Analysing Web Service Composition with PEPA

Bryce Mitchell* Jane Hillston*

June 4, 2004

1 Introduction

Web services are an emerging paradigm aiming to offer the interoperability afforded by web applications combined with rich client interaction. Web service composition allows a distributed application to be constructed from a number of previously published web services. This component-based style of implementation offers many benefits to the developer, but the reliance of third party hardware and software only exacerbates the unpredictability of application performance. In this paper we present a first step towards addressing this problem, demonstrating an automatic mechanism for developing a performance model based on a web service composition.

The web service composition language that we consider is the *Business Process Execution Language for Web Services* (*BPEL4WS* or simply *BPEL*) [3]. It is an XML-based language and is implemented as a high level specification. This specification provides a means for using a composition of web services to perform applications, normally the automation of a *business process* incorporating:

- the ordering constraints between operations provided by different web services;
- how data is shared between web services; and
- the different organisations or *partners* involved in the process.

BPEL is described in more detail in the following section.

The performance models we generate are written in the stochastic process algebra, PEPA [2]. PEPA was chosen because it was anticipated that its compositional structure would be a good match to the compositional specification in a BPEL description.

The rest of the paper is structured as follows. In Section 2 we give a brief introduction to web services in general and BPEL in particular. The mapping between BPEL and PEPA is presented in detail in Section 3, including some information

*School of Informatics, University of Edinburgh, Edinburgh EH9 3JZ

about the implementation. This is illustrated in Section 4 by an example based on a loan approval application. We conclude in Section 5 with a brief summary of what has been achieved and an overview of the outstanding issues.

2 Web Service Composition

Web services are applications that maintain a modular structure and use the Internet to communicate using XML messages. Web service applications are built using a stack of specifications to define, invoke and find web services. Each protocol is an open standard developed by industry and overseen by the W3C [4]. The diagram in Figure 1 shows the stack, and each layer is briefly described below.

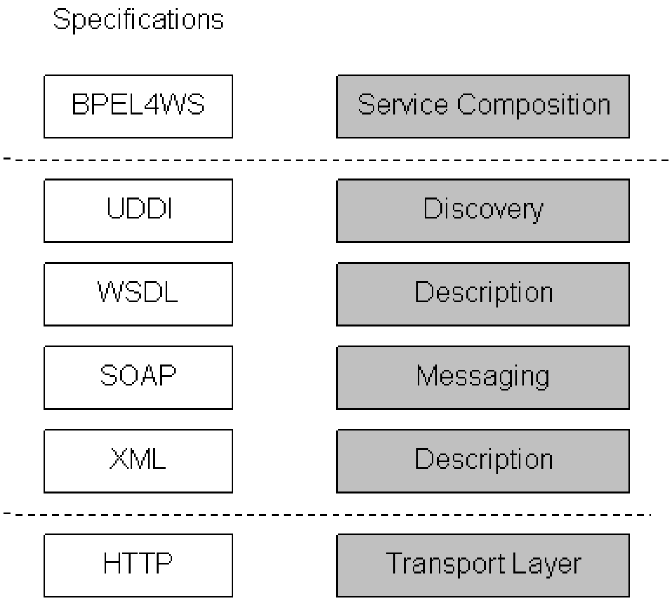

Figure 1: The Web Service Stack

XML: XML or *Extensible Markup Language* is a markup language similar to HTML. However, XML focuses on describing data, rather than describing formatting and layout as HTML does. The extensible nature of XML comes from the lack of a fixed set of tags — an XML document can contain any tag or element the author wishes.

SOAP: SOAP is the specification used for communicating between web services. SOAP is a very lightweight protocol and is generally implemented on top of the HTTP message layer; however SOAP is a protocol independent specification.

WSDL: The WSDL or *Web Service Description Language* describes a web service in the sense of providing details of the operations offered by that web service. These details include type information about the parameters of the operation and how the service likes to interact i.e. what protocols may be used to invoke it.

UDDI: The UDDI is similar to a directory service for looking up a required service. UDDI is an industry-wide project to standardise the discovery of web services.

