SOCA (2010) 4:63–79
DOI 10.1007/s11761-010-0055-6

ORIGINAL RESEARCH PAPER

A two-phase framework for quality-aware Web service selection

Qi Yu · Manjeet Rege · Athman Bouguettaya · Brahim Medjahed · Mourad Ouzzani

Received: 20 April 2009 / Revised: 28 December 2009 / Accepted: 21 January 2010 / Published online: 5 March 2010

Abstract Service-oriented computing is gaining momentum as the next technological tool to leverage the huge investments in Web application development. The expected large number of Web services poses a set of new challenges for efficiently accessing these services. We propose an integrated service query framework that facilitates users in accessing their desired services. The framework incorporates a service query model and a two-phase optimization strategy. The query model defines service communities that are used to organize the large and heterogeneous service space. The service communities allow users to use declarative queries to retrieve their desired services without worrying about the underlying technical details. The two-phase optimization strategy automatically generates feasible service execution plans and selects the plan with the best user-desired quality. In particular, we present an evolutionary algorithm that is able to "co-evolve" multiple feasible execution plans simultaneously and allows them to compete with each other to generate the best plan. We conduct a set of experiments to assess the performance of the proposed algorithms.

Q. Yu (✉) · M. Rege
Rochester Institute of Technology, Rochester, NY, USA
e-mail: qi.yu@rit.edu

M. Rege
e-mail: mr@cs.rit.edu

A. Bouguettaya
CSIRO ICT Center, Canberra, ACT, Australia
e-mail: athman.bouguettaya@csiro.au

B. Medjahed
University of Michigan, Dearborn, MI, USA
e-mail: brahim@umich.edu

M. Ouzzani
Purdue University, West Lafayette, IN, USA
e-mail: mourad@cs.purdue.edu

Keywords Web service · Service selection · Quality of Web service

1 Introduction

The Web is a distributed, dynamic, and large information repository. It has now evolved to encompass various information resources accessible worldwide. However, a major problem of the Web is that it contains mostly human consumable contents. The *Semantic Web* is gaining momentum as a non-trivial extension to the current Web. It would provide well-defined machine interpretable Web content [3,16]. It could help computers understand the actual meaning of the information on the Web.

Enabling technologies for the Semantic Web are experiencing a fast growth. *Web services* have become *de facto* the most significant technological by-product. A Web service is a set of related functionalities that can be programmatically accessed through the Web [25]. More precisely, a Web service can be captured by two key properties: *functionality* and *quality*. Functionality is about what the service can offer, and it is typically represented by a set of operations provided by the service. Each service operation can be described by using input, output, precondition, and effect. Quality is about how well a service provider delivers the service, and it is usually captured by a set of quality parameters, such as response time, price, reputation. Semantic web technologