Web Component: A Substrate for Web Service Reuse and Composition

Jian Yang and Mike. P. Papazoglou

Tilburg University, Infolab
PO Box 90153, 5000 LE, Tilburg, Netherlands
{jian,mikep}@kub.nl

Abstract. Web services are becoming the prominent paradigm for distributed computing and electronic business. This has raised the opportunity for service providers and application developers to develop value-added services by combining existing web services. Emerging web service standards and web service composition solutions have not addressed the issues of service re-use and extension yet. In this paper we propose the concept of *web component* that packages together elementary or complex services and presents their interfaces and operations in a consistent and uniform manner in the form of a class definition. Web components are internally synthesized out of reused, specialized, or extended elementary or complex web services. They are published externally as normal web services and can thus be employed by any web-based application.

1 Introduction

The Web has become the means for organizations to deliver goods and services and for customers to discover services that match their needs. By web service, we mean a self-contained, internet-enabled applications capable not only of performing business activities on its own, but also possessing the ability to engage other web services in order to complete higher-order business transactions. Examples of such services include catalogue browsing, ordering products, making payments and so on. The platform neutral nature of the web services creates the opportunity for building *composite services* by using existing elementary or complex services possibly offered by different enterprises. For example, a `travel plan` service can be developed by combining several elementary services such as `hotel reservation`, `ticket booking`, `car rental`, `sightseeing package`, etc., based on their WSDL description [13]. Web services that are used by a composite service are called *constituent services*.

Web service design and composition is a distributed programming activity. It requires software engineering principles and technology support for service reuse, specialization and extension such as those used, for example, in component based software development. Although web service provides the possibility for offering new services by specialization and extension instead of designing them from

A. Banks Pidduck et al. (Eds.): CAISE 2002, LNCS 2348, pp. 21–36, 2002.

scratch, to this date there is little research initiative in this context. In this paper we introduce the concept of *web component* to facilitate this very idea of web service reuse, specialization and extension.

Web components are a packaging mechanism for developing web-based distributed applications in terms of combining existing (published) web services. Web components have a recursive nature in that they can be composed of published web services while in turn they are also considered to be themselves web services (albeit complex in nature). Once a web component class is defined, it can be reused, specialized, and extended. The same principle applies to service composition activities if we view a composite service as a special web service, which contains composition constructs and logic.

Normally, composite services are developed by hard-coding business logic in application programs. However, the development of business applications would be greatly facilitated if methodologies and tools for supporting the development and delivery of composite services in a co-ordinated and effectively reusable manner were to be devised. Some preliminary work has been conducted in the area of service composition, mostly in aspects of workflow-like service integration [3], service conversation [7], and B2B protocol definition [1]. However, these approaches are either not flexible or too limited as they lack proper support for reusability and extensibility.

Services should be capable of combination at different levels of granularity, while composite services should be synthesized and orchestrated by reusing or specializing their constituent services. Therefore, before complex applications can be built on simple services or composite services, we need to look at a fundamental aspect of composition: *composition logic*. Composition logic dictates how the component services can be combined, synchronised, and co-ordinated. Composite logic is beyond conversation logic (which is modeled as a sequence of interactions between two services) and forms a sound basis for expressing the business logic that underlies business applications.

We use the concept of web component as a means to encapsulate the composition logic and the construction scripts which oversee the combination of existing web services. These constructs are private (non-externally visible) to a web component. The public interface definition provided by a web component can be published and then searched, discovered, and used in applications as any other normal web service. Web components can also serve as building blocks to construct complex applications on the basis of reuse and extension.

In this paper we will concentrate on how web components are used for composite service planning, definition and construction. The contribution of this paper is three-fold:

- it proposes the concept of a web component for creating composite services and web service re-use, specilization, and extension;
- it also proposes a light-weight service composition language that can be used as the script for controlling the execution sequence of service compositions. Since this language is expressed in XML it can be exchanged easily across the network;

– Finally, it provides a complete framework for web service composition so that composite web services can be planned, defined, and invoked on the basis of web components.

The paper is organized as follows. Section 2 presents a framework for service composition. Section 3 discusses different forms of service composition so that basic composition logic can be derived, and introduces the Service Composition Specification Language (SCSL) that defines a web component and provides implementation scripts. Section 4 outlines the features of the Service Composition Planning Language (SCPL) and Service Composition Execution Graphs (SCEG). In section 5 we demonstrate how SCSL, SCPL, and SCEG work together to fulfill the tasks of planning, design, implementation, and execution of composite web services. Section 6 presents related work and summarizes our main contributions. Finally, section 7 concludes the paper.

2 Service Composition: Technical Challenges

In this section we will first analyze the nature of service composition, provide a framework for service composition and application development based on web services. Subsequently, we illustrate the characteristics of composition logic which lays the foundation for creating web components.

2.1 A Framework for Service Composition

The real challenge in service composition lies in how to provide a complete solution. This means to develop a tool that supports the entire life cycle of service composition, i.e., discovery, consistency checking and composition in terms of re-use and extendibility. This comes in contrast to the solutions provided by classical workflow integration practices, where service composition is pre-planned, pre-specified, has narrow applicability and is almost impossible to specialise and extend.

Service composition spans three phases: (1) planning, (2) definition, and (3) implementation. By *planning*, we mean the candidate services (elementary or composite) that are discovered and checked for composability and conformance. During this phase alternative composition plans may be generated and proposed to the application developer. The outcome of this phase is the synthesis of a composite service out of desirable or potentially available/matching constituent services. At the *definition* phase, the actual composition structure is generated. The output of this phase is the specification of service composition. Finally, the *implementation* phase implements the composite service bindings based on the service composition specification. The following types of service composition are used throughout these phases:

– *Explorative composition*: service composition is generated on the fly based on a customer (application developer's) request. The customer describes the

desired service, the service broker then compares the desired composite service features with potentially matching published constituent service specifications and may generate feasible (alternative) service composition plans. These plans result in alternative service compositions that can be ranked or chosen by service customers depending a criteria such as availability, cost and performance. This type of service composition is specified on the fly and requires dynamically structuring and co-ordination of constituent services.
- *Semi-fixed composition:* Here some of the actual service bindings are decided at run time. When a composite service is invoked, the actual composition plan will be generated based on a matching between the constituent services specified in the composition and the possible available services. In this case, the definition of the composite service is registered in an e-marketplace, and it can be used just as any other normal service, i.e., it can be searched, selected, and combined with other services.
- *Fixed composition:* a fixed composite service synthesizes fixed (pre-specified) constituent services. The composition structure and the component services are statically bound. Requests to such composite services are performed by sending sub-requests to constituent services.

We can conclude that the following main elements are needed to develop a complete solution for service composition: (1) service request description, (2) service matching and compatibility checking, (3) description of service composition, and (4) service execution monitoring and coordination. In this paper we shall concentrate on the first three items in some detail.

2.2 Composition Logic

Composition logic we refer to the way a composite service is constructed in terms of its constituent services. Here, we assume that all publicly available services are described in WSDL. Composite logic has the following two features:

- *Composition type:* this signifies the nature of the composition and can take two forms:
  - *Order:* indicates whether the constituent services in a composition are executed in a serial or parallel fashion.
  - *Alternative service execution:* indicates whether alternative services can be invoked in a service composition. Alternative services can be tried out either in a sequential or in a parallel manner until one succeeds.
- *Message dependency:* indicates whether there is message dependency among the parameters of the constituent services and those of the composite service. We distinguish between three types of necessary message dependency handling routines in a composition:
  - *message synthesis:* this construct combines the output messages of constituent services to form the output message of the composite service.
  - *message decomposition:* this construct decomposes the input message of the composite service to generate the input messages of the constituent services;

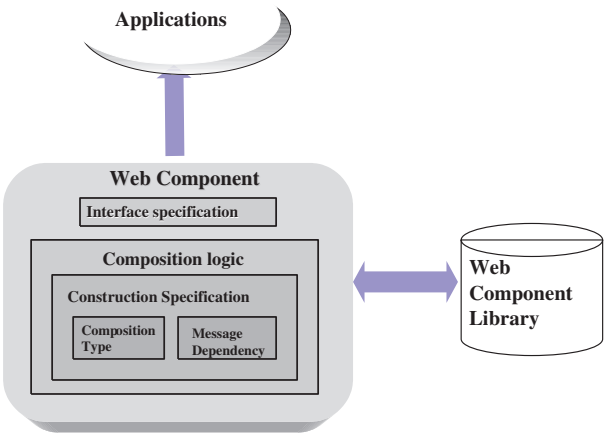

Fig. 1. Web component ingredients

- *message mapping:* it specifies the mappings between the inputs and outputs of the constituent services. For example, the output message of one component service is the input message of another service.

