

QUT Digital Repository:
<http://eprints.qut.edu.au/>



Bruno, Giorgio and La Rosa, Marcello (2006) From Collaboration Models to BPEL Processes Through Service Models. In Bussler, C., Eds. *Proceedings Workshop on Web Service Choreography and Orchestration for Business Process Management 2005* 3812/2006, pages pp. 75-88, Nancy, France.

© Copyright 2006 Springer

This is the author-version of the work. Conference proceedings published, by Springer Verlag, will be available via SpringerLink.

<http://www.springer.de/comp/lncs/> Lecture Notes in Computer Science

From Collaboration Models to BPEL Processes Through Service Models

Giorgio Bruno and Marcello La Rosa

Dip. Automatica e Informatica,
Politecnico di Torino, Torino, Italy
giorgio.bruno@polito.it, marcello.larosa@fastwebnet.it

Abstract. This paper proposes a model-based lifecycle for the development of web services, which is based on two kinds of models, collaboration models and service ones. After agreeing upon a collaboration model, which is a public specification, each party can work out a service model and then can turn it into a process written in an orchestration language such as BPEL. As the conceptual gap between a service model and its BPEL implementation is relevant, this paper is concerned with the automatic mapping of service models to BPEL processes, in line with model-based development. Moreover it discusses how to validate services with respect to collaboration models both at-design time and at run-time, and presents the bProgress software environment, which is made up of a number tools developed during this research.

1 Introduction

The technology of web services is gaining growing consensus as the platform of choice for carrying out collaborations within and across enterprise boundaries.

In its simplest form a collaboration takes place through a request-response interaction between two services, a requester and a provider: the requester sends a request, the provider reacts to the request by performing an action and then replies with a response.

Real cases are more complicated as a collaboration may require a number of interactions between the parties: for this reason the parties have to agree in advance on the message flow by working out a common model, called a collaboration model.

However in order to properly support a given collaboration, a service has to arrange its activities (receiving, sending and processing ones) within a control structure, hence it turns out to be a process. Therefore the emerging technology of orchestration languages and processes is a good choice for implementing such services.

On the other hand moving directly from a collaboration model, which is a rather neutral specification of the interactions between two services, to an orchestration process is too long a jump to be afforded in real applications and it is like skipping the design phase in software development.

As a matter of fact developing a collaboration is a process and, as such, it entails the usual phases of specification, design, implementation and operation.

At specification-time a collaboration is a model specifying the messages to be exchanged between the parties as well as the ordering and the timing constraints of those messages. A collaboration model is a public specification which the parties will use to develop and test their own services.

At design-time each party works out a service model, i.e. a more detailed model which, in addition to the activities concerned with sending/receiving the messages established in the collaboration model, includes the activities necessary for producing and processing such messages.

At implementation-time a service model is turned into a working solution based on an orchestration language, such as BPEL [1].

The contribution of this paper basically consists in defining strong connections between the specification phase and the design one and between the design phase and the implementation one.

Verifying the conformity of a service model to a collaboration model is a key issue of the first of the above-mentioned connections; such a verification is based on the relationships existing between services and collaborations (with respect to a given collaboration a service can act as a provider or as a requester for one or more instances) and will be discussed in two major cases, i.e. when the service provides a single collaboration instance or requires multiple instances.

As the conceptual gap between a service model and its BPEL implementation is relevant, the second of the above-mentioned connections is concerned with the automatic mapping of service models to BPEL processes, in line with model-based development [2].

This paper is organized as follows. Section 2 presents collaboration models and introduces the example of a selling collaboration, which will be used throughout this paper. Section 3 illustrates service models and discusses the assumptions the automatic mapping to BPEL processes rely on. Sections 4 and 5 describe how WSDL documents and BPEL processes are automatically generated. Section 6 discusses how service models can be validated with respect to a given collaboration model. Section 7 gives a short account of the bProgress environment, which is made up of a number tools developed during this research. A comparison with related work is the subject of section 8, while section 9 presents the conclusion.

2 Collaboration Models

A well-known example of collaboration is the purchasing of goods or services, whose description is as follows: the requester sends a request for quote (rfQ) to the provider, which may respond with a quote; if the requester accepts the quote, it will then send an order to the provider. That collaboration will be used throughout this paper and will be referred to as the selling collaboration, according to the provider perspective (the requester would call it a purchasing collaboration).

Basically a collaboration model in bProgress consists of messages placed within a control structure providing for sequential, alternative, repetitive and timeout-related paths. The model of the selling collaboration is shown in Fig. 1.

