Composite Web Services

Kung-Kiu Lau and Cuong Tran

Keywords. Web service, composite service, service composition, component model.

Abstract. Currently, composition of web services is done by orchestration. An orchestration is a workflow that combines invocations of individual operations of the web services involved. It is therefore a composition of individual operations, rather than a composition of entire web services. In this paper we propose a different approach to web service composition, whereby entire services are composed into composite services. The latter are again entire web services, that is, they can be further composed using our composition, or they can be used in an orchestration. We show how these composite services can be constructed hierarchically and used in practice.

1. Introduction

In a service-oriented architecture [18], individual services are combined into a single workflow that reflects the business process in question. Although services can be defined in a general way, in practice the most widely used services are web services [13, 2].

Currently, composition of web services is carried out by orchestration [14]. An orchestration is a workflow that combines invocations of individual operations of the web services involved. It is therefore a composition of individual operations, rather than a composition of entire web services.

In this paper, we propose a different approach to web service composition, whereby entire services are composed into composite services. The latter are again entire web services, that is they can be further composed using our composition, or they can be used in an orchestration.

The key difference between our approach and web service orchestration lies in the nature of a composite web service created by our approach. A composite service has all its operations available for composition or orchestration. By contrast, in an orchestration, only the chosen individual operations of the member services are available for invocation. A composite service is a service, whereas an orchestration is a workflow. By the same token, a composite service is also different from a choreography [14] (which is defined on a chosen set of individual operations).

Another important feature of our approach is that composition is hierarchical. This means that a composite service can be constructed step by step from sub-services in a systematic manner.

Our approach is based on our component model [11, 10]. In our model, components are built from computation units. These units provide operations but do not invoke other units, and so behave like web services. Our components are composed in a hierarchical manner by using special connectors, which we call exogenous connectors [11]. It is these connectors that make the difference between our model and other component models, and the difference between our approach to web service composition and current practice in web service composition.

2. Motivation

Currently, web service composition is done by orchestration [6]. A web service orchestration is a coordination of web service invocations, and can be represented by a workflow. It can therefore be defined as a function *ORC* with the following type:

$$ORC : op \times op \cdots \times op \rightarrow wf \tag{1}$$

where *op* is the type of operations in web services, and *wf* is the type of workflows for invoking a set of such operations.

An orchestration is defined using workflow languages such as BPEL [3], BPML [4] and XLANG [17]. A workflow in these languages can be converted into a web service by giving it a WSDL [13] interface. The resulting web service can then be orchestrated with other web services.

To motivate composite web services, in this section we use a simple example to show how composition is different from orchestration.

Consider a bank system with just one ATM that serves two bank consortia $BC1$ and $BC2$, each with two bank branches, $B1$ and $B2$, $B3$ and $B4$ respectively. The ATM reads the customer's card, performs a security check, identifies the customer's bank consortium and passes customer requests together with customer details to the customer's bank consortium. The customer's bank consortium checks customer details and identifies the customer's bank branch, and then passes on the customer requests and customer details to the customer's bank branch. The bank branch checks customer details and provides the usual services of withdrawal, deposit, balance check, etc.

Suppose all the elements of the bank system are available as web services, each providing appropriate operations. Then we can build a web service for the bank system by orchestrating all these web services, and converting the resulting workflow into a web service. For any particular orchestration, operations to be invoked in the web services have to be chosen, and one specific corresponding workflow is defined. Figure 1 shows one possible orchestration.

In this workflow, the operation *pc* (processCard) of the ATM is invoked to identify the customer's bank consortium. The operation *gb* (getBank) of bank consortium $BC1$ or $BC2$ is invoked to get the customer's bank branch. The operations *dp* (deposit) or *wd*

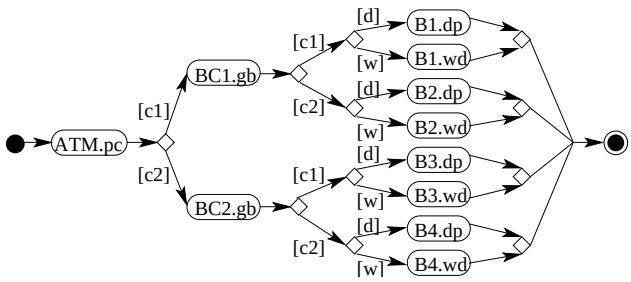

FIGURE 1. Bank orchestration.

(withdraw) are invoked in the bank branches ($B1$, $B2$, $B3$ or $B4$). This workflow can be converted into a bank web service that provides the *deposit* and *withdrawal* operations.

Orchestration is not compositional with respect to the operations invoked. That is, given an orchestration, it is not possible to add to its set of invoked operations and hence its workflow. For example in Figure 1 it is not possible to add an invocation of *security check* to *ATM*, or a *balance check* operation to the bank branches. Any such change would require an entirely new orchestration.

This is true even if the orchestration is defined in a hierarchical workflow language like YAWL [19]. Figure 2 shows how the bank system workflow in Figure 1 can be defined in YAWL.

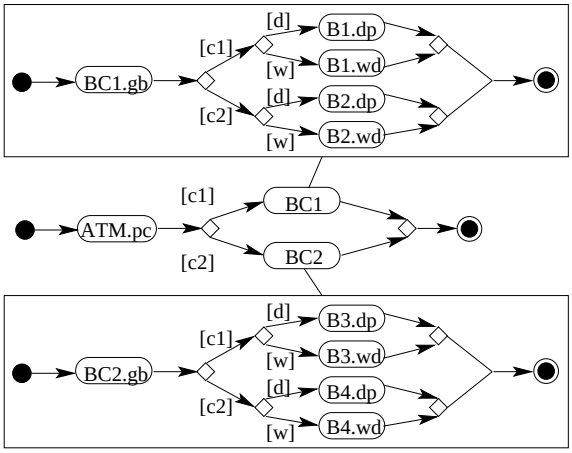

FIGURE 2. Bank with nested workflows.

To add security checks to *ATM*, it would be necessary to change the top-level workflow. To add *balance check* to bank branches, it would be necessary to change the sub-workflows for *BC1* and *BC2*.

Of course in an orchestration, it is possible to include all the operations of all the web services involved. However, such a workflow can potentially be very large, complex

and cumbersome. Furthermore, it will contain many redundancies and repetitions because many sub-workflows are duplicated, as can be seen in Figures 1 and 2.

By contrast, we define a composite web service as a web service that is composed from sub-services. A composite web service is not just one orchestration, but is a web service that provides all the operations of all the sub-services, i.e. it contains all possible orchestrations of these operations. For the bank system, the composite service would have the workflow shown in Figure 3, where # denotes a parameter. This workflow is

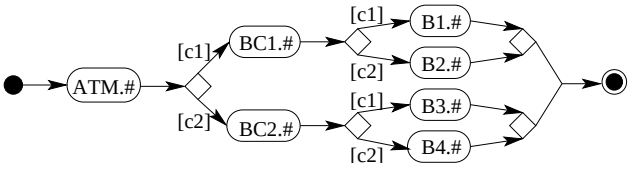

FIGURE 3. Bank composite.

parameterised over all the operations of every web service involved.

3. Web Service Composition

So we want to define web service composition differently from web service orchestration. In particular, we want to define it hierarchically, that is, we want to be able to compose services into composite services, which in turn can be composed into even bigger composite services. This is illustrated by Figure 4

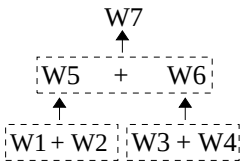

FIGURE 4. Web service composition.

where web services W1 and W2 are composed into a composite service W5, and web services W3 and W4 are composed into a composite service W6. W5 and W6 are in turn composed into W7.

A composition can be defined as a function *COMP* with the following type:

$$COMP : ws \times ws \times \cdots \times ws \rightarrow ws \tag{2}$$

where *ws* is the type of web services.

The difference between orchestration and composition can be seen clearly by comparing (1) and (2): an orchestration takes named operations (in the web services involved)

as arguments and returns a workflow (for the invocations of these operations); whereas a composition takes web services and returns a (composite) web service.

Our definition of web service composition is based on a component model that we have defined [10], in particular composition in the design phase [9]. This model defines what components are, as well as composition operators for them, for different phases, namely design and deployment phases. We will show that our model can serve as a component model for web services and their composition.

3.1. A Component Model for Web Services

In our model [10], components have the distinguishing features of *encapsulation* and *compositionality*. Components are constructed from two kinds of basic entities: (i) *computation units*, and (ii) *connectors* (Figure 5). A computation unit *CU* encapsulates computation. It provides a set of methods (or operations). *Encapsulation* means that *CU*'s methods do not call methods in other computation units; rather, when invoked, all their computation occurs in *CU*. Thus *CU* could be thought of as a web service.

There are two kinds of connectors: (i) *invocation*, and (ii) *composition* (Figure 5). An invocation connector is connected to a computation unit *CU* so as to provide access to the methods of *CU*.

A composition connector encapsulates *control*. It is used to define and coordinate the control for a set of components (atomic or composite). Composition connectors can be defined for the usual control structures for sequencing and branching. A *sequencer* connector that composes components $C_1, \ldots, C_n$ can call methods in $C_1, \ldots, C_n$ in that order. A *pipe* connector is similar to a sequencer, but additionally passes the results of calls to methods in C_i to those in C_{i+1}. A *selector* connector that composes components $C_1, \ldots, C_n$ can select one component out of $C_1, \ldots, C_n$ and call methods in that component only. The control structure for looping is defined as iterators on individual composition connectors (and invocation connectors, see below). Our composition connectors are thus a Turing complete set [12, 5], for defining control flow.

Clearly composition connectors can define (and encapsulate) *workflow* for a set of connected components. They can define workflow control-flow for sequencing, branching and looping, as described in e.g. [20].

Components are defined in terms of computation units and connectors. There are two kinds of components: (i) *atomic*, and (ii) *composite* (Figure 5). An atomic component

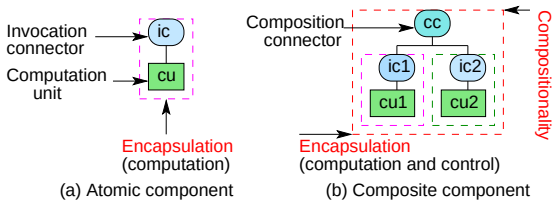

FIGURE 5. Our component model.

consists of a computation unit with an invocation connector that provides an interface to

the component. An atomic component encapsulates *computation* (Figure 5(a)). A composite component consists of a set of components (atomic or composite) composed by a composition connector. The composition connector provides an interface to the composite. A composite component encapsulates *computation* and *control* (Figure 5(b)).

An atomic component can thus be a web service, its invocation connector being the WSDL interface. A composite component can be a (composite) web service that contains sub-services as well as workflow between the sub-services. Its top-level composition connector is its interface. However, this interface cannot be described in standard WSDL since the web service now contains workflow (in the composition connector).

Our components are also *compositional*, i.e. the composition of two components C_1 and C_2 yields another component C_3. In particular, $C3$ also has the defining characteristics of encapsulation and compositionality. Thus compositionality implies that composition preserves encapsulation (Figure 5(b)).

Encapsulation and compositionality lead to *self-similarity* of composite components, as can be clearly seen in Figure 5(b). Self-similarity provides the basis for a hierarchical way of composing systems from components.

Encapsulation and compositionality result form the nature of our connectors. They are in fact *exogenous connectors* [11], and encapsulate control outside of computation units in a system. Exogenous composition connectors are defined in a hierarchical way. For example, a *sequencer* connector, or a *pipe* connector, that composes two atomic components A_1 and A_2 is clearly defined in terms of the invocation connectors in A_1 and A_2. In general, exogenous composition connectors form a hierarchy built on top of invocation connectors for atomic components. Connectors at level n for any $n > 1$ can be defined in terms of connectors at levels 1 to $(n - 1)$. Indeed, exogenous connectors have a hierarchical type system [11].

The hierarchical nature of exogenous connectors entails a strictly hierarchical way of constructing systems by composing components. In such a system, atomic components form a flat layer, and the entire control structure (of composition connectors) sits on top of this. The precise choice of connectors, the number of levels of connectors, and the connection structure, depend on the relationship between the behaviour of the individual components and the behaviour that the whole system is supposed to achieve. Whatever the control structure, however, it is strictly hierarchical, which means that there is always only one connector at the top level. This is the connector that initiates control flow in the whole system.

As an example, the bank system can be constructed using our component model as shown in Figure 6. $P1$, $P2$ and $P3$ are pipe composition connectors; $S1$, $S2$ and $S3$ are selector composition connectors; and $I1 \ldots I7$ are invocation connectors. The top-level connector $P1$ is the interface to the system, and is where control flow starts.

4. Composite Web Services

Using our model as a component model for web services, we can use standard web services as atomic components, composite web services as composite components, and use

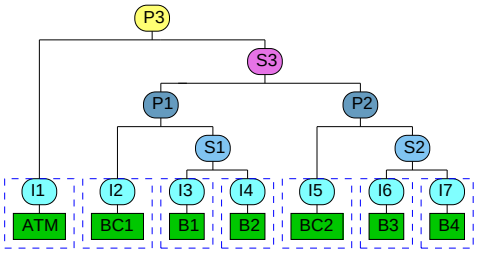

FIGURE 6. The bank system.

the composition connectors[1] as composition operators for web services. This is illustrated

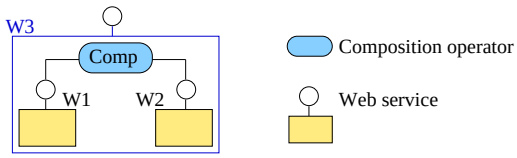

FIGURE 7. Composite web services.

in Fig 7, where two services *W1* and *W2* are composed by a composition operator *Comp* into a composite service *W3*.

W3 is a web service, just like *W1* and *W2*. However, whereas *W1* and *W2* have interfaces described in standard WSDL, *W3* has an interface that cannot be described in standard WSDL, because *W3* contains workflow embodied in the composition operator *Comp*. Therefore, in order to define *W3* as a web service, we need to extend standard WSDL in order to incorporate workflow description. Then we need to devise a method to generate its interface in the extended WSDL from the standard WSDL interfaces of *W1* and *W2*.

The bank system in Figure 6 can be built as a composite web service composed from standard web services for *ATM*, *BC1, BC2, B1, B2, B3* and *B4* (Figure 8). The structure of this composite is of course identical to that of the bank system in Figure 6.