BPEL4WS: BPEL provides an extension to the web services architecture to allow both the abstract description of these processes and a method for executing these processes. In this work we focus on the abstract description properties of BPEL to allow us to create a model of the interaction of web services needed to provide a process, and predict key metrics such as throughput.

2.1 BPEL Syntax

This section provides a brief overview of the syntax of a BPEL document. The definition of any service consists of four key sections:

Process: This is the root element of the document. It provides information such as the name of the process, namespace prefixes and also allows for a process to be declared as abstract.

Partner Links/Partner Link Types: These define the relationship between the different partners that interact in the process. Typically a partner will be seen as both a consumer and provider of a service, for example a buyer of some goods and a seller of others. Within partner link types *roles* specify the `portType` provided by each service for use in the process. Each partner may have more than one role, e.g. one for buying and one for selling.

Variables: These are used to store intermediate values that relate to the state or history of the process.

Fault Handlers: These provide contingency activities in the event of a failure. Failures may occur due to internal or invoked events. This aspect of the specification is not addressed in the current work.

A BPEL process allows operations to be executed sequentially or concurrently as appropriate and also supports familiar programming structures such as `while` loops and `if then else` statements.

The basic structure of the process follows the definition provided by the above sections and defined by outer structural elements, `<sequence>`, `<flow>` and `<pick>`. Within the structure a number of activities can be performed, these can include other structure activities in a hierarchical fashion. For example, it is common to find a <flow> activity within a <sequence>. The full set of valid BPEL activities is:

<receive>	A receive element is used to wait for a request.
<reply>	A reply activity is used to send a response to a request received through a <receive>.
<invoke>	The basic activity of invoking an operation provided by a service.
<assign>	This activity is used to assign data from one variable to another.
<throw>	Allows a process to throw an exception.
<terminate>	Used to finish the entire process.
<wait>	This activity causes a delay for a specified period of time.
<empty>	An activity that does nothing, for example to suppress an error.
<**sequence**>	Activities in a sequence must be executed *sequentially* in the order specified in the document.
<switch>	Allows for conditional control structures.
<while>	This activity provides the typical `while` loop functionality.
<**pick**>	A number of activities are specified in a pick and they are executed *concurrently* with execution continuing as soon as the **first** of th ese activities returns or a specified *timeout* has elapsed in which case the activity corresponding to the timeout is executed and the process continues.
<**flow**>	Activities within a flow activity may be executed *concurrently* in the sense that there is no causal ordering between them. Note that activities within flows may be sequence activities.
<scope>	This element is optional, but can be used to define the scope of a group of activities.
<compensate>	This activity provides for the cancellation of activities if a web service is not available.

Of these, our current work focuses on <sequence>, <flow> and <pick>. These three elements define the order in which operations are executed within the process.

3 Automatically generating PEPA from BPEL

In this section we present the mapping from BPEL to PEPA which is at the core of the tool, and discuss some issues relating to its implementation.

3.1 PEPA Mapping

There are several key elements to our mapping from BPEL to PEPA which can be summarised as follows:

- We create a separate PEPA representation, as a component, for each of the involved partners.

- Each BPEL atomic activity is represented as a PEPA activity. For clarity the name of the BPEL activity is used as the type for the corresponding PEPA activity.

- Furthermore we create an additional PEPA component which represents the BPEL process itself and serves to coordinate the behaviours of the other components. Data and state handling activities are attributed to this component. For example, an `<assign>` activity allocates data between the local variables in the process. In the XML representation such activities do not have a name attribute; therefore such an activity is mapped to a PEPA activity of type "`unnamedElement`" followed by a unique number.

- We assume that each partner runs on one single-threaded processor, meaning that it has bounded capacity to carry out computation. Thus when a flow of activities are all assigned to a single partner we assume that they are carried out in an arbitrary sequential order, reflecting this bounded capacity and the lack of causality constraints.

- The interactions within the process are captured in a system equation defining the cooperations between the PEPA components.

When the BPEL structuring activities are encountered they are mapped to the PEPA combinators as follows. A `<sequence>` activity is mapped to the PEPA prefix (.) in the sense that all activities that are within `<sequence>` tags are mapped to a chain of PEPA activities separated by the prefix combinator.

An example of this mapping is shown in Figure 2 where a customer and airline interact to book a flight.

When creating a PEPA representation for activities within `<flow>` elements, based on the assumptions outlined above, we also map to a sequence of PEPA activities linked by the prefix operator.

The final structural element, `<pick>`, is mapped to the PEPA choice (+) operator. The race condition in operation in a BPEL `<pick>` activity maps well the race condition governing the choice operator in PEPA. When creating the mapping we include all activities within the `<pick>` element and also the timeout activity. The continuous nature of the probability distributions in PEPA ensures that two actions in the choice cannot occur simultaneously. This maintains the property that a BPEL `<pick>` activity completes as soon as one of the nested activities or the timeout has completed. Figure 3 continues the travel example to

```
<sequence>
  <receive name="sendItinerary"
    partnerLink="customer" portType="itineraryPT"
    operation="sendItinerary" variable="itinerary"/>
  <invoke name="requestTickets"
    partnerLink="airline" portType="ticketOrderPT"
    operation="requestTickets" inputVariable="itinerary"/>
</sequence>
```

maps to...

$$P \stackrel{\mathit{def}}{=} (sendItinerary, r).(requestTickets, r').P$$

Figure 2: PEPA Mapping of <sequence> element

illustrate the pick mapping. The example shows a web service attempting to book a room at one of two hotels (Hilton or Marriott). The booking is made with the first hotel to return a price, or if no price is received by either within an hour just the airline tickets are sent to the customer.

Further, the provision for nesting of elements within XML allows the possibility of nesting of activities within BPEL. This nesting can result in more complicated structures, such as flows, within picks, within sequences.

To control the interaction between activities we create a PEPA component to represent the entire process. The component is built up by applying the above mappings for each of the structural activities. This component then enforces the scheduling of activities.

Finally we represent the interaction between the process and the partners by using the PEPA co-operation operator ($\bowtie$). This defines the interaction between the partners and the component representing the scheduling of the activities.

3.2 Implementation

The JAXB framework is used for accessing the data held within the BPEL document. To use the framework a set of classes must be created to represent the data. These classes are generated automatically using the `xjc` compiler, provided with the JAXB framework, from the BPEL Schema. Once the classes have been generated it is then possible to marshall documents between XML and Java.

Once a process object is available it is possible to access the other elements in the document in order to construct the internal representation that is used to build the PEPA model.

The information about the partners in the process is extracted first. We extract in this order because we need to know what partners are available in order to be able to match activities to the correct partners later.

```
<pick>
    <onMessage partnerLink="hilton" portType="roomPT"
      operation="requestRoom" inputVariable="itinerary">
      <invoke name="bookHilton"
        partnerLink="hilton" portType="roomPT"
        operation="reserve" inputVariable="itinerary" />
    </onMessage>
    <onMessage partnerLink="marriott" portType="roomPT"
      operation="getRoom" inputVariable="itinerary">
      <invoke name="bookMarriot"
        partnerLink="hilton" portType="roomPT"
        operation="reserveRoom" inputVariable="itinerary" />
    </onMessage>
    <onAlarm (until="1H")>
      <receive  name="justSendTickets"
        partnerLink="customer" portType="itineraryPT"
        operation="sendTickets" variable="tickets"/>
    </onAlarm>
  </pick>
```

maps to...

$$P \stackrel{def}{=} (bookHilton, r).P + (bookMarriott, r').P + (justSendTickets, r'').P$$

Figure 3: PEPA Mapping of <`pick`> element

A substantial part of the implementation deals with the activity information. It is currently assumed that all processes begin with a sequence — although this not a requirement imposed by the formal specification or schema. A distinct method deals with each of the three structure elements (<`sequence`>, <`flow`> and <`pick`>). Each of these methods uses the list of objects attached to its element. For each element in the list we use the `instanceof` keyword to determine the type of activity. If the activity is not an <`invoke`>, <`receive`>, <`reply`>, <`flow`>, <`sequence`> or <`pick`> we map it to the special *Bpel* partner used to represent internal events; this applies to activities such as <`assign`>, <`copy`> etc.