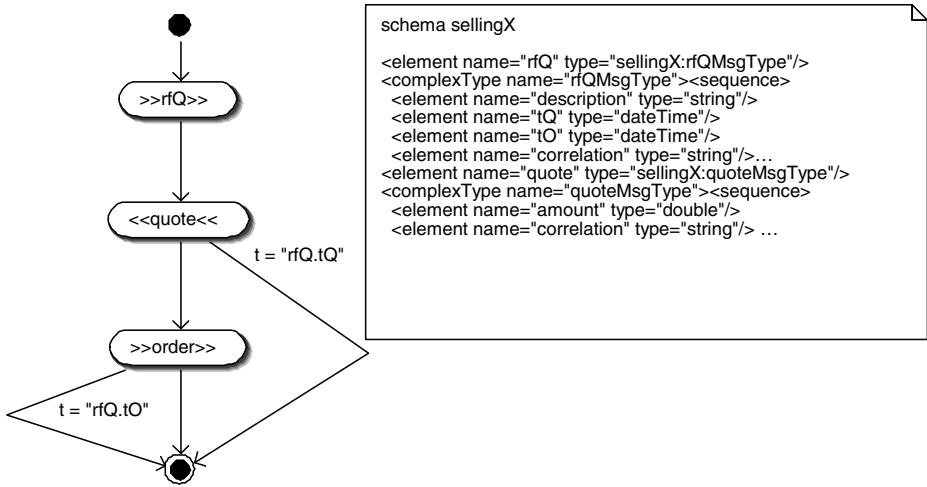


Fig. 1. The selling collaboration model (sellingC)

A message has a left-to-right direction (e.g. rfQ and order), or a right-to-left direction (quote), a left-to-right message being sent from the requester to the provider and a right-to-left one being sent from the provider to the requester.

Unlabelled links represent precedence constraints.

As a collaboration is assumed to be started by the requester, the first message is a left-to-right one, like rfQ, and is called the initial message. In some cases a collaboration could be started with two or more alternative messages, then the first element in the model would not be a left-to-right message but a branch leading to the various initial messages.

Each message (except the initial ones) must have a deadline. The meaning of a deadline is as follows: the receiver is bound to wait for a message until its deadline expires and no longer; it is useless for the sender to send a message if its deadline has expired. When a deadline expires, a timeout will occur; timeout links (i.e. those labelled with keyword “t”) establish the effects of timeouts. In the model shown in Fig.1 there are two deadlines, tQ and tO, and both are attributes of message rfQ: tQ is the time-limit for sending/receiving a quote, tO is the time-limit for sending/receiving an order.

The selling collaboration model is to be interpreted as follows. The provider can send a quote only after receiving an rfQ; the requester will wait for the quote until time-limit tQ and no longer, therefore if the quote is not received by that time, the collaboration will be ended. After receiving a quote, the requester can send an order; after sending the quote, the provider will wait for the order until time-limit tO and no longer, hence if the order is not received by that time, the collaboration will be ended. Other kinds of purchasing/selling collaborations can be found in [3].

Each message has a name and a type. By convention the name of the type is that of the message followed by suffix “MsgType”: if so, the type is not shown in the model (therefore “>>rfQ>>” is equivalent to “>>rfQ, rfQMsgType>>”).

Types are defined in an XML schema associated with the collaboration model. An excerpt from `sellingX`, i.e. the schema related to the selling collaboration, is shown in Fig.1. Correlation attributes are explained in the next section.

3 Service Models

A service model in `bProgress` is an abstract graphical representation of a service that is to be automatically mapped to a BPEL process. A service model is not a graphical representation of a BPEL process as it is based on higher-level abstractions. However in order to be automatically translated into BPEL processes, `bProgress` service models adopt the same conventions as BPEL as regards the generation of process instances and the correlation of messages to process instances.

In general a BPEL process begins by receiving the initial message (the case of multiple initial messages is left apart). When the BPEL run-time system receives the initial message for a given process, it generates an instance of that process and delivers it the message. As to the collaboration started by the initial message, the newly generated process instance is said to be its provider, while the process instance that sent that message is said to be its requester. Likewise, by extension, for a process and a service model.

At the very heart of a collaboration there is the possibility for the same pair of process instances to exchange messages over a period of time. In fact a message is directed to a process (more precisely to its endpoint) and, if it is not an initial message, it is also assumed to contain the information about the process instance it is to be delivered to. BPEL correlates a message to a process instance on the basis of the value of one or more attributes (called properties) of the message. This solution, relying on the payload of messages, is completely transparent, i.e. free from implementation details.

