Service Components for Managing the Life-Cycle of Service Compositions [1]

Jian Yang and Mike. P. Papazoglou
Tilburg University, INFOLAB,
Dept. of Information Systems and Management,
PO Box 90153
5000 LE, Tilburg
Netherlands
{jian,mikep}@uvt.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. However, current web service composition solutions do not address software engineering principles for raising the level of abstraction in web-services by providing facilities for packaging, re-using, specializing and customizing service compositions.

In this paper we propose the concept of *service component* that packages together complex services and presents their interfaces and operations in a consistent and uniform manner in the form of an abstract class definition. Service components are internally synthesized out of reused, specialized, or extended complex web services and just like normal web services are published and can thus be invoked by any service-based application. In addition, we present an integrated framework and prototype system that manage the entire life-cycle of service components ranging from abstract service component definition, scheduling, and construction to execution.

Keywords: web services, service scheduling and execution, service composition life-cycle, composition logic, service components, service reusability.

1 Introduction

The Web has become the means for organizations to deliver goods and services and for customers to search and retrieve services that match their needs. Web services are self-contained, Internet-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. Simple web services may provide simple functions such as credit checking and authorization, inventory status checking, or weather reporting, while complex services may appropriately unify disparate business functionality to provide a whole range of automated processes such as insurance brokering, travel planning, insurance liability services or package tracking. Several software vendors and consortia are providing platforms (such as IBM's WebSphere, or Microsoft's .NET), languages and description models for service representation and discovery such as WSDL

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

and UDDI, which offer uniform representation of and access to web services, respectively. The platform neutral nature of web services creates the opportunity for building *composite services* by combining existing elementary or complex services, possibly offered by different enterprizes. 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 [27]. We use the term *composite service* to signify a service that employs and synthesizes other services. The services that are used in the context of a composite service are called its *constituent services*.

The current standard web service definition language is WSDL [27]. WSDL consists of the following main constructs: `messages, portTypes, bindings, ports`. `portTypes` define the interface of a web service in terms of operations and their input and output messages. `Bindings` provide the implementation of a web service by specifying the protocol(s) used for service invocation, e.g., they can be SOAP binding, HTTP GET/POST binding, or MIME binding. `Messages` define the structures and types of the inputs and outputs of operations. A service consists of a set of `ports` that define the bindings used in the service.

Web service applications developed in terms of WSDL alone are isolated and opaque and moreover, cannot be inter-linked to express the business semantics of web services. Breaking this isolation requires connecting web services and specifying how collections of web services work jointly to realize more complex functionality typified by business processes.

The recently proposed standard Business Process Execution Language for Web Services (BPEL4WS or BPEL for short) [31] (which suprceeds both WSFL and XLANG specifications) is an XML-based specification language that addresses the above problem by supporting the definition of a new web service in terms of compositions of existing (constituent) services. BPEL models the actual behaviour of a participant in a business interaction as well as the visible message exchange behaviour of each of the parties involved in the business protocol. A BPEL process is defined "in the abstract" by referencing and inter-linking portTypes specified in the WSDL definitions of the web services involved in a process.

Web service design and composition is a distributed programming activity. It requires in addition to BPEL specifications software engineering principles and technology support for service reuse, extension, specialization and inheritance such as those used, for example, in component based software development [23]. Despite the fact that web service technology offers the potential for deriving new services and applications on the basis of service extension, specialization and parameterization, to this date there is little research initiative in this area.

Currently, the web service application space, 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. For example, in order to obtain contextual information, e.g., all services relating to a

complex service called `travel plan`, all relevant service APIs need to be appropriately inter-linked. This service inter-linking is particular to the logic of the application using the `travel plan` service. If, for instance, the requirements of the application change or need to be adjusted, then a service like `travel plan` will have to be re-specified and recreated by possibly inter-linking additional or modified service interfaces. This approach leads to a proliferation of service APIs and results in unmanageable and cluttered solutions.

To address this limitation of web service technology, we introduce in this paper the concept of *service component*. Aim of a service component is to raise the level of abstraction in web services by modularising synthesized service functionality and by facilitating web service reuse, extension, specialization and service inheritance. 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, parameterised, customized, and generally inherited, to assist in the creation of new applications. Service components package together a number of related service messages and functionality, provided by diverse service providers, into a self-contained software module (called the *service component class*). This module exposes a well-defined interface and contains the business (or composition) logic that is responsible for inter-linking the service message and operation interfaces of the disparate services contained in it. In contrast to the service component interface that is public, the business logic that helps inter-link the services contained in a service component is private. Therefore, service components can be encapsulated (made discrete) and can be connected together to create more complex, highly-functional applications by means of reuse, extension, restriction, parameterization or specialization.

The paper is organized as follows. Section 2 presents the concept of service component while section 3 presents a framework for service composition that supports the phases of service composition and facilitates the development of applications and composite services in terms of service components. Section 4 discusses the different aspects of service composition, explains how basic composition logic can be derived, and introduces the Service Composition Specification Language (SCSL) that provides an XML-based implementation script for service component classes. Section 5 outlines the features of the Service Scheduling Language (SSL) and Service Composition Execution Graph (SCEG) that are used for managing the scheduling and execution of service activities, respectively. In section 6 we describe how SCSL, SSL and SCEG work together to fulfil the service life-cycle phases of definition, scheduling, construction and execution. Prototype development activities are discussed in Section 7, while Section 8 presents related research work and summarizes our main contributions. Finally, section 9 concludes the paper and presents future research directions.

2 Service components

Normally composite services are developed by hard-coding business logic into application programs. The development of business applications would be greatly facilitated if methodologies and tools for supporting the development and delivery of composite services in a coordinated and effectively reusable manner were to be devised. Some preliminary work has been conducted in the area of service composition, mostly in aspects of composition modelling and workflow- like service integration [5], service conversation [15], and B2B protocol definition [4]. However, these approaches are either not flexible or too limited as they lack proper support for reusability, extensibility, and specialization.

2.1 General Characteristics

Services should be capable of combination at different levels of granularity. Furthermore, composite services should be able rely on facilities that provide support for service synthesis and orchestration on the basis of constituent service reuse, inheritance and specialization. Therefore, before complex applications can be built out of composite services, we need to address a fundamental aspect of service composition: *composition logic*. Composition logic dictates how the component services can be combined, synchronized, and co-ordinated. Composition logic forms a sound basis for expressing the business logic that underlies business applications. In current web service standards and implementations, e.g., the BPEL, composition logic is kept apart from its associated service interface specifications.

In the service component framework we present in this paper, we remedy this situation: process structure, partner roles, portTypes, etc, which are all specified in BPEL, are encapsulated as composition logic and together with their associated service interface specifications are uniformly represented as an abstract service component class. The interface is used to specify the publicly accessible behaviour of the service component. The service component interface is a set of operations offered to the service component invokers. The service component interface is constructed out of existing constituent service interfaces or extensions/specializations thereof. A service component can thus be viewed as a high-level self-contained composite service that presents a public interface and includes a private part comprised of the composition constructs and logic that are required for its manifestation. The public interface definition provided by a service component describes its messages and operations. The service component messages and operations can be published and then searched, discovered, and used just like any normal web service. The encapsulated composition logic and construction scripts that administer the combination of distributed web service messages and operations into a unique service composition class are private (internal and thus non-visible) to a service component.

A delivered service component can be specialized or extended to accommodate new desirable

functionality or to allow selective modification of already existing functionality. This concept is known as *dynamic inheritance* [12]. In summary, in contrast to conventional web services, a service component follows object-oriented principles such as encapsulation (for a business concept, be it entity or process), instantiation, generalization and specialization (not currently supported by web service technology) and plugs into an environment that provides a compatible "socket".

2.2 Representing service components as abstract classes

Service components are a packaging mechanism for developing web service based distributed applications in terms of combining existing (published) web services. Service 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).

To be able to reuse, specialize, and extend services, we rely on the definition of a service component class. A service component can be specified in two isomorphic forms: the definition of an abstract *service component class* and an equivalent *XML version* that corresponds and conforms to the class definition of a service component. WDSL/BPEL compliant service components are defined in terms of an XML-based sub-language, which we name *Service Composition Specification Language (SCSL).*

The SCSL version of a service component is used to describe in XML/WSDL concrete service components, mappings to and between constituent services contained in service components and, in general, implementing executable processes out of abstract class definitions. These issues are of no concern to an application developer who wishes to see and use a high-level abstract specification of services and processes free of mapping and implementation details. Consequently, the abstract service component class definition is used for reuse, extension, specialization, and versioning purposes, whereas its corresponding XML/WSDL, form, viz. SCSL, is used for construction purposes, distributed message exchange, remote service invocation and execution, as well as for communication across the network.

The *abstract* class definition of a service component takes the following form:

```
ServiceComponentclass ServiceComponentName {
  Definition
    ** message and operation definitions **
  Construction
    ** how activities are scheduled **
  PortType
    ** link activities with portType specified in WSDL **
  MessageHandling
    ** define how messages are decomposed, composed, and mapped **
  Provider
```

```
    ** link portTypes with service providers **
}
```

The class definition for a service component provides five ingredients that can be seen as the basis for expressing service reuse and specialization. These are: `Definition, Construction, PortType, Provider`, and `MessageHandling`. With the concept of service component, the process of developing web service compositions becomes a matter of reusing, specializing, extending and possibly customizing available service components. This enables a great deal of flexibility and reusability of service compositions and delivers highly-functional applications.

Service components normally comprise two inter-related parts. They comprise a typical *business process* that operates on specific *entity (data) components* in a particular business domain. A business process is a set of logically related tasks performed to achieve a well defined business outcome [22]. An automated business process is a precisely choreographed sequence of activities systematically directed towards performing a certain business task to completion. For example, processing a credit claim, ordering goods from a supplier, creating a marketing plan, processing and paying an insurance claim, and so on, are all examples of typical business processes. An *activity*, which coincides with service portTypes and operations in a web service, is an element that performs a specific function within a process. Activities can be as simple as sending or receiving a message, or as complex as coordinating the execution of other processes and activities. Entity components provide *entity services* - often eminently re-usable data elements in different processes. An entity component may "own" a specific set of logical database tables, so separating the enterprize data and its schemas from the process business logic. Aside from providing read and update services to processes, they can also manage validation of data related to themselves. Entity components provide private data services to service components, and thus are not visible as an external web service. The advantage of this approach is that it separates concerns between process and entity. For example, a `Sales Order` entity component manages all the data aspects of a sales order, including data validation, while a `Sales Order` service component is all about providing services to manage and process sales orders. This not only reduces dependencies, but also reduces the complexity of process service components. In addition, complex entity components become eminently re-usable at run-time, so reducing maintenance and testing when service components need to be updated.

In the following we will concentrate only on the process nature of service components as the data handling part follows well-known practises from the object-oriented world. In particular, in this paper we will concentrate on how service components are used for composite service definition and construction. The contribution of this paper is three-fold:

- it proposes the concept of a service component for creating composite services via reuse, specialization, and extension;
- it introduces an XML-based light-weighted specification language and class definition for

service components that can be used for service component definition, construction, and execution. The specification language is conformant with BPEL;

- Finally, it provides an integratedframework for web service composition development, which manages the entire life-cycle of service composition, viz. definition, scheduling and execution.

Service component classes and SCSL are revisited in section-4 where we explain how services are materialized.

3 Service composition: An overview

With the rapid expansion of web service related applications in fields such as e-business, e-government and e-health, there is a clear need for infrastructures and frameworks that can be used to develop applications on the basis of web service compositions. In this section we first analyze the nature of service composition and introduce a framework for service composition and application development based on web services. Subsequently, we illustrate the characteristics of composition logic that lays the foundation for creating service components.

3.1 Service composition categories

It is obvious that service composition is far more than merely an interoperability problem. The real challenge in service composition lies in how to provide a complete solution. This means developing a toolset that manages the entire life cycle of service composition. This comes in contrast to solutions provided by classical workflow integration practices, where service composition is pre-predetermined and pre-specified, has narrow applicability and is almost impossible to specialize and extend.

Web service composition falls under three major categories:

- *Explorative composition*: This category requires that service compositions are generated on the fly on the basis of a request expressed by a client (application developer). The client specifies the desired service functionality in a high-level request language and the ensuing services are then compared with potentially matching UDDI published constituent service specifications. The matching process may generate a series of feasible service composition alternatives. These alternative service compositions can be ranked or can be chosen by service clients on the basis of non-functional criteria such as availability, cost or performance. This type of service composition requires that the composite service is dynamically orchestrated out of constituent services.

- *Semi-fixed composition:* Semi-fixed compositions require that the entire service composition is specified statically but the actual service bindings are decided at run time. When a com-

posite service is invoked, the actual composition specification is generated on the basis of a matching between the constituent services that are specified in the composition and possibly available services. In this case, the definition of the composite service is registered in a service composition repository, and can then be used just as any other normal service i.e., it can be searched, selected, and combined with other web services.

- *Fixed composition:* A fixed composite service requires that its constituent services be synthesized in a fixed (pre-specified) manner. The composition structure and the component services are statically bound. Requests to such composite services are performed by sending sub-requests to its constituent services.

3.2 Service composition life-cycle

This paper advocates a phased approach to service composition. The activities in this phased approach are collectively referred to as the *service composition life-cycle*. The purpose of these activities, or phases, is to first describe services in the abstract and then to generate executable service processes from these abstract specifications. Abstract service descriptions can be either derived from a request specified in a high-level language (explorative compositions) or by a client specified service definition (semi-fixed and fixed compositions). Hence, the service composition life-cycle spans all three modes of service composition and is characterized by the following five phases:

1. *Planning phase:* the planning phase assists in determining the series of service operations (or planned activities) that need to be retrieved and aggregated in order to satisfy a given user supplied service request. A service request language provides for a formal means of describing desired service attributes and functionality, including temporal and non-temporal constraints between services, and scheduling service preferences. Service requests and planned activities are executed on the basis of information supplied by a domain model. Each vertical marketplace domain model includes [1]:

    - A standard business processes that formally describe business interactions between organizations. In the world of web services business processes are published via directories such as the UDDI.
    - A document model for defining structured XML business documents exchanged between trading partners or service requesters and providers over the Internet. This includes request (input) and reply (output) business documents for business actions that are part of standard business processes.

    The planning phase is used only in conjunction with explorative compositions.

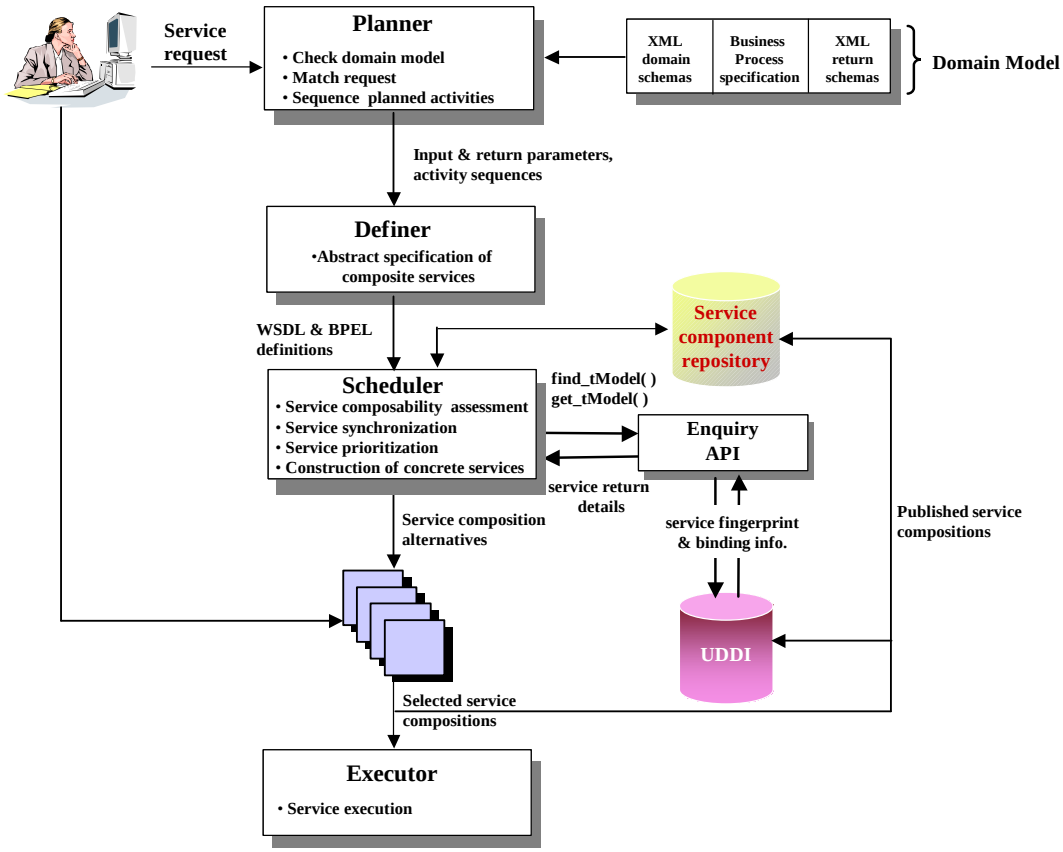

Figure 1: Phases involved in an explorative service composition.

2. *Definition phase:* the definition phase allows defining *abstractly* composite services. Composite service definitions employ WSDL in conjunction with a language that allows defining business processes by orchestrating web services, viz., BPEL. In the case of service components, abstract service component classes can be employed during this phase.

3. *Scheduling phase:* the scheduling phase is responsible for determining how and when services will run and preparing them for execution. Its main purpose is to give *concrete* definitions to the constructs supplied by the definition phase by composing abstract services, by assessing their composability and conformance capabilities, by correlating messages and operations, and then by synchronizing and prioritizing the execution of constituent web services according to their definition. During this phase alternative composition schedules may be generated and proposed to the application developer for choice.

4. *Construction phase:* The outcome of this phase is the construction of a concrete and unambiguous composition of services – out of a set of desirable or potentially available/matching constituent services – that are ready for execution.

5. *Execution phase:* the execution phase implements composite service bindings on the basis of the scheduled service composition specifications and executes the services in question.

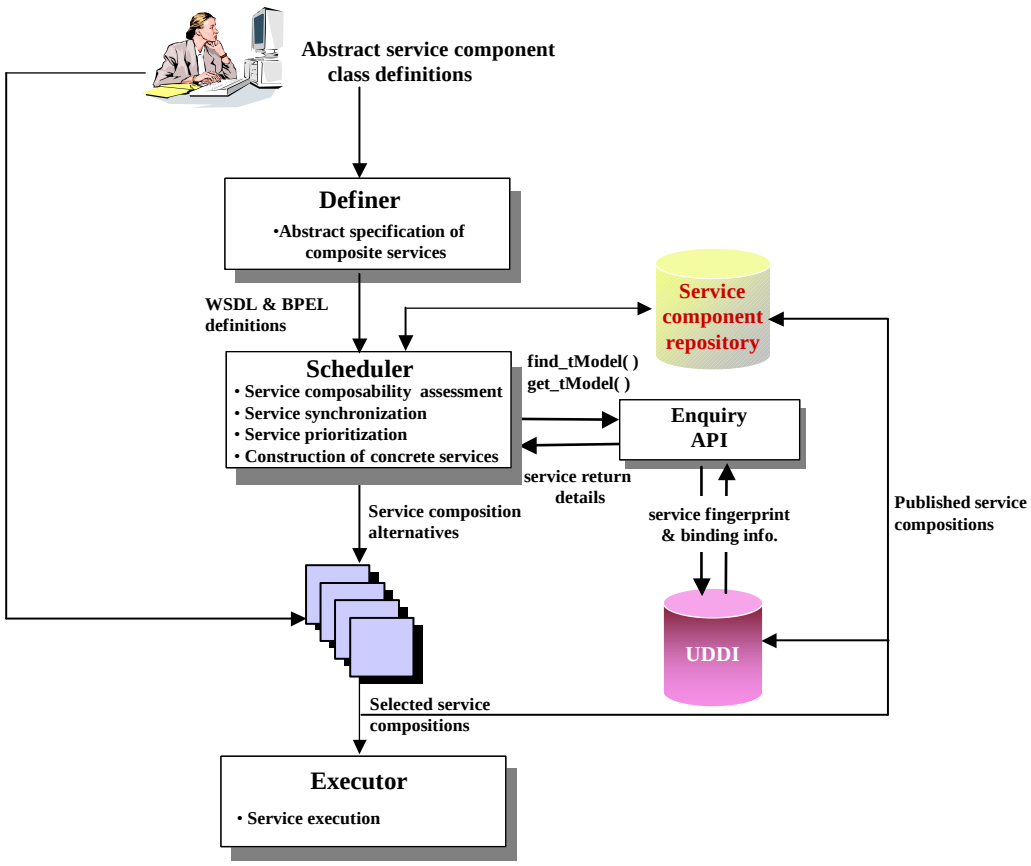

Figure 2: Phases involved in semi-fixed and fixed service composition.

Figure 1 depicts the service life-cycle phases required for an explorative service composition. We assume that this procedure is initiated by a client who is aware of the format of the input and the output parameters defined in the XML schemas (document model) and the standard business process specifications for a particular domain. The *planner* module checks the consistency of the request with respect to the business process specification by finding appropriate paths of activities (service operations) potentially satisfying the client request. If the request is consistent with the domain model specifications, a set of activities is returned for further processing. The planner returns activity paths along with the business process specification that could potentially satisfy the client request. Subsequently, the service definition (*definer*) module constructs abstract WSDL and BPEL service definitions for the planned activity sequences. These are then passed to the *scheduler* module. The scheduler needs to interact first with the service providers to be able invoke the service operations specified in the abstract definitions. The scheduler makes the

abstract definitions concrete by first appropriately invoking the UDDI enquiry API. For this purpose the scheduler calls the enquiry UDDI operations to `find` and `get_detail` of the UDDI API to retrieve detailed information about the service port-types, elements and bindings of the services. The scheduler searches the UDDI for the services required and uses the information found in the UDDI registry to establish the particular invocation pattern needed for the specific service being employed. Subsequently, the scheduler correlates the constituent services and checks for compatibility. Alternative service compositions, based on non-functional service characteristics, such as performance, security and pricing models, are then proposed to the client for selection and approval. Finally, once the selected services are made concrete, they are stored in a service repository for future use and passed to the *executor* module for execution.

Figure 2 depicts the phases required for semi-fixed and fixed service composition that are the two service composition schemes used in conjunction with service components. Semi-fixed and fixed service composition is a much simpler affair when compared explorative service composition. Semi-fixed and fixed service composition does not require any planning activities as the client provides composite service definitions in the form of abstract service component classes that are further processed by the scheduler.

Explorative service composition and service composition planning has been studied in [1]. In this paper we concentrate on how service components can be used in the context of semi-fixed and fixed service compositions.

3.3 Composition logic

Service components are used as building blocks for generating web applications based on packaging together composed service functionality. Consequently, the process of web service composition becomes a matter of reusing, specializing, and extending the available service components. This enables a great deal of flexibility and reusability of service compositions. To understand this procedure we need to examine the inner structure and inner workings of a service component.

Figure 3 depicts the ingredients of a service component. It illustrates that a service component presents a single *public interface* to the outside world that is constructed in terms of a uniform representation of the operation signatures, message types, and portTypes of its constituent services. A service component contains a *composition logic* part that specifies internally how it is constructed out of constituent web services in terms of *composition type* and *message dependency* constructs.

Composition logic refers to the manner according to which a service component is constructed in terms of its constituent services. Here, we assume that all publicly available services are described in WSDL. Composition logic comprises the following two constructs:

- *Composition type:* this construct signifies the nature of the composition, which can take one of two forms:

- *Order:* this construct indicates whether the constituent services in a composition are executed in a serial or parallel manner.
- *Alternative service execution:* this construct indicates whether alternative services can be invoked in a given service composition. Alternative services can be tried out either in a sequential or in a parallel fashion.

- *Message dependency:* this construct signifies the types of message dependency between constructs of the constituent services within a service component as well as message dependencies between constructs of the constituent services and those of their surrounding service component. We distinguish between three types of message dependency handling mechanisms necessary for service composition:
  - *message synthesis:* this construct combines the output messages of constituent services to form the output message of a composite service.
  - *message decomposition:* this construct decomposes the input message of the composite service to generate the input messages of its constituent services;
  - *message mapping:* this construct specifies input/output mappings between its constituent services. For example, the output message of one constituent service could be the input message of another service.

  Examples of message dependency are given in section-4.3 and Figure 6.

At this stage what remains to be examined are the core service component constructs for expressing composition logic, how they are defined and how they can be reused and specialized. These issues are addressed in the following section.

4 Materialization of service components

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

4.1 Basic constructs for service composition

The following basic composition types have been identified to serve as a sound basis for representing service compositions [29]:

1. **Sequential service composition (sequ):** With this type the constituent services are invoked successively. The execution of a constituent service depends on its preceding service, i.e., a new service cannot begin unless its preceding service has committed. For example,

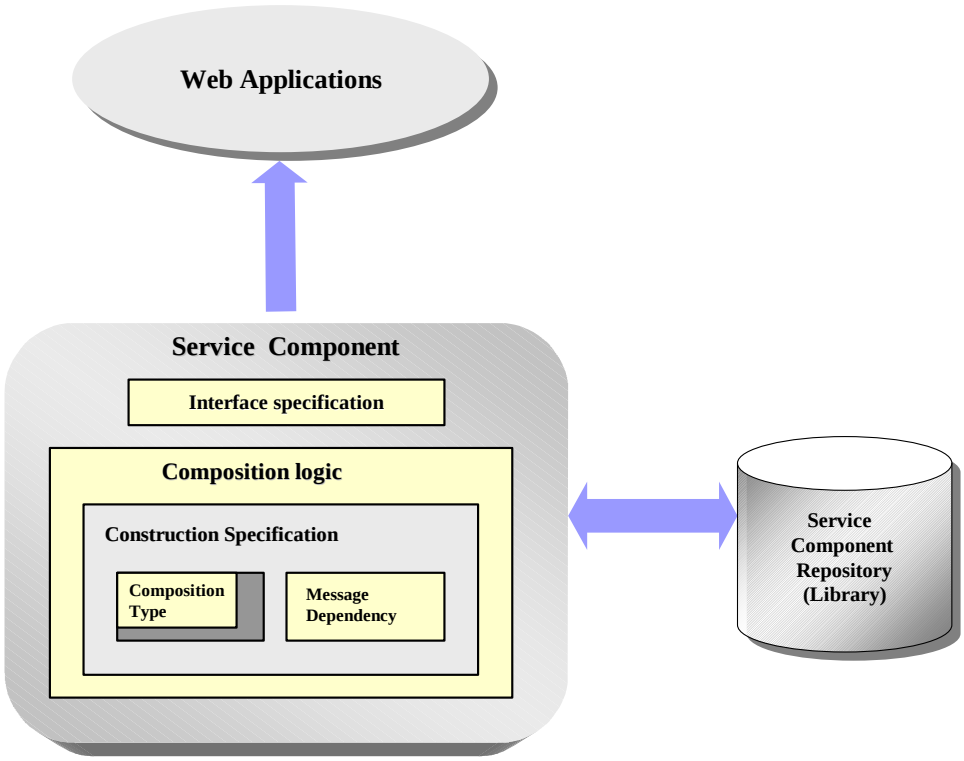

Figure 3: Service component ingredients.

