The VLDB Journal (2003) 12: 333–351 / Digital Object Identifier (DOI) 10.1007/s00778-003-0101-5

Composing Web services on the Semantic Web

Brahim Medjahed[1], **Athman Bouguettaya**[1⋆], **Ahmed K. Elmagarmid**[2⋆⋆]

[1] Department of Computer Science, Virginia Tech, 7054 Haycock Road, Falls Church, VA 22043 USA
e-mail: {brahim,athman}@vt.edu
[2] Department of Computer Sciences, Purdue University, 250 N. University Street, West Lafayette, IN 47907 USA
e-mail: ake@cs.purdue.edu

Edited by V. Atluri, A. Joshi, and Y. Yesha. Received: December 15, 2002 / Accepted: April 16, 2003
Published online: September 23, 2003 –

Abstract. Service composition is gaining momentum as the potential *silver bullet* for the envisioned *Semantic Web*. It purports to take the Web to unexplored efficiencies and provide a flexible approach for promoting all types of activities in tomorrow's Web. Applications expected to heavily take advantage of Web service composition include B2B E-commerce and E-government. To date, enabling composite services has largely been an ad hoc, time-consuming, and error-prone process involving repetitive low-level programming. In this paper, we propose an *ontology*-based framework for the automatic composition of Web services. We present a technique to generate composite services from high-level declarative descriptions. We define formal safeguards for meaningful composition through the use of *composability* rules. These rules compare the *syntactic* and *semantic* features of Web services to determine whether two services are composable. We provide an implementation using an E-government application offering customized services to indigent citizens. Finally, we present an exhaustive performance experiment to assess the scalability of our approach.

Keywords: Semantic Web – Web services – Service composition – Ontology

1 Introduction

The Web was originally created to enable the sharing of information among scientists. It has since evolved to cater to governments, businesses, and individuals to make their data Web accessible. However, a large proportion of today's data on the Web are "understandable" only to humans or custom-developed applications. The *Semantic Web* purports to address this issue [5,6,43]. It is defined as an extension of the "existing" Web in which information is given a well-defined meaning [43]. The ultimate goal of the Semantic Web is to transform the Web into a medium through which data can be shared, understood, and processed by automated tools.

The development of enabling technologies for the Semantic Web is the priority of various research communities. One key technology is the emerging concept of *Web services* [5, 7]. Simply put, a *Web service* is a set of related functionalities that can be programmatically accessed through the Web [38]. Examples of Web services include stock trading, credit check, and electronic tax filing. This powerful concept is gradually taking root because of the convergence of business and government efforts to making the Web the place of choice for all types of activities. The widespread adoption of XML standards including WSDL [46], SOAP [44], and UDDI [45] has spurred intense activity in industry and academia to address Web service research issues. One of the most important issues is the use of the Web as a facilitator of service *outsourcing* [9,38]. This new model would enable companies to significantly reduce their overhead, deploy business solutions quickly, and open up new business opportunities. We identify two types of services: *simple* and *composite*. *Simple* services are Internet-based applications that do not rely on other Web services to fulfill consumer requests. An example of a simple service is a *lemon check* service that provides history information about cars (e.g., whether the car has failed in a previous emission/inspection test). A *composite* service is defined as a conglomeration of outsourced services (simple and/or composite) working in tandem to offer a *value-added* service. An example of a composite service is a *car broker* that outsources from *car dealer*, *financing*, and *insurance* services to provide "complete" car sale solutions.

Service composition has recently taken a central stage as an emerging research area. Several techniques have been proposed in this area [3,4,10,21,29,34]. However, they generally require dealing with low-level programming details, thus making the process of composing services demanding for non-expert users. Composers need to identify the way operations are interconnected, services invoked, and messages mapped to one another. For example, assume that the *car broker* provides an operation `insuranceQuote` that requests quotes

⋆ This author's work is supported by the National Science Foundation's Digital Government Program under grant 9983249-EIA and by a grant from the Commonwealth Information Security Center (CISC).
⋆⋆ This author's work is supported by the National Science Foundation's Digital Government Program under grant 9983249-EIA.

from an *insurance* service. Assume that `insuranceQuote` sends the following information to the *insurance* service: `firstName`, `lastName`, `SSN` (social security number), and `dateOfBirth`. To enable interactions between both services, the composer would have to select the *insurance* service operation to invoke and ensure that this operation does not require additional information (e.g., customer's address) to process the request. Additionally, the composer needs to check that the types of data expected by the *insurance* service operation are compatible with the types of data sent by `insuranceQuote`.

The *automatic* composition of Web services is a recent trend to deal with the aforementioned problems [6,23]. This would include the automatic selection and interoperation of Web services. Automatic composition is slated to play a major role in enabling the envisioned *Semantic Web* [48]. Composers would specify the *what* part of the desired composition (i.e., the actions to be performed), but will not concern themselves with the *how* part (services to be outsourced, how to interact with those services, etc). The process of composing Web services (selecting Web services, plugging their operations, mapping their messages, etc.) must be transparent to users. Detailed descriptions of composite services would be automatically generated from composers' specifications.

The semantics of Web services is crucial to enabling automatic service composition. It is important to insure that selected services for composition offer the "right" features. Such features may be *syntactic* (e.g., number of parameters included in a message sent or received by a service). They may also be *semantic* (e.g., the business functionality offered by a service operation or the domain of interest of the service). To help capture Web services' semantic features, we use the concept of *ontology*. An *ontology* is a shared conceptualization based on the semantic proximity of terms in a specific domain of interest [23]. Ontologies are increasingly seen as key to enabling semantics-driven data access and processing [7]. They are expected to play a central role in the Semantic Web, extending syntactic service interoperability to semantic interoperability [16].

In this paper, we propose a framework for the automatic composition of Web services. Combining the emerging concepts of Web service and ontology is at the core of our approach. Two case study applications are used: B2B E-commerce and E-government. The B2B application (car dealership) is used to illustrate the proposed framework for Web service composition. The E-government application is showcased in our implementation. More precisely, this paper's contribution focuses on the following:

- **Composability model for Web services**: A major issue in the automatic composition of Web services is whether those services are *composable* [5]. *Composability* refers to the process of checking if Web services to be composed can actually interact with each other. We propose a *composability model* for comparing *syntactic* and *semantic* features of Web services.
- **Automatic generation of composite services**: We propose a technique to generate composite service descriptions while preserving the aforementioned composability rules. The proposed technique uses as input a high-level specification of the desired composition. This specification contains the list of operations to be performed through composition without referring to any component service.
- **Prototype implementation and experiments**: We provide a prototype implementation of our approach using emerging Web service standards including WSDL, UDDI, and SOAP. We also conduct a set of experiments to evaluate the performance and scalability of our approach.

The remainder of this paper is organized as follows. In Sect. 2, we present our approach for the semantic description of Web services. In Sect. 3, we describe the proposed composability model for Web services. In Sect. 4, we present a novel technique for the automatic generation of composite service descriptions. In Sect. 5, we use an E-government application to showcase the implementation of the proposed approach. In Sect. 6, we discuss the performance of the proposed approach. In Sect. 7, we give a brief survey of the related work. Finally, we provide concluding remarks in Sect. 8.

2 Semantic description of Web services

Composing Web services requires the description of each service so that other services can understand its features and learn how to interact with it. An emerging language for describing operational features of Web services is WSDL (*Web Services Description Language*) [46]. WSDL is being standardized within the W3C consortium. Major industry leaders are supporting and participating in WSDL development. Hence WSDL will likely gain considerable momentum as *the* language for Web service description. However, WSDL provides little or no support for *semantic* description of Web services. It mainly includes constructs that describe Web services from a syntactic point of view. To cater to Semantic Web-enabled Web services, we extend WSDL with semantic capabilities. This would lay the groundwork for the *automatic* selection and composition of Web services. We define an ontology for Web services and specify it using the emerging DAML+OIL language (Fig. 1). DAML+OIL adopts an object-oriented approach, describing ontologies in terms of classes, properties, and axioms (e.g., subsumption relationships between classes or properties) [16]. DAML+OIL builds on earlier Web ontology standards such as RDF and RDF Schema and extends those languages with richer modeling primitives (e.g., cardinality). Other Web ontology languages such as OWL [42] may also be used to specify the proposed ontology.

We model the proposed ontology using a directed graph (Fig. 1). Nodes represent the ontology's concepts. Unfilled nodes refer to WSDL concepts (e.g., name, binding, input). Gray nodes refer to extended features introduced to augment WSDL descriptions with semantic capabilities. Edges represent relationships between the ontology's concepts. They are labeled with the cardinality of the corresponding relationship. For example, the edge *service* $\rightarrow$ *operation* states that a service has one or more operations. The edge *operation* $\rightarrow$ *input* states that an operation has at most one input message. A Web service is defined by instantiating each ontology concept.

We consider three types of participants in our approach: *providers*, *composers*, and *consumers*. *Providers* are the entities (e.g., credit reporting agency) that offer simple Web services (e.g., *Credit History* service). The provider

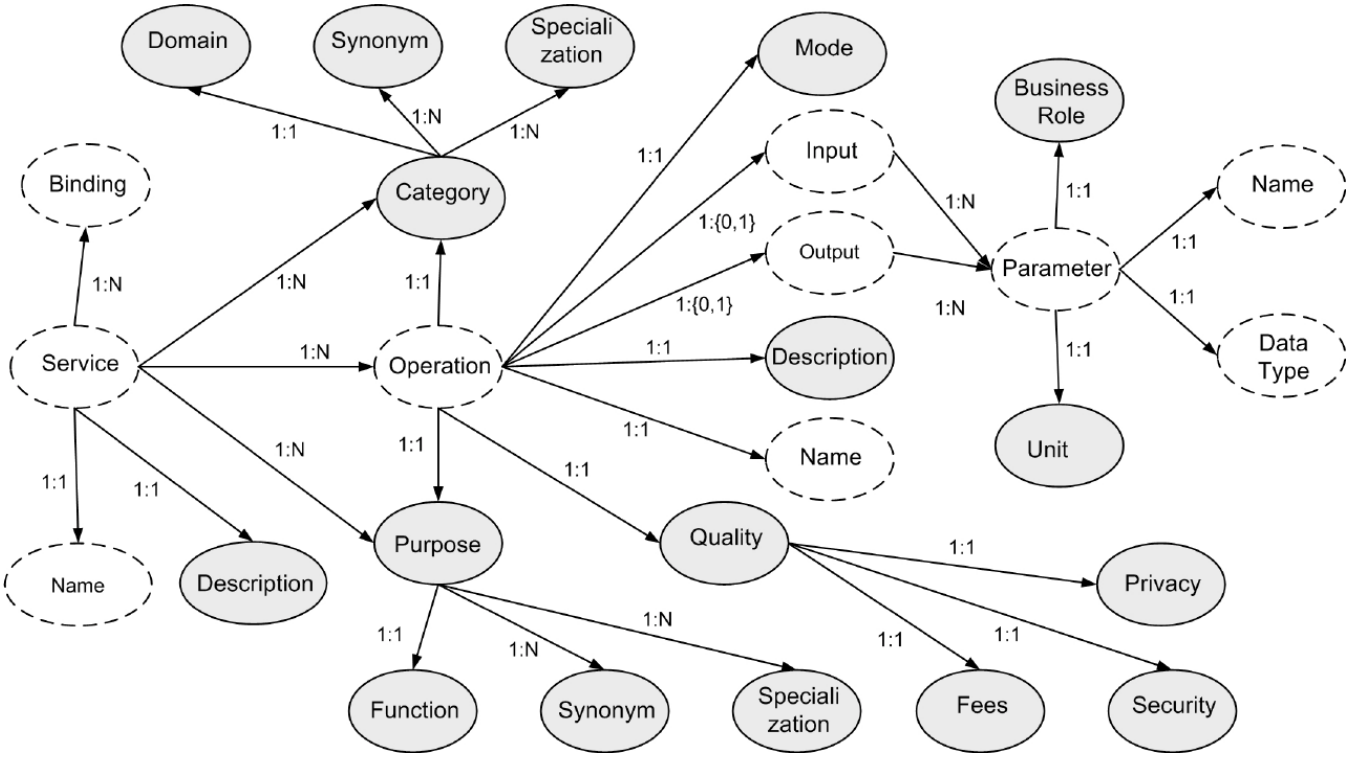

Fig. 1. Ontology-based description of Web services

is responsible for describing its Web service by assigning a value to each ontology concept. *Composers* (e.g., *car broker*'s company) are responsible for specifying composite services. Once generated, composite service descriptions are advertised in a service registry so that they can be discovered. *Consumers* may be end users (e.g., *car broker*'s customers) or other Web services that invoke a Web service (simple or composite).

Example 1. As a running example, we consider the car brokerage application (Fig. 2). Assume that a company provides a *car broker* (CB) composite service that offers car sale packages. The company's customers submit their requests to CB. CB outsources from other Web services to handle each request. Examples of outsourced services include *insurance* (IN), *car dealer* (CD), *lemon check* (LC), *financing* (FI), and *credit history* (CH). A typical scenario would be of a customer using CB service to buy a car having a specific model, make, and mileage. The customer would start by invoking CB's `sendMePriceQuote` operation to get a price quote (step 1). To get a quote, CB would transparently interact with a *car dealer* via CD's `priceQuote` operation (step 1.1). If interested in a used car, the customer would check its history report by invoking CB's `askForProblemCheck` operation (step 2). This operation is processed by outsourcing from LC's `problemCheck` operation (step 2.1). The customer would then apply for financing by invoking the operation `applyForFinancing` provided by CB (step 3). Before accepting a financing plan, CB would check the customer's credit by invoking CH's `payingHistory` operation (step 3.1). If the credit is positive, CB would invoke the `financingQuote` operation offered by the *financing* service (step 3.2). Finally, the customer would request an insurance quote through CB's `insuranceQuote` operation (step 4). CB would transparently invoke the operation `applyforInsurance` offered by the *insurance* service (step 4.1). This service would outsource from DH's `drivingRecord` operation before issuing insurance quotes (step 4.2). ♢

2.1 Mode and messages

The functionalities provided by a Web service are accessible through operation invocations. We consider four operation *modes*: *notification*, *one-way*, *solicit-response*, and *request-response*. A *notification* operation sends an output message but does not expect to receive any response message. In a *one-way* operation, the service receives an input message, consumes it, but does not produce any output message. In a *solicit-response* operation, the service generates an output message and receives an input message in return. In a *request-response* operation, the service receives an input message, processes it, and sends a correlated output message. `CB::sendMePriceQuote` is an example of *solicit-response* operation. The input of this operation includes a `VIN` (vehicle identification number) and `price`. Its output message contains four parameters: `make`, `model`, `year`, and `mileage`. `FI::paymentCalculator` is an example of a *request-response* operation. Its input includes a `purchasePrice`, `downPayment`, and `loanTerm`. The output message of this operation contains `interestRate` and `monthlyPayment`. `CD::specialOffers` is a *notification* operation whose output includes a `make`, `model`, `color`, `year`, `mileage`, and `price`.

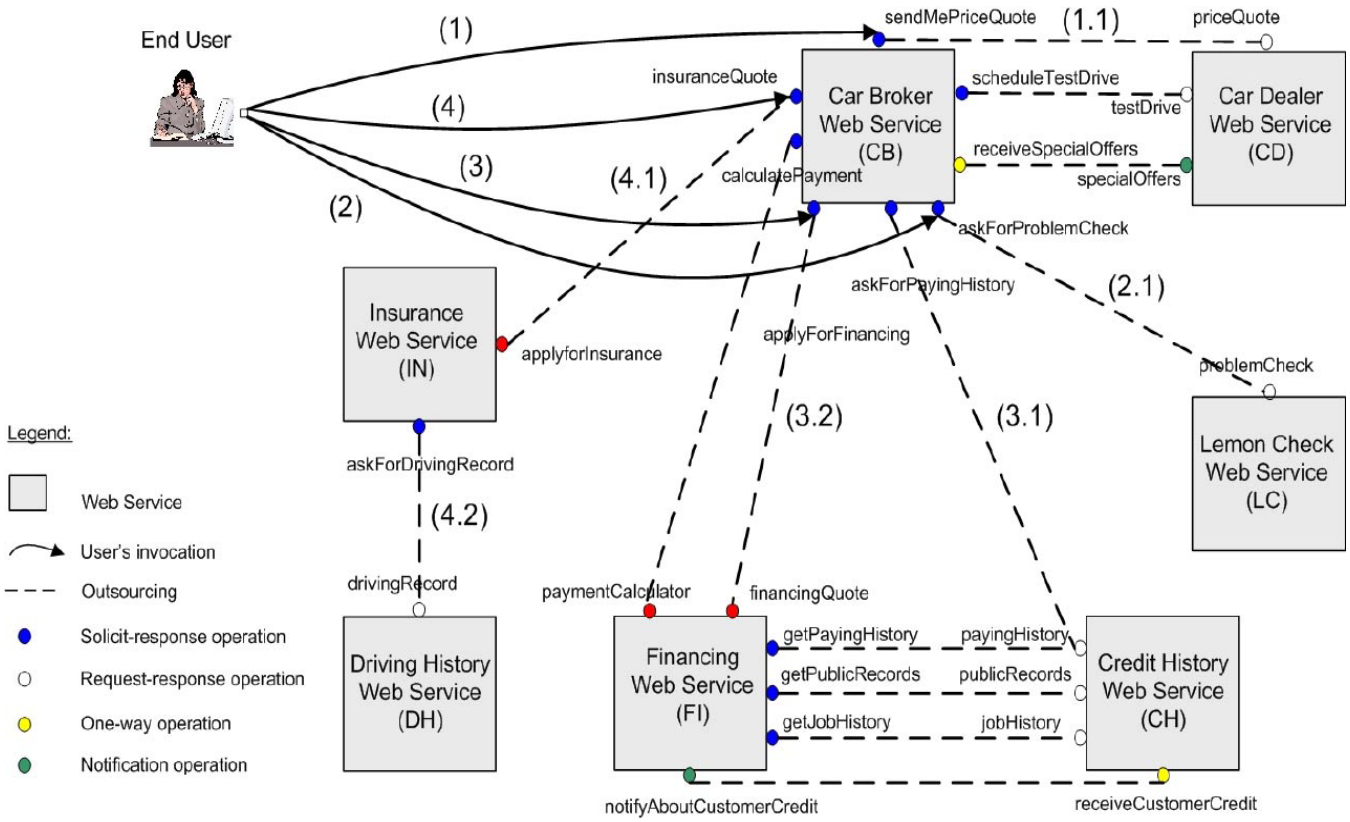

Fig. 2. Scenario: car brokerage application

`CH::receiveCustomerCredit` is a *one-way* operation whose input contains a `firstName`, `lastName`, `SSN`, `dateOfBirth`, and `creditInformation`.

An operation has an *input* and/or *output* message depending on its mode. Request-response and solicit-response operations have both *input* and *output* messages. Notification (with respect to one-way) operations have only *output* (with respect to *input*) messages. Each message contains one or more parameters (called *parts* in WSDL). A parameter has a *name* and *data type*. The data type gives the range of values that may be assigned to the parameter. We use XML Schema's *built-in* data types as the typing system [41]. *Built-in* types are predefined in the XML schema specification. They can be either *primitive* or *derived*. Unlike primitive types (e.g., `string`, `decimal`), derived types are defined in terms of other types. For example, `integer` is derived from the `decimal` primitive type.

While data types are important for the automatic matching of message parameters, they do not capture the semantics of those parameters. For example, the `price` parameter may be in US dollars, yen, or euro. Additionally, it may represent a total price or price without taxes. To model such constraints, we associate a *unit* and a *business role* to each parameter. We use standard measurement units (length, area, weight, money code, etc.) to assign values to parameter units. If a parameter does not have a unit (e.g., `firstName`), its unit is equal to "none". The *business role* gives the semantics of the corresponding parameter. It takes its value from a predefined taxonomy for business roles. Every parameter would have a well-defined semantics according to that taxonomy. An example of such taxonomy is *RosettaNet*'s business dictionary [25]. It contains a common vocabulary that can be used to describe business properties. For example, if the `price` parameter has an "extendedPrice" role (defined in *RosettaNet*), then it represents a "total price for a product quantity". For flexibility purposes, different Web services may adopt different taxonomies to specify their parameters' business roles. We use XML namespacesto prefix business roles with the taxonomy in which they are defined:[1]

Definition 1 – *Message*. A *message M* is defined as a tuple $(P,\mathcal{T},\mathcal{U},\mathcal{R})$ where:

- P is a set of parameter names.
- $\mathcal{T}$: $P \longrightarrow$ *DataTypes* is a function that assigns a data type to each parameter. *DataTypes* is a set of XML data types.
- $\mathcal{U}$: $P \longrightarrow$ *Units* is a function that gives the unit of measurement used for each parameter. *Units* is a taxonomy for measurement units.
- $\mathcal{R}$: $P \longrightarrow$ *Roles* is a function that assigns a business role to each parameter. *Roles* is a taxonomy for business roles. ♢

2.2 Purpose and category

Each operation is semantically described through its *purpose* and *category*. The purpose contains three attributes: *function*, *synonyms*, and *specialization*. The function gives the *business*

[1] XML namespaces provide a method for qualifying element and attribute names used in XML documents by associating them with URI references.

functionality provided by the operation. Examples of functions include "request for quotation", "purchase order", and "delivery order". As for parameters' business roles, a wide range of taxonomies may be used to define the function attribute. Examples of such taxonomies include *RosettaNet*, cXML (*commerce XML*), and EDI (*electronic data interchange*) [25]. Each purpose is prefixed with an XML namespace that points to the corresponding taxonomy. For example, an operation's purpose may be preceded by a namespace that points to *RosettaNet*'s taxonomy for business transactions. Another operation's purpose may be preceded by *cXML* taxonomy's namespace. The *synonyms* attribute contains the set of alternative function names for the operation. For example, "quotation" is a synonym of "request for quotation". The *specialization* attribute defines a set of characteristics of the current function. For example, a "request for quotation" may be "for your information" or "legally binding". It may also include "destination charge" or not, be "valid until a specific date", etc. We summarize below the notion of operation purpose:

Definition 2 – *Purpose*. The *purpose* of an operation op_{ik} is defined by a tuple *(Function, Synonyms, Specialization)* where *Function* is op_{ik}'s business functionality defined within a given taxonomy, *Synonyms* is a set of alternative function names, and *Specialization* is a set of characteristics of op_{ik}'s function. ♢

The *category* of an operation is defined in the same way as its purpose. Each category contains three attributes: *domain*, *synonyms*, and *specialization*. The domain gives the area of interest for the current operation. Examples of domains include "automobile dealers" and "insurance". Taxonomies such as NAICS (*North American Industry Classification System*) and UNSPSC (*Universal Standard Products and Services Classification*) may be used to define the domain attribute. *Synonyms* and *Specialization* attributes work just as they do in the operation's purpose. Synonyms of the domain "automobile dealers" include "car dealers" and "car sellers". An example of a specialization attribute associated with "insurance" is {"car", "home"}. This means that the corresponding operation provides both car and home insurance. We define below the notion of operation category:

Definition 3 – *Category*. The *category* of an operation op_{ik} is defined by a tuple *(Domain, Synonyms, Specialization)* where *Domain* is op_{ik}'s area of interest defined within a given taxonomy, *Synonyms* is a set of alternative domains, and *Specialization* is a set of characteristics of op_{ik}'s domain. ♢

2.3 Operation quality

Several Web services may provide "similar" operations in terms of their mode, message, purpose, and category. It is thus important to define *qualitative properties* that help composers select the "best" Web services [35]. We identify three qualitative properties for operations: *fees*, *security*, and *privacy*. Other properties such as time, availability, and latency may also be added. The fees property indicates the dollar amount required to execute the operation. Security and privacy are two important requirements of Web services [33]. Businesses collect, store, process, and share information about millions of users who have different preferences regarding the privacy and security of their information. Privacy and security are particularly important in E-government applications where citizens are sensitive about their personal information. The security property is a boolean that indicates whether the operation's messages are securely exchanged (e.g., using encryption techniques) between servers and clients. The privacy property contains the input and output parameters that should not be divulged to external entities (i.e., other than the service provider). If a parameter does not belong to the *privacy* set, then no privacy constraint is specified on that parameter. Assume that *privacy* = {`SSN`, `Credit Card Number`} where `SSN`, `Credit Card Number` are two input parameters. This property states that those parameters are kept private by the service provider. Based on the aforementioned properties, we define below the notion of operation quality:

Definition 4 – *Quality*. The *quality* of an operation op_{ik} is defined by a tuple $(Fees_{ik}, Security_{ik}, Privacy_{ik})$. $Fees_{ik}$ is the dollar amount needed to execute op_{ik}. $Security_{ik}$ is a boolean that specifies whether op_{ik}'s messages are securely exchanged. $Privacy_{ik}$ is the set of input and output parameters that are not divulged to external entities. ♢

2.4 Defining operations and Web services

Web services are accessible via operations. Each operation is identified by a *name* and a text *description* that summarizes the operation's features. It also has a *mode*, *input* and/or *output* messages, *purpose*, and *category*. We present below a definition of a service operation:

Definition 5 – *Operation*. An operation op_{ik} is defined by a tuple $(Description_{ik}, Mode_{ik}, In_{ik}, Out_{ik}, Purpose_{ik}, Category_{ik}, Quality_{ik})$ where:

- $Description_{ik}$ is a text summary about the operation features.
- $Mode_{ik} \in$ {"one-way", "notification", "solicit-response", "request-response"}.
- In_{ik} and Out_{ik} are the input and output messages, respectively. $In_{ik} = (\emptyset, \mathcal{T}_{ik})$ and $Out_{ik} = (\emptyset, \mathcal{T}_{ik})$ for notification and one-way operations, respectively.
- $Purpose_{ik}$ describes the business function offered by the operation (cf. Definition 2).
- $Category_{ik}$ describes the operation's domain of interest (cf. Definition 3)
- $Quality_{ik}$ gives the operation's qualitative properties (cf. Definition 4). ♢

Example 2. The operation `CB::sendMePriceQuote` is defined by the tuple *(Desc, Mode, In, Out, P, C, Q)* where:

- *Desc* = "this operation returns the price quotation for a given car"; *Mode*="solicit-response".
- *In* = $(P, \mathcal{T}, \mathcal{U}, \mathcal{R})$, so that P = {`VIN`, `price`}; $\mathcal{T}$(`VIN`) = "positiveInteger"; $\mathcal{T}$(`price`) = "float". $\mathcal{U}$(`VIN`) = "none"; $\mathcal{U}$(`price`) = "US dollar"; $\mathcal{R}$(`price`) = "extendedPrice".
- *Out* = $(P, \mathcal{T})$ so that P = {`make`, `model`, `year`, `mileage`}; $\mathcal{T}$(`make`) = "string"; $\mathcal{T}$(`model`) = "string"; $\mathcal{T}$(`year`) = "gYear"; $\mathcal{T}$(`mileage`) = "positiveInteger"; $\mathcal{U}$(`mileage`) = "mile".

- *P* is the operation purpose defined as follows: *P.Function* = "request for quotation"; *P.Specialization* = {"for your information"}; and *P.Synonyms* = {"quotation"}.
- *C* is the operation category defined as follows: *C.Domain* = "automobile dealers"; *C.Synonyms* = {"car dealers"}; and *C.Specialization* = {"used cars"}.
- *Q* is the operation's quality defined as follows: *Q.Fees* = 0; *Q.Security* = "false"; and *Q.Privacy* = ∅ (i.e., no sensitive information is exchanged). ♢

A Web service is identified by a *name* and a text *description* that summarizes the service features. Interactions with the service are performed according to a specific *binding* [46]. The *binding* defines message format and protocol details for service operations and messages. Examples of bindings include *SOAP* (*Simple Object Access Protocol*), *HTTP Get/Post* and MIME (*Multipurpose Internet Mail Extensions*). A service may have several bindings associated with it. For each Web service, we also associate a *purpose* and *category*. The *purpose* describes the business functionalities offered by the service operations. Each element in the service *purpose* refers to the business functionality offered by a specific operation. The *category* describes the service domain of interest. It includes the categories of all operations provided by the service. It also contains an element that corresponds to the category of the service. Indeed, the domain of interest of a composite service may be different from the domains of interest of its operations. For example, the *car dealer* composite service is related to the "automobile dealers" industry. Yet it includes operations related to "insurance" (e.g., `insuranceQuote`) and "mortgage and nonmortgage loan" (e.g., `financingQuote`) industries. Below we give a formal definition of a Web service (composite or simple):

Definition 6 – *Web service*. A *Web service* WS_i is defined by a tuple $(Description_i, OP_i, Bindings_i, Purpose_i, Category_i)$ where:

- $Description_i$ is a text summary about the service features.
- OP_i is a set of operations provided by WS_i.
- $Bindings_i$ is the set of binding protocols supported WS_i.
- $Purpose_i = \{Purpose_{ik}(op_{ik}) \mid op_{ik} \in OP_i\}$ is a set of WS operations' purpose.
- $Category_i = \{Category_{ik}(op_{ik}) \mid op_{ik} \in OP_i\} \cup \{Category_i(WS_i)\}$ is a set of WS_i operations' categories. ♢

Example 3. We consider the *car dealer* (CD) service depicted in Fig. 2. This service is defined by the tuple *(Desc, OP, B, P, C)*, where *OP* = {`priceQuote`, `testDrive`, `specialOffers`}, and *B* = {"SOAP"}. *P* is the service purpose defined by the set {*purpose(*`priceQuote`*)*, *purpose(*`testDrive`*)*, *purpose(*`specialOffers`*)*}. *C* is the service category defined by the set {*category(*`priceQuote`*)*, *category(*`testDrive`*)*, *category(*`specialOffers`*)*} ∪ {*category(CD)*}. Operation purposes and categories are defined in the same way as in Example 2. The *category(CD)* element is defined as follows: *category(CD).domain* = "automobile dealers"; *category(CD).synonyms* = {"car dealers", "car sellers"}; and *category(CD).specialization* = {"used cars"}. ♢

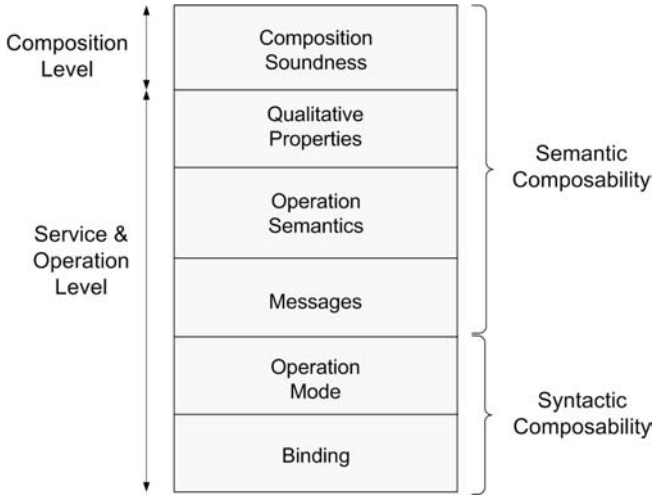

Fig. 3. Composability model for Web services

3 Composability model for Web services

A major issue when defining a composite service is whether its component services are *composable* [5]. For example, it would be difficult to invoke an operation if there were no mapping between the parameters requested by this operation (e.g., data types, number of parameters) and those transmitted by the client service. In this section, we identify two sets of *composability rules* to compare *syntactic* and *semantic* properties of Web services (Fig. 3). Syntactic rules include: (1) *mode composability*, which compares operation modes, and (2) *binding composability*, which compares the binding protocols of interacting services. Semantic rules include (1) *message composability*, which compares the number of message parameters, their data types, business roles, and units; (2) *operation semantics composability*, which compares the semantics of service operations; (3) *qualitative composability*, which compares qualitative properties of Web services; and (4) *composition soundness*, which checks whether combining Web services in a specific way is worthwhile. *Composition soundness* checks composability at the composition level, unlike the other rules that deal with composability at the service and operation levels.

In our model, clients and servers refer to *outsourcing* (e.g., *car broker*) and *outsourced* (e.g., *insurance*) services, respectively. We adopt the approach defined in XLANG [28], WSFL [18], and BPEL4WS [2] standards, that is, Web services at both sides (client and server) are defined in WSDL augmented with semantic capabilities. As in those standards, we assume that each operation on the server side has at most one matching operation at the client side and vice versa [28,18,2].

3.1 Mode and binding composability

For Web service interactions to take place, operations at client and server sides must have "dual" modes [18, 28]. A *notification* operation at one service must be connected to a *one-way* operation at a partner service. Similarly, a *solicit-response* operation maps to a *request-response* operation at a partner service. For ex-

ample, the operation `CB::sendMePriceQuote` (solicit-response) maps to `CD::priceQuote` (request-response), and `CB::receiveSpecialOffers` (one-way) maps to `CD::specialOffers` (notification). The following rule, called *mode composability*, checks whether two operations have "dual" modes.

Definition 7 – *Mode composability*. Two operations op_{ik}=(D_{ik}, M_{ik}, In_{ik}, Out_{ik}, P_{ik}, C_{ik}, Q_{ik}) and op_{jl}=(D_{jl}, M_{jl}, In_{jl}, Out_{jl}, P_{jl}, C_{jl}, Q_{jl}) are *mode composable* if (i) M_{ik} = "notification" and M_{jl} = "one-way"; or (ii) M_{ik} = "one-way" and M_{jl} = "notification"; or (iii) M_{ik} = "solicit-response" and M_{jl} = "request-response"; or (iv) M_{ik} = "request-response" and M_{jl} = "solicit-response". ♢

Assume now that two Web services are communicating through operations that are mode composable. Since these Web services may support different binding protocols (e.g., SOAP, HTTP, or MIME), it is important to insure that they "understand" each other at the message format and protocol level. At least one of the protocols expected by a Web service must be supported by the other. For example, it would be difficult for a service that expects to receive messages in MIME protocol to interact with another service that formats its messages in HTTP. The following rule, called *binding composability*, checks that Web services support at least one common binding protocol.

Definition 8 – *Binding composability*. Two services WS_i = (D_i, O_i, B_i, P_i, C_i) and WS_j = (D_j, O_j, B_j, P_j, C_j) are *binding composable* if $B_i \cap B_j \neq \emptyset$. ♢

3.2 Message composability

Interactions between Web services involve the exchange of messages. A message consists of one or more parameters, each having a certain data type. Hence it is important to check that the data types of the parameters sent by a service are compatible with the data types of the parameters received by its partner. We consider two primary data-type-compatibility methods: *direct* and *indirect compatibility*. Two parameters are *directly compatible* if they have the same data type. A parameter p is *indirectly compatible* with a p' if the type of p is derived from the type of p'. For example, a parameter with a `positiveInteger` or `short` type is indirectly compatible with an `integer` parameter. Note that, contrary to direct compatibility, indirect compatibility is asymmetric.

We extend the notion of data type compatibility to messages as follows: A message M is *data type compatible* with a message M' if every parameter of M is directly or indirectly compatible with a parameter of M'. Note that not all parameters of M' need to be mapped to the parameters of M. The rationale is that an input message of a service operation may use only a subset of the parameters sent through an output message of a "dual" operation. For example, assume that the *car broker* provider is not interested in knowing the color of cars advertised as special offers. In this case, she/he defines the input message of the `CB::receiveSpecialOffers` as composed of the following parameters: "make", "model", "year", "mileage", "price". The input message is compatible with the output message of `CD::specialOffers`, although the output message contains an additional parameter.

Assume now that a parameter p is compatible (directly or indirectly) with a parameter p'. For p and p' to be mapped together, they must also have compatible semantics. For example, a total price should not be mapped with a price before taxes. Similarly, a price in US dollars should not be mapped to a price in yen. To this end, we compare the units and business roles of p and p'. Both parameters should have the same units and business roles. The *message composability* rule compares the input and output messages of every pair of operations. The idea is to check that each input of an operation is data type compatible with the output of the other operation. The input's unit and business role should be the same as the output's unit and business role. This means that the parameters of each input message map to all or some of the parameters contained in the output message of the other operation:

Definition 9 – *Message composability*. Two operations op_{ik} = (D_{ik}, M_{ik}, In_{ik}, Out_{ik}, P_{ik}, C_{ik}, Q_{ik}) and op_{jl}=(D_{jl}, M_{jl}, In_{jl}, Out_{jl}, P_{jl}, C_{jl}, Q_{jl}) are *message composable* if:

1. $\forall\, p \in In_{ik}$, $\exists\, p' \in In_{jl} \mid p$ is data type compatible with p', and $\mathcal{U}(p) = \mathcal{U}(p')$, and $\mathcal{R}(p) = \mathcal{R}(p')$.
2. $\forall\, p \in In_{jl}$, $\exists\, p' \in In_{ik} \mid p$ is data type compatible with p', and $\mathcal{U}(p) = \mathcal{U}(p')$, and $\mathcal{R}(p') = \mathcal{R}(p')$. ♢

3.3 Operation semantics composability

This rule ensures that interconnected operations have "*compatible*" purposes and categories. For example, the function of `CB::sendMePriceQuote` (i.e., "request for quotation") is different from the function of `CD::testDrive` (i.e.,"reservation"). It would be semantically "incorrect" to map these operations since they offer different business functions. Similarly, `IN::applyForInsurance` is "not semantically compatible" with `CB::calculatePayment` since these operations have different domains ("insurance" and "mortgage and nonmortgage loan", respectively). To define *compatibility* between operation categories, let us consider the two operations op_{ik} = (D_{ik}, M_{ik}, In_{ik}, Out_{ik}, P_{ik}, C_{ik}, Q_{ik}) and op_{jl} = (D_{jl}, M_{jl}, In_{jl}, Out_{jl}, P_{jl}, C_{jl}, Q_{jl}). We say that C_{ik} is *compatible with* C_{jl} if:

1. (C_{ik}.Domain = C_{jl}.Domain) or (C_{ik}.Domain $\in$ C_{jl}.Synonyms) or (C_{jl}.Domain $\in$ C_{ik}.Synonyms); or (C_{ik}.Synonyms $\cap$ C_{jl}.Synonyms $\neq \emptyset$); and
2. C_{ik}.Specialization $\subseteq$ C_{jl}.Specialization

The first condition verifies that the domains of interest are similar (first disjunct) or synonyms (second, third, and fourth disjuncts). The second condition ensures that op_{jl} provides all the characteristics of op_{ik}'s category. Compatibility between purposes is defined in the same way as between categories. Based on the notion of *compatibility* between categories and purposes, we define the *operation semantics composability* rule. We say that op_{ik} is *operation semantics composable* with op_{jl} if the purpose of op_{ik} is compatible with the purpose of op_{jl} and the category of op_{ik} is compatible with the category of op_{jl}:

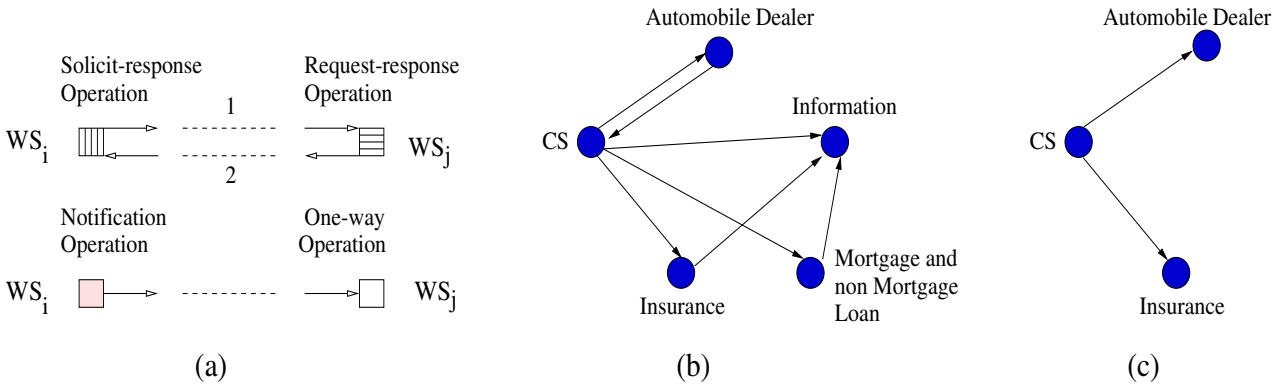

Fig. 4a–c. Composition soundness

Definition 10 – *Operation semantics composability*. We say that op_{ik} = (D_{ik}, M_{ik}, In_{ik}, Out_{ik}, P_{ik}, C_{ik}, Q_{ik}) is *operation semantics composable* with op_{jl} = (D_{jl}, M_{jl}, In_{jl}, Out_{jl}, P_{jl}, C_{jl}, Q_{jl}) if (i) P_{ik} is *compatible with* P_{jl} and (ii) C_{ik} is *compatible with* C_{jl}. ♢

3.4 Qualitative composability

Composers generally have preferences regarding the quality of operations they would like to outsource. Qualitative composability rules check the qualitative properties of interacting operations. Let us consider an operation op_{ik} that outsources from another operation op_{jl}. The *fees* composability verifies that the dollar amount op_{ik} is willing to pay is at least equal to the amount required by op_{jl}. *Security* composability guarantees that if op_{ik} uses security mechanisms (e.g., encryption and nonrepudiation) to exchange messages, then op_{jl} uses them also. *Privacy* composability compares op_{ik}'s and op_{jl}'s privacy features. The privacy preferences of op_{ik} should be subsumed by the privacy features exposed by op_{jl}. If op_{ik}'s provider does not want a parameter p to be divulged (i.e., $p \in Quality_{ik}.privacy$), then p should also belong to $Quality_{jl}.privacy$. The following definition summarizes the qualitative composability rule:

Definition 11 – *Qualitative composability*. We say that op_{ik} = (D_{ik}, M_{ik}, In_{ik}, Out_{ik}, P_{ik}, C_{ik}, Q_{ik}) is *qualitatively composable with* op_{jl} = (D_{jl}, M_{jl}, In_{jl}, Out_{jl}, P_{jl}, C_{jl}, Q_{jl}) if:

1. Q_{ik}.Fees $\geq$ Q_{jl}.Fees; and
2. (Q_{ik}.Security = true) $\Rightarrow$ (Q_{jl}.Security = true); and
3. Q_{ik}.Privacy $\subseteq$ Q_{jl}.Privacy. ♢

3.5 Composition soundness

Another important aspect to consider in service composition is whether combining a set of services in a specific way provides an added value. For example, it would probably be "unusual" to combine a *lemon check* service with a *hotel* service. However, combining *airline* and *hotel* services would provide a *travel preparation* composite service. The idea is to define a rule, called *composition soundness*, to test whether composite services are *sound*. By *sound*, we mean that the way component services are composed provides an added value. To this end, we introduce the notion of *composition templates*. These are graphs built using *precedence relationships*. As depicted in Fig. 4a, a Web service WS_i *precedes* another service WS_j if an operation of WS_i invokes an operation of WS_j. We give below a formal definition of the precedence relationship:

Definition 12 – *Precedence relationship*. Let WS_i = (D_i, O_i, B_i, P_i, C_i) and WS_j = (D_j, O_j, B_j, P_j, C_j) be two Web services. WS_i *precedes* WS_j if $\exists$ $op_{ik} \in O_i$ $\exists$ $op_{jl} \in O_j$ | (i) (M_{ik}="notification" and M_{jl}="one-way"); or (ii) (M_{ik}="solicit-response" and M_{jl}="request-response"). ♢

A *composition template* is associated with each composite service and gives the general structure of that service. It is modeled by a directed graph (V, E) where V is a set of service category names and E is a set of edges. A special vertex corresponds to the composite service and has the special value "CS". An edge (v_i, v_j) $\in$ E means that a service of category name v_i precedes a service of category name v_j. Figure 4b gives the template corresponding to the *car broker* composite service. *Lemon check*, *credit history*, and *driving history* services are represented by the same node in the graph since they have the same category name (i.e., "information").

Composition templates are used to compare the values added by different compositions. For example, consider the template depicted in Fig. 4c. This template is a subgraph of the template depicted in Fig. 4b. This means that the second composite service would provide a subset of the functionalities offered by the first one. For example, it does not provide "financing" operations since it does not outsource from a "mortgage and nonmortgage loan" service.

To check whether a composition of services is *sound*, we define the notion of *stored templates*. *Stored templates* are divided into two groups. The first group includes templates that are predefined by domain experts (e.g., cars, travel, computers). For example, the travel industry would agree that a *travel preparation* composite service combines *airline*, *hotel*, and *car rental* services. The second group includes templates that are "learned" by the system. Each time a composite service is defined, the system stores its template in the repository. For example, assume that the composer defines a service whose template is depicted in Fig. 4b. If the template does not already exist in the repository, the system would store it for future use. Since stored templates inherently provide added values, they are used to test the soundness of composition plans.

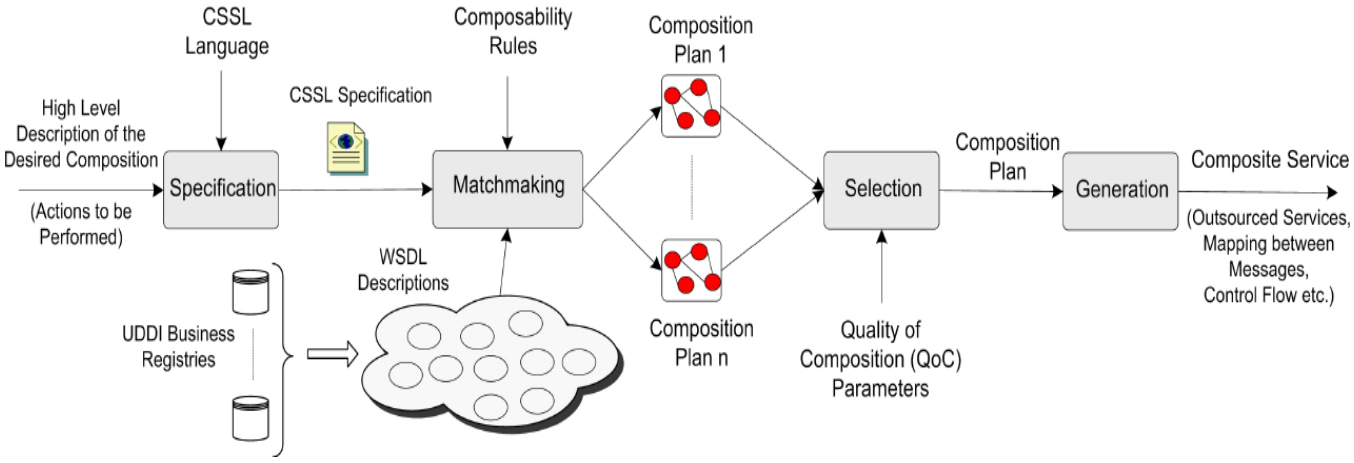

Fig. 5. Overview of the proposed approach for service composition

Definition 13 – *Composition soundness*. A composition of services is *sound* if its template is a subgraph of a *stored template*. ♢

Stored templates are different from *process templates* and *reference processes* used in [10] and [34], respectively. Indeed, process templates and reference processes are used as an a priori "canvas" when defining composite services. In contrast, stored templates are used a posteriori to check the soundness of composite services, i.e., once they have been generated. It is important to note that the composition soundness rule is not used to determine if Web services are composable. It is rather used to determine if composing a given set of services provides an added value. Even if a composition is not sound, composers have the flexibility to decide whether they are willing to consider such composition as "acceptable".

4 Automatic composition of Web services

Based on our composability model, we propose an approach for the automatic composition of Web services. This approach consists of four conceptually separate phases: *specification*, *matchmaking*, *selection*, and *generation* (Fig. 5). The *specification* phase enables high-level descriptions of the desired compositions using a language called CSSL (*Composite Service Specification Language*). The *matchmaking* phase uses composability rules to generate *composition plans* that conform to composers' specifications. By *composition plan*, we mean the list of component services and their interactions with each other (plugging operations, mapping messages, etc.) to form the composite service. The matchmaking algorithm uses as input the composer's specification and a repository (e.g., UDDI [45]) of preexisting service interfaces described in WSDL (extended with semantic constructs). Composers select a generated plan (*selection phase*) based on *quality of composition* (*QoC*) parameters (e.g., ranking, cost). Using the selected plan, a detailed description of the composite service is automatically generated (*generation phase*). This description includes the list of outsourced services, mappings between the composite and oustourced services operations and messages, and the control flow of outsourced operations. The control flow refers to the execution order of the operations outsourced by the composite service.

Specification is the phase that requires the composer's intervention to describe the desired composition. The matchmaking phase automatically generates composition plans based on the composer's specification. The selection phase uses QoC parameters (composition and relevance thresholds) to select the best plan. Those parameters are given by composers based on their *profiles* defined in the specification process. The generation phase automatically provides a description of the generated composite service in a given "target" language (e.g., WSFL).

4.1 Specification phase

We define an XML language, called *CSSL* (*Composite Service Specification Language*), for the specification of composite services. CSSL is simple enough to enable high-level descriptions of composite services. Composers only need to have a general idea about the service they are interested in offering (e.g., the operations to be outsourced). They are not required to be aware of the full technical details such as descriptions of the component services, their characteristics (e.g., data types), and how they are plugged together. There are several differences between CSSL and existing service composition languages ([18,28,2,13,21,10,34]). First, CSSL adopts the ontology-based model introduced in Sect. 2 to cater to Semantic Web-enabled Web services. Most of the existing languages do not consider semantic capabilities of Web services. Second, the CSSL specification of a composite service does not refer to any oustourced service. This is in contrast to other languages where composers insert references to component services in their composite service specifications. Third, CSSL specifications are used as the entry point for the (semi-)automatic generation of composite services. Finally, CSSL defines a WSDL-like language for composite services. It extends WSDL language to allow: (1) the description of semantic features of Web services and (2) specification of the control flow between composite service operations. This makes the definition of composite services as simple as the definition of simple (i.e., noncomposite) services. Additionally, it allows the support of recursive composition of services. Composite services can be considered as WSDL services and hence be used as components for new compositions.

Table 1. CSSL specification of the car broker composite service

```
<service name="car broker"/>
 <category domain="brokerage">
 .....
 <binding name="SOAP"/>
 <message name="offer">
 <parameter name="price" type="float" unit="US dollar" role="extendedPrice"/>
 <parameter name="make" type="string" ...../>
 <parameter name="model" type="string" ...../>
 <pararmeter name="year" type="gYear" ...../>
 <parameter name="mileage" type="integer" ...../>
 </message>
 .....
 <operation name="receiveSpecialOffers" mode="one way"/>
 <input name="offer"/>
 <category domain="automobile dealer">
 <synonyms>
 <synonym value="car dealer"/>
 </synonyms>
 .....
 <purpose function="price-sales catalogue"/>
 .....
 <quality>
 <fees value=0/>
 .....
 </operation>
 <flow source="getPayingHistory" target="applyForFinancing">
```

We illustrate the main features of CSSL through the example depicted in Table 1. For the sake of clarity, we omit references to XML namespaces. The top element of a CSSL specification is *service* which includes the name (*car broker*) of the composite service. The service category attributes (i.e., domain, synonyms, and specialization) are specified within the *category* element. Each *operation* element allows the specification of the operation name, description, mode, purpose, category, and quality. It also contains *input* and/or *output* elements. For example, the `receiveSpecialOffers` operation is one-way (*mode* attribute) and provides price and sales information (*purpose* element) for the automobile industry (*category* element). The operation's input message, named `offer`, contains five parameters (e.g., `price`). Each parameter has an XML Schema data type (e.g., "float"), unit (e.g., "US dollar"), and role (e.g., extendedPrice"). To facilitate the definition of input and output message, we provide a set of *predefined messages*; these are used by composers as a basis for defining composite service operations. Composers can define new messages, use predefined messages, or modify predefined messages at their own convenience. For example, they may decide to remove *TOD* (*terms of delivery or transport*) part from a "request for quotation" predefined message if they are not interested in offering such information.

CSSL also enables the specification of the control flow of composite service operations (*flow* attribute). Operations may be executed sequentially or in parallel. For example, the specification of the *car broker* service shows that this service first checks the payment history of its customers (*source* attribute) before applying for financing on their behalf (*target* attribute). Note that CSSL also allows the specification of conditions on control flows. For example, the *car broker* service would apply for financing only if the current customer has a payment history.

4.2 Matchmaking phase

Once CSSL specifications are provided, the next step is to generate corresponding composition plans using a *matchmaking algorithm* (Table 2). Since the number of generated plans may be large, composers have the possibility to control the number of generated plans through the *nb_requested_plans* input. The algorithm uses service interfaces rather than "whole" descriptions of component services for checking compatibility. This has the important advantage of decreasing the number of services to be accessed. Indeed, the same interface may be used by several existing services with different implementations [46]. For example, the car industry may define an interface for selling cars. *Car dealer* services would then reuse this interface to create their own services.

The general premise of the matchmaking algorithm is to map each operation $op_{ik} = (D_{ik}, M_{ik}, In_{ik}, Out_{ik}, P_{ik}, C_{ik}, Q_{ik})$ of the composite service WS_i to one or more operations $op_{jl} = (D_{jl}, M_{jl}, In_{jl}, Out_{jl}, P_{jl}, C_{jl}, Q_{jl})$ of existing service WS_j. The algorithm looks for Web services $WS_j = (D_j,, O_j, B_j, P_j, C_j)$ so that P_{ik} and C_{ik} are compatible with at least one element of P_j and C_j, respectively (lines 8 and 9). Then the algorithm verifies that interacting services are binding composable (second condition in line 8). We organize Web services into *communities*. Communities provide means for an ontological organization of the available service space. Each community clusters Web services based on their category. All

services that have similar categories belong to the same category (purpose) community. A service may belong to different communities. The use of service communities accelerates the process of discovering relevant component services. Assume that the category C_{ik} is not compatible with WS_j's category. In this case, C_{ik} is not compatible with the services that belong to WS_j's community. Hence those Web services would be pruned from the service space.

For every pair of operations (op_{ik},op_{jl}), it also checks mode composability (line 10), operation semantics composability (line 11), and message composability (line 12). Each iteration of the `while` statement generates a composition plan. The *matched* set contains all component operations that have already been "plugged into" a composite service operation (lines 10 and 15). The use of this set prevents the generation of composition plans that can be inferred from previously generated plans. For example, assume that the following two plans, $plan_1 = \{(op_{i1},op_{j1}), (op_{i2},op_{j2})\}$ and $plan_2 = \{(op_{i1},op_{j3}), (op_{i2},op_{j4})\}$, have been generated. Since $plan_3 = \{(op_{i1},op_{j1}), (op_{i2},op_{j4})\}$ and $plan_4 = \{(op_{i1},op_{j3}), (op_{i2},op_{j2})\}$ can be inferred from $plan_1$ and $plan_2$, there is no need to generate them again. The statements in lines 26–28 check composition soundness based on QoC parameters. They use two functions, *complete()* and *relevant()*. Details about the selection process including QoC, completeness, and relevance are given in the next section.

The matchmaking algorithm uses the following functions to check composability: *purpose_compatible()*, *category_compatible()*, *quality_composable()*, *message_composable()*, and *sound()*. The functions *purpose_compatible()* and *category_compatible()* return *true* or *false* depending on whether a composite service operation has a purpose or category compatible with the purpose or category of a component service operation. *quality_composable()* returns *true* if a composite service operation is qualitatively composable with a component service operation. The two other functions are given in Table 3. The *message_compatible()* function returns *true* or *false* depending on whether a message M_i is message compatible with M_j. To allow a one-to-one mapping between M_i's and M_j's parameters, we use the *matched* set (line 10). This set contains M_j's parameters that have already been mapped to M_i's. The *sound()* function checks the soundness of the generated plan. Once a template has been computed for the generated plan (lines 2–5), it is compared with stored templates (lines 6–14). Note that, as stated in Sect. 3.5, composition plans are returned to composers even if they are not sound.

Example 4. To illustrate the matchmaking algorithm and without loss of generality, we give below the execution trace of the `receiveSpecialOffers` operation of the *car broker* composite service:

- **Step 1** (lines 1–5). The set variable *plan* contains the "plug-ins" for the currently generated plan. It is reinitialized to the empty set for each iteration of the `while` statement.
- **Step 2** (lines 6–9). Look for component services (e.g., *car dealer*) supporting SOAP protocol so that `receiveSpecialOffers`' purpose and category are compatible with the service purpose and category.
- **Step 3** (line 10). Determine operations of the *car dealer* service that are mode composable with `receiveSpecialOffers`. Since `receiveSpecialOffers` is a *one-way* operation, only the `specialOffers` operation is returned. The algorithm also checks that the two operations have not already been "plugged" together.
- **Step 4** (line 11). The `receiveSpecialOffers` and `specialOffers` operations have similar functions ("price-sales catalogue") and domains ("automobile dealer"). Thus they are operation semantics composable. We assume that both operations are qualitatively composable.
- **Step 5** (line 12). The operations are tested for message composability. The input of `receiveSpecialOffers` (i.e., `offer`) is compared with the output of `specialOffers`. Except for the parameter "color", which does not belong to `offer`, all of `specialOffers` output's parameters are mapped to `offer`'s. Hence the two operations are message composable.
- **Step 6** (lines 13–22). Since both operations are syntactically and semantically composable, a "plug-in" between the operations is inserted in *plan*. The information is also kept in the set *matched*. Steps 1 to 6 are iteratively performed for the remaining composite service operations.
- **Step 7** (lines 22–29). Once all operations of the *car broker* service have been "plugged", the algorithm checks whether the generated plan is sound. Other plans are then generated depending on the number of requested plans. ♢

4.3 Selection phase

At the end of the matchmaking phase, several composition plans may have been generated. To facilitate the selection of relevant plans, we define three *quality of composition* (*QoC*) parameters: *ranking*, *relevance*, and *completeness*. Other *QoC* parameters based on cost and time may also be defined. We present below definitions of *ranking*, *relevance*, and *completeness*:

- *Composition ranking*: The *ranking* of a composition gives an approximation of its "importance". For each plan, we determine its composition template CT. Assume that CT is a subgraph of a stored template ST_i. We use a function $\mathcal{R}$ ($\mathcal{R}$ for *reference*) defined on the set of stored templates; $\mathcal{R}(ST_i)$ gives the number of times that services with templates that are subgraphs of ST_i have been created. The *ranking* of CT with respect to ST_i is the proportion of references to ST. It is defined as follows (*n* is the number of stored templates):

$$Ranking(CT, ST_i) = \frac{\mathcal{R}(ST_i)}{\sum_{k=1}^{n} \mathcal{R}(ST_k)}$$

- *Composition relevance*: This parameter, denoted by *CR*, gives an approximation of a composition soundness. It compares edges of a composition template, CT, with the edges of a stored template ST_i. *CR*(CT,ST_i) is the ratio of CT's edges that occur in ST_i. It is defined as follows (E and E_i are the edges of CT and ST_i, respectively):

$$CR(CT, ST_i) = \frac{\mid E \cap E_i \mid}{\mid E \mid}$$

Table 2. Matchmaking algorithm

```
(01) Input: WS_i, repository, nb_requested_plans {
(02)  nb_generated_plans = 0
(03)  matched = ∅
(04)  while nb_generated_plans ≤ nb_requested_plans do
(05)   { plan = ∅
(06)    for each operation op_ik ∈ O_i do
(07)     { found = false
(08)      for each service WS_j from repository | category_compatible(C_ik,C_j)
(09)                                  and purpose_compatible(P_ik,P_j) and (B_i ∩ B_j ≠ ∅) do
(10)      { for each operation op_jl ∈ O_j | (mode_ik and mode_jl are dual) and (op_jl ∉ matched)
(11)         if purpose_compatible(P_ik,P_jl) and category_compatible(C_ik,C_jl) and quality_composable(op_ik,op_jl)
(12)            and message_composable(in_ik,out_jl) and message_composable(in_jl,out_ik)
(13)          then { found = true
(14)                 plan = plan ∪ {(op_ik,op_jl)}
(15)                 matched = matched ∪ {op_jl}
(16)                 break }
(17)       if found then break
(18)      } /* for in line (08) */
(19)      if ¬found
(20)         then { output("no matchmaking for",op_ik)
(21)                break }
(22)     } /* for in line (06) */
(23)    if ¬found then break
(24)              else if sound(plan)
(25)                   then output(plan,ST) /* ST is a Stored Template */
(26)                   else if relevant(plan,τ_relevance) and complete(plan,τ_completness)
(27)                      then output(plan,ST,τ_relevance,τ_completness) /* Test for QoC parameters */
(28)                      else output(plan,"not sound",τ_relevance,τ_completness)
(29)    nb_generated_plans = nb_generated_plans + 1
(30)   } /* while in line (04) */ }
```

Table 3. Message composability and soundness checking functions

```
(01) function message_composable(M_i,M_j):boolean {
(02)   matched = ∅
(03)   for each param p_ik ∈ P_i do
(04)    { found = false
(05)     for each param p_jl ∈ P_j | p_jl ∉ matched do
(06)      if (T(p_ik) = T(p_jl) or
(07)         T(p_jl) is derived from T(p_ik)) and
(08)         (U(p_ik) = U(p_jl)) and (R(p_ik) = R(p_jl))
(09)         then { found = true
(10)                matched = matched ∪ {p_jl}
(11)                break }
(12)     if ¬found then return false
(13)    } /* for in line (03) *
(14)  return true
(15) }
(16)
(17)
```

```
function sound(plan):boolean {
  for each element (op_ik,WS_j,op_jl) ∈ plan do
   if mode_ik ∈ {"notification","solicit-response"}
    then template = template ∪ ("CS",C_j)
    else template = template ∪ (C_j,"CS")
  for each stored template ST do
   for each pair (v_p,v_q) ∈ template do
    { found = false
      for each pair (v_r,v_s) ∈ ST do
       if (v_p,v_q) = (v_r,v_s)
          then { found = true
                 break }
     if ¬ found then break
    } /* for in line (07) */
  if found then return true
           else return false
}
```

– *Composition completeness*: This parameter, denoted by *CC*, gives the proportion of composite service operations that are composable with component service operations. *CC* allows the generation of plans whose composite services may not be "fully" composable with component services. The value of *CC* is set by service composers and depends on their level of expertise. Indeed, if the value *CC* is relatively low (e.g., 25%), the algorithm might return plans in which 75% of the composite service operations are not composable with component service operations. In this case, composers may need to change their specification (e.g., data types) so that the desired service can deal with other services' features. The following formula defines the *CC* parameter for a composite service WS_i:

$$CC(WS_i) = \frac{\mid Composable(O_i) \mid}{\mid O_i \mid}$$

Table 4. WSFL description generated for the car broker service

Flow Model	Global Model
<serviceProvider name="myCarDealer"> <locator type="static" service="carDealer.com"> </serviceProvider> ..... <activity name="Activity_1"> <performedBy serviceProvider="myCarDealer"> <implement> <export> <target portType="Port_1" operation="receiveSpecialOffers"/> </export> </implement> </activity> <activity name="Activity_2"> ..... <target portType="Port_1" operation="askForPayingHistory"/> ..... <activity name="Activity_3"> ..... <target portType="Port_1" operation="applyForFinancing"/> ..... <controlLink source="Activity_2" target="Activity_3">	<plugLink> <source serviceProvider="carBroker" portType="Port_1" operation="receiveSpecialOffers"/> <target serviceProvider="carDealer" portType="portCarDealer" operation="specialOffers"> </plugLink> <plugLink> <source serviceProvider="carBroker" portType="Port_1" operation="insuranceQuote"> <target serviceProvider="insurance" portType="portInsurance" operation="applyForInsurance"> </plugLink> <plugLink> <source serviceProvider="carBroker" portType="Port_1" operation="applyForFinancing"> <target serviceProvider="financing" portType="portFinancing" operation="financingQuote"> </plugLink>

where *Composable*(O_i) = $\{op_{ik} \in O_i \mid \exists\, WS_j\ \exists\, op_{jl} \in O_j$ so that op_{ik} is syntactically and semantically composable with $op_{jl}\}$.

Composition plans are sorted and returned according to their ranking. Plans with the highest ranking are returned first. This assumes that a ranking coefficient is maintained for each stored template. Composers define thresholds $\tau_{relevance}$ and $\tau_{completness}$ corresponding to relevance and completeness parameters, respectively. Plans are returned to composers if their relevance and completeness are greater than their respective thresholds. QoC parameters may be specified within CSSL specifications so that the "best" plans are automatically selected and returned to users.

4.4 Generation phase

The last phase in our approach aims at generating a detailed description of a composite service. This description includes the list of outsourced services, mappings between composite service and component service operations, mappings between messages and parameters, and flow of control and data between component services. Two important features of this phase are *customization* and *extensibility*. *Customization* refers to the ability to generate composite service descriptions in different languages such as WSFL (*Web Services Flow Language*) [18], XLANG [28], and BPEL4WS (*Business Process Execution Language for Web Services*) [2]. Composers specify the "target" language in their CSSL specification. *Extensibility* refers to the potential to include additional composition languages [10,34]. Indeed the structure of composition plans is abstract enough to be used at an intermediate level between CSSL specifications and most existing Web service composition languages. Below we illustrate the generation of a WSFL description from a composition plan.

WSFL defines two complementary descriptions for a composite service: *flow model* and *global model*. The *flow model* specifies the execution sequence between component services. The *global model* specifies how component services interact. Table 4 depicts parts of the flow and global models generated for the *car broker* service. The *flow model* contains a set of *activities*. Each activity represents a single step of the overall business goal. Activities are bound to services through the *locator* element. *Static* binding means that the service is directly specified in the locator. The information assigned to the *name* (i.e., *myCarDealer*) and *service* (*carDealer.com*) attributes is obtained during the matchmaking phase (Table 2, line 08). We use the *UDDI inquiry interface* (i.e., *find()* operation) to retrieve such information from the *businessEntity* of each Web service stored in the UDDI registry. Each activity in the flow model is implemented by an operation specified in the *implement* element. The nested *export* element means that this operation is outsourced. For example, the operation `receiveSpecialOffers` (corresponding to *Activity_1*) is outsourced from the *myCarDealer* service. The name of this operation is provided by the composer in the CSSL specification. Activities are connected together through *control links*, which specify the order in which activities are executed. Each *control link* is generated from a *flow* element provided by composers in their CSSL specifications. For example, *Activity_2* (corresponding to the `askForPayingHistory`

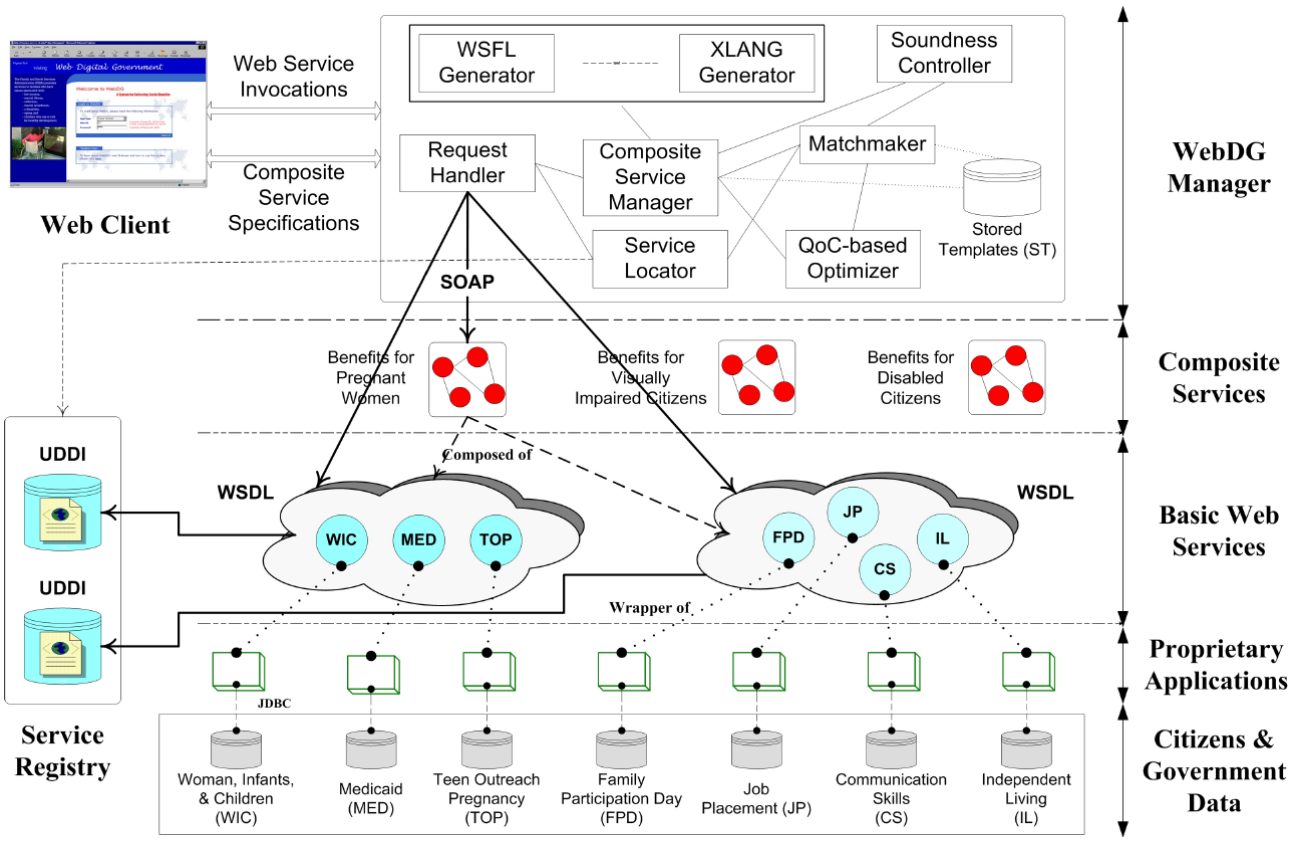

Fig. 6. WebDG architecture

operation) is executed before *Activity_3* (corresponding to `applyForFinancing`).

The *global model* includes a set of *plug link* elements. A *plug link* connects each operation of the composite service to an operation of a component service. It indicates that an interaction has to take place between these two operations in order to completely implement an activity. Each *plug link* element corresponds to a mapping in a composition plan (Table 2, line 14). For example, the operation `insuranceQuote` is mapped to the operation `applyForInsurance` offered by the *insurance* service. The *port type* and operation names of the outsourced services are obtained from the WSDL description of those services.

5 Case study: E-government Web services

A typical and emerging area that involves access to Web services is *E-government*. As an application domain of our research in Web services, we partnered with the *Family and Social Services Administration* (FSSA) [26]. The FSSA serves families facing issues associated with low income, mental illness, addiction, mental retardation, disability, aging, and children at risk for healthy development. The aim is to help the needy citizens in collecting benefits to which they are entitled. However, the current process within FSSA is time-consuming and frustrating to both citizens and case officers. To facilitate the use of FSSA welfare applications and hence expeditiously satisfy citizens' needs, we organize these applications into Web services. The implementation of our approach is showcased using the FSSA case study. The resulting system, called *WebDG*, provides customized services to indigent citizens. In this section, we first describe WebDG implementation. We then illustrate our approach by using a scenario from social and welfare services.

5.1 WebDG implementation

The WebDG system is implemented across a network of Solaris workstations. Citizens and case officers access WebDG via a graphical user interface (GUI) implemented using HTML/Servlet (Fig. 6). WebDG currently includes seven (7) FSSA applications implemented in Java (JDK 1.3). These applications are wrapped by WSDL descriptions. Examples of services implemented in WebDG include `WIC` (a federally funded food program for women, infant, and children), `Medicaid` (a healthcare program for low income citizens), and `Teen Outreach Pregnancy` (a program that offers childbirth and postpartum educational support to pregnant teens). Each service accesses a database (Oracle or Informix) in the backend to retrieve and/or update citizens and government information.

We use the *Axis Java2WSDL* utility in IBM's Web Services Toolkit to automatically generate WSDL descriptions from Java class files. WSDL service descriptions are published into a UDDI registry. We adopt Systinet's *WASP UDDI Standard 3.1* as our UDDI toolkit. A Cloudscape (4.0) database is used as a UDDI registry. WebDG services are deployed using *Apache SOAP* (2.2). Apache SOAP provides not only

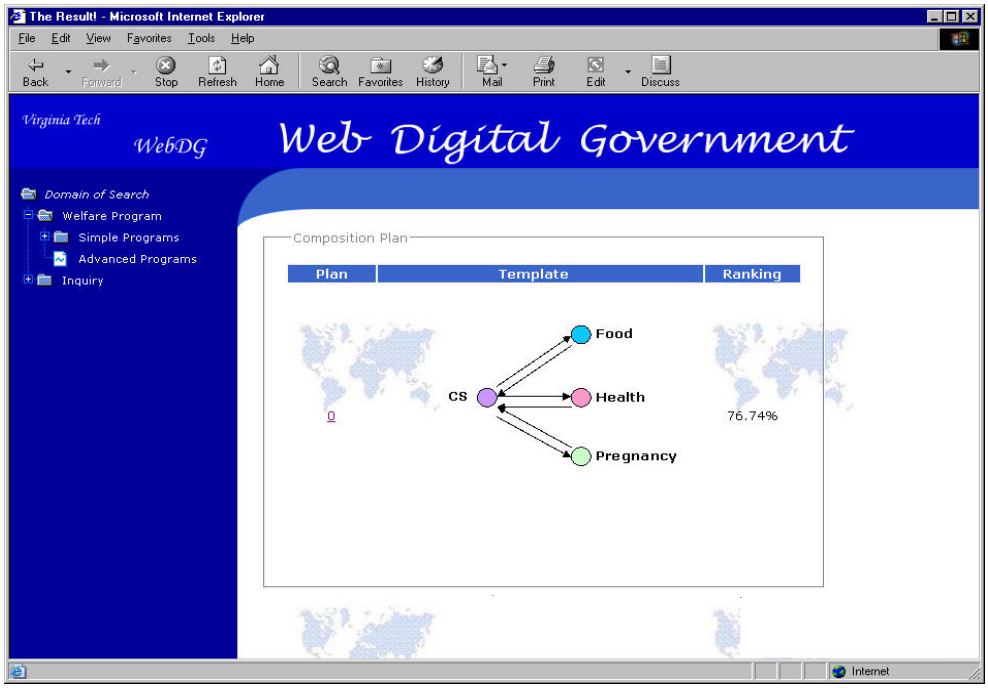

Fig. 7. Stored template corresponding to the pregnancy benefits service

server-side infrastructure for deploying and managing service but also client-side API for invoking those services. Each service has a *deployment descriptor*. The descriptor includes the unique identifier of the Java class to be invoked, session scope of the class, and operations in the class available for the clients. Each service is deployed using the *service management client* by providing its descriptor and the URL of the *Apache SOAP servlet rpcrouter*.