The composition is hierarchical (composite services are denoted by dotted boxes): *B1* and *B2* are composed into the composite service *C1* by using the selection connector *S1*; the composite *C1* is in turn composed with *BC1* using the pipe connector *P1*, creating the composite *C2*; similarly *B3* and *B4* are composed into *C3* by using the selection connector *S2*; the composite C3 is then composed with *BC1* using the pipe connector *P2*, creating the composite *C4*; the composite *C2* is in turn composed with *C4* by using the selector connector *S3* to create the composite *C5*; the composite *C5* is composed with *ATM* by using another pipe connector *P3*, creating the composite *C6*.

The composite service C6 provides all the operations offered by its sub-services.

[1] In the design phase.

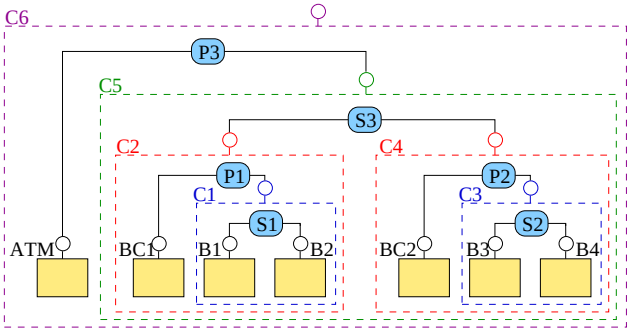

FIGURE 8. The bank composite web service.

4.1. Defining Composite Web Services

In order to define composite web services, we need to extend standard WSDL to incorporate the workflow added by connectors in composition. To this end, we define a new extensible element for WSDL documents, called *workflow*. It contains child elements which describe the details of the workflow structure. The extended WSDL document for a composite service consists of standard elements such as types, messages, portType, binding and services, together with the additional *workflow* element, as shown in Figure 9.

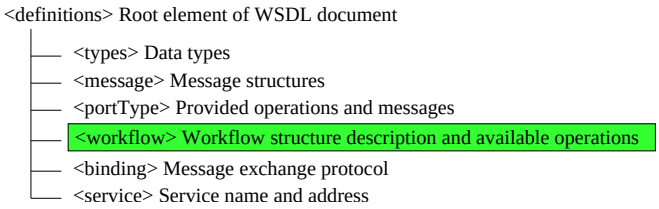

FIGURE 9. An extended WSDL document.

Under the *workflow* tag, there are extensible tags describing workflow structures. We define such a tag for each of our composition connectors. The behaviour of the connectors is defined by their implementation on the web server concerned.

The tag for each connector in turn contains child tags specifying the services (and operations) involved. If a connector provides sequential invocation, e.g. sequencer and pipe, then the child tags describe the sequence of services involved. If a connector provides a branching structure, e.g. selector, then the child tags specify the branching condition and the corresponding services.

The schema for *workflow* consisting of pipe and selector connectors is depicted in Figure 10. This workflow element has either a pipe or choice child element. Each contains a number of services (and operations). Furthermore, workflow structures (pipe and choice) may in turn contain one another.

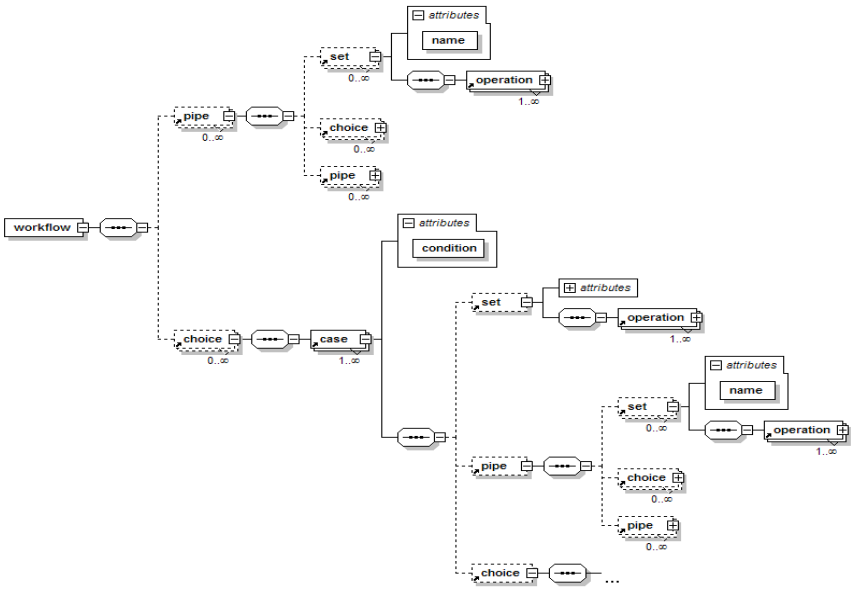

FIGURE 10. Schema for workflow and available operations.