Once we have determined the activity type we extract useful information such as the name of the activity, the partner needed to execute the activity, the portType and operation. The portType and operation data is required for looking up the operation in the WSDL at a later stage for validation purposes.

The information for each individual activity is stored in a *PepaActivity* object. Each activity is then added to a List in the object representing the current structure element, e.g. a *SequenceActivity* or *FlowActivity* object. Objects that represent

```
<partnerLinks>
    <partnerLink name="customer"
          partnerLinkType="lns:loanApprovalLinkType"
          myRole="approver"/>
    <partnerLink name="approver"
          partnerLinkType="lns:loanApprovalLinkType"
          partnerRole="approver"/>
    <partnerLink name="assessor"
          partnerLinkType="lns:riskAssessmentLinkType"
          partnerRole="assessor" />
</partnerLinks>
```

Figure 4: Partners in the Loan Approval Process[1]

structural elements can include other objects, also representing structural elements. This allows for the nesting of activities. These objects are used to generate the PEPA syntax when the separate *PepaExtractor* class is creating the model in a format suitable for the PEPA Workbench. The separation of the generation of the PEPA, from the actual syntax implementation allows for changes to be made to either independently of one another.

3.3 Annotated WSDL

The BPEL specification gives us the structure of a PEPA model but gives us no information of suitable rates for the activities represented. Moreover, apart from the internal activites attributed to the *BPEL* component, this does not seem to be the natural place for such information to reside. Nevertheless such information is needed if the model is to be satisfactorily parameterised.

Some recent efforts have linked “quality of service” (QoS) information with WSDL, so we enriched the XML schema for WSDL with an optional latency estimate for each offered service.

Once the internal representation of the PEPA model is constructed, the associated WSDL definition is accessed for each used web service. This allows the activities in the component corresponding to the partner to be validated against the operations offered according to the WSDL, and to extract the latency information which is then used to set the rate of the corresponding activity. The rate is calculated as $1/latency$.

This value does not take into account any communication delay that a request for the service may encounter. The communication delay is not taken into account because a WSDL only describes the service and should be independent of the platform and network the service is operating on.

4 Example

In this section we present a small case study illustrating the use of the tool to create and solve a model for an example BPEL process.

```
<sequence>
 <receive name="receive1" partnerLink="customer"
   portType="apns:loanApprovalPT" operation="approve"
            variable="request" createInstance="yes" />
 <invoke name="invokeAssessor"
   partnerLink="assessor" portType="asns:riskAssessmentPT"
   operation="check" inputVariable="request"
   outputVariable="riskAssessment" />
 <assign name="assign">
   <copy>
     <from expression="'yes'" />
     <to variable="approvalInfo" part="accept" />
   </copy>
 </assign>
  <invoke name="invokeapprover" partnerLink="approver"
   portType="apns:loanApprovalPT" operation="approve"
   inputVariable="request" outputVariable="approvalInfo" />
 <reply name="reply" partnerLink="customer"
       portType="apns:loanApprovalPT" operation="approve"
                               variable="approvalInfo" />
</sequence>
```

Figure 5: Activities in the Loan Approval Process[1]

The example is based on a tutorial on BPEL by IBM[1] that describes a process for handling a loan application submitted by a customer. In this process we have the customer and the two web services provided by the financial institution: *loan assessor* and *loan approver*. In BPEL each of these are represented as a *partner* in the process and the relevant extract from the BPEL document is shown in Figure 4

Within the model, a PEPA component is created to represent each partner, and the activities the partner is involved in. Figure 5 shows the BPEL extract that defines the activities within the process. A brief explanation of the process is provided below:

- The process waits to receive a request.
- The process invokes a web service to perform a risk assessment based on the application details
- The result of the assessment is assigned to an internal variable within the process

- A further web service is invoked to decide if the application should be approved
- The result is sent back to the customer