Correlations are automatically inserted by the `bProgress` code generator, provided that each message includes an attribute named `correlation` (this is the reason why messages types in Fig.1 include that attribute), whose value is able to identify the proper process instance on both sides of the collaboration. That value is set by the requester. As far as the selling collaboration is concerned, the correlation value is made up of the URL of the requester endpoint and of the id of the `rfQ` (i.e. the primary key of the corresponding data object managed by the information system on the requester side).

When the requester sends an `rfQ`, the BPEL sending activity reads the correlation value from the output message and associates a tag having that value with the sending process instance. Such tags are called correlation sets in BPEL and have to be declared in the process, as shown in the next section. When the `rfQ` is received, the BPEL run-time system generates an instance of the receiving process, reads the correlation value from the input message and associates a tag having that value with the newly generated instance. The quote is sent back to the requester with the same correlation value as the `rfQ`, hence it will be delivered to the requester process instance that previously sent the `rfQ`. When an order is sent, since it has the same correlation set as the quote and the `rfQ`, it will be delivered to the provider process instance that previously sent the winning quote.

An example of a selling service model is shown in Fig. 2. It provides a single selling collaboration, as shown in the “provides” clause. A new instance is started when an rfQ is received, hence the state of the process instance coincides with the state of the collaboration.

The selling service model is basically an extension of the collaboration model, in which left-to-right messages have been turned into receiving activities and right-to-left messages have been turned into sending activities, and some links have been expanded into processing activities.

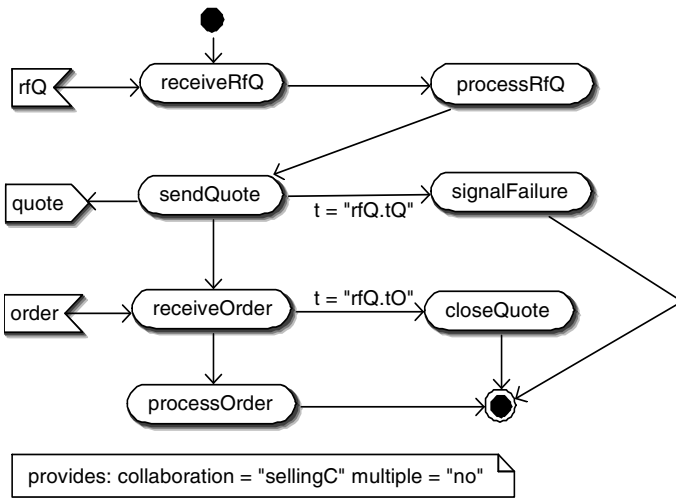


Fig. 2. The selling service model (sellingS)

In bProgress a service model is an extended UML activity diagram, including communication (i.e. sending and receiving) activities and processing ones. A communication activity is completely defined by the model, while a processing activity is to be supplemented with a BPEL content (as a textual addendum). The tasks of the processing activities are as follows: processRfQ writes the rfQ in the information system (on the provider side) and makes it generate the quote to be sent, processOrder writes the order in the information system, closeQuote and signalFailure report to the information system that the quote has been unsuccessful or has not been prepared in due time, respectively.

An example of a purchasing service model is shown in Fig. 3. Its task is basically to select the best supplier for a given request for quote on behalf of the information system (on the requester side). In fact it begins by receiving message purchasingInfo (from the information system), which contains the request for quote along with a list of suppliers to be involved. Then it sends the request for quote to each supplier, receives all the quotes (until all the quotes expected have been received or the time-limit established in attribute purchasingInfo.t has been reached), selects the best one, sends an order to the winning supplier and finally reports the result back to the

Types are defined in an XML schema associated with the collaboration model. An excerpt from *sellingX*, i.e. the schema related to the selling collaboration, is shown in Fig.1. Correlation attributes are explained in the next section.

3 Service Models

A service model in *bProgress* is an abstract graphical representation of a service that is to be automatically mapped to a BPEL process. A service model is not a graphical representation of a BPEL process as it is based on higher-level abstractions. However in order to be automatically translated into BPEL processes, *bProgress* service models adopt the same conventions as BPEL as regards the generation of process instances and the correlation of messages to process instances.

In general a BPEL process begins by receiving the initial message (the case of multiple initial messages is left apart). When the BPEL run-time system receives the initial message for a given process, it generates an instance of that process and delivers it the message. As to the collaboration started by the initial message, the newly generated process instance is said to be its provider, while the process instance that sent that message is said to be its requester. Likewise, by extension, for a process and a service model.

At the very heart of a collaboration there is the possibility for the same pair of process instances to exchange messages over a period of time. In fact a message is directed to a process (more precisely to its endpoint) and, if it is not an initial message, it is also assumed to contain the information about the process instance it is to be delivered to. BPEL correlates a message to a process instance on the basis of the value of one or more attributes (called properties) of the message. This solution, relying on the payload of messages, is completely transparent, i.e. free from implementation details.

Correlations are automatically inserted by the *bProgress* code generator, provided that each message includes an attribute named *correlation* (this is the reason why messages types in Fig.1 include that attribute), whose value is able to identify the proper process instance on both sides of the collaboration. That value is set by the requester. As far as the *selling* collaboration is concerned, the correlation value is made up of the URL of the requester endpoint and of the id of the *rfQ* (i.e. the primary key of the corresponding data object managed by the information system on the requester side).

When the requester sends an *rfQ*, the BPEL sending activity reads the correlation value from the output message and associates a tag having that value with the sending process instance. Such tags are called correlation sets in BPEL and have to be declared in the process, as shown in the next section. When the *rfQ* is received, the BPEL run-time system generates an instance of the receiving process, reads the correlation value from the input message and associates a tag having that value with the newly generated instance. The quote is sent back to the requester with the same correlation value as the *rfQ*, hence it will be delivered to the requester process instance that previously sent the *rfQ*. When an order is sent, since it has the same correlation set as the quote and the *rfQ*, it will be delivered to the provider process instance that previously sent the winning quote.

if there is one, it sets attribute `purchasingResult.quoteSelected` to “true” and attribute `purchasingResult.winner` to the endpoint of the winner (after retrieving it from list `purchasingInfo.providers`) and also prepares the order to be sent.

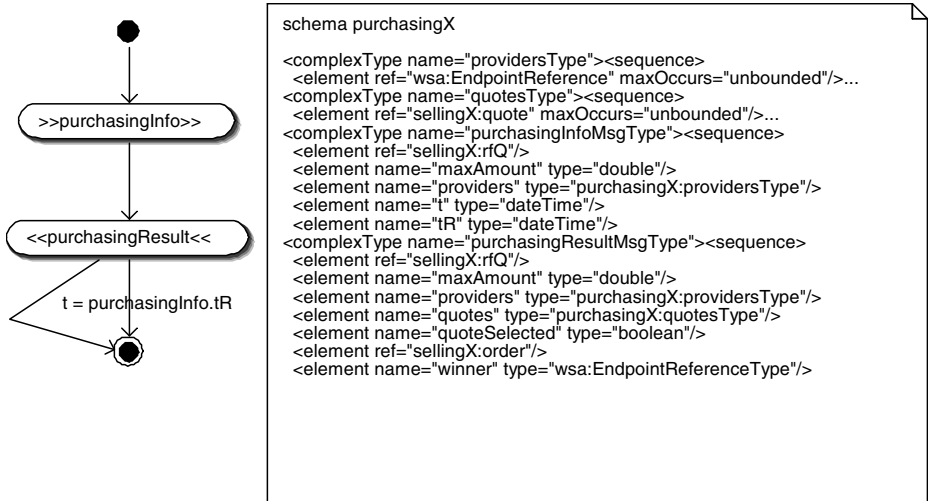


Fig. 4. The purchasing collaboration model (purchasingC)

4 Generating a WSDL Document from a Collaboration Model

This section explains how the bProgress code generator maps a collaboration model, such as `sellingC`, to a WSDL document.

A collaboration model implies the existence of two services, one on the provider side, the other on the requester side. The interfaces of such services are defined in the same WSDL document; the behavior of each service, instead, is defined in a distinct BPEL process, which relies on that WSDL document.

In general, a WSDL document can provide abstract information (messages, operations and portTypes) as well as deployment information (bindings and network addresses) for a number of web services. However, since a BPEL process is intended to be a reusable definition, which can be deployed in different scenarios, it is directly concerned only with the abstract information part of the WSDL documents it relies on.

The current version of the bProgress code generator assumes that all interactions are asynchronous, hence they correspond to WSDL one-way operations. Therefore an abstract web service turns out to correspond to a WSDL portType grouping a number of one-way operations.

An excerpt from the WSDL document generated from `sellingC` follows (“tns”, “sellingX”, “plnk” and “bpws” denote the current document, the schema shown in Fig. 1, the document defining partner link types and the BPEL schema, respectively).

```

<definitions name="sellingC"
  targetNamespace="http://www.polito.it/bProgress/sellingC" ...
<message name="rfQMsg"><part name="payload" element="sellingX:rfQ"/>
<portType name="providerPT">
  <operation name="rfQ"><input message="tns:rfQMsg"/>...
  <operation name="order"><input message="tns:orderMsg"/>...
<portType name="requesterPT">
  <operation name="quote"><input message="tns:quoteMsg"/>...
<plnk:partnerLinkType name="sellingCPLT">
  <plnk:role name="provider"><plnk:portType name="tns:providerPT"/>...
  <plnk:role name="requester"><plnk:portType name="tns:requesterPT"/>...
<bpws:property name="correlation" type="xsd:string"/>...
<bpws:propertyAlias propertyName="tns:correlationId" messageType=
"tns:rfQMsg" part="payload"
query="/sellingX:rfQ/sellingX:correlationId"/>

```

There are two portTypes called providerPT and requesterPT: the former groups all the input messages of the service model, the latter all the output messages.

BPEL requires the portTypes involved in a collaboration to be included in a partnerLinkType construct together with their corresponding role.

When generating a partnerLinkType, bProgress uses two roles, provider and requester, and assigns them to the portType grouping the input messages and to the one grouping the output messages, respectively.

5 Generating a BPEL Process from a Service Model

A BPEL process is basically a hierarchical structure of sending (invoke), receiving (receive) and processing (assign) activities. The structure of a process is determined by compound activities, such as sequence, switch, while, flow, and pick.

An excerpt from the BPEL selling process follows.

```

<process name="sellingP"
  xmlns:sellingC="http://www.polito.it/bProgress/sellingC" ...
<partnerLinks>
  <partnerLink name="sellingCPL" partnerLinkType="sellingC:sellingCPLT"
    myRole="provider" partnerRole="requester"/>...
<variables>
  <variable name="rfQ" messageType="sellingC:rfQMsg"/>
  <variable name="quote" messageType="sellingC:quoteMsg"/>
  <variable name="order" messageType="sellingC:orderMsg"/>
  <variable name="deadlineNotExpired" type="xsd:boolean"/>...
<correlationSets>
  <correlationSet name="sellingCCS" properties="sellingC:correlation"/>.
<sequence>
  <receive name="receiveRfQ" partnerLink="sellingCPL"
    portType="sellingC:providerPT" operation="rfQ" variable="rfQ"
    createInstance="yes">
    <correlations><correlation initiate="yes" set="sellingCCS"/>...
  <assign name="processRfQ"> ... prepares the quote
  <assign name="sendQuote_d"> ... sets inner variable deadlineNotExpired
    to true if the deadline of sendQuote has not expired
  <switch name="sendQuote_s">
    <case condition="bpws:getVariableData('deadlineNotExpired')">
      <invoke name="sendQuote" partnerLink="sellingCPL"
        portType="sellingC:requesterPT" operation="quote"

```

```

inputVariable="quote">
  <correlations>
    <correlation initiate="no" set="sellingCCS" pattern="out"/>...
  <otherwise><sequence><empty name="signalFailure"/><terminate/>...
</pick name="receiveOrder" createInstance="no">
  <onMessage partnerLink="sellingCPL" portType="sellingC:providerPT"
    operation="order" variable="order">
    <correlations><correlation initiate="no" set="sellingCCS"/>...
    <sequence><empty name="processOrder"/><terminate/>...
  <onAlarm until="bpws:getVariableData('rfQ','payload',
    '/sellingX:rfQ/sellingX:t0')">
    <sequence><empty name="closeQuote"/><terminate/>...

```

A BPEL process sends and receives messages only through channels which are called partnerLinks. PartnerLinks are declared at the beginning of the process. A partnerLink refers to a partnerLinkType (taken from one of the WSDL documents referred to by the process) and establishes the role (the value of attribute myRole) played by the process with respect to the partnerLinkType.

For each collaboration model provided or requested by the service model there is a partner link in the BPEL process. In the selling process there is only one partnerLink, sellingCPL (named after the collaboration it provides), hence the process can receive an rfQ, or send a quote, through that channel. When a message is received, it is copied into a variable; when it is sent, it is read from a variable. Variables in the process correspond to the messages in the service model.

The behavior of the selling process is basically a sequence of five major activities, as shown in the model.

The first activity, receiveRfQ, copies the initial message into variable rfQ and tags the instance with the value read from attribute correlation of the message received. Such tags are called correlation sets in BPEL and have to be declared in the process, e.g. sellingCCS in the code shown above. When the requester sends an rfQ, as will be shown later on, the sending activity reads the correlation value from the output message and initializes the correlation set associated with the sending process instance, using that value. A quote must not be sent if it is late. In order to comply with such a constraint, the BPEL code generator produces two activities: the first one, sendQuote_d, sets an inner variable, deadlineNotExpired, to “true”, if the deadline of the quote has not expired, to “false” otherwise; the second activity is a two-branch switch structure, the first branch being taken if deadlineNotExpired is “true”.

The collaboration will be closed, if the order is not received within a given time-limit. Therefore activity receiveOrder is mapped to a pick structure whose purpose is to wait for the order to arrive or for the corresponding timeout alarm to go off: when one of those triggers occurs, the associated activity is carried out and the pick completes. Since activities closeQuote, processOrder and signalFailure have been left undefined in the model, they are mapped to BPEL empty activities.

The purchasing process

For lack of space only the BPEL code corresponding to multiple-sending activity sendRfQs is illustrated in detail. Given its parameters, that activity is turned into a

loop that iterates over the list of providers contained in attribute `providers` of variable `purchasingInfo` and sends the `rfQ` to each provider. Providers are simply denoted by their endpoint references as shown in Fig. 4.

The same `rfQ` is sent to each provider through partner link `sellingCPL`, which is a variable partner link as its partner endpoint reference has to be set before each sending operation. In fact, if the WSDL documents contain no deployment information, as is the case of the WSDL documents generated by `bProgress`, partnerLinks lack the information needed to reach the intended web services. For this reason, before using a partnerLink to send an initial message (like `rfQ`), the BPEL process has to set the partner endpoint reference within the partnerLink. An endpoint reference is a structure (based on WS-Addressing [4]) including the network address of the web service to be called along with other deployment information.

An excerpt from the code of `sendRfQs` follows.

```
<scope name="sendRfQs_s"><sequence>
  <assign>
    <copy><from expression="ora:countNodes('purchasingInfo', 'payload',
      '/purchasingX:purchasingInfo/purchasingX:providers/
      wsa:EndpointReference')"/><to variable="sendRfQs_c"/>...
    <copy><from expression="0"/><to variable="i"/>...
    <while name="sendRfQs_w" condition="bpws:getVariableData('i')
      &lt; bpws:getVariableData('sendRfQs_c')"><sequence>
      <assign>
        <copy><from variable="purchasingInfo" part="payload"
          query="/purchasingX:purchasingInfo/purchasingX:
            providers/wsa:EndpointReference[bpws:getVariableData('i')+1]"/>
          <to partnerLink="sellingCPL"/>...
        <copy><from expression="bpws:getVariableData('i')+1"/>
          <to variable="i"/></copy>...
      <invoke name="sendRfQs" partnerLink="sellingCPL"
        inputVariable="rfQ" portType="sellingC:providerPT"
        operation="rfQ">
      <correlations>
        <correlation initiate="yes" set="sellingCCS" pattern="out"/>...
```

The outer scope, `sendRfQs_s`, encompasses two sequential BPEL activities. The first one (the assign activity) determines the number of iterations (which corresponds to the number of endpoint references contained in attribute `purchasingInfo.providers`) and sets inner variable `sendRfQs_c` to that value; then it initializes the index of the loop (i.e. inner variable `i`) to 0. The second activity is a “while” whose body is performed as long as `i < sendRfQs_c`. At each iteration the endpoint reference of the current provider is copied into partner link `sellingCPL` and the index is incremented, then the `rfQ` is sent to the current provider.

The need for multiple-sending activities as well as multiple-receiving ones in BPEL has also been pointed out in [5], where the introduction of two new primitives, `broadcast` and `collect`, is proposed. The multiple-sending and multiple-receiving activities presented in this section can be used to implement well-known patterns, such as those dealing with multiple instances [6] and those related to asynchronous communication (`publish/subscribe`, `broadcast`) [7].

6 Validating Service Models

Validating a service model means making sure that it conforms with the collaborations it provides or requests.

If a service model is concerned (as a provider or a requester) with a single instance of a given collaboration, then it can be viewed as an extension of the model of that collaboration, just as the selling service shown in Fig. 2 is an extension of the selling collaboration shown in Fig. 1. In this case it is simply a matter of proving that the service model can be transformed, by means of suitable reduction rules, into a model which is equivalent to the collaboration one.

In fact in the selling service shown in Fig. 2 activity processRfQ can be viewed as a refinement of the precedence link connecting receiveRfQ to sendQuote, so it can be replaced with a simple link from receiveRfQ to sendQuote; likewise for activity processOrder. Moreover activity closeQuote can be viewed as a refinement of the timeout link connecting receiveOrder to the final state and hence it can be replaced with a simple link; likewise for activity signalFailure. At this point the resulting model turns out to be equivalent to the selling collaboration model. Such a reduction rule is the inverse of the second inheritance-preserving transformation rule (i.e. rule PJS) presented in [8].