Examples of message dependency are given in Section 3.3 and Figure 3. The composition types together with message dependency constructs form the basis of composition logic, which is specified in web components. Figure 1 depicts the ingredients of a web component. It illustrates that a web component presents a single *public interface* to the outside world in terms of a uniform representation of the futures and exported functionality of its constituent services. It also internally specifies its *composition logic* in terms of *composition type* and *message dependency* constructs. The process of web service composition design becomes a matter of reusing, specializing, and extending the available web components. This enables a great deal of flexibility and reusability of service compositions.

3 Creation of Web Components

In this section, we first present the basic constructs for expressing composition logic, then we demonstrate how web components are specified in SCSL and in terms of web component classes.

3.1 Basic Constructs for Service Composition

The following constructs have been identified to serve as the basis for compositions [12]:

1. **Sequential service composition (sequ).** In this case, the constituent services are invoked successively. The execution of a constituent service is

dependant on its preceding service, i.e., one cannot begin unless its preceding service has committed. For example, when a composite travel plan service - composed of an `air ticket reservation service`, a `hotel booking service`, and a `car rental service` - makes a travel plan for a customer, the execution order should be `hotel booking`, `air ticket reservation`, and `car rental`. The invocation of the `hotel booking` service is dependent on the successful execution of the `air ticket reservation` because without a successful air ticket reservation, hotel booking can not go ahead.

2. **Sequential alternative composition (seqAlt).** This situation indicates that alternative services could be part of the composition and these are ordered on the basis of some criterion (e.g., cost, time, etc). They will be attempted in succession until one service succeeds.
3. **Parallel service composition.** In this case, all the component services may execute independently. Here two types of scenarios may prevail:
   (a) **Parallel with result syncronization (paraWithSyn).** This situation arises when the constituent services can run concurrently, however, the results of their execution need to be combined. For example, purchasing a PC may involve sending inquiries to different companies which manufacture its parts. These inquires may run in parallel, however, they all need to execute to completion in order to obtain the total configuration and price.
   (b) **Parallel alternative composition (paraAlt).** In this situation alternative services are pursued in parallel until one service is chosen. As soon as one service succeeds the remainder are discarded.

Although these constructs represent the most common cases of service composition, to make the composition logic complete, we also need to introduce two additional control constructs: **condition** and **while_do**. The former is used to decide which execution path to take, while the latter is a conventional iteration construct.

The various composition types may result in different message dependencies and therefore require different message handling constructs. Table 1 summarizes the message dependency handling constructs required for different types of service composition.

The basic composition types together with message dependency handling constructs provide a sound basis for forming the composition logic in a web component.

Similar basic routing mechanisms such as *sequential* and *parallel* for use in distributed computations and workflows can be found in [9,10]. The main difference between this work and ours is that they provide basic constructs for control flow execution, whereas we use them as part of the abstraction mechanism for defining web service interfaces and the composition logic for building composite services.

Table 1. Message handling used in different types of composition

	messageSynthesis	messageDecomposition	messageMapping
sequ	X	X	X
seqAlt			X
paraWithSyn	X	X	X
paraAlt			X
condition			X
while_do			X

3.2 The Web Component Class Library

Web component classes are abstract classes used as a mechanism for packaging, reusing, specializing, extending and versioning web services by converting a published WSDL specification into an equivalent object-oriented notation. Any kind of web service (composite or not) can be seen as a web component class provided by an organization and can be used in the development of distributed applications.

The web component class library is a collection of general purpose and specialized classes implementing the primitives and constructs discussed in the previous section. Classes in the web component library act as abstract data types i.e., they cannot be instantiated. They provide basic constructs and functionality that can be further specialized depending on the needs of an application. A distributed web application can be build by re-using, specializing, and extending the web component library classes. The web component library classes can be categorized as follows:

- *web service construction class:* this abstract class is used for creating definitions out of WSDL specifications. This construction class will generate a web component class for a registered web service defined in WSDL, i.e., the service messages, interface and implementation.
- *web component class:* this class represents all elementary or composite web services. There are six subclasses in this category which implement the composition constructs discussed in the previous section (see Table-1).
- *application program class:* this class is used to develop application programs by using the web component classes. Since application program classes are web components, they also can be reused, specialized, and extended.