when a composite service such as `travel plan` - composed of an `ticket booking` service, a `hotel booking` service, and a `car rental` service - suggests a travel plan to a customer, the execution order should be `ticket booking`, `hotel booking`, and `car rental`. The invocation of the `hotel booking` service dependents on a successful execution of the `ticket booking` service because without a successful ticket booking, a hotel booking can not go ahead.

2. **Sequential alternative composition (seqAlt).** This type indicates that alternative services could be part of the composition and that these are ordered according to some criterion (e.g., cost, time, etc). Alternative services are attempted in succession until one of them succeeds.

3. **Parallel service composition.** In this situation, all the component services may execute independently. Here, two types of scenarios may prevail:

   (a) **Parallel with result synchronization (paraWithSyn).** This situation arises when the constituent services can run concurrently, however, the results of their execution need to be combined. For example, the services `restaurantReservation` and `sightseeingBooking` can be executed in parallel; however, they need to execute to completion and their results need to be combined in order to obtain a valid itinerary for the day.

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

Table 1: Message handling constructs for service composition.

(b) **Parallel alternative composition (paraAlt).** With this type of composition alternative services are pursued in parallel until one service is chosen. As soon as a service succeeds the remainder are discarded.

Although the above basic types of composition are adequate for representing the most common features of service composition, two ancillary control constructs are required to make the composition logic complete: `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 hence require different message handling capabilities. Table 1 summarizes the message dependency handling constructs required for the different types of service composition.

The basic composition types in conjunction with message dependency handling constructs provide a sound basis for developing the composition logic in a service component.

Similar basic routing mechanisms such as *sequential* and *parallel* for use in distributed computations and workflows can be found in [21]. The main difference between this publication and the work reported herein is that this publication discusses basic constructs for control flow execution, whereas in this paper the basic routing constructs are used as part of the abstraction mechanism for realizing the composition logic for service components.

4.2 The service component class library

Service 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 represented as a service component class advertised by a given service provider and can thus be used in the development of distributed service applications.

In Figure 4 we define a service component class for a composite service named `TravelPlan` that describes `ticketBooking` and `hotelBooking` activities. The `Definition` construct defines the two messages `tripOrderMsg` and `tripResMsg` and one *public* operation `travelPlanning`. The two messages are input and output of the operation `travelPlanning`. The statement `Construction` specifies the manner in which the two activities are composed. The construct `PortType` specifies

```
ServiceComponentClass TravelPlan {
  Definition
    TripOrderMessage tripOrderMsg
    TripResultMessage tripResDetails
    TravelPlanning (in tripOrderMsg, out tripResDetails)
  Construction
    sequ(ticketBooking, hotelBooking)
  PortType
    ticketBookingPT.makeRes ticketBooking
    hotelBookingPT.makeBooking hotelBooking
  Provider
    TicketBookingProvider ticketBookingPT
    HotelBookingProvider hotelBookingPT
  MessageHandling
    messageDecomposition(TravelPlanning.tripOrderMsg,
                         ticketBookingPT.makeRes.ticketBookingMsg,
                         hotelBookingPT.makeBooking.hotelBookingMsg)
    messageSynthesis(ticketBookingPT.makeRes.e-ticket,
                     hotelBookingPT.makeBooking.hotelBookingDetails,
                     TravelPlanning.tripResDetails)
}
```

Figure 4: A service component class definition.

the port types and operations that the activities refer to. In this example, operation `makeBooking` of PortType `hotelBookingPT` and operation `makeRes` of PortType `ticketBookingPT` are used for the activities `hotelBooking` and `ticketBooking`, respectively. The construct `Provider` defines the web service providers that provide these service activities. The construct `MessageHandling` defines a message dependency among the service component operations and their constituent activities. For example, the construct `messageDecomposition` decomposes the message `tripOrderMsg` of operation `TravelPlanning` into two input messages `hotelBookingMsg` and `ticketBookingMsg` of `hotelBookingPT.makeBooking` and `ticketBookingPT.makeRes`, respectively. Note that the input/output message of port type operations are defined in WSDL.

Using the class definition of a service component, we can reuse and specialize it much in the same way that object oriented systems do. For example, if we need to provide an additional activity `sightseeing` that runs after the two existing activities `ticketBooking` and `hotelBooking`, we can define a new service component `NewTravelPlan` as a subclass of the existing `TravelPlan` service component as follows:

```
ServiceComponentClass NewTravelPlan SubclassOf TravelPlan {
  Construction
    sequ(ticketBooking, hotelBooking, sightseeing}
  PortType
    sightseeingPT.sightseeing sightseeing
  Provider
    sightseeingProvider sightseeingPT
  MessageHandling
    messageDecomposition(TravelPlanning.tripOrderMsg,
```

```
                        ticketBookingPT.makeRes.ticketBookingMsg,
                        hotelBookingPT.makeBooking.hotelBookingMsg,
                        sightseeingPT.sightseeing.sightseeingBMsg)
    messageSynthesis(ticketBookingPT.makeRes.e-ticket,
                     hotelBookingPT.makeBooking.hotelBookingDetails,
                     sightseeingPT.sightseeing.sightseeingRes
                     TravelPlanning.tripResDetails)
}
```

In this example, the constructs `Construction` and `MessageHandling` are refined in the subclass `NewTravelPlan` while the constructs `PortType` and `Provider` are extended. More details about how service components can be reused and specialized can be found in [30].

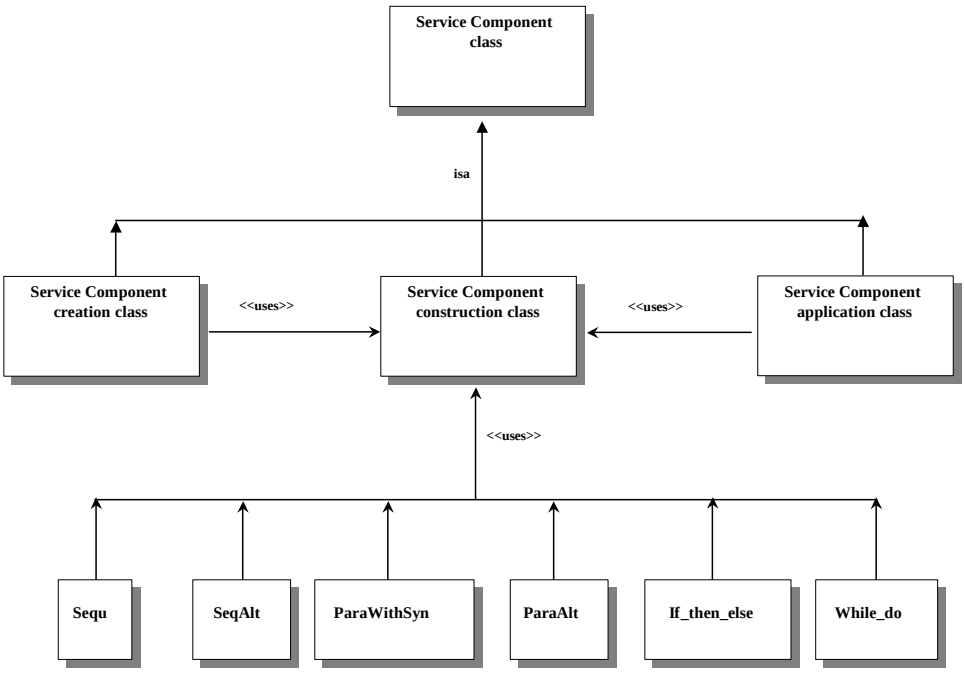

Figure 5: Primitive class constructs.

The service component class library is a collection of general purpose and customized service component classes, e.g., `TravelPlan`, that are used in the context of service component based applications. These classes employ the primitives and constructs discussed in the previous section. The service component library also contains a set of primitive classes that act as abstract data types i.e., they cannot be instantiated. These primitive classes provide basic constructs and functionality that can be further specialized according to the needs of an application resulting thus in highly-functional web-service based applications. A distributed web application can be build by re-using, specializing, and customizing the service component library classes. The primitive classes of the service component library include the following constructs:

- *service component creation class:* this primitive class is used for creating service component classes out of WSDL/XML specifications. Service component classes for registered web services defined in WSDL/XML are created by using construction classes. This implies that `Definitions, Construction, PortType, MessageHandling,` and `Providers` will be generated for the service component classes.

- *service component construction classes:* this class provides the semantics and functions to implement the composition constructs discussed in the previous section. These primitive classes are: `sequ, seqAlt, paraWithSyn, paraAlt, if_then_else,` and `while_do`. The construction classes are used for building new service components.

- *service component application class:* this class is used as a basis for developing distributed application programs that employ service components. As application program classes are essentially service components, they can also be reused, specialized, and extended.

The example illustrated in Figure 4 is a service component application class. The primitive class constructs and their relationships are summarized in Figure 5.

Interacting service components and services (across the network) can only communicate on the basis of exchanging XML Web service specifications and SOAP messages. Thus although service 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 over the network. For this purpose we use the SCSL version of a service component class, which is presented in the next section.

4.3 Service component specification in XML

There are two parts in an SCSL definition: 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 6). These are isomorphic to the `Definition` and `Construction` parts of a service component shown in Figure 4. The `construct` part of an SCSL definition 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 attain a certain service 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 presented in section-4.1, while message handling specifies how service and activity messages are processed.

SCSL is designed to comply with the web service standards. In other words, we adopt the same conventions as WSDL [27], e.g., the SCSL construct `PortType` coincides with the WSDL construct `portType` and is used for grouping `operations`. Operations represent a single unit of work for the service being described. The example of `travelPlan` illustrated in Figure 6 is an WSDL/XML equivalent of the service component class illustrated in Figure 4 and provides a single `PortType`

```
<webService name="travelPlan">
<!---= Message definition ==-->
  <definition>
    <message name="tripOrderMsg">
      <part name="ticketBookingMsg" element="ticketBooking"/>
      <part name="hotelBookingMsg" element="hotelBooking"/>
    </message>
    <message name="tripResDetailsMsg">
      <part name="e-ticketMsg" element="e-ticket"/>
      <part name="hotelBookingDetailsMsg" element="hotelBookingDetails"/>
    </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="ticketBooking">
        <input message="ticketBookingMsg"/>
        <output message="e-ticket"/>
        <performedBy serviceProvider="Disney Land"/>
        <use portType="ticketBookingPT" operation="makeRes"/>
      </activity>
      <activity name="hotelBooking">
        <input message="hotelBookingMsg"/>
        <output message="hotelBookingDetails"/>
        <performedBy serviceProvider="Paradise"/>
        <use portType="hotelBookingPT" operation="makeBooking"/>
      </activity>
      <messageHandling>
        <messageDecomposition>
          <source message="tripOrderMsg"/>
          <target message="hotelBookingMsg" query="Query1"/>
          <target message="ticketBookingMsg" query="Query2"/>
        </messageDecomposition>
        <messageSynthesis>
          <source message="hotelBookingDetailsMsg"/>
          <source message="e-ticketMsg"/>
          <target message="tripBookingDetailsMsg" query="Query3"/>
        </messageSynthesis>
      </messageHandling>
    </composition>
  </construct>
</webService>
```

Figure 6: Service composition specification language of class in Figure 4.

named `travelPlaner` with one operation `travelPlanning`. The activity `hotelBooking` uses the operation `makeBooking` of port type `hotelBookingPT`, while the activity `ticketBooking` uses the operation `makeRes` of port type `ticketBookingPT`.

We also specify how input and output messages of constituent service operations are linked from (to) those of the composite service. Here we provide the three message handling types: (1) message synthesis, (2) message decomposition, and (3) message mapping. For example, 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 `ticketBookingMsg` of constituent operation `makeRes`. 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.

To guarantee consistent handling of messages and operations between distributed web services provided by diverse providers, standard naming schemes and business processes must be employed. This is common practise in vertical e-marketplaces, such as chemicals, travel industry, pharmaceutical, semiconductors, etc, where service components can be deployed. For instance, in the business domain of e-travelling the open travel agency (OTA, www.opentravel.org), has specified a common naming scheme (ontology) and a set of standard business processes for searching for availability and booking a reservation in the airline, hotel and car rental industry, as well as the purchase of travel insurance in conjunction with these services. OTA specifications use XML for structured data messages to be exchanged over the Internet. In this paper we adopt a similar philosophy and assume that all message and operation names as well as process specifications are standard and OTA conformant.

In order to be able to specify how messages between service components and their constituent services are mapped to each other, the following three queries, based on XQuery syntax [28], are used:

```
Query1:
<hotelBooking>
  for $hb in (document($tripOrder.xml)//hotelBooking)
  return
   $hb
</hotelBooking>

Query2:
<ticketBooking>
  for $tr in (document($tripOrder.xml)//ticketRes)
  return $tr
</ticketBooking>

Query3:
<tripResDetails>
  for $hbd in (document($hotelDetails.xml))
```

```
  return $hbd
</tripResDetails>
<e-ticket>
  for $et in (document($e-ticket.xml))
  return $et
</e-ticket>
```

In each `query` specification, an input `document` must conform to a `source` message type in an SCSL definition. In addition, a return document must conform to a `target` message type in an SCSL definition. For instance, in `Query1`, the document variable `tripOrder.xml` is of message type `tripOrderMsg`, and the result document must be of message type `hotelBookingMsg`.

A similar idea is also used in [17]. This publication discusses how a data-centric service can be integrated by means of input schema decompositions and output schema compositions. In this paper, an integrated input XML schema is constructed from any constituent (data) service input schema or from the sequential composition of these input schemas. Each output XML schema is synthesized out of the output schemas of the constituent services and is specified in a template based on XML-QL syntax [9].

The BPEL standard also follows a similar philosophy. In BPEL specifications, data mappings and correspondence among services are specified in XPath based queries.

Although the above example is based on sequential composition, other types of service compositions can be specified in a similar fashion. Note that the code snippet in Figure 6 is a simplified version of the SCSL and serves only for illustration purposes. Binding specifications are not included in this figure. The complete XML schema of SCSL can be found in Appendix-A.

5 Service composition scheduling and execution

As already explained in section-3.1, service compositions in a service component need be scheduled and generated according to a client provided abstract definition. This means that the abstract SCSL definitions in Figure 6 need to be converted into an intermediate representation that can be handled by the scheduler, (see Figure 2). With this in mind, we have developed a *Service Scheduling Language (SSL)* that specifies how a service component is built up in terms of its constituent services by considering how they are inter-related, for instance, by taking into account their execution order and dependencies. Concrete service component definitions specified in SSL result in a service execution structure represented in the form of a *Service Composition Execution Graph (SCEG)*. The SGEC is then passed to and executed by the executor module in Figure 2. In the following, we will first introduce the concepts underlying the SSL and SCEG by means of our running example and then we will present their formal characteristics.

Figure 7 illustrates how a composite service called `HolidayPlan` can be scheduled in SSL by combining three component services `hotelBooking`, `restaurantReservation`, and `sightseeing`.

In this example, we specify that `hotelBooking` and `restaurantReservation` have to run sequentially, and there is data dependency between them, i.e., the location of the hotel determines the location of the restaurant, while `Sightseeing` can run in parallel with the two services.

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

Figure 7: Service Scheduling

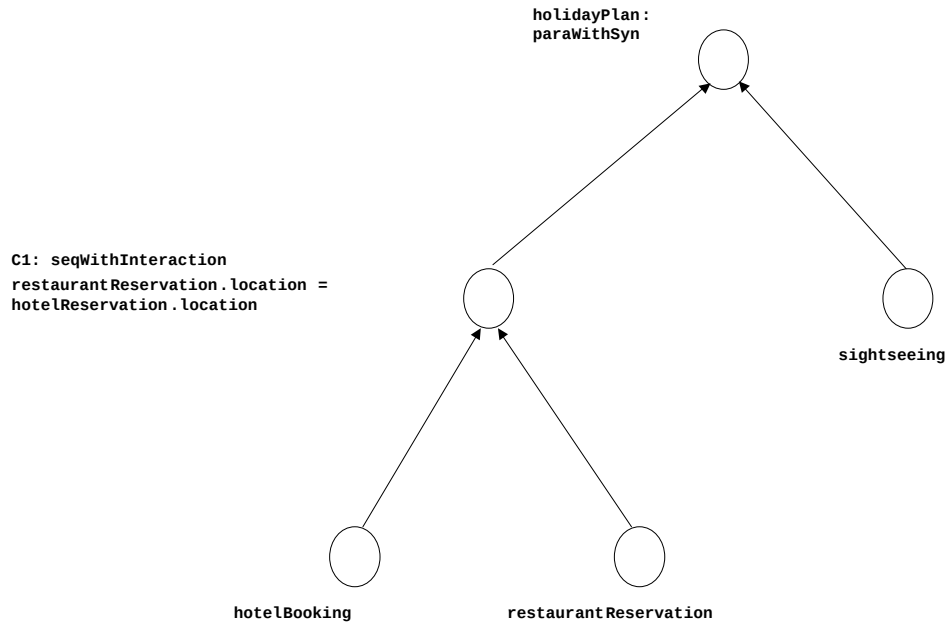

Figure 8: The Service Composition Execution Graph

SSL provides a simple light-weight but powerful mechanism for service composition nesting and substitution, composition extension, and dynamic service selection. The formal syntax specification of SSL in BNF can be found in Figure 9.