The *pipe* tag is used to represent the pipe workflow structure provided by the pipe connector. The pipe connects a sequence of services specified by *set* tags, or other workflow structures (pipe or choice). The pipe invokes every service, or passes requests to structures, in the sequence. An invocation result is used as input to the next invocation.

The *choice* tag represents the branching workflow structure embodied by the selector connector. It contains a number of cases specified by the *case* tag. A case is a combination of a matching condition and an operation set (i.e. service) or another workflow structure. Different cases have different matching conditions. The choice workflow invokes a service or a structure if the corresponding matching condition is satisfied.

As an example, the workflow description for the composite bank service in Figure 8 can be described by the following outline:

```
<workflow>
 <pipe> <set name="ATM">
   <operation name="procCard">...</operation>
  </set>
  <choice>
   <case condition="1">
    <pipe>
     <set name="BC1">
      <operation name="getBank">...</operation> </set>
     <choice>
      <case condition="1"><set name="B1">
       <operation name="withdraw">...</operation>
       <operation name="deposit">...</operation> ...</set> </case>
      <case condition="2">
       <set name="B2">
        <operation name="withdraw">...</operation>
        <operation name="deposit">...</operation> ...</set></case>
     </choice></pipe></case>
```

```
<case condition="2">
 <pipe>
  <set name="BC2">
   <operation name="getBank">...</operation> </set>
...
```

The intended meaning of this workflow is that it first invokes *any* operation of ATM, and pipes the result to the branching structure; if the result is 1, then *any* one of BC1's operations can be invoked or if the result is 2 then *any* one of BC2's operations can be invoked; the result of BC1's operation is used to compare with the branching condition; if the value is 1, then *any* one of B1's operations can be invoked, or if the value is 2, then *any* one of B2's operations can be invoked. Similarly the result of BC2's operations is used in comparison with branching condition; if the condition is 1 then *any* one of B3's operations or B4's operations will be invoked. After that, the workflow ends and the result of the last invocation is returned.

4.2. Implementing Composite Services

Given the extended WSDL document for a composite service, we need to implement the service on a web server. This implementation consists of the implementation of the intended behaviour of the workflow defined in the extended WSDL document, as well as the implementation of the interface of the composite service, also defined in the extended WSDL document.

For every composition connector, we need to implement its workflow defining its intended behaviour. To this end, we implement our connectors as Java classes which are stored as templates. Every time we use connectors to create composite services, these templates are used to generate real Java classes.

In general, for a composite service, the Java class for the top-level connector always has one operation *invoke*, that is the operation provided by the composite service to the outside world. Clients use a composite service via its *invoke* operation. Depending on the behaviour of each connector, the *invoke* operation may have different signatures. Basically, the signature of *invoke* comprises three main elements, viz. condition, operation names and operation parameters. The condition is used in a branching workflow structure for selecting sub-services. Operation names indicate which operations of the selected sub-services are invoked. Operation parameters are parameters passed to the invoked operations. Also, the signature of *invoke* includes the results returned by the composite.

The signature of *invoke* is reflected in the definitions of types, messages and portType of the extended WSDL document for a composite service. We implement message exchange style as RPC, and transport as SOAP over HTTP, in the popular manner. This information is contained inside the binding section of the extended WSDL document. The composite service address is specified at design time and contained in the service section.

The Java classes for connectors after generation are compiled and deployed to a web service engine, which is Axis [1] in our implementation. We now show our implementation for the *pipe* and *selector* connectors. For simplicity, our implementation only deals with parameters of primitive data types, e.g. string, integer, float, etc. We use String as intermediate type because other primitive types can be converted to String and vice versa.

The Pipe class template has one method:

```
invoke(String[] methods, String[] params);
```

The pipe connector receives a list of operations and a list of parameters for these operations. The *invoke* method is used to sequentially call every operation in the list. Each operation is provided by a sub-service.

If a sub-service is standard service, the connector identifies the number of parameters for every operation so that the parameters can be taken out of the parameter list and passed to the operations invoked. The connector also does type conversion for parameters if the invoked operations use primitive types different from String. If it is at the beginning or the middle of the operation list, the result of an invoked operation will be inserted into the first position of the parameter list for subsequent operation invocations. Otherwise, the result is returned as the output of the composite. The completed operation and used parameters are thus removed from the operation and parameter lists.

If a sub-service is a composite service, the connector just passes the whole operation list at that point to the *invoke* operation of the sub-service. However, if the (composite) sub-service has a branching structure, then the connector extracts the first element of the parameter list before passing a call to the sub-service operation.