The *WebDG manager* is at the core of the WebDG system. The *Service Locator* (SL) allows the discovery of WSDL descriptions by accessing the UDDI registry. The SL implements *UDDI Inquiry Client* using WASP UDDI API. Once a service is discovered, its operations are invoked through *SOAP Binding Stub*, which is implemented using Apache SOAP API. CSSL specifications are handled by the *Composite Service Manager* (CSM). The CSM uses JAXP (*Java API for XML Processing*) to parse those specifications and returns them to the *matchmaker*. The matchmaker sends the category of each composite service operation to the SL. Only services with a category compatible with the operation's category are retrieved. The SL parses the WSDL description of each located service and returns it back to the matchmaker. After checking composability, the matchmaker generates composition plans and sends them to the *QoC-based Optimizer*. The optimizer selects plans based on QoC parameters. The matchmaker forwards the selected plans to the *Soundness Controller* (SC).

The SC checks the soundness of each generated plan by accessing a stored templates (ST) repository (Oracle database). Stored templates are kept in a relational table containing four attributes: *template number* (unique), *source category*, *target category*, and *ranking*. The SC returns composition plans with their compatible stored templates (if any) to the CSM. The CSM finally forwards the results to the composer, who selects the appropriate plan (e.g., plan with the highest ranking). If the selected plan is not sound, the CSM module determines its template and stores it in the ST repository. This new template can be used to test the soundness of future composition plans. Once a composition plan is selected, the CSM module forwards it to the appropriate *generator*. For example, if the composer has specified WSFL as a "target" language, the composition plan is sent to the *WSFL generator*. In this case, a WSFL description is generated as described in Sect. 4.4 and returned to the composer via the CSM module.

5.2 Scenario: collecting social and welfare benefits

To illustrate the main features of WebDG, we present the following scenario. Let us consider the case of a pregnant teen Mary visiting case officer John to collect social benefits to which she is entitled. Mary would like to apply for a government-funded health insurance program. She also needs to consult a nutritionist to maintain an appropriate diet during her pregnancy. As Mary will not be able to take care of the future newborn, she is interested in finding a foster family. The fulfillment of Mary's needs requires accessing different services scattered in and outside the local agency. It would be more efficient if all Mary's needs are addressed together and specified only once. John would, as result, seamlessly access all related services through one single access point. He would specify Mary's needs through one single composite service called *Pregnancy Benefits* (PB).

Case Officer John would select the "Advanced Programs" node (Fig. 7) to specify PB composite service. He would give the list of operations to be outsourced by PB. Examples of such operations include `Find_Available_Nutritionist`, `Find_PCP_Providers` (which looks for primary care providers), and `Find_Pregnancy_Mentors`. After checking composability rules, WebDG would return *composition plans* that conform to PB specifications. Each plan has an ID

Table 5. Simulation settings

Variable	Range
Service interfaces	3000–30000
Composite services	100–1000
Operations per service	10–50
Parameters per message	50–100
Requested plans	50–100
Stored templates	100–500
Vertices per stored template	10–20

(number), a graphical description, and a ranking. The ranking gives an approximation about the relevance of the corresponding plan (Fig. 7). John would click on the plan's ID to display the list of outsourced services. In our scenario, WIC, Medicaid, and TOP services would be outsourced by PB.

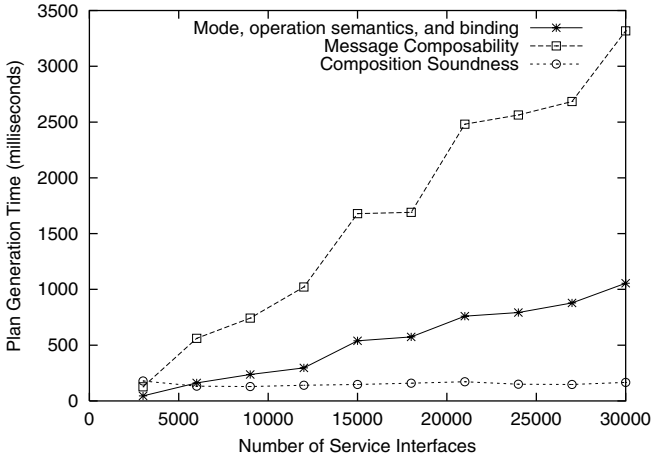

Fig. 8. Plan generation time

6 Performance evaluation

The purpose of our experiment is to assess the scalability of our approach, i.e., the possibility of generating plans for a large number of service interfaces. We mainly focus on the matchmaking phase since it is the one that may require access to a large number of Web services to check composability rules. The aim is to evaluate the effectiveness and speed of the matchmaking algorithm. We also assess the role of the selection phase (QoC parameters) in reducing the number of generated plans. We ran our experiments on a Sun Enterprise Ultra 10 server with a 440-MHz UltraSPARC-IIi processor, 1-GB of RAM, and under a Solaris operating system. Although the algorithm is implemented in a WebDG prototype, we built a simulation testbed to run the experiments. This allows the generation of a large number of service interfaces that would be difficult to achieve in the current WebDG version. The testbed generates XML documents that store CSSL composite service specifications and WSDL service interfaces. CSSL and WSDL documents are manipulated using JDOM, a Java-based document object model for XML documents. JDOM provides means to represent XML documents for easy manipulation.

The testbed allows the generation of WSDL descriptions at an arrival rate that follows a statistical model (e.g., *Poisson*, *uniform*) specified by the user. The arrival rate represents the duration (in seconds) between the generation of two consecutive services. This parameter is particularly important for simulating the dynamics of Web service environments. The testbed also allows users to give the range and statistical models for several quantitative attributes such as the number of operations per service, messages, and parts per message. In our experiments, we randomly generate (using uniform distribution) service interfaces, composite services, and stored templates. We varied different parameters including the number of service interfaces, composite services, operation per services, parameters per message, requested plans, stored templates, and nodes per templates (Table 5). We first set the number of services (from 3,000 to 30,000 with an iteration range of 3,000). For each service, we then generate a category (1 out of 50), binding, and number of operations. We also generate the mode, purpose (1 out of 100), category, and quality of each operation. Finally, for each input and output message, we randomly generate the number of parameters, data type, unit, and business role of each parameter (1 out of 37 built-in data types).

We first evaluate the time for generating composition plans (Fig. 8). We consider three execution times. The first execution time includes mode, binding, and operation semantics composability. The second execution time corresponds to message composability. The last execution time corresponds to composition soundness. The results show that most of the time is spent on checking message composability (Fig. 8). Indeed, the second time requires comparing the parameters of each composite service operation with the parameters of each operation of a component service. In contrast, the first time includes comparing operation modes, categories, purposes, and binding protocols that are less CPU-intensive. Composition soundness is the property that consumes the least generation time. Indeed, syntactic and operation semantics composability compare composite services with all service interfaces in the business registry. In contrast, composition soundness compares generated plans with stored templates whose number is much smaller than the number of service interfaces (100–500 templates vs. 3,000–30,000 interfaces). This also explains the relative stability of the composition soundness time. Note that the plan generation time shown in Fig. 8 does not consider access time to UDDI business registry and stored templates repository.

We also assess the impact of QoC parameters on the number of generated plans (Fig. 9). We particularly consider the composition completeness (CC) ratio. We conducted experiments for CC = 33% and CC = 66%. The results show that the number of generated plans is higher for CC = 33% (Fig. 9). Indeed, for CC = 33%, plans are generated if at least 33% of composer operations are composable with component operations. However, for CC = 66%, plans are generated if at least 66% of composer operations are composable with component operations. The results also show that the number of generated plans for CC = 33% is, on average, greater than 50 (i.e., minimum number of requested plans). This means that plans are generated for almost every specified composite service. However, for CC = 66%, the number of generated plans is at most equal to 30 (i.e., less than the minimum number of requested plans). This means that for some composite services, no plan

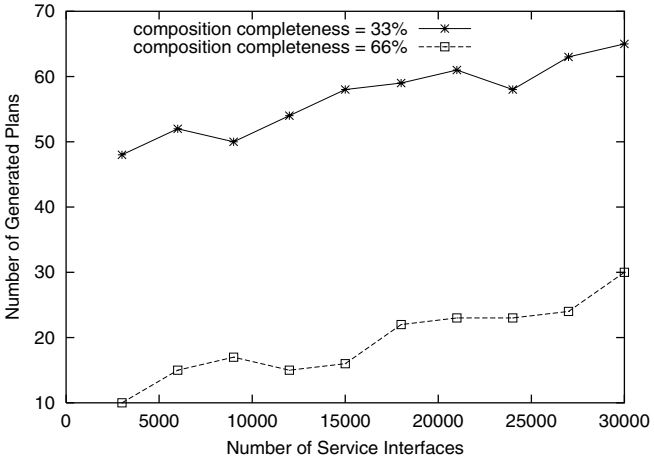

Fig. 9. Number of generated plans

has been generated. This confirms our expectation about the impact of CC on the number of generated plans; a relatively low value of CC generates more plans, each plan containing a small number of composable operations. In contrast, a high value of this ratio generates a smaller number of plans, each plan having more composable operations.

7 Related work

Web services are slated to be a very active research area. We overview major techniques, standards, and platforms for Web services that are most closely related to our research.

7.1 Automatic service composition

Automatic service composition has been the focus of several recent projects. *DAML-S* (the *DARPA Agent Markup Language*) [23] defines a semantic markup for Web services based on the use of ontologies. DAML-S introduces the notions of prerequisites (called Preconditions) and consequences (called Effects) of Web services to cater to automatic composition. It is not clear, however, how composite services are generated using DAML-S specifications. DAML-S does not define the notion of service composability. Additionally, it does not consider semantic properties such as purpose, parameter unit, and business role. An architecture for service composition in pervasive computing environments is presented in [12]. Service descriptions are provided in DAML-S. They also include platform specific information such as processor type, speed, and memory availability. The composition manager uses a semantic service discovery mechanism to select component services. This mechanism is based on *DReggie*, a Jini-based semantic discovery framework [11]. The matching mechanism focuses mostly on comparing service attributes. In contrast, the matchmaking algorithm proposed in this paper is based on a set of composability rules that compare the structure of messages, their business function, the semantics and data types of their parameters, qualitative properties, and the soundness of composite services.

WSMF (Web Service Modeling Framework) combines the concepts of Web services and ontologies to cater to Semantic Web-enabled services [7]. WSMF is still in its early stage. The techniques for the semantic description and composition of Web services are still ongoing. Furthermore, WSMF does not address the issue of service composability. An approach to ontology-based composition of Web services is proposed in [8]. It uses DAML-S for describing Web services. Semantic matching of Web service capabilities is limited to comparing QoS (quality of service) dimensions and input/output parameters of Web services. In our approach, we compare additional features such as category, purpose, parameters' data types, units, and business roles. Additionally, [8] focuses mostly on the automatic selection of Web services. Only a few details are given about the way composite services are generated. *Ninja* [14] introduces a technique called automatic path creation (APC) to cater to automatic service composition. When an APC receives requests for composite service execution, the APC creates a path that includes a sequence of operators that perform computation on data and connectors that provide data transport between operators. Ninja focuses mostly on fault tolerance by replicating services on multiple workstations. It uses a limited operator functional classification (four categories) to automate the selection of operators. It is also mainly based on input-output matching of services. *SWORD* [32] uses a rule-base expert system to automatically determine whether a desired composite service can be achieved using existing services. SWORD does not seem to focus on service composability and semantic description of Web services.

