

By Jian Yang

WEB SERVICE COMPONENTIZATION

Developing a framework for analyzing service composition reuse and specialization.

SERVICE-ORIENTED COMPUTING IS BECOMING the prominent paradigm for distributed computing and e-commerce, creating opportunities for service providers and application developers to develop value-added services by combining Web services. Web services are self-contained, Web-enabled applications capable not only of performing business activities on their own, but also possessing the ability to engage other Web services in order to complete higher-order business transactions. Examples of such services include checking credit, ordering products, and procurement. The platform-neutral nature of Web services creates the opportunity for building composite services by using existing elementary or complex services, possibly offered by different service providers.



The recently proposed standard Business Process Execution Language for Web Services (BPEL4WS or BPEL for short)¹ is an XML-based specification language that specifies how to define a business process in terms of compositions of existing Web services. BPEL models the actual behavior of a participant in a business interaction as well as the visible message exchange behavior of each of the parties involved in the business protocol. A BPEL process is defined “in the abstract” by referencing and interlinking portTypes specified in the WSDL definitions of the Web services involved in a process.

Currently, the Web service application realm, even for applications developed on the basis of BPEL, is rather unstructured and flat. The reason being that services are composed in a rather ad hoc and opportunistic manner by simply combining their operations and input and output messages. If the

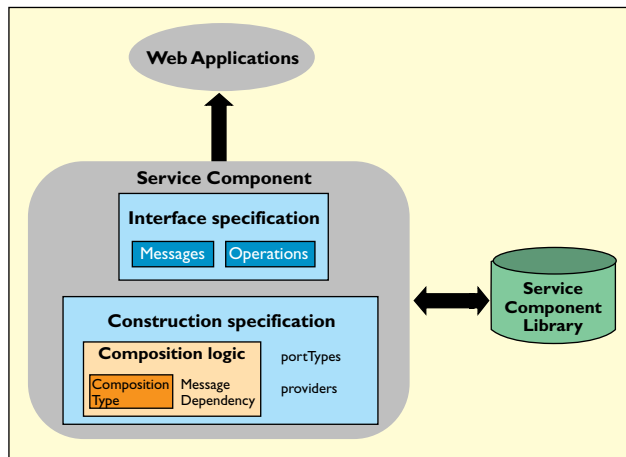


Figure 1. Service component ingredients.

requirements of the application change or need to be adjusted, then the composition will have to be respecified and recreated by possibly interlinking additional or modified service interfaces. This approach leads to a proliferation of service operations and results in unmanageable and cluttered solutions.

Web service design and composition is essentially a distributed programming activity. There are cases where application developers want to reuse the design and implementation of existing Web services only by extension or restriction without developing them from scratch. Therefore, designing service-based distributed applications is an activity very similar to the original vision of component-based development, which is not just about development, but also about managing the assembly of collabora-

tive assets to provide a highly adaptive application solution. This requires software engineering principles and technology support for service reuse, specialization, and extension such as those used, for example, in component-based software development [4]. Currently, it is not possible to define and implement a Web service composition once and use it in similar designs with some variations in a later stage.

To address this limitation of Web service technology, we introduce here the concept of service component. Aim of a service component is to raise the level of abstraction in Web services composition. Service components represent modularized service-based applications that package and wire together service interfaces with associated business logic into a single cohesive conceptual module. These modules can be extended, specialized, and generally inherited, to assist in the creation of new applications.

Service Component for Web Service Composition

As a starting point for the discussion of Web service composition reuse and specialization, we need to identify several key elements required in specifying Web service compositions and commonly used in the currently proposed standards such as BPEL and the Business Process Modeling Language (BPML)—see www.bpmi.org.

Elements of a Service Component. In most proposed Web service composition and business process modeling languages the basic elements are processes, activities, data flows, and control flow. A process is viewed as a series of activities where an activity represents a well-defined business function. For example, hotel booking and air ticket reservation can be two activities in a travel plan business process. Activities can be composed into complex activities. Data flow represents the data flowing from one activity to another. For example, if we want to reserve an air ticket before booking a hotel room, then the arrival/departure dates from the air ticket reservation activity will be sent to hotel booking activity as the check-in/check-out dates. Control flow, on the other hand, specifies how a set of activities is executed in terms of sequence, conditions, and coordination actions.

Figure 1 depicts the ingredients of a service component. It illustrates that a service component presents a uniform public interface to the outside world in terms of a uniform representation of exported operations of its constituent services and input/output messages. It also illustrates how the service com-

¹See www-106.ibm.com/developerworks/library/ws-bpel/.

ponent is constructed internally in terms of the composition logic, portTypes used in the composition, and the providers who provide the constituent services.

Composition logic refers to the way a composite service is constructed in terms of its constituent services. It covers the aspects of control flow and data flow in a process. A block of Web service operations is composed according to the composition logic specified in a service component. Here, we assume all publicly available services are described in WSDL. Composition logic comprises the following two fundamentals:

- **Composition type:** this construct signifies the nature that the composition can take on the basis of combining Order, Alternative service execution, and Conditioned execution. Most commonly supported composition types in current business process specifications include: parallel, sequential, choice, while, all, and foreach. Examples in this article are formulated on the basis of these types.
- **Message dependency:** this construct signifies whether there is a message dependency among the constituent services and the composite service. We distinguish between three types of message handling constructs necessary for compositions: message synthesis combines the output messages of constituent services to form the output message of the composite service; message decomposition decomposes the input message of the composite service to generate the input messages of its constituent services; and message mapping specifies the mappings among the inputs and outputs of the constituent services. For example, the output message of one constituent service could be the input message of another service.

Defining a Service Component Class

A service component can be specified in two isomorphic forms: a class definition and an XML-based composition specification that corresponds and conforms to the class definition of a service component and is used to implement the service class. The XML-based composition specification can be

BPEL or other XML-based business process specifications. The class definition of a service component is used for discovery, reuse, extension, specialization, and versioning purposes, whereas its corresponding XML/WSDL form is used for construction purposes, message exchange, service invocation and communication across the network. Here, we only concentrate on service component

class definition; the XML specification for the service component we developed can be found in [5].

The following example illustrates the concept of a service component class. Suppose we need to construct a travel plan Web service, which involves activities such as air ticket reservation, hotel booking, and sightseeing booking where air ticket reservation

Figure 2. A service component class definition.

```
webComponentClass TravelPlan {
  Definition
  d1: TripOrderMessage tripOrderMsg
  d2: TripResultMessage tripResDetails
  d3: travelPlanning (in tripOrderMsg, out tripResDetails)
  Construction
  c1: sequential (ticketRes, booking)
  c2: parallel (c1, sightSeeing)
  PortType
  pt1: TicketReservationPT.tReservation ticketRes
  pt2: HotelBookingPT.hBooking booking
  pt3: SightSeeingPT.sightSeeing sightseeing
  Provider
  pv1: TicketReservationProvider TicketReservationPT
  pv2: HotelBookingProvider HotelBookingPT
  pv3: SightSeeingProvider SightSeeingPT
  MessageHandling
  m1: messageDecomposition(TravelPlanning.tripOrderMsg,
    HotelBookingPT.hBooking.hotelBookingMsg,
    TicketReservationPT.tReservation.ticketResMsg)
  m2: messageSynthesis(HotelBookingPT.hBooking.hotelBookingDetails,
    TicketReservationPT.tReservation.e-ticket,
    SightSeeingPT.sightSeeing.sightSeeubgDetails)
```

needs to be executed before hotel booking, and sightseeing can run parallel with the other two. Presumably these activities are provided by different Web service portTypes offered by different providers.

In Figure 2, we define a service component class for the composite service named TravelPlan. The **Definition** construct defines the types of two messages tripOrderMsg and tripResDetails and one public operation travelPlanning. The two messages are the input and output of operation travelPlanning. **Construction** specifies the way the three constituent activities are composed. The construct **PortType** specifies the port types and operations that the activities refer to. In this example, operation tReservation of port type TicketReservationPT, operation hBooking of port type HotelBookingPT, and operation sightSeeing of port type SightseeingPT are used for activities ticketRes, booking, and sightseeing respectively. The **Provider** construct defines the Web service providers that provide the service activities. The construct **MessageHandling** defines message dependency among service component operations and their constituent activities. For example, **messageDecomposition** decomposes tripOrderMsg of operation TravelPlanning into two input messages hotelBookingMsg and ticketResMsg of HotelBookingPT.hBooking and TicketReservationPT.tReservation, respectively.

Reusing and Specializing Service Components

The class definition for a service component provides five aspects that can be seen as a basis for reuse and specialization, namely **Definition**, **Construction**, **PortType**, **Provider**, and **MessageHandling**. The service component class itself can also be specialized and reused.

Reuse and Specialization of Definitions. If we, for example, wish to provide an additional output message `explanationMsg` besides the two messages defined in the class `TravelPlan` in Figure 2 to explain the reason why a travel plan is failed, we can define a new class `AddTravelPlan` as follows:

```
serviceComponentClass AddTravelPlan SubclassOf
TravelPlan {
    Definitions
        PlanFailReasonMessage reasonMsg
        TravelPlanning(in tripOrderMsg, out
tripResDetails, out reasonMsg)
        ...
}
```

In this example, the operation `travelPlanning` is refined by providing two output messages instead of one defined in superclass `TravelPlan`.

Reuse and Specialization of Construction. Suppose a new travel planning service includes a fourth service `RestaurantBooking` that can run parallel with the others. We can then specialize the original construction in Figure 2 as follows:

```
serviceComponentClass additionalTravelPlan
SubclassOf TravelPlan {
...
    Construction
        c3: parallel(c2, restaurantBooking)
...
}
```

In this case, `c1` and `c2` are inherited from the super class `TravelPlan`.

If we decide that sightseeing should be booked after the ticket reservation and hotel booking, instead of rewriting the whole specification, we can introduce a new subclass as follows:

```
serviceComponentClass ModifiedTravelPlan Sub-
classOf TravelPlan{
...
    Construction
        c2: sequential(c1, sightseeing)
...
}
```

In this case, `c1` is inherited from the superclass, while `c2` is redefined.

Reuse and Specialization of PortType, Providers and MessageHandling Constructs. We can also add new elements, such as new service providers and new message handling constructs in the sub-service component classes. Overriding an element in **Provider** indicates we want to replace the old service provider with a new one in the subclass. Adding a new definition in the **messageHandling** construct normally associates with the changes occurred in the element(s) of the **Construction**. For example, adding a new **Construction** element `c3` in the previous example may result in the necessity of adding a new element in **messageHandling** that handles the synthesizing of the restaurant booking result with other activity results.

Service Component Reuse and Specialization. Since a service component is specified as a class, it can be reused whenever a service is required. If any part of the service component is used (for example, its messages and operations), then they have to be prefixed with the name of the service component. As illustrated previously, overriding and extending any service component element is a specialization of a service component.

The keyword **SubclassOf** means the defining service component class inherits all the aspects from the super service component class, except the parts that are redefined and modified in the subclass. Any new names introduced in any aspects of the subclass service component that are not defined in the superclass are treated as extensions to the superclass. All the names that appear in the subclass that are also used in the superclasses override corresponding construct definitions in the superclasses.

Service Component Creation

There are two approaches for creating service components:

- Converting from a WSDL/XML specification: in this case, a published WSDL service service or a XML-based service composition (for example, in BPEL) is converted into an equivalent object-oriented notion (class) as illustrated in the previous section. Any kind of service (composite or not) can be represented and stored as a service component class and can be used in the development of distributed applications.
- Defining a service component class from scratch—there are two scenarios here: Defining a concrete service component class, that is, all the aspect elements in the service component: **Definition**, **Construction**, **PortType**, **Provider**, and **MessageHandling** are provided. The example in Figure 2 is a concrete class; Defining an abstract

service component class, that is, only **Definition** aspect is specified in the abstract class. Concrete service component classes can be used to implement the abstract classes.

There are two ways to realize an abstract class by a concrete class: Defining concrete a class as subclass of the abstract class. In this case, the concrete subclass not only realizes the abstract class, but also can extend its definition (adding more operations); and defining concrete classes by implementing the abstract class. Similar to the practices followed by Java, we can in this case have several service component classes implementing the same abstract class. We can also have one service component class implementing several abstract service component classes.

The Service Component Class Library and Repository. The service component class library is a collection of general-purpose and specialized classes implementing the primitives and constructs discussed previously. Classes in the service component library provide basic constructs and functionality that can be

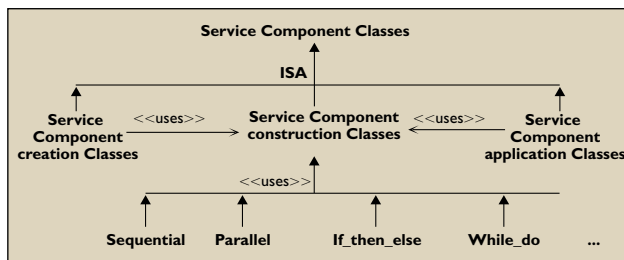


Figure 3. Service component classes.

used to create service component classes. The service component library provides system classes that can be categorized as follows:

- Service component creation classes: these classes are used for creating service component classes out of WSDL/BPEL specifications, that is, the **Definitions, Construction, portType, Message-Handling, and Providers** will be generated for the service component classes.
- Service component construction classes: these classes provide the semantics and functions to implement the composition types discussed earlier. These classes include: **sequential, parallel, if_then_else, and while_do**. The construction classes are used for building new service components.