There are two aspects of SSL which make it flexible and extensible: (1) a labelling system that 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 schedule recursively by labelling existing compositions; (2) variables and macros that can be used in the place where the service (operation) and composition types (such as sequential, paraWithSyn etc) are required. This second (and conventional) aspect SSL of is not discussed in this paper due to space limitations.

```
<SSL>               ::= <statement>+
<statement>         ::= <label> ":" <planExpr>
<planExpr>          ::= <compositionType> "(" <compositionPara> ")"
                        <mappingExpr>
<compositionType>   ::= "seqNoInteraction" | "seqWithInteraction" |
                        "seqAlt" | "paraWithSyn" | "paraAlt" |
                        "condition" | "while_do" | variables
<compositionPara>   ::= <paraExpr> "," <paraExpr>*
<paraExpr>          ::= <label> | <planExpr> | <serviceIdentifier>
<mappingExpr>       ::= "mapping" "(" <mappingElem> "," <mappingElem>*
")"
<mappingElem>        ::= <message>+ "->" <message>+
<label>             ::= string_literal
<variables>         ::= string_literal
<serviceIdentifier> ::= the existing service identifier
```

Figure 9: The formal syntax specification of SSL

SCEG is generated on the basis of an SSL specification. Figure 8 depicts the SCEG representation corresponding to the SSL specification in Figure 7. Formally speaking, 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 service used in SSL, we create a vertex;
- for $v \in V$, $spe(v)$ represents the type of composition and the mapping specification;
- for $u, v \in V$, if u is used in v for composition, we introduce an arc $u \rightarrow v$.

6 Service composition: a complete picture

In this section, we explain how the SCSL, the SSL and the SCEG work together to create service compositions. Recall that as already stated in section-3.1, we will concentrate on service component definition, scheduling, construction and execution.

6.1 Definition and scheduling

The definition and scheduling phases involve abstract service definition, service discovery, composability and compatibility checking, and synchronization. The output of these two phases is a series of concrete service composition alternatives specified in SSL for user selection and approval.

To exemplify these two phases we assume that we are dealing with an abstract service component class definition called `holiday plan` that is similar to, but slightly more complicated than the class

`travel plan` illustrated in Figure 4. We also assume that its isomorphic SCSL version has been derived and has the following simplified form:

```
<HolidayPlan>
  <input message="holidayBookingMsg>
  <output message="holidayBookingDetails>
  <paraWithSyn>
    <sequ>
      <activity name="ticket booking"/>
      <activity name="hotel booking"/>
      <activity name="restaurant reservation"/>
      <mapping="ticketBooking.date=hotelBooking.date">
      <mapping="restarurantReservation.location=hotelBooking.location">
    </sequ>
    <activity name="sightseeing"/>
    <synthesis="holidayBookingDetails=ticketBookingRes
                                    +restarurantReservationRes+sightseeingRes">
  </paraWithSyn>
</Holiday Plan>
```

This definition can realized provided that we find the services that match the required activities. 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 service descriptions found in a repository (UDDI or service component library) to determine how closely related they are. Services with a high degree of relatedness will be selected as relevant services for subsequent capability checking.
- Capability analysis: the capabilities of the services selected from the previous step are checked in terms of the functionality they provide to determine whether they can accomplish completely or partially the tasks of the requested service.
- Syntactic analysis: matching services have their syntax of their interfaces checked to determine how they can be combined to achieve the requested higher-order service component functionality.

Currently, service discovery is conducted by interacting with UDDI to find details regarding the technical capabilities of the required services. For this purpose the `find` and `get_Detail` operations of the UDDI enquiry API are used. These operations are used to discover and retrieve the technical fingerprint that can be used to recognize a web-service that implements a particular behaviour or its programming interface. This procedure is described in section-3.1. If UDDI and WSDL are used together, the `overviewDoc` element of the `tModel`, that is used to provide an overview description of the `tModel` and its intended use, is a WSDL service interface definition.

In addition the *service component repository* is checked to ascertain whether there exist service components that can form the basis for reuse or extension in the composition. In this way we avoid developing service compositions from scratch as much as possible.

Once a candidate service is found on the basis of the first two steps discussed above, the ensuing web services (or service components), which can be used to perform the actions specified by scheduler, need to be analyzed to determine their syntactic compatibility and their conformance. To understand service conformance and compatibility issues we first need to give a formal definition of web services, which is given below.

A web service (S) can be represented 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 syntactic checking.

A service can be represented as $S =< C, A, P >$, 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 activities of S' are compatible with those of S $(S'.A \diamond S.A)$. This is given in definition-2.

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 $a.O \diamond a'.O$.

In the context of web services, inputs and outputs are specified in XML schemas. We can then

say that an input/output XML schema *schema-1* is compatible with another *schema-2* if and only if *schema-1* is a supertype of *schema-2*. This relates to work on XML schema subtyping that can be found in [19, 16].

Definition-4

S' conforms to S ($S' \triangleright S$) if:

(1) the contents $S'C$ and $S.C$ are overlapping and

(2) $\exists a' \in S'.A, \exists a \in S.A$ such that $a'.O \diamond a.I$.

To exemplify these issues, we assume that we can choose between two schedules specifying candidate services in SSL after conformance and compatibility checking has been successfully completed. For this purpose we use the definition of the service component `HolidayPlan` that was given earlier in this subsection. These two schedules are named `HolidayPlan1` and `HolidayPlan2`.

```
HolidayPlan1
C1: sequential(ticketBooking, hotelBooking, restaurantReservation)
    Mapping (ticketBooking.arrive_date=hotelBooking.date,
             restarurantReservation.location=hotelBooking.location)
C2: paraWithSyn (C1, sightseeing)
    Sythesis (holidayPlanning.schedule=C1.schedule+sightseeing.schedule)


holidayPlan2
C1: sequential (travelPlan, restaurantReservation)
    Mapping (restarurantReservation.location=hotelBooking.location)
C2: paraWithSyn (c1, sightseeing)
    Sythesis (holidayPlanning.schedule=C1.schedule+sightseeing.schedule)
```

The schedule `HolidayPlan1` 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. The schedule `HolidayPlan2` contains two services one of which is a composite service defined and constructed as shown in Figure 6. The mapping `ticketBooking.arrive_date=hotelBooking.date` is assumed to be accomplished by the composite service `travelPlan`.

Related work on compatibility can be found in the areas of capability matching in software agents [25] and more importantly in software component compatibility assessment [32, 2, 3]. In [25] the authors introduce the agent capability description language LARKS and how it can be used in matching processes. In the area of software component compatibility representative research results can be found in [32]. This work is strictly based on syntactic specification and relates the comparability issue to simple subtyping checks. Recent trends in this area use a declarative language and develop a reasoning mechanisms to check component comparability [2, 3].

In this paper we use simple conventional compatibility and conformance mechanisms found mostly in the theory of programming languages. These mechanisms can be substantially improved by combining them with recent results in the area of software component comparability checking

```
ServiceComponentClass HolidayPlan1 subcalssof sequ, parawithSyn {
  Definition
    HolidaySchedule holidaySchedule;
    ... //public operations
    Construction
      parawithSyn(sequ(TicketBooking, HotelBooking, RestaurantReservation), Sightseeing);
    PortType
     ...
    Provider
     ...
    MessageHandling
      messageMapping(TicketBooking.date, HotelBooking.date);
      messageMapping(RestaurantReservation.location,
                              HotelBooking.location);
      messageSynthesizing(HolidayPackaging.holidaySchedule,
                 C1.TravelSchedule, Sightseeing.Schedule);
}
```

Figure 10: Constructing the service component class HolidayPlan1.

that appear to be particularly promising for application to the context of web services. However, before this is accomplished further research is required.

6.2 Construction of concrete service component classes

To choose among alternative composition schedules generated by the scheduling phase, the end point properties of the candidate constituent services (such as cost, performance, binding requirements) need to be assessed. Suppose that the schedule `holidayPlan1` in section-6.1 is selected. In this schedule the primitive classes `sequ` and `paraWithSyn`, see Figure 5, are then used in conjunction to define and construct the composition and the ensuing service component class in an incremental fashion.