Other techniques for composing Web services include *WISE* [21], *eFlow* [10], and *CMI* [34]. These techniques generally assume that composers are responsible of checking service composability. WISE [21] defines the notion of *virtual business process* to help users compose services. Composite service specifications, however, require dealing with such technical details as interservice communication and event management. eFlow [10] uses the notion of *process template* to model composite services. Composers need to browse the *process library* to search for process templates of interest. Furthermore, they need to handle interactions between component services when defining composite services. CMI [34] introduces the notion of *polymorphic process model* for describing collaborations among activities. This model requires handling details about activities and their interactions (e.g., defining activity state machines, interfaces, and implementations).

7.2 Service matching and composability

Techniques have recently been proposed to deal with service matching and composability. Paolucci et al. [30] proposes a solution-based on DAML-S for semantic matching between service advertisements and capabilities. The matching algorithm defined in [30] is limited to comparing inputs and outputs of the advertisement with inputs and outputs of the request. *LARKS* defines five techniques for service matchmaking: context matching, profile comparison, similarity matching, signature matching, and constraint matching [37]. Those techniques mostly compare service text descriptions, signatures (inputs and outputs), and logical constraints about inputs and outputs. The *ATLAS* matchmaker defines two methods for comparing service capabilities described in DAML-S [31]. The first method compares functional attributes to check

whether advertisements support the required type of service or deliver sufficient quality of service. The second compares the functional capabilities of Web services in terms of inputs and outputs. No evaluation study is presented to determine the effectiveness and speed of the ATLAS matchmaker. Li and Horrocks [22] describe the design of a service matchmaker that uses DAML-S-based ontology. It uses techniques from knowledge representation to match service capabilities. In particular, it defines a description logic (DL) reasoner; advertisements and requests are represented in DL notations. Baina et al. [1] present a composability property that compares service categories. However, features such as operation purposes, modes, message data types, and soundness are not considered. No algorithm is proposed to check composability or to automatically generate composite services. In [24], an algorithm for checking Composability is presented. Unlike our approach, this algorithm is limited to checking syntactic features (input and output events of component services). In addition, it only checks composability on an a posteriori basis to replace a component service by another. Composers are still responsible for providing detailed descriptions of their services. Heuvel et al. [15] define composability properties to compare service categories and messages. This method not seem to consider properties such as mode, purpose, binding protocol, and composition soundness. More importantly, it does not seem to provide algorithms for the automatic generation of composite services.

7.3 Standards and commercial platforms

Standardization efforts are ongoing to enable service composition [39]. These include XLANG [28], WSFL [18], BPEL4WS [2], and XL [13]. XLANG [28] and WSFL [18] extend WSDL language to provide constructs for combining Web services to build multiparty business processes. BPEL4WS (*Business Process Execution Language for Web Services*) [2] combines the features of both WSFL (support for graph oriented processes) and XLANG (structural constructs for processes) for composing Web services. However, XLANG, WSFL, and BPEL4WS assume that service composers are responsible for checking service and operation compatibility. Semantic properties of Web services are not considered. XL (XML Language) [13] aims at providing an XML language for service specification and composition. It uses a mix of imperative, parallel, and workflow constructs. XL is still in its initial design stage. In its current form, it does not explicitly define primitives for composition.

Commercial platforms are increasingly targeting Web services [40]. Microsoft's .NET [27] enables service composition through *Biztalk Orchestration* tools that use XLANG. .NET does not check service composability. IBM's WebSphere [19] supports key Web service standards. To the best of our knowledge, it provides little or no support for service composition. HP's Netaction Internet Operating Environment (IOE) [17] is an integrated platform for building Web services. Like .NET, Netaction does not support service composability. HP announced in July 2002 it was discontinuing its development and support of NetAction. The WebMethods Enterprise Server [47] defines *Flow*, a process-oriented language used to visually compose services. This language is very simple and is fairly limited to a small number of services. IONA's Orbix E2A [20] includes the *Orbix E2A Web Services Integration Platform*. It provides a set of tools for business integration using Web service standards. Developers create Web services from existing applications including EJBs and CORBA objects. It is unclear how Web services would be composed. Sun ONE (*Sun Open Net Environment*) [36] is a platform for Web services developed by Sun. Sun began its Web services efforts only recently, and few details have so far emerged.

8 Conclusion

In this paper, we propose a rigorous framework for composing Web services. We present an algorithm to automatically generate composite services from high-level specifications of the desired composition. We define a model for checking service composability. This model provides a set of composability rules that compare syntactic and semantic features of Web services. We provide an implementation of the proposed approach in the WebDG prototype. Finally, we conducted experiments to illustrate the scalability of our approach. Future work includes extending our composability model to include additional semantic features such as temporal and spatial availability of Web services and operation preconditions and effects. Other extensions would be to consider XML Schema's user-defined data types and define data type compatibility among message parameters in terms of XML Schema inference. Finally, we are investigating the definition of an "optimization" model for composite services based on our quality of composition (QoC) parameters.

Acknowledgements. The authors would like to thank the anonymous reviewers for their valuable comments on earlier drafts of this paper.

References

1. Baina K, Benali K, Godart C (2001) A process service model for dynamic enterprise process interconnection. In: Proceedings of the CoopIS conference, Trento, Italy, September 2001, pp 239–254
2. BEA, IBM, Microsoft (2003) Business Process Execution Language for Web Services (BPEL4WS). http://xml.coverpages.org/bpel4ws.html
3. Benatallah B, Dumas M, Shen M, Ngu AHH (2002) Declarative composition and peer-to-peer provisioning of dynamic Web services. In: Proceedings of the ICDE conference, San Jose, CA, February 2002, pp 297–308
4. Benatallah B, Medjahed B, Bouguettaya A, Elmagarmid A, Beard J (2000) Composing and maintaining Web-based virtual enterprises. In: Proceedings of the 1st VLDB TES workshop, Cairo, Egypt, September 2000, pp 155–174
5. Berners-Lee T (2001) Services and semantics: Web architecture. http://www.w3.org/2001/04/30-tbl
6. Berners-Lee T, Hendler J, Lassila O (2001) The Semantic Web. Sci Am 7–15
7. Bussler C, Fensel D, Maedche A (2002) A conceptual architecture for Semantic Web enabled Web services. SIGMOD Rec 31(4):24–29
8. Cardoso J, Sheth A (2002) Semantic e-workflow composition. Technical report, LSDIS Lab, Computer Science, University of Georgia

9. Casati F, Georgakopoulos D, Shan MC (eds) In: Proceedings of the 2nd VLDB TES workshop, Rome, September 2001. Lecture notes in computer science, vol 2193, Springer, Berlin Heidelberg New York
10. Casati F, Ilnicki S, Jin L, Krishnamoorthy V, Shan MC (2000) Adaptive and dynamic service composition in eFlow. In: Proceedings of the CAiSE conference, Stockholm, June 2000, pp 13–31
11. Chakraborty D, Perich F, Avancha S, Joshi A (2001) DReggie: a smart service discovery technique for e-commerce applications. In: Proceedings of the workshop at the 20th symposium on reliable distributed systems, New Orleans, October 2001
12. Chakraborty D, Perich F, Joshi A, Finin T, Yesha Y (2002) A reactive service composition architecture for pervasive computing environments. In: Proceedings of the 7th personal wireless communications conference, Singapore, October 2002, pp 53–62
13. Florescu D, Grünhagen A, Kossmann D (2002) XL: an XML Programming Language for Web service specification and composition. In: Proceedings of the WWW 2002 conference, Honolulu, May 2002, pp 65–76
14. Gribble SD, Brewer EA, Hellerstein JM, Culler D (2000) Scalable, distributed data structures for Internet service construction. In: Proceedings of the 4th symposium on operating systems design and implementation, San Diego, CA, October 2000, pp 319–332
15. Heuvel JVD, Yang J, Papazoglou MP (2001) Service representation, discovery and composition for E-marketplaces. In: Proceedings of the CoopIS conference, Trento, Italy, September 2001, pp 270–284
16. Horrocks I (2002) DAML+OIL: a description logic for the Semantic Web. IEEE Data Eng Bull 25(1):4–9
17. HP (2003) NetAction. http://www.hp.com
18. IBM (2003) Web Services Flow Language (WSFL). http://xml.coverpages.org/wsfl.html
19. IBM (2003) WebSphere. http://www-3.ibm.com/software/info1/websphere
20. IONA (2003) Orbix E2A. http://www.iona.com
21. Lazcano A, Alonso G, Schuldt H, Schuler C (2000) The WISE approach to electronic commerce. Int J Comput Sys Sci Eng 15(5):343–355
22. Li L, Horrocks I (2003) A software framework for matchmaking based on Semantic Web technology. In: Proceedings of the WWW 2003 conference, Budapest, May 2003, pp 331–339
23. McIlraith SA, Son TC, Zeng H (2001) Semantic Web services. IEEE Intell Sys 16(2):46–53
24. Mecella M, Pernici B, Craca P (2001) Compatibility of e-services in a cooperative multi-platform environment. In: Proceedings of the 2nd VLDB TES workshop, Rome, September 2001, pp 44–57
25. Medjahed B, Benatallah B, Bouguettaya A, Ngu A, Elmagarmid A (2003) Business-to-business interactions: issues and enabling technologies. VLDB J 12(1):59–85
26. Medjahed B, Rezgui A, Bouguettaya A, Ouzzani M (2003) Infrastructure for e-government Web services. IEEE Internet Comput 7(1):58–65
27. Microsoft (2002) .NET. http://www.microsoft.com/net
28. Microsoft (2003) Web Services for Business Process Design (XLANG). http://xml.coverpages.org/xlang.html
29. Muth P, Wodtke D, Weissenfels J, Dittrich AK, Weikum G (1998) From centralized workflow specification to distributed workflow execution. J Intell Inform Sys 10(2):159–184
30. Paolucci M, Kawamura T, Payne TR, Sycara K (2002) Semantic matching of Web services capabilities. In: Proceedings of the 1st international Semantic Web conference, Sardinia, Italy, June 2002, pp 318–332
31. Payne TR, Paolucci M, Sycara K (2001) Advertising and matching DAML-S service descriptions. In: Proceedings of the international Semantic Web working symposium, Stanford, CA, July 2001
32. Ponnekanti SR, Fox A (2002) SWORD: A developer toolkit for Web service composition. In: Proceedings of the WWW 2002 conference, Honolulu, May 2002
33. Rezgui A, Ouzzani M, Bouguettaya A, Medjahed B (2002) Preserving privacy in Web services. In: Proceedings of the the 4th international ACM workshop on Web information and data management, McLean, VA, November 2002, pp 56–62
34. Schuster H, Georgakopoulos D, Cichocki A, Baker D (2000) Modeling and composing service-based and reference process-based multi-enterprise processes. In: Proceedings of the CAiSE conference, Stockholm, June 2000, pp 247–263
35. Sheth A, Miller J (2003) Web services: technical evolution yet practical revolution. IEEE Intell Sys 18(1):78–80
36. Sun (2003) Sun ONE. http://wwws.sun.com
37. Sycara K, Klush M, Widoff S (1999) Dynamic service matchmaking among agents in open information environments. ACM SIGMOD Rec 28(1):47–53
38. Tsur S, Abiteboul S, Agrawal R, Dayal U, Klein J, Weikum G (2001) Are Web services the next revolution in e-commerce? (Panel). In: Proceedings of the VLDB conference, Rome, September 2001, pp 614–617
39. Van der Aalst W Don't go with the flow: Web services composition standards exposed. IEEE Intell Sys 18(1):72–76
40. Vaughan-Nichols SJ (2002) Web services: beyond the hype. IEEE Comput 35(2):18–21
41. W3C (2001) XML Schema. http://www.w3.org/XML/Schema
42. W3C (2003) OWL Web Ontology Language overview. http://www.w3.org/TR/owl-features
43. W3C (2003) Semantic Web. http://www.w3.org/2001/sw
44. W3C (2003) Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap
45. W3C (2003) Universal description, discovery, and integration (UDDI). http://www.uddi.org
46. W3C (2003) Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl
47. WebMethods (2003) http://www.webmethods.com
48. Weikum G (ed) (2002) Special issue on organizing and discovering the Semantic Web. IEEE Data Eng Bull 25(1): 1–58