Conversely, the service component class repository has a collection of concrete service component classes

ONCE SERVICE COMPONENT CLASSES ARE DEFINED, THEY CAN BE REUSED AND SPECIALIZED BY AIDING OR OVERRIDING THEIR DEFINITIONS.



known as service component application classes. The example in Figure 2 is a service component application class. All the classes stored in the repository also have WSDL Web service descriptions published in UDDI, for instance. Therefore they can be searched and discovered. Developing Web service-oriented applications becomes a job of reusing and specializing service component application classes; the various types of service component classes and their relationships are summarized in Figure 3.

Related Work

Subtyping and inheritance are both widely supported in object-oriented systems, however, the concepts involved in subtyping and inheritance are classes, attributes, and methods. While in the context of service component, subtyping and inheritance need to be extended as illustrated here. BPEL, XPDL,² and BPML are three standards for XML-based process definition languages. BPEL is designed specifically for Web service orchestration. However, none of the proposed standards addresses the issue of service composition reuse and specialization.

Work similar to what is reported in this article can be found in the area of workflow class inheritance. In [1] the author presents a framework to analyze the requirements for supporting workflow class inheritance. Different perspectives of inheritance are discussed and a workflow class definition language is proposed. In [2] the authors present a workflow class specification consisting of a set of object classes as well as a set of rules. Subclass and inheritance is supported, at least partially. A system called TOWE that implements a set of classes providing basic mechanisms for workflow execution is described in [3]. Normal workflow classes can then be developed by inheriting the functionality from these basic classes.

Although all these publications are relevant, the basic difference between the work conducted in the workflow area and what is proposed in this article lies in that all the constructs in the service component class can be considered as reuse points: they can be referred to both inside or outside the class definition. Consequently, these are the points that can be specialized and overridden.

Conclusion

A framework for analyzing Web service composition reuse and specialization has been introduced and described here. The service component concept was

proposed and used as a packaging mechanism for composing Web services. Once service component classes are defined, they can be reused and specialized by adding or overriding their definitions. Currently, a prototype system for service components and their underlying support framework is being implemented in Java. It is based on a set of widely accepted industry standards, including but not limited to WSDL, SOAP, and DOM. The prototype provides an integrated environment for service composition, which includes composition specification, planning, implementation, and execution. The resulting composite Web services can be published as new services and their configuration is stored in a repository and can thus be reused and extended. ■

REFERENCES

1. Bussler, C. Workflow class inheritance and dynamic workflow class binding. In *Proceedings of the Workshop Software Architectures for Business Process Management at the 11th Conference on Advanced Information Systems Engineering (CAiSE*99)*, Heidelberg, Germany, June 1999.
2. Kappel, G., Lang, P., Rausch-Schott, S., and Retschitzegger, W. Workflow management based on objects, rules, and roles. *Bulletin of the Technical Committee on Data Engineering* 18, 1 (Mar. 1995).
3. Papazoglou, M.P., Delis, A., Bouguettaya, A., Haghjoo, M. Class library support for workflow environments and applications. *IEEE Transactions on Computers* 46, 6 (June 1997).
4. Papazoglou, M.P. and Yang, J. Design methodology for Web services and business processes. In *Proceedings of the Third VLDB-TES Workshop*, (August, Hong Kong), Springer, 2002.
5. Yang, J. and Papazoglou, M.P. Service components: A substrate for Web service reuse and composition. In *Proceedings of 14th Conference on Advanced Information Systems Engineering (CAiSE02)*, Toronto, May 2002.

JIAN YANG (jian.yang@uvt.nl) is an assistant professor at Tilburg University in The Netherlands.

Part of this work has been funded by the Telematica Institute under the project PIEC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

²See www.wfmc.org/standards/docs/xpdl_010522.pdf.