Firstly services `ticketBooking, hotelBooking, restarurantReservation` are combined by employing the primitive class `sequ` and are further extended with the necessary message types and operations. Subsequently, the result is combined with the service `sightseeing` by employing the primitive class `paraWithSyn`to generate the application program class called `holidayPlan1`. The code in Figure 10 illustrates how the *concrete* service component class `holidayPlan1` is constructed and defined. This concrete class definition is internally represented in SCSL.

6.3 Execution

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 denotes an entire application under execution. The type of composition and the message dependencies are 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 the basis of the *depth-first search*. The process of construction and execution of composite services is illustrated in Figure 11.

7 Implementation

The service component framework has been implemented into a prototype system called Service-Com. The current prototype version is implemented in Java. It is based upon a set of widely accepted standards: (1) WSDL [27], the implementation from IBM is used to read and write WSDL files. (2) SOAP, an implementation from Sun is used to create and send SOAP messages. (3) DOM, the DOM implementation from Apache is used to read and write XML files. This implementation is part of the Java XML package, incorporated into the JDK version 1.4.

Figure 12 shows the main window of the Service-Com prototype system. The toolbar provides access to the major functionalities of the tool. In the `File` menu options are provided to start with a new web service composition, load an existing web service composition or save a created web service composition. The `Options` menu enables the user to modify the settings for the tool as well as maintaining an error log. In the `Build` menu the functionalities are offered with regard to the definition, scheduling and invocation (execution) of a web service composition. The main window itself shows the dialogue interface for creating and editing a web service composition. The general properties of the composition can be specified, activities (service operations and portTypes) can be added, edited or removed, and condition(s) can be specified.

For the specification and editing of an activity in a web service composition the tool provides the activity dialogue illustrated in Figure 13. In the activity dialogue the user can specify the general characteristics of the activity as well as its binding type, which can be either dynamic or fixed. The dialog in Figure 13 shows an activity with a fixed binding type. In this case the user can specify which web service is to be used. This can be done by clicking the `Select WSDL interface description file` button.

This results in the opening of the view WSDL dialog, which provides an overview of the services offered by the selected web service provider. Additional information concerning a web service, its ports and its operations can be viewed.

Once the composition specification is created, the next step is to assemble the specified composition. During this step, a set of files are generated, which function together as a stub for composition invocation. These files include: a SCSL file for the composition specification, Java source and binary files for the composition for each activity, and the WSDL file for the composite service. These files can be re-used and extended for the same or similar type of composition. During the service assembly, service inter-linking is conducted internally in terms of SSL.

```
Let G represent the graph, L(v) denote the set of children notes of v,
let spe(v).type be the type of the composition,
and let spe(v).M be the mapping specifications of the composition.
begin
   for each vertex u ∈ G.V
       mark(u) := "unvisited";
   endfor;
   for each vertex u ∈ G.V
       if mark(u) = "unvisited";
          DFSE(u);
       endif;
   endfor;
endbegin;
Procedure DFSE(v: vertex)
begin
   mark(v) := "visiting";
   for each w ∈ L(v)
       if mark(w) = "unvisited"
          DFSE(w);
       endif;
   endfor;
   mark(v) := "visited";
   case spe(v).type = sequ
         call route(sequ);
         call mapping(spe(v).M);
   case ... // all primitive class constructs shown in
         // Figure-4 are tested and handled here.
   endcase;
endbegin;
```

Figure 11: The Algorithm for execution of service components.

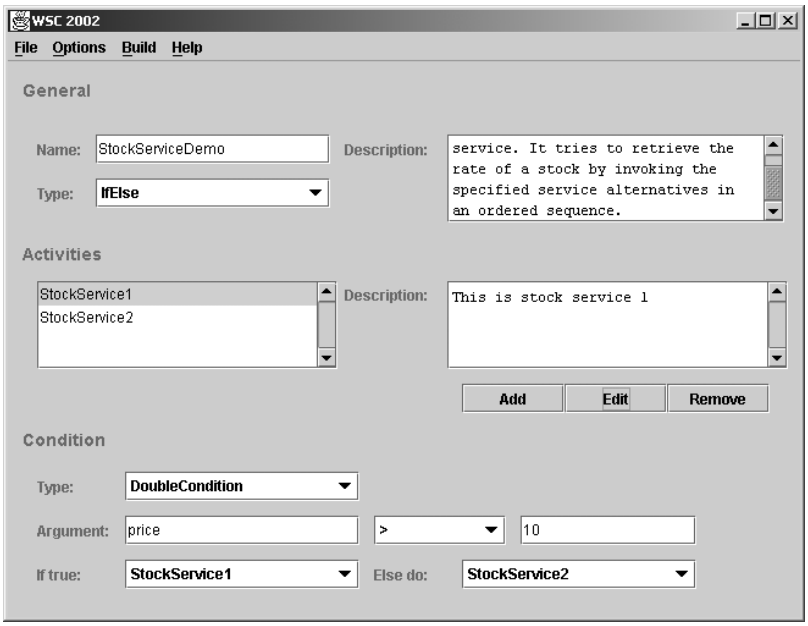

Figure 12: The main window of Service-Com.

The final part of the prototype deals with the invocation (execution) of a web service composition and the corresponding result handling. This part of the prototype system traverses the SCEG paths specified for service component construction. Two Java dialogs are generated in the scheduling phase, which can be used to respectively invoke a composition and display the invocation's results.

At this stage the prototype system provides for service component definition, scheduling, construction and execution on the basis of relatively simple service reuse, and revision (restriction and extension) mechanisms. No explicit support is provided service inheritance and specialization. These features are currently under implementation together with an XQuery extension that handles message mappings.

8 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 [6] 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 systems which present some overlaps with the ideas reported in the above. For example, work on flexible workflows has focused on dynamic process modification [14]. In this publication workflow

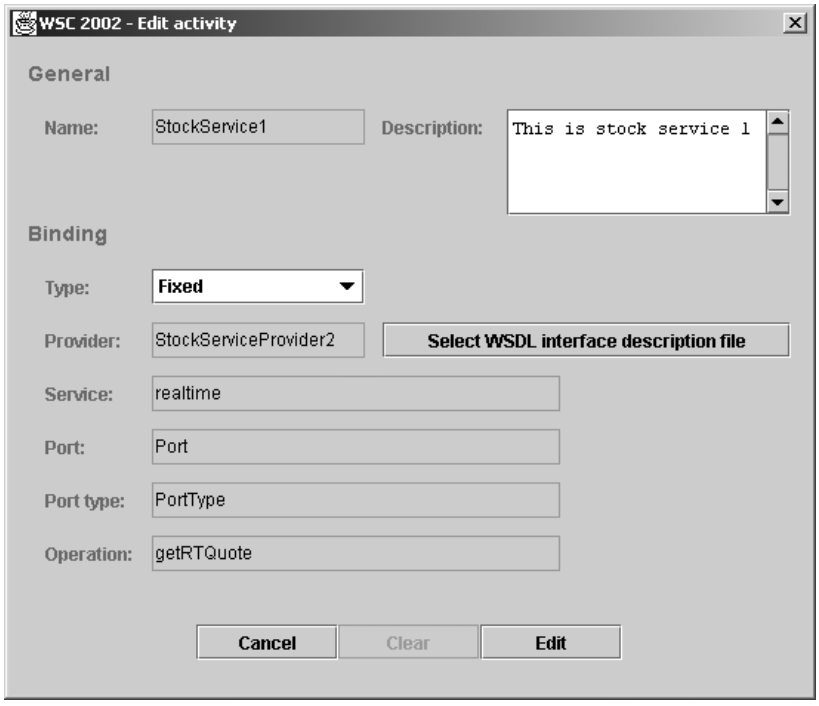

Figure 13: The activity dialogue window for a fixed binding

changes are specified by transformation rules composed of a source schema, a destination schema and of conditions. The workflow system checks for parts of the process that are isomorphic with the source schema and replaces them with the destination schema for all instances for which the conditions are satisfied.

The approach described in [11] 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 [7] 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.

Work related to web-services and coordination/composability can be found in CSCW and groupware publications [20]. In this publication the authors examine the potential of using coordination technology to model electronic business activities and illustrate the benefits of such an approach.

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 [4, 5], 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 enterprizes 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.

Based on the above arguments [4] proposes the idea of defining B2B protocols for inter-enterprize process execution. B2B protocols expose the public processes while WfMSs implement the private processes of an enterprize. This approach provides an interesting way of binding private and public processes together which lays a foundation for service description, monitoring and contracts. However, it is not clear how these can be used in service composition.

In [5], a Composition Service Definition Language (CSDL) was proposed, which supports dynamic service selection, data mappings and extraction. The Composite Service Engine is very much like a workflow engine.