Figure 2 the web component class definition for a `travelPlan` service. We assume that a travel plan is a composite service which combines the two services `hotelBooking` and `ticketReservation` which are published by the two service providers `Paradise` and `DisneyLand`, respectively. In this figure, class `TravelPlan` is defined as a subclass of class `sequ` which provides the basic operations for the sequential type of composition and message dependency handling. The TripOrderMessage in the travelPlan interface includes such information as the dates, the type of rooms, the location, and the preferred airline for

the trip. These are the combined input messages from the WSDL specification of the `hotelBooking` and `ticketReservation` web services. In the construction specification of the `travelPlan` service the TripOrderMessage is shown to be decomposed to its constituent messages `Paradise.HotelBookingMsg` and `DisneyLand.TicketResMsg` which need to be executed at the `Paradise` and `DisneyLand` service provider sites, respectively.

```
class TravelPlan is sequ {
  public  TripOrderMessage tripOrderMsg;
  public  TripResultMessage tripResDetails;
  public  travelPlanning(TripOrderMessage) -> TripResultMessage;
  private void compose(Paradise.makeBooking, DisneyLand.makeRes);
  private void messageDecomposition(TravelPlan.TripOrderMsg,
                                    Paradise.HotelBookingMsg,
                                    DisneyLand.TicketResMsg);
  private void messageSynthesis(TravelPlan.TripResDetails,
                                     Paradise.HotelBookingDetails,
                                     DisneyLand.E-ticket);
}
```

Fig. 2. Web component class definition for travelPlan service

A web component is specified in two isomorphic forms: a class definition (discussed above), and an XML specification in terms of a Service Composition Specification Language (SCSL). Interacting web components and services (across the network) can only communicate on the basis of exchanging XML Web service specifications and SOAP messages. Thus although web component classes serve as a means for specification and reusability they need to be converted to an equivalent XML representation in order to be transmitted across the network. For this purpose we use the SCSL.

3.3 Web Component Specification in XML

There are two parts in SCSL: the interface of the composite service specified in its `defn` part and the construction of the composition logic is specified in its `construct` part, (see Figure 3). These are isomorphic to the interface and construction parts of a web component shown in Figure 2. The `construct` part of an SCSL specification consists of a `compositionType`, a series of activities, and message handling constructs. Activities are internal (non-visible) elementary tasks that need to be performed to achieve a certain web component operation. These are executed remotely in the web sites hosting the web service constituents. The composition type in SCSL specifies the nature of activity execution according to the discussion in section 3.1, while message handling specifies how service and activity messages are processed.

```
<webService name="travelPlan">
<!---- Message definition ==-->
  <definition>
    <message name="tripOrderMsg">
      <part name="hotelBookingMsg" element="hotelBookingMsg"/>
      <part name="ticketResMsg" element="ticketResMsg"/>
    </message>
    <message name="tripResDetails">
      <part name="hotelBookingDetails" element="hotelBookingDetails"/>
      <part name="e-ticket" element="e-ticket"/>
    </message>
  </definition>
<!-== The composite service interface definition ==-->
  <defn>
    <portType name="travelPlaner">
      <operation name="travelPlanning">
        <input message="tripOrderMsg"/>
        <output message="tripResDetails"/>
      </operation>
     </portType>
  </defn>
<!---- The composite service implementation details ==-->
  <construct>
    <composition type="sequ">
      <activity name="hotelBooking">
        <input message="hotelBookingMsg"/>
        <output message="hotelBookingDetails"/>
        <performedBy serviceProvider="Paradise"/>
        <use portType="hotelBookingHandler" operation="makeBooking"/>
       </activity>
       <activity name="ticketReservation">
         <input message="ticketResMsg"/>
         <output message="e-ticket"/>
         <performedBy serviceProvider="Disney Land"/>
         <use portType="ticketResHandler" operation="makeRes"/>
        </activity>
        <messageHandling>
          <messageDecomposition>
            <source message="tripOrderMsg"/>
            <target message="hotelBookingMsg"/>
            <target message="ticketResMsg"/>
          </messageDecomposition>
          <messageSynthesis>
            <source message="hotelBookingDetails"/>
            <source message="e-ticket"/>
            <target message="tripResDetails"/>
           </messageHandling>
    </composition>
  </construct>
</webService>
```

Fig. 3. Service Composition Specification Language (SCSL)

In SCSL we adopt the same convention as WSDL [13], i.e., the `portType` is used for grouping `operations`. Operations represent a single unit of work for the service being described. The example of `travelPlan` illustrated in Figreu 3 corresponds to the web component class isslustrated in Figure2 and provides a single `portType` named `travelPlaner` with one operation `travelPlanning`. The activity `hotelBooking` uses the operation `makeBooking` of port type `hotelBookingHandler`. The activity `ticketReservation` uses the operation `makeRes` of port type `ticketResHandler`.

We also specify how input and output messages of constituent services operations are linked from (to) those of the composite service. Here we rely on the three types message handling: (1) message synthesis, (2) message decomposition, and (3) message mapping described in section 2.2. For example, the output message `hotelBookingDetails` of the constituent operation `makeBooking` and the output message `e-ticket` of the constituent operation `makeRes` are composed into the output message `tripResDetails` of the composite service `travelPlan` in the `messageSynthesis` part. The input message of the composite service `travelPlan` called `tripOrderMsg` is decomposed into two messages: the input message `hotelBookingMsg` of constituent operation of `makeBooking` and input message `ticketResMsg` of constituent operation `makeRes`.

Although the above example is meant for sequential compositions, the other types of composition can be specified in a similar fashion. Note Figure 3 is a much simplifier version of the SCSL for illustrative purposes. The binding specifications are not included in this figure.

The XML schema of SCSL is not provided for reasons of brevity.

4 Service Composition Planning and Composition Execution Languages

As discussed in section 2, service composition should be planned and generated according to service developer's request. With this in mind, we are developing a **Service Composition Planning Language (SCPL)** that specifies how a composite service is built up in terms of the relationships among constituent services such as execution order and dependency. The resulting specification combines services from the web component library and presents them in the form of web component classes as described in the previous. These specifications will subsequently generate a service execution structure in the form of a **Service Composition Execution Graph (SCEG)**. When the SGEC is executed it invokes the corresponding services at remote site and co-ordinates them. In the following, we will first introduce the concepts of SCPL and SCPG by means of a simple example then we will present their formal specifications.

Figure 4 illustrates how a composite service called `HolidayPlan` can be planed in SCPL by combining three component services `restaurantBooking`, `hotelReservation`, and `sightseeing`. In this example, we specify that `restaurantBooking` and `hotelReservation` have to run sequentially, and there is data dependency between them, i.e., the location of the hotel determines the

```
Composition holidayPlanning
  C1: sequ (hotelReservation, restaurantBooking)
      mapping (hotelBooking.location = restaurantBooking.location)
  C2: paraWithSyn (C1, sightseeing)
```

Fig. 4. Specification of a service composition plan

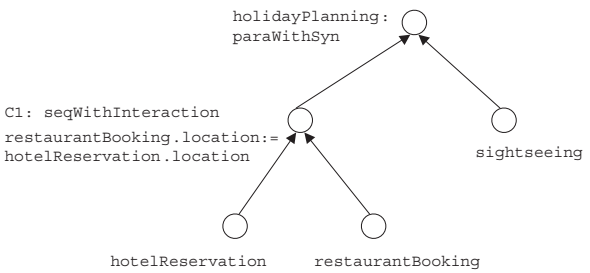

Fig. 5. The service composition execution graph

location of the restaurant, while `Sightseeing` can run in parallel with the other two services.

There are two aspects of SCPL which make it flexible and extensible: (1) a labelling system which can be used to label any composition. The labels can be used in any place where services (operations) are required. We can build a composition plan recursively by labelling existing composition plans. (2) plan variables and macros which can be used in the place where the service (operation) and composition types (such as sequential, paraWithSyn etc) are required. This second aspect is not discussed in this paper due to space limitations. SCPL provides a simple light-weight but powerful mechanism for service composition nesting and substitution, composition extension, and dynamic service selection.