For this example the WSDL for the risk assessment web service has been annotated to reflect that the service, on average, takes 4 seconds to complete the assessment. Figure 6 shows an extract of the annotated WSDL.

```
<portType name="riskAssessmentPT">
  <operation name="check" latency="4">
    <input message="loandef:creditInformationMessage"/>
    <output message="tns:riskAssessmentMessage"/>
    <fault name="loanProcessFault"
           message="loandef:loanRequestErrorMessage"/>
  </operation>
</portType>
```

Figure 6: Annotated WSDL for Risk Assessment Web Service

When the BPEL document is loaded into the tool the source document is accepted by the JAXB framework, and a model successfully generated. The model created is shown in Figure 7.

```
/*
 * This file is autogenerated from a BPEL4WS document
 * by s0094815 using bpel2pepa
 * on Mon May 24 22:24:05 BST 2004
 */

BPEL = (assign,1.0).BPEL;
CUSTOMER = (receive1,1.0).CUSTOMER+(reply,1.0).CUSTOMER;
APPROVER = (invokeapprover,1.0).APPROVER;
ASSESSOR = (invokeAssessor,0.25).ASSESSOR;

P = (receive1,1.0).(invokeAssessor,0.25).(assign,1.0)
  .(invokeapprover,1.0).(reply,1.0).P;

P<receive1,invokeAssessor,assign,invokeapprover,reply>
 (BPEL||CUSTOMER||APPROVER||ASSESSOR)
```

Figure 7: Model generated for the Loan Approval Process

By comparing the structure of the process and the partners shown (Figures 5 and 4 respectively) and the output shown in Figure 7 we can confirm the behaviour of the program is as expected. For each partner a PEPA component has been created showing the activities the partner is involved in. In the case of the *Customer*, *receive* and *reply* activities have been associated. The assignment of the loan approval information to an internal variable has been correctly mapped to the *Bpel* component for representing the internal activities.

Figure 7 also shows the rate for the *invokeAssessor* activity reflects the value placed within the operation attribute in the annotated WSDL.

All of the activities in this process are within `<sequence>` elements, and thus mapped to the PEPA prefix operator(.). The PEPA representation of the process, (denoted as P) shows this mapping has been successfully applied.

```
1  0.12500000000000022
2  0.5000000000000007
3  0.12499999999999997
4  0.12499999999999993
5  0.12499999999999997
```

Figure 8: Solution for model generated for the Loan Approval Process

Once the model has been created the next step is to solve the model, this step also ensures the PEPA syntax generated is valid. The PEPA workbench is successfully called from within the tool and Figure 8 shows the results that are calculated. From these results we can calculate measures such as the throughput of the process.

5 Conclusions

We have presented a proof-of-concept implementation of a tool to automatically generate compositional performance models from a web service composition described in BPEL. Moreover we have proposed a means to parametrise the web service element of the model via an annotated version of WSDL.

This is preliminary work and several issues remain to be addressed. In particular more work is needed on appropriate parameterisation of the model.

- At the moment all communication costs are ignored;
- The Annotated WSDL will necessarily give a static view of the timing of each operation, not taking into account the current load on the host offering the service.
- The timing of local operations, within the component *BPEL* are currently allowed to default to 1.0, and a more realistic and differentiated estimate should be used.

Nevertheless we believe this offers a valuable first investigation into the automatic generation of PEPA models from BPEL and WSDL specifications.

References

[1] IBM. Business Process with BPEL4WS: Learning BPEL. `http://www-106.ibm.com/developerworks/webservices/library/ws-bpelcol2/`.

[2] Hillston J. *A Compositional Approach to Performance Modelling*. PhD thesis, University of Edinburgh, 1994.

[3] Andrews T., Curbera F., Dholhkia H., Goland Y., Klein J., Leymann F., Liuk K., Koller D., Smith D., Thattle S., Trickovid T, and Weerawaran A. Specification: Business Process Execution Language for Web Services Version 1.1. `http://www-106.ibm.com/developerworks/library/ws-bpel`, May 2003.

[4] W3C. Web Services. `http://www.w3.org/2002/ws/`.