The definitions of types, messages and portType of the extended WSDL document for composite services having a pipe as the top-level connector look like the following:

```
<wsdl:types>
 <schema targetNamespace="urn:cbsd" .../>
  <complexType name="ArrayOfString">
   <sequence><element name="item" type="xsd:string"/>
   </sequence></complexType></schema>...</wsdl:types>

<wsdl:message name="invokeRequest">
 <wsdl:part name="operations" type="ArrayOfString"/>
 <wsdl:part name="params" type="ArrayOfString"/>  </wsdl:message>

<wsdl:message name="invokeResponse">
 <wsdl:part name="result" type="xsd:string"/>  </wsdl:message>

<wsdl:portType name="...">
 <wsdl:operation name="invoke"  parameterOrder="operations params">
  <wsdl:input message="invokeRequest".. />
  <wsdl:output message="invokeResponse".../> </wsdl:operation></wsdl:portType>
```

ArrayOfString is not a primitive data type, so we need to define it in the extended WSDL document. The *invoke* operation has the input message *invokeRequest* consisting of two arrays of string containing the operation and parameter lists. The output message *invokeResponse* contains the result of the *invoke* operation.

The Selector class template also has just one method:

```
invoke(String condition, String[] operations, String[] params);
```

Like the pipe connector, the selector connector receives a list of operations (provided by the sub-services) and a list of parameters for these operations. In addition, it also receives a *condition* for selecting one of the sub-services. The *invoke* method is used to call one operation in the selected sub-service, i.e. one which matches the *condition* passed to the method. If the selected sub-service is a standard service, when the selector selects whichever operation, it will identify the number of parameters and their types for the selected operation, extract parameters from the parameter list, convert to appropriate types if needed, and pass the extracted parameters to the selected operation. The result

of the selected operation is the output of the composite. If the selected sub-service is a composite service, the connector will extract the first parameter from the parameter list, and put it together with the operation and parameter lists into a call to the *invoke* operation of the selected sub-service.

The definitions of types and output message of the extended WSDL document for composite services having selector as the top-level connector are similar to those for pipe connector. However, the *invoke* operation of selector has a different signature (with the addition of *condition*), which affects input message and portType definitions. These definitions are as follows:

```
...
<wsdl:message name="invokeRequest">
 <wsdl:part name="condition" type="xsd:string"/>
 <wsdl:part name="operations" type="impl:ArrayOfString"/>
 <wsdl:part name="params" type="impl:ArrayOfString"/></wsdl:message>
...
<wsdl:portType name="...">
 <wsdl:operation name="invoke" parameterOrder="condition operations params">
  <wsdl:input message="impl:invokeRequest".../>
  <wsdl:output message="impl:invokeResponse".../>
 </wsdl:operation> </wsdl:portType>
```

The binding and service sections for composite services are shown below:

```
<wsdl:binding name="..." type="...">
 <wsdlsoap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
 <wsdl:operation name="invoke">  <wsdlsoap:operation soapAction=""/>
  <wsdl:input name="invokeRequest">
   <wsdlsoap:body  encodingStyle="..." namespace="urn:cbsd"
    use="encoded"/></wsdl:input> <wsdl:output name="invokeResponse">
...
<wsdl:service name="..."> <wsdl:port binding="..." name="...">
  <wsdlsoap:address  location="http://server/composite_service"/>
 </wsdl:port></wsdl:service>
```

As mentioned before, the message exchange style is RPC, and the transport is SOAP over HTTP.

4.3. Tool Support

To support our approach to service composition, we have implemented a tool. The tool can be used by a service designer to construct a composite web service, and also by a client to invoke a composite web service.

The process of creating composite services with our tool is illustrated in Figure 11. We start with WSDL documents of standard web services, or extended WSDL (WSDL'

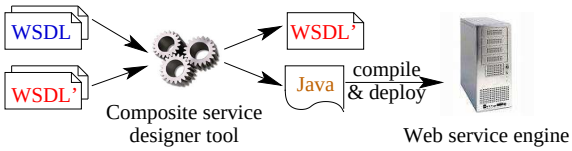

FIGURE 11. Construction process.

in Figure 11) documents for composite services as inputs. The tool generates Java classes

and the associated extended WSDL document for the resulting composite service. The Java code of the composite service is compiled and the binary is deployed on to the web service engine. This construction process can be applied hierarchically, by building (composite) services and composing them successively.

Figure 12 shows a screen shot of our tool being used to create a composite service. Through the user interface, the tool allows the service designer to choose WSDL

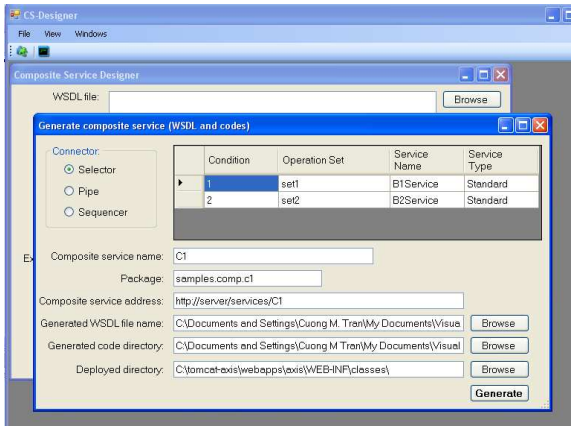

FIGURE 12. Composite service designer tool.

or extended WSDL files as its input. The designer can also choose a desired connector, give a name to the composite service, assign composite service address, and specify the directory for the generated code.

The example in Figure 12 shows the creation of the composite web service $C1$ by composing two standard web services $B1$ and $B2$ using a selector connector. The composite service address is `http://server/services/C1`. The composite service Java code is generated and compiled. The composite service in binary is deployed to the server at the directory `c:\tomcat-axis\webapps\...\classes\`.

Figure 13 shows a screenshot of our tool being used by a client to invoke a composite service. Our tool allows clients to input a composite service description file (WSDL' file). The tool then draws a diagram of the workflow structure embodied in the composite service. The client clicks at each activity in the diagram and chooses one operation to be invoked. Based on the chosen operations, our tool generates the syntax for client calls to the composite service.

The example in Figure 13 shows a client using our composite bank service. The client can see the workflow embodied in the service, chooses the operation *withdraw* of the bank branch, and clicks the *Generate* button. The tool then shows the syntax to invoke the composite bank service, and the code for this invocation is also generated in a directory.

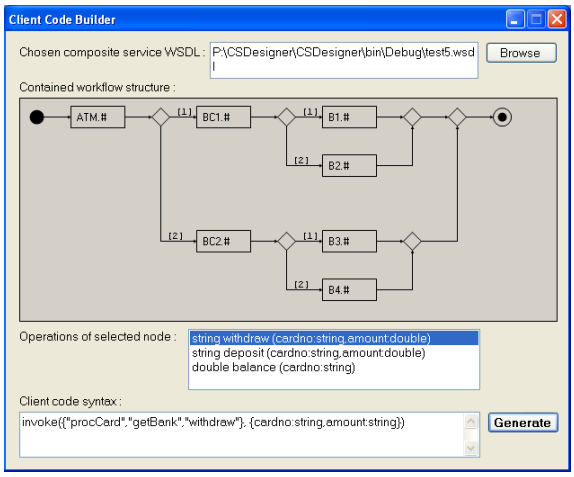

FIGURE 13. Composite service client tool.

5. Discussion

The key difference between composition and orchestration lies in the nature of a composite web service created by our approach. A composite service has all its operations available for composition or orchestration. By contrast, in an orchestration, only the chosen individual operations of the member services are available for invocation. Because it contains a workflow structure, a composite service can specify many different business logics involving the operations of its sub-services. In other words, a composite service contains many workflows, whereas an orchestration defines just one.

Our approach is distinctive compared with other current approaches. Our composite service now is truly a composite which captures entire element services and composite exists at every composition. Composite service is constructed by using our special connector as composition operators. Moreover, composite has separation between invocation control structure (given by connector). and services. This leads to our approach brings up some strong benefits. Because composition is fundamentally different from orchestration, our approach is novel. For practical purposes, we believe our approach also has some advantages over current approaches to web service orchestration.

First, our approach eases the creation of composite services. Developers need only focus on building up a structure of available services. Composition does not involve fixed operations. By allowing parameterisation of operations to be invoked, it enables clients to choose the operations based on their business logic. Thus, a composite service, once built, can be used in many different applications. In our example, the composite bank service C6 can allow multiple applications.

The second benefit of our approach is the reduction in effort of creating and maintaining web service orchestrations that belong to the same composite. Instead of incurring cost for creating and keeping separate multiple workflows working, developers of applications can just use an appropriate composite service which is already constructed to fulfil

their needs. For example, in our bank composite, only the parametric workflow needs to be maintained, instead of the individual workflows that it contains. Thus our approach minimises the maintenance problem as maintenance only happens on the composite service, and the client need not change the code of his application for each business logic embodied in the composite service.

The third benefit is the hierarchical manner of building composite services. After every composition at every level of the whole system, there exist composite services. These individual composite services can be used separately by other applications. For instance, in the bank example, two composites C2 and C4 could be used in an application involving multiple bank consortia. In this case, C6 would allow customers belonging to multiple consortia to use its sub-services. Another benefit is easing service composition maintenance. Thanks to the hierarchical nature of our composition approach, if one sub-service has changed its location, only one composite service containing this sub-service is affected. The composite can be updated locally by its developer and the change can happen without requiring updates to other related services. For instance, in the bank example, if B1 changes, then only C1 is affected.

Finally, as mentioned before, our composite service is still a web service. Thus it can be used in orchestration. As shown in the previous section, our tool allows a composite service to be invoked, yielding a workflow. However, our tool is not yet integrated with standard orchestration tools. For such an integration we need to extend existing workflow designer tools such as Eclipse-BPEL. Such a tool would combine a standard WSDL processor with a processor for extended WSDL as defined in Section 4.1. Our tool can provide the processor for extended WSDL, and we are currently working on its integration with Eclipse-BPEL.