A Service Composition Execution Graph (SCEG) is generated on the basis of an SCPL specification. Figure 5 illustrates the SCEG corresponding to the SCPL specification in Figure 4 .

Formally an SCEG is a labelled DAG $P =< N, A, spe >$ where N is a set of vertices, A is a set of arcs over N, such that

- for every operand and label in a statement in SCPL, create a vertex;
- for $v \in V$, $spe(v)$ represents the type of composition and the mapping specification;
- if u is used in v for composition, introduce an arc $u \rightarrow v$.

The service composition execution graph is used for coordinating constituent service execution at remote sites and run time service selection based on the user requirement specifications. These issues are further discussed in the next section.

5 Service Composition: A Complete Picture

In this section, we explain how the SCSL, the SCPL and the SCEG work together to create service compositions. Recall that as already explained in section 2, there are three stages of service composition: planning, definition, and implementation. We address these in the following.

The *planning* stage involves service discovery, composability and compatibility checking. The output of this stage is a composition plan specified in SCPL. Assume we have the following request:

```
<request>
  <from src="http://www.infolab.nl/jian />
  <vocabulary name="holiday planning" />
  <service name="ticket booking" />
  <service name="hotel booking" />
  <service name="restarurant booking", option="optional" />
  <service name="sightseeing" />
  <result> $serviceInfo </result>
  <condition condition="ticketBooking.date=hotelBooking.date">
  <condition condition="restarurantBooking.loc=hotelBooking.place"
</request>
```

This request can be satisfied if we find the services which match the required constituent services either completely or partially. For service discovery, it is important to find an appropriate service with the right capability. Service discovery relies on the following steps:

- Semantic relatedness: during this step, the requested service is compared against the service description in the repository in terms of service contents to decide how closely related they are. Services with a high degree of relatedness will be selected as relevant services for further capability checking.
- Capability analysis: the capabilities of the services selected from the previous step are checked in terms of the functions they provide to determine whether they can accomplish completely or partially the tasks of the requested service.
- Syntactic analysis: "capable" services have their syntax of their interfaces checked to determine how they can be combined to achieve the requested higher-order service.

We can view a web service (S) as a triple: $< C, A, P >$ where C, A, P stand for contents, activities (capabilities), and properties respectively. Contents refer to what the service is about. Activities are a set of operations the service provides. Properties refer to some end point information about the service such as payment methods, cost, etc. C is used in conjunction with semantic relatedness checks, A is used in capability and syntax check, while P is used for selecting alternative composition plans.

We can identify two types of checking depending on the nature of composition: *compatibility checking* and *conformance checking*. Service $S1$ is compatible with $S2$ when $S1$ is at least as capable as $S2$ and $S1$ can substitute $S2$. Service S conforms to S' when S and S' can be combined in a way that the output of S can be taken as the input of S'. Here, we introduce two symbols: $\diamond$ for "compatibility" and $\triangleright$ for "conformance". As P does not play an important role in service discovery, we only consider C and A for the purpose of semantic and syntactic checking.
Service $S =< C, A, P >$, where $\forall a \in A$, we define $a =< op, I, O >$, where op, I, and O stand for operation, inputs and outputs respectively. For input, we have $I =< p_1, \ldots, p_m >$, and for output, we have $O =< q_1, \ldots, q_n >$., where every p_i $(i = 1 \ldots m)$ and q_j $(j = 1 \ldots n)$, takes the form $< name >:< type >$.

Definition 1. *Service S' is compatible with S ($S' \diamond S$) if the contents of S are a subset those of S' ($S.C \subset S'.C$) and the operations of S' are compatible with those of S ($S'.A \diamond S.A$).*

Definition 2. *Activities in Service S' are compatible with the activities in service S ($S'.A \diamond S.A$) when $\forall a \in S.A$, if we can find an operation $a' \in S'.A$ such that $a' \diamond a$.*

Definition 3. *Operations $a' \diamond a$ if*
(1) the pre-condition and the post-condition of $a'.op$ are equivalent to $a.op$,
(2) the inputs $a'.I \diamond a.I$ and
(3) the outputs $a'.O \diamond a.O$.