Validating a service model that requests multiple instances of a given collaboration, such as the purchasing service shown in Fig. 3, in general cannot take place at design-time, since the states of the various collaboration instances can differ during the execution of the service. In fact if activity selectQuote worked badly, the order could be mistakenly sent to a supplier that did not provide any quote. In this case run-time checks are needed in order to prevent a service from sending or receiving a message in wrong order (or not complying with timing constraints). Such checks can be performed only if the states of ongoing collaborations are available and can be updated when needed.

In a previous paper [9] a solution based on collaboration objects was presented: their purpose is to separate validation logic from process logic as well as to provide high-level sending and receiving operations to workflow processes.

This paper presents a different solution in which for each collaboration a descriptor is maintained in the requesting BPEL process: the reason is to take advantage of the persistency features provided by the BPEL run-time system. However the handling of such descriptors (i.e. checking and updating actions) takes place by means of external stateless Java objects.

A collaboration descriptor basically contains the endpoint reference of the partner service, the state of the collaboration (i.e. the name of the next message to be sent or received) and the current deadline.

The bProgress code generator adds run-time checks, in terms of Java-based activities, to each sending or receiving operation. Therefore a BPEL process performs a pre-sending check before each sending activity in order to make sure that the message name is included in the state of the descriptor of the corresponding collaboration instance and the deadline has not expired. If that check succeeds, the state and the current deadline are updated according to the collaboration model; otherwise an exception is thrown. The service model must include fault paths leading to the proper exception-handling activities. Moreover a BPEL process performs a

post-receiving check after each receiving activity (except for initial messages); since the partner link involved contains the endpoint reference of the partner, the corresponding collaboration descriptor can be retrieved and then suitable checks and updates can be performed.

7 The bProgress Environment

The bProgress environment is a set of tools based on the Eclipse platform, intended to support new approaches for business processes and services.

Collaboration models and service ones are produced with a UML 2.0 visual tool. We have defined a set of profiles (i.e. <<receive>>, <<send>> and <<timeout>>) so as to specialize the graphical elements. The models are first exported to XMI 1.1 documents and then transformed into a simpler XML representation by means of an XSL transformation.

The bProgress code generator produces WSDL documents and BPEL processes from such internal XML representations, according to the rules presented in section 5; if required, it can add run-time checks.

The BPEL processes presented in this paper have been tested using Oracle BPEL Process Manager 10.1.2 Beta-3; the tests have been carried out on one purchasing process requesting collaborations of a number of different selling processes (providing the same selling collaboration).

8 Comparison with Related Work

Orchestration languages, such as BPEL, are a good implementation platform, however more abstract representations, i.e. service models, are needed to help designers focus on process logic and get rid of technical details. On the other hand this is an area suitable for applying model-based development [2] with the purpose of automatically deriving orchestration processes from service models.

Mapping models to BPEL processes has been addressed in several papers from different starting points: extended state machines are used in [10], while UML activity diagrams are adopted in [11]; however their focus is on control aspects rather than on collaboration issues.

Web service composition is a fundamental issue in service-oriented computing: it basically refers to the possibility of building a new service on top of some existing services. A survey of existing proposals is presented in [12] together with a comparative analysis with respect to some key requirements including composition correctness. As to the correctness it is shown in [13] that a composite service, made up of component services (modelled as Petri nets having one input place and one output place) and of standard composition operators, can be checked for deadlock and incorrect termination.

This paper addresses web service composition in terms of collaboration requests: the purchasing service, shown in Fig. 3, is in effect a composition of selling collaborations. The reason is to offer a broader perspective: in fact a component service can be involved in several interactions with the composite service (as is the

case of the selling service with respect to the purchasing service), not only in an initial request and in a final reply; moreover it may happen that a multiple composition of similar component services is needed (such as the multiple selling collaborations requested by the purchasing service). In such cases, it is hard to prove composition correctness; therefore section 6 has presented an approach based on run-time checks, which are meant to prevent a service from sending or receiving a message in wrong order (or not complying with timing constraints).

Providing a standard solution to web service composition entails a number of practical issues, such as the handling of endpoint references and the correlation of messages to process instances, to be taken into account at the same time. This paper has presented multiple-sending and multiple-receiving activities, which work under a number of assumptions: each message has a correlation attribute, there is a variable containing the endpoint references of the services to be involved in a multiple-sending activity, there is a variable where the messages received with a multiple-receiving activity can be stored. The combination of all such aspects can be thought of as a realization of well-known patterns, such as those dealing with multiple instances [6] and those related to asynchronous communication (publish/subscribe, broadcast) [7].