6. Related Work

Although orchestration and choreography are related to our work, we have already pointed out that our approach is fundamentally different. In orchestration, an orchestration language, such as BPEL [3], is used for defining executable workflows in XML-based format, consisting of series of activities. Every activity requires a particular service operation as input. The workflow can be deployed onto a workflow enactment system, such as the BPEL engine, which manages the workflow execution. However, existing orchestration languages like BPEL and YAWL [19] cannot describe parametric workflows as embodied in our composite web services.

Choreography focuses on describing interactions between services by specifying operations in structures such as sequence, choice, etc., using a language like WS-CDL [8]. The approach still explicitly requires specific operations to be named in the choreography document. Furthermore, choreography of services does not result in a service which can be executed.

Aspect-oriented Web Service (AOWS) [7, 16] is web service based on AOCE (Aspect-Oriented Component Engineering). A service is enriched with an aspectual description which supports automated service discovery. This approach uses an AOConnector object

which serves as a gateway to a client. The connector receives client requests and relays them to an appropriate AOWS. Their connector is unlike our composition connector because it does not define a workflow structure, and using their connector on an AOWS does not produce a service.

Web Transact [6, 15] is a framework for providing transactional features to service composition. It suggests to compose web services in hierarchical architectures. Standard web services providing similar functional capabilities are bundled using the mediator pattern to create mediator services. Mediator services are later composed to create composite services by using WSTL (Web Service Transaction Language) to specify the execution sequence of specific mediator service operations. Thus, a composite service in this approach still involves invocation of specific operations. Also, a composite does not exist at every level of composition, unlike our approach. Therefore we believe their approach is not hierarchical.

7. Conclusion

We have presented a new approach for web service composition using exogenous connectors as composition operators on web services. The composite service captures all the operations provided by the sub-services, and it allows the operations to be invoked in a defined workflow structure. A composite service thus represents a rich service, giving clients a choice of many operations. Our approach appears to have benefits compared with current approaches, especially orchestration.

In future, we plan to work on outstanding issues such as complex data structure manipulation in service communication, and error propagation among composite services.

In addition, it will be interesting to test the practicality of our approach, with regard to SOAs for larger real-world applications. To this end, we will need to investigate how to publish composite services in a suitable registry, along the lines of UDDI [13].

References

[1] Axis - web services framework web site. http://ws.apache.org/axis/.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeragwarana. Business process execution language for web services - version 1.1. Technical report, IBM, 2003.

[4] A. Arkin. Business process modeling language. Technical report, BPMI Organisation, 2005.

[5] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. Comm. ACM, 9(5):366–371, 1966.

[6] S. Dustdar and W.Schreiner. Survey of web service composition. *Int. J. Web and Grid Services*, 1(1):1–30, 2005.

[7] J. Grundy, T. Panas, S. Singh, and H. Stockle. An approach to developing web services with aspect-oriented component engineering. In *In Proceedings of the 2nd Nordic Conference on Web Services*, 2003.

[8] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web services choreography description language version 1.0. Technical report, W3C, 2004.

[9] K.-K. Lau, L. Ling, and Z. Wang. Composing components in design phase using exogenous connectors. In *In* Proc. 32nd Euromicro Conference on Software Engineering and Advanced Applications, pages 12–19, 2006.

[10] K.-K. Lau, M. Ornaghi, and Z. Wang. A software component model and its preliminary formalisation. In F. de Boer *et al.*, editor, Proc. 4th Int. Symp. on Formal Methods for Components and Objects, LNCS 4111, pages 1–21. Springer-Verlag, 2006.

[11] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In G. Heineman *et al.*, editor, Proc. 8th Int. Symp. on Component-based Software Engineering, LNCS 3489. Springer, 2005.

[12] D. Le Métayer, V.-A. Nicolas, and O. Ridoux. Exploring the software development trilogy. In IEEE Softw., volume 15, pages 75–81, 1998.

[13] E. Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI.* Addison-Wesley, 2002.

[14] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[15] P. Pires. Webtransact: A framework for specifying and coordinating reliable web services compositions. Technical report, Federal University of Rio De Janeiro, 2002.

[16] J. Hosking S. Singh, J. Grundy and J. Sun. An architecture for developing aspect-oriented web services. In *Proceedings of European Conference on Web Services, Vaxjo, Sweden*, 2005.

[17] S. Thatte. Xlang: Web services for business process design. Technical report, Microsoft, 2001.

[18] E. Thomas. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, 2005.

[19] W. van der Aalst, L. Aldred, M. Dumas, , and A. ter Hofstede. Design and implementation of the YAWL system. In 16th Int. Conf. on Advanced Information Systems Engineering, 2004.

[20] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. In Distributed and Parallel Databases, pages 5–51, 2003.

Kung-Kiu Lau and Cuong Tran
School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom

e-mail: {kung-kiu,ctran}@cs.man.ac.uk