Definition 4. *S' conforms to S ($S' \triangleright S$) if:*
(1) the the contents $S'C$ and $S.C$ are overlapping and
(2) $\exists a' \in S'.A, \exists a \in S.A$ such that $a'.O \diamond a.I$.

There is still a lot of research that needs to be done in the area of compatibility and conformity checking. However this is beyond the scope of this paper. Some primary research results can be found in [11,8].

To exemplify these issues, we use the above service request `holidayPlanning` as an example and assume we have a choice between the following two plans specified in SCPL after semantic and capability checking:

```
CompositionPlan1 holidayPlanning
C1: sequential (ticketBooking, hotelBooking, restaurantBooking)
    Mapping (ticketBooking.arrive_date=hotelBooking.date,
             RestarurantBooking.loc=hotelBooking.place)
C2: paraWithSyn (C1, sightseeing)
  Sythesizing (holidayPlanning.schedule=C1.schedule+sightseeing.schedule)

CompositionPlan2 holidayPlanning
C1: sequential (travelPlan, restaurantBooking)
    Mapping (RestarurantBooking.loc=hotelBooking.place)
C2: paraWithSyn (c1, sightseeing)
  Sythesizing (holidayPlanning.schedule=C1.schedule+sightseeing.schedule)
```

CompositionPlan1 contains three services and defines two mappings. The first mapping indicates that the arrival date must be the same as the hotel check-in date. The second mapping indicates that the restaurant and the hotel must be located at the same place. CompositionPlan2 contains two services one of which is a composite service defined and constructed in Figure3. The mapping ticketBooking.arrive_date=hotelBooking.date is assumed to be accomplished by the composite service travelPlan.

To choose among alternative composition plans generated by the planning stage, we need look at the properties of the candidate constituent end point services (such as cost, performance, binding requirements). Suppose that CompositionPlan1 is selected, we can then use the sequ and paraWithSyn web component classes to *define* and *construct* the plan in an incremental fashion. We first generate C1 by using the web component class sequ as a super-class and further extend it with new message types and operations. Then we use the web component class paraWithSyn as a super-class to construct an application program class holidayPlanning by linking C1 together with another service sightseeing and specifying the appropriate message dependency handlings. The final class definition then can be transformed into SCSL. The code in Figure 6 illustrates how the web component class holidayPlanning is defined.

To *execute* a composite service, an SCEG graph is generated. As already stated in Section 4, the SCEG is a labelled DAG. Every node in this graph is a composite service with its children representing constituent services. The root node is the application we want to build. The type of composition and the message dependency is indicated in the label of the node. The node in the SCEG bind to and execute web services at different sites while the overall control is situated at the site which launches the application. The algorithm for SCEG execution has been developed on basis of the *depth-first search*.

```
class C1 is sequential {
  public TravelSchedule travelSchedule;
    ... //public operations
  private void compose(TicketBooking, HotelBooking, RestaurantBooking);
  private void messageMapping(TicketBooking.date, HotelBooking.date);
  private void messageMapping(RestaurantBooking.location,
                              HotelBooking.location);
}
class HolidayPlanning is paraWithSyn {
  public HolidaySchedule holidaySchedule;
    ... //public operations
  private void compose(C1, Sightseeing);
  private void messageSynthesizing(HolidayPlanning.holidaySchedule,
                C1.TravelSchedule, Sightseeing.Schedule);
}
```

Fig. 6. An application program class HolidayPlanning

6 Related Work

Most of the work in service composition has focussed on using work flows either as a engine for distributed activity coordination or as a tool to model and define service composition. Representative work is described in [2] where the authors discuss the development of a platform specifying and enacting composite services in the context of a workflow engine. The eFlow system provides a number of features that support service specification and management, including a simple composition language, events and exception handling.

The workflow community has recently paid attention to configurable or extensible workflow. The approach described in [6] allows for automatic process adaptation. The authors present a workflow model that contains a placeholder activity, which is an abstract activity replaced at run-time with a concrete activity type. This concrete activity must have the same input and output parameter types as those defined as part of the placeholder. In addition, the model allows to specify a selection policy to indicate which activity should be executed.

The work presented in [5] proposes some interesting ideas in workflow interoperation. It provides infrastructure to support dynamic aspects in planning, scheduling, and execution by introducing workflow schema templates. Reuse of existing workflow schema and templates can be achieved by schema splicing. However how this approach can be used in service composition is not clear.