There are many similarities between collaboration models, as presented in this paper, and choreography description languages, such as WS-CDL [14]. The purpose of our research is different as it mainly consists in providing a model-based approach to the development of a collaboration (or choreography). For this reason it is essential to mediate between the capabilities of current technology (BPEL in this case) and conceptual requirements, such as the need for multiple-sending activities (and multiple-receiving ones).

9 Conclusion

This paper has shown that the notions of collaboration and service are strongly related, a collaboration being an abstract representation of the interactions between two services and a service being a process that can be involved in one or more collaborations as a provider or a requester.

According to the principles of model-based development this paper has illustrated how service models can be automatically mapped to BPEL orchestration processes, and has discussed the major issues to be taken into account.

Current work proceeds in several directions, including: the extension from binary collaborations to multi-party ones, the introduction of transactional support [15], and the integration with other notations, such as BPMN [16].

References

1. Andrews, T. et al.: Business Process Execution Language for Web Services Version 1.1. BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems (2003). <http://www.ibm.com/developerworks/library/specification/ws-bpel/>
2. Mellor, S., Clark, A. N., Futagami, T.: Special Issue on Model-Driven Development. IEEE Software, Vol. 20 (5). IEEE Computer Society (2003)

3. Dijk, A. V.: Contracting workflows and protocol patterns. In: van der Aalst, W.M.P., Hofstede, A.H.M. ter, Weske, M. (eds.): BPM 2003. Lecture Notes in Computer Science, Vol. 2678. Springer (2003) 152-167
4. Bosworth, A. et al.: Web Services Addressing (WS-Addressing). BEA, IBM, Microsoft (2003). <http://msdn.microsoft.com/ws/2003/03/ws-addressing/>
5. Mendling, J., Strembeck, M., Neumann, G.: Extending BPEL4WS for multiple instantiation. In: Dadam, P., Reichert, M. (eds.): INFORMATIK 2004. Lecture Notes in Informatics (LNI), Vol. 51. German Computer Science Association (2004) 524-529
6. van der Aalst, W. M. P., Hofstede, A. H. M. ter, Kiepuszewski, B., Barros, A. P.: Workflow patterns. Distributed and Parallel Databases, Vol. 14(1). Springer (2003) 5-51
7. Wohed, P., van der Aalst, W. M. P., Dumas, M., Hofstede, A. H. M. ter: Analysis of web service composition languages: the case of BPEL4WS. In: Song, I.Y., Liddle, S. W., Ling, T. W., Scheuermann, P. (eds.): 22nd Int. Conf. ER 2003. Lecture Notes in Computer Science, Vol. 2813. Springer (2003) 200-215
8. van der Aalst, W.M.P., Weske, M.: The P2P approach to interorganizational workflows. In: Dittrich, K. R., Geppert, A., Norrie, M.C. (eds.): 13th Int. Conf. CAiSE 2001. Lecture Notes in Computer Science, Vol. 2068. Springer (2001) 140-156
9. Bruno, G.: Modeling and using business collaborations. In: Pre-proceedings of the 1st Int. Conf. on Interoperability of enterprise software and applications, Geneva (2005) 114-125
10. Baina, K., Benatallah, B., Casati, F., Toumani, F.: Model-driven web service development. In: Persson, A., Stirna, J. (eds): 16th Int. Conf. CAiSE 2004. Lecture Notes in Computer Science, Vol. 3084. Springer (2004) 290-306
11. Mantell, K.: From UML to BPEL, IBM (2003). <http://www-128.ibm.com/developerworks/webservices/library/ws-uml2bpel/>
12. Milanovic, N., Malek, M.: Current solutions for web service composition. IEEE Internet Computing, Vol. 8 (6). IEEE Computer Society (2004) 51-59
13. Hamadi, R., Benatallah, B.: A Petri-net-based model for web service composition. In: Proceedings of the 14th Australasian Database Conference. Australian Computer Society (2003) 191-200
14. Kavantzias, N. et al. (eds): Web Services Choreography Description Language Version 1.0. W3C (2004). <http://www.w3.org/TR/ws-cdl-10/>
15. Dalal, S.; Temel, S.; Little, M.; Potts, M.; Webber, J.: Coordinating business transactions on the web. IEEE Internet Computing, Vol. 7 (1). IEEE Computer Society (2003) 30-39
16. White, S. A.: Introduction to BPMN, IBM (2004). <http://www.bpmn.org>