         inputVariable="DriverInsuranceInfoRequest"
         outputVariable="DriverInsuranceInfoResponse" >
   <source linkname="current-to-A"/> <source linkname="current-to-B"/>
 </invoke>
 <!-- Invoke Insurance A web service -->
 <invoke partnerLink="insuranceA" portType="ComputeInsurancePremiumPT"
         operation="ComputeInsurancePremium" inputVariable="InsuranceRequest"
         outputVariable="InsuranceAResponse">
   <target linkname="current-to-A"/>
 </invoke>
 <!-- Invoke Insurance B web service -->
 <invoke ...> <target linkname="current-to-B"/> </invoke>
</flow>
<!-- Select the best offer and construct the response -->
<switch>
 <case ... InsuranceAResponse ... <= ... InsuranceBResponse ... >
    <!-- Select Insurance A -->
    <assign>
     <copy>
      <from variable="InsuranceAResponse" />
      <to variable="InsuranceSelectionResponse" />
     </copy>
    </assign>
 </case>
 <otherwise> <!-- Select Insurance B --> </otherwise>
 </switch> <!-- Send a response to the client -->
     <reply partnerLink="client" portType="InsuranceSelectionPT"
            operation="SelectInsurance" variable="InsuranceSelectionResponse"/>
 </sequence>
</process>
```

Fig. 4. WSBPEL process activity definition.

or the choice '+' operator) with other processes. Atomic actions are given by the internal (or silent) action τ, or by input/output actions $a?(x)$, $a!(x)$ (a message x is received or sent through a channel a, respectively). For any process identifier A there must be a unique defining equation $A(\tilde{x}) = P$. Then, $A(\tilde{y})$ behaves like $P\{\tilde{y}/\tilde{x}\}$. Defining equations provide recursion, since P may contain any process identifier, even A itself.

In order to see things clearer, we give a simple example: if we have a channel a, $a!(x).P$ represents a process that sends the value x through a

channel a and then behaves like process P. Conversely, $a?(y).Q$ is a process that waits for a value x to be received through channel a, binding the variable y to the received value (let's say x), and then behaves like $Q\{x/y\}$, where $Q\{x/y\}$ indicates the substitution of the variable y by the value x in the body of Q. Process communication is synchronous in CCS, and names of channels and data values are separate sets. Hence, there is no mobility in CCS.

Despite its simplicity, CCS presents a high expressive power, capable of capturing most WSBPEL constructs and mechanisms. In order to describe our proposal for the formalization of WSBPEL in CCS we follow a three-step process. We first consider an isomorphic "untagged" version of WSBPEL (shown in Figure 5). This syntax removes the syntactic sugar associated to XML and provides a clearer view of the concepts represented to the reader. Then we show how this untagged version of WSBPEL can be encoded in CCS. And finally, the connections between operations in the WSBPEL model are translated into CCS by putting the corresponding Web Service specifications with the business process specification in parallel, and linking their channels.

Instead of providing a full definition of the WSBPEL to CCS specification, we focus in the main WSBPEL constructs (following the syntax defined in Figure 5), and show the resulting CCS processes. This basic specification can then be extended in order to give full support to the language, but it is enough in a first approach to illustrate the process to the reader.

3.2 Mapping WSBPEL language constructs to Process Algebra

A number of the WSBPEL constructs correspond trivially to some of the CCS operators. This is the case of WSBPEL atomic actions for sending and receiving messages. Each WSDL message is represented by a CCS channel, identified by a name formed by the `partnerLink`, `portType`, operation, and message names (e.g., `"plnk/port/op/msg"`). Hence, WSBPEL 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:

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

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

The empty action corresponds to the silent transition τ:

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

The WSBPEL `flow` construct can be easily translated by considering the parallel composition of CCS processes