Our approach differs from the above activities in the following manner:

- We propose an integrated approach towards service composition, which covers the entire service composition life-cycle for service components spanning abstract service definition, scheduling, construction and execution.
- The concept of service component is introduced for web service reuse, specialization, and extension.
- During the scheduling stage, variables and macros can be introduced in the SSL that 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.

9 Conclusion and future work

It is obvious that service composition is not just an interoperability problem. The real challenge in service composition lies in providing a complete solution in terms of a framework and a toolset that manage the entire life-cycle of service composition. If this approach is not followed, solutions suffer from the same problem as classical workflow integration practices: service composition is ad- hoc, pre-determined and pre-specified, almost impossible to specialize and extend, and applicability is limited to only a few narrow cases and applications.

In this paper, we analyzed the different forms of service composition and their essential characteristics. In order to support the need for flexible, scalable, extensible service compositions, we introduced the concept of service component that raises the level of abstraction in web services by packaging together elementary or complex services and presenting their interfaces and composition logic in a consistent and uniform manner in the form of customizable class definitions. Based on the concept of service component we proposed an integrated framework that manages its entire life-cycle ranging from abstract service component definition, scheduling, and construction of concrete service components to their execution.

Service components are fully executable and portable between service component-conformant environments. Service components interoperate with WSDL or BPEL conformant web services, irrespectively whether these are represented in terms of service components or not.

The service component approach is light-weight, flexible, and leads to reusable and customizable service components when compared with current popular workflow solutions for web services.

Future research activities concentrate on combining our previous work on service planning [1] with the work reported herein to automatically generate service components out of service request language expressions. In addition, we also consider how non-functional service properties, such as price, security mechanisms, and performance, may impact the choice of service alternatives that are generated by the scheduler.

A The Schema of SCSL

```
<element name="compositeService" type="compositeServiceType"/>

<complexType name="compositeServiceType">
  <sequence>
    <element name="defn" type="Defn"/>
    <element name="construction" type="Construction"/>
  </sequence>
  <attribute name="name" type="string"/>
</complexType>

<complexType name="Defn">
  <element name="PortType" minOccurs="1">
    <complexType>
      <element name="operation" minOccurs="1">
        <complexType>
          <element ref="wsdl:input"/>
          <element ref="wsdl:output"/>
          <attribute name="name" type="string"/>
        </complexType>
      <attribute name="name" type="string"/>
```

```
    </complexType>
</complexType>

<complexType name="Construction">
  <element name="composition">
    <complexType>
      <sequence>
        <element name="activity" minOccurs="1">
          <complexType>
            <element ref="wsdl:input"/>
            <element ref="wsdl:output"/>
            <element name="performedBy">
              <complexType>
                <attribute name="serviceProvider" type="string"/>
              </complexType>
        <attribute name="name" type="string"/>
      </sequence>
      <element name="messageHandling" minOccurs="1">
        <complexType>
          <element name="messageMapping" minOccurs="0">
            <complexType>
              <element name="source">
                <complexType>
                  <attribute name="message" type="messageType"/>
                </complexType>
              <element name="target">
                <complexType>
                  <attribute name="message" type="messageType"/>
                  <attribute name="query" type="XQuery"/>
                </complexType>
            </complexType>
           <element name="messageDecomposing">
             <complexType>
               <element name="source" minOccurs="1" maxOccurs="1">
                 <complexType>
                   <attribute name="message" type="messageType"/>
                 </complexType>
               <element name="target">
                 <complexType>
                   <attribute name="message" type="messageType"/>
                   <attribute name="query" type="XQuery"/>
                 </complexType>
             </complexType>
           <element name="messageComposing">
             <complexType>
               <element name="source" minOccurs="2">
                 <complexType>
```

```
                <attribute name="message" type="messageType"/>
              </complexType>
            <element name="target" minOccurs="1" maxOccurs="1">
              <complexType>
                <attribute name="message" type="messageType"/>
                <attribute name="query" type="XQuery"/>
              </complexType>
          </complexType>
      </complexType>
    <attribute name="type">
      <simpleType>
        <restriction base="string">
          <enumeration value="seqNoInteraction"/>
          <enumeration value="seqWithInteraction"/>
          <enumeration value="seqAlt"/>
          <enumeration value="paraWithSyn"/>
          <enumeration value="paraAlt"/>
        </restriction>
      </simpleType>
</complexType>
```

Acknowledgements: We wish to thank Bart Orriëns for the implementation of the Serv-Co system and his constructive ideas. We also wish to thank the anonymous reviewers of this paper for their constructive comments and criticism, which resulted in considerably improving the quality of this manuscript.

References

[1] M. Aiello et al. A Request Language for Web-Services Based on Planning and Constraint Satisfaction. in VLDB Workshop on Technologies for E-Services (TES02), 2002.

[2] P. Boinot et al. A Declarative Approach for Designing and Developing Adaptive Components. in Procs of the 15th IEEE Conference on Automated Software Engineering, September 2000.

[3] P. Brada. Towards Automated Component Compatibility Assessment. in Procs of ECOOP Workshop on Component-oriented programming, Budapest, 2001.

[4] C. Bussler. The Role of B2B Protocols in Inter-Enterprise Process Execution. Procs. of the 2nd VLDB-TES Workshop, Rome, 2001.

[5] F. Casati and Ming-Chien Shan. Dynamic and Adaptive Composition of e-services, *Information Systems*, 26(2001), page 143-163, 2001.

[6] 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*.

[7] 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.

[8] V. Christophides, R. Hull, and M. Xiong. Beyond Discreate E-Services: Composing Session-Oriented Services in Telecommunications. Procs. of the 2nd VLDB-TES Workshop, Rome, 2001.

[9] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language doe XML, W3C note, http://www.w3.org/TR/NOTE-xml-ql, 1998.

[10] Eliëns, A.. Principles of Object-Oriented Software Development, Addison-Wesley, Harlow, England, 2nd Edition, 2000.

[11] 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.

[12] P. Herzum, O. Sims. Business Component Factory. J. Wiley & Sons, 2000.

[13] 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)*, September 2001.

[14] G. Joeris and O. Herzog. Managing Evolving Workflow Specifications with Schema Versioning and Migration Rules. TZI Technical Report 15, University of Bremen, 1999.

[15] H. Kuno, M. Lemon, A. Karp, and D. Beringer. Conversations + Interface = Business Logic. Procs. of the 2nd VLDB-TES Workshop, Rome, 2001.

[16] G.M. Kuper and J. Simeon. Subsumption for XML Types. in Procs of International Conference on Database Theory (ICDT'01), London, January 2001.

[17] J. Lu, J. Mylopoulos, J. Ho. Towards Extensible Information Brokers Based on XML. in Procs of 12th Conference on Advanced Information Systems Engineering, Stockholm, June, 2000.

[18] M. Mecella, B. Pernici, and P. Craca. Compatibility of e-Services in a Cooperative Multiplatform Environment. Procs. Of the 2nd VLDB-TES Workshop, Rome, September 2001.

[19] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. in Procs of Extreme Markup Languages, Canada, August, 2000.

[20] G. A. Papadopoulos and F. Arbab. Modelling Electronic Commerce Activities Using Control-Driven Coordination, Ninth International Workshop on Database and Expert Systems Applications, Vienna, Austria, August 1998, IEEE Press.

[21] 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.

[22] M. Papazoglou. The World of e-Business: Web Services, Workflows and Business Transactions. WWW Journal, Kluwer Academic, to appear March 2003.

[23] M. P. Papazoglou, J. Yang Design Methodology for Web Services and Business Processes Procs. of the 3rd VLDB-TES Workshop, Hong-Kong, 2002.

[24] Szyperski, C. Component Software: Beyond Object-Oriented Programming. *Addison-Wesley/ACM-Press,* NY. 1999.

[25] K. Sycara, J. Lu, M. Klusch, S. Widoff. Matchmaking among Heterogeneous Agents on the Internet. in Procs of 1999 AAAI Spring Symposium on Intelligent Agent in Cyberspace, March 1999.

[26] UDDI.org *UDDI Technical White paper*, *http* : *//www.uddi.org/pubs/lru_UDDI_Technical_Paper.pdf*, 2001

[27] Web Service Definition Language. http://www.w3.org/TR/wsdl.

[28] "An XML Query Language, http://www.w3.org/TR/xquery, 2002.

[29] J. Yang, M.P. Papazoglou, and W-J Van Heuvel. Tackling the Challanges of Service Composition. ICDE-RIDE workshop on Engineering E-Commerce/E-Business, San Jose, 2002.

[30] J. Yang. Web Service Componentization: Towards Service Reuse and Specilization. To appear in Communication of ACM, October 2003.

[31] http://www-106.ibm.com/developerworks/library/ws-bpel/

[32] A.M. Zaremski and J.M. Wing. Specification Matching of Software Components. ACM Transactions on Software Engineering and Methodology, Vol 6(4), October 1997.