The workflow approaches provide some basic mechanisms that can be used for supporting dynamic service co-ordination and composition. However as the authors pointed out in [1,4], workflow systems do not cater for the dynamic and distributed nature of service composition for two reasons: (1) a common workflow modelling and management environment is impossible to achieve especially across different enterprises since no WFMS vendor shares the same workflow syntax and semantics; (2) workflow systems do not offer facilities such as changing flow definitions which is a fundamental requirement for service composition. Therefore, these solutions may work only for semi-fixed and fixed compositions, however, they do not work well with explorative composition which requires the service composition structure to be generated on the fly and the composition itself to be changeable. Moreover, they do not support parameterization, reuse, specialization, and nesting of service compositions.

Our approach differs from the above in the following ways:

- in this paper we propose an integrated approach towards service composition, which includes composition planning, specification, implementation and execution.
- The concept of web component is introduced for web service reuse, specialization, and extension.
- At the planning stage, variables and macros can be introduced in the SCPL which can be used for service substitution.
- Unlike workflow schemas SCSL is a light-weight specification language in XML which can be executed in different organizational settings without too much implementation overhead.

7 Conclusion

It is obvious that service composition is not just an interoperability problem. The real challenge in service composition is how to provide a complete solution in terms of tools that support the entire cycle of service composition, i.e., discovery, consistency checking, composition, re-use, and extendibility.

In this paper, we presented a framework to discuss the different forms of service composition and their essential characteristics. Based on this framework, we presented an approach for composition planning, definition, implementation, and execution. In order to support the need for flexible, scalable, extensible service composition, we introduced the concept of web component that packages together elementary or complex services and presents their interfaces and operations in a consistent and uniform manner in the form of class definitions. This approach is light weight, flexible, and leads to reusable web components when compared with current popular workflow solutions.

References

1. C. Bussler. The Role of B2B Protocols in Inter-Enterprise Process Execution. Proc. Of the 2nd VLDB-TES Workshop, Rome, 2001. 22, 35
2. F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, M. C. Shan. Adaptive and Dynamic Service Composition in eFlow, *HP Lab. Techn. Report, HPL-2000-39* . 35
3. F. Casati and Ming-Chien Shan. Dynamic and adaptive composition of e-services, *Information Systems*, 26(2001), page 143-163, 2001. 22
4. F. Casati, M. Sayal, and M. C. Shan Developing E-Services for Composing E-Services. Proc. Of the 13th CAiSE conference, Switzerland, 2001 35
5. V. Christophides, R. Hull, A. Kumar, and J. Simeon Workflow Mediation using VorteXML. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2000. 35
6. D. Georgakopoulos, H. Schuster, D. Baker, and A. Cichocki. Managing Escalation of Collaboration Processes in Crisis Mitigation Situations. Proceedings of ICDE 2000, San Diego, CA, USA, 2000. 35
7. H. Kuno, M. Lemon, A. Karp, and D. Beringer. Conversations + Interface = Business Logic. Proc. Of the 2nd VLDB-TES Workshop, Rome, 2001. 22
8. M. Mecella, B. Pernici, and P. Craca. Compatibility of e-Services in a Cooperative Multi-platform Environment. Proc. Of the 2nd VLDB-TES Workshop, Rome, 2001. 33
9. M. P. Papazoglou, A. Delis, A. Bouguettaya, M. Haghjoo. "Class Library Support for Workflow Environments and Applications". IEEE Transactions on Computer Systems , vol. 46, no.6, June 1997. 26
10. W. M. P. van der Aalst and A. Kumar. XML Based Schema Definition for Support of Inter-organizational Workflow. Information System Research (accepted) 26
11. W-J Van Heuvel, J. Yang, and M. P. Papazoglou. Service Representation, Discovery, and Composition for E-Marketplaces, *Proc. Of International Conference on Cooperative Information Systems (cooPIS01)*, Sep, 2001. 33
12. J. Yang, M. P. Papazoglou, and W-J Van Heuvel. Tackling the Challenges of Service Composition. Proc. of ICDE-RIDE workshop, San Jose, 2002. 25
13. Web Service Definition Language. http://www.w3.org/TR/wsdl. 21, 30