```
Flw = flow P1 P2 ... Pn
```

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

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

processDef	::=	`process` *name partnerLinkDef*$^+$
		{ *variableDef*$^+$ *faultHandlerDef*$^+$ *activityDef*$^+$ | *scopeDef* }
scopeDef	::=	`scope` *stdParameters* [*variableDef*] [*faultHandlerDef*] *activityDef*
stdParameters	::=	[*name*] *stdElements*
stdElements	::=	*sourceOperation** *targetOperation**
variableDef	::=	`variable` *name messageType*
partnerLinkDef	::=	`partnerlink` *name partnerLinkType* [*myRole*] [*partnerRole*]
activityDef	::=	*primitiveActivityDef* | *structuredActivityDef*
primitiveActivityDef	::=	*receiveOperation* [*variable*] *stdParameters*
	|	*replyOperation* [*variable*] [*faultName*] *stdParameters*
	|	*invokeOperation* [*inputVariable*] [*outputVariable*]
		stdParameters faultHandlerDef
	|	`throw` *fault stdParameters*
	|	`empty` *stdParameters*
	|	`wait` *timeExp*
structuredActivityDef	::=	`sequence` *stdParameters activityDef*$^+$
	|	`switch` *stdParameters caseDef*
	|	`pick` *stdParameters onMessageDef*$^+$ *onAlarmDef**
	|	`while` *condition stdParameters activityDef*
	|	`flow` *stdParameters linkDef** *activityDef*$^+$
msg	::=	*partnerLink*/*wsdlPortType*/*wsdlOperation*/*msgName*
receiveOperation	::=	`in` *msg*
replyOperation	::=	`out` *msg*
invokeOperation	::=	`out` *msg* [`in` *msg*]
sourceOperation	::=	`out` *msg*
targetOperation	::=	`in` *msg*
faultHandlerDef	::=	{ `catch` *fault activityDef* }*
fault	::=	*faultName faultVariable*
caseDef	::=	{ `case` *condition activityDef* }$^+$ [`otherwise` *activityDef*]
onMessageDef	::=	`onMessage` *msg variable activityDef*
onAlarmDef	::=	`onAlarm` *timeExp activityDef*
timeExp	::=	`for` *duration* | `until` *deadline*
linkDef	::=	`link` *name*

Fig. 5. Grammar for the untagged version of WSBPEL.

Sequential composition of activities is not directly translatable. In fact, a sequence has to be defined using parallel composition

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

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

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

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.

In WSBPEL, faults are raised by means of the throw construct, which also indicates a fault name. That name will be used to catch the fault. Faults are modelled by channels named after their correspondent fault name. To raise a fault, we will perform an output on the corresponding channel. The `catch` construct is included into the `FaultHandler` section, which may be declared as a part of the global process or restricted to a specific scope.

```
Ctch  =  catch fault1 P1
         catch fault2 P2
         ...
         catch faultN Pn
```

If we consider the former situation, we can translate the `catch` construct as follows:

$$\begin{aligned}[\![\,\texttt{Ctch}\,]\!](cancel) = {} & [\![\,\texttt{P}\,]\!](cancel) \\ \parallel\ & fault_1?().\,[\![\,\texttt{P1}\,]\!] \\ \parallel\ & fault_2?().\,[\![\,\texttt{P2}\,]\!] \\ & \cdots \\ \parallel\ & faultN?().\,[\![\,\texttt{Pn}\,]\!]\end{aligned}$$

It is worth noting that the translation to CCS abstracts away some details irrelevant to the orchestration itself like the duration of the timeout. In the case of using catch into a `FaultHandler` belonging to a restricted scope S, we would just consider the substitution of P in the translation by the set of activities defined within the scope S. The WSBPEL conditional construct `switch`, in which one out of several sets of activities is executed based on the evaluation of conditions

```
Sw  =  switch
          case cond1 P1
          case cond2 P2
          ...
```

```
case condN Pn
otherwise  Q
```

is modelled using the choice operator ('+') in CCS:

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

where, once again, the translation process abstracts away some computational details (e.g., the conditions), because we are not modelling data. Hence, we are aware that several issues may arise, such as the detection of "phantom" deadlocks. This limitation is not critical in this first approach, so we will try to address it in the future. 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 $[\![\texttt{Sw}]\!]$. Finally, the WSBPEL `pick` construct waits for given messages or timeouts to happen.

```
Pck  =  pick
          onMessage msg1 P1
          onMessage msg2 P2
          ...
          onMessage msgN Pn
          onAlarm timeExp R
```

The `pick` construct can be modelled based on the choice operator ('+'):

$$\begin{aligned}[\![\texttt{Pck}]\!] = {} & msg1?().[\![\texttt{P1}]\!] \\ & + msg2?().[\![\texttt{P2}]\!] \\ & + \cdots \\ & + msgN?().[\![\texttt{Pn}]\!] \\ & + \tau.[\![\texttt{Q}]\!]\end{aligned}$$

CCS does not model time. As a consequence, when we are dealing with silent actions τ, we can consider that the `timeExp` time expression associated to the `onAlarm` part of the declaration of `pick` designates a time which has already arrived.

Finally, a while loop

```
Whl  =  while cond P
```

will be translated to:

$$[\![\texttt{Whl}]\!] = \tau.\ 0 + ([\![\texttt{P}]\!](loop) \parallel loop?().[\![\texttt{Whl}]\!])$$

Once again, we abstract the conditional clause for the loop using τ since we are not modelling data. In order to model the construct, we emit the loop signal as long as we want to perform another iteration. In this case, τ will

determine when to stop the iterations. Data abstraction may produce some undesirable effects, such as divergence of the CCS processes, but, again, this can be dealt with at a different level (see, e.g., [2]).

4 Business Process behaviour

Ideally, from the point of view of WSBPEL orchestration, Web Services should interoperate perfectly with each other, or at least, we should be able to know beforehand when two services have a compatible behaviour. Unfortunately, full automatic analysis of complex software systems is still not possible. However, there are a number of aspects that can be analyzed with our approach. Two key aspects to interoperability are compatibility and replaceability. For our purposes, 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. However, this definition must be extended as detailed in [2] since client/server systems do not terminate, and we must consider infinite sequences of silent actions. A formal notion of behavioural compatibility has been developed throughout several works such as [4] for software architectures and CORBA components. These notions can be directly applied to Web Services. In [7] a model-based approach is proposed for verifying Web Services composition, using Message Sequence Charts (MSCs) and WSBPEL. In order to illustrate the example that we described in section 2, the translation of the business process specification in CCS is shown in Figure 6. Note that `SelectInsuranceA` and `SelectInsuranceB` are activities carried out in the `switch/case` section of the specification. They correspond to internal actions. Link dependencies in the `flow` construct are represented by synchronization channels `current-to-A` and `current-to-B`.

Once we have the specification of the business process in CCS, we can check its compatibility against the specification of the Web Services it interacts with. However, in most situations WSBPEL processes interact either with stateless Web Services or with services that do not have a behavioural description available (we have to assume that they are going to behave in the expected way, message order is going to be correct, etc.). Because of that, we need to complement the static description of the Web Services involved in the business process with behavioural information. This description may be elaborated using different notations, such as MSCs, UML descriptions, or WSBPEL itself [15]. In general, any notation susceptible of being translated to standard process algebra will be valid for such a purpose. This approach provides analytic power since we can infer interesting properties from the composition of heterogeneous systems which may be described using different techniques. Having behavioural descriptions available of all the entities involved in the process we can analyze their compatibility by putting in parallel these descriptions, and connecting the different channels for communication between

```
InsuranceSelectionProcess =

Client/InsuranceSelectionPT/SelectInsurance/InsuranceRequest?() .
 ( (
     CurrentInsurance/GetDriverInsuranceInfoPT/GetDriverInsuranceInfo/
                     DriverInsuranceInfoRequest?() .
     CurrentInsurance/GetDriverInsuranceInfoPT/GetDriverInsuranceInfo/
                     DriverInsuranceInfoRequest?() .
     current-to-A!() . current-to-B!()  )
     || ( current-to-A?() .
         InsuranceA/ComputeInsurancePremiumPT/ComputeInsurancePremium/
                     InsuranceRequest!() .
         InsuranceA/ComputeInsurancePremiumPT/ComputeInsurancePremium/
                     InsuranceAResponse!() )
     || ( current-to-B?() .
         InsuranceB/ComputeInsurancePremiumPT/ComputeInsurancePremium/
                     InsuranceRequest!() .
         InsuranceB/ComputeInsurancePremiumPT/ComputeInsurancePremium/
                     InsuranceBResponse!() )
  ) .   (tau . SelectInsuranceA  + tau . SelectInsuranceB ) .
Client/InsuranceSelectionPT/SelectInsurance/InsuranceSelectionResponse!().0
```

Fig. 6. CCS translation of the WSBPEL Insurance Selection Process specification.

them. In this respect, we have to consider several issues related to the inherent differences between the choreographic and the orchestration approaches, described in [3].

Replaceability refers to the ability of a software system to substitute another, in such a way that the change is transparent to external clients. In stateless Web Services, replaceability is fairly easy to check. We only have to test that the WSDL description of the new service contains all the operations of the replaced service. However, the situation is different at the behavioural level. First, we need to check that the dependencies of the new service when implementing the methods of the old one, are a subset of the dependencies of the old service. Second, we have to check that the relative order of incoming and outgoing messages of the old service is preserved by the messages of the new one.

Additionally, in case of detecting a mismatching or incompatible behaviour between a Web Service and the business process, we could attempt to perform adaptation between the two of them. The formalization previously described would allow adaptation of the services at the behavioural level using algorithms similar to the one described in [1].

5 Conclusions and future work

Throughout this work we have described a potential approach to the formalization of Web Service orchestration, with a specific interest in WSBPEL, the current industry standard, using a process algebra (CCS).

By formalizing core WSBPEL language constructs we have provided a basic system to reason about Web Service compositions in a relatively simple but expressive way. A previous proposal of a behavioural WSBPEL description was developed in [11], using an algebraic-style abstract syntax (along with its operational semantics). In this paper we have presented a more pragmatic approach to formalization by making use of a standard and well-known process algebra such as CCS. This also provides access to model checking and other analysis tools available for that notation. In particular, the Concurrency Workbench of the New Century (CWB-NC) [10], includes a model checker for determining whether a system satisfies a given formula written in an expressive temporal logic, the modal mu-calculus, or in a standard process algebra through a front end tool (there are several available for CCS, CSP, Basic Lotos, etc.). The CWB-NC explores every possible state that the system may reach during execution, and checks to see whether any invalid state is reachable. If such an incorrect state is detected, a description of the execution sequence leading to the state is reported to the user. Deadlock detection during the execution of a business process, for example, is feasible using this approach.

The Concurrency Factory [9] is an integrated environment for the modelling and verification of concurrent systems. Like the CWB-NC, the Concurrency Factory uses standard process algebra as the theoretical basis for its formal modelling notation and model checking as its primary verification technique. In this case, the tool supports automatic code generation. Verified designs may be automatically translated into Java code, for instance.

Other issues related to WSBPEL behavioural modelling still remain open. For example, in this first approach we have not considered mobility, since dynamic service binding is not featured in the standard yet. It is true, however, that a number of vendors are starting to support this non-standard characteristic in their tools, as it is the case of [6], where "computed" partner links can be created at design time, and later completed with partner link information resolved at execution time. This feature could be modelled using another standard process algebra such as the π-calculus, which allows mobility, giving the possibility of sending and receiving channel names as values. Furthermore, for this study, we have considered only basic WSBPEL language constructs and mechanisms. The idea is extending this initial set by also providing modelling mechanisms for data and other WSBPEL features. In this sense, this study has demonstrated to what point CCS can be stretched in order to model WSBPEL constructs.

As a last remark, formalization of Web Service composition specification through a process algebra such as CCS, allows a higher degree of independence between analysis algorithms and description language syntax. In our opinion, following this approach will enable the software engineering community to reuse analysis techniques as well as to adapt them to new standard specifications whenever required.

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 **74** (2005), pp. 45–54.

[2] Brogi, A., C. Canal, E. Pimentel and A. Vallecillo, *Formalizing Web Service Choreographies*, in: *Proc. of the 1st International Workshop on Web Services and Formal Methods (WS-FM'04)*, Electronic Notes in Theoretical Computer Science **105** (2004), pp. 73–94.

[3] Camara, J., C. Canal and J. Cubo, *Issues in the formalization of Web Service Orchestrations*, in: *Second International Workshop on Coordination and Application Techniques for Software Entities (WCAT'05)(in press)*, 2005.

[4] 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.

[5] Curbera, F. et al., "Updated: Business Process Execution Language for Web Services (WSBPEL 1.1)," BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems (2003).
URL
www-128.ibm.com/developerworks/library/specification/ws-bpel/

[6] Duerrstein, B., *Dynamic service binding with WebSphere Process Choreographer* (2004).
URL www-106.ibm.com/developerworks/library/ws-dbind/

[7] Foster, H., S. Uchitel, J. Kramer and J. Magee, *Model-based verification of Web Service compositions*, in: *Proc. of Automated Software Engineering (ASE'03)*, 2003.

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

[9] SUNY, "The Concurrency Factory: A graphical verification toolset for concurrent systems," Stony Brook - SUNY (1999).
URL www.cs.sunysb.edu/~concurr/

[10] SUNY, "Concurrency Workbench of the New Century (CWB-NC)," Stony Brook - SUNY (2000).
URL www.cs.sunysb.edu/~cwb/

[11] Viroli, M., *Towards a Formal Foundation to Orchestration Languages*, in: *Proc. of the 1st International Workshop on Web Services and Formal Methods (WS-FM'04)*, Electronic Notes in Theoretical Computer Science **105** (2004), pp. 51–71.

[12] Voigt, H., "Model-based Analysis of executable Business Processes for Web Services," Database and Information Systems Group (2004).
URL www.fots.ua.ac.be/graphtransfo_refactoring/slides/Voigt-ModelBasedBPEL.pdf

[13] W3C, "Web Service Choreography Description Language (WS-CDL) 1.0," World Wide Web Consortium (2004).
URL www.w3.org/TR/ws-cdl-10/

[14] WS-I Organization, "Interoperability: Ensuring the Success of Web Services," Web Services Interoperability Organization (2004).
URL www.ws-i.org/docs/20041130.introduction.ppt

[15] Yellin, D. M. and R. E. Strom, *Protocol Specifications and Components Adaptors*, ACM Transactions on Programming Languages and Systems **19** (1997), pp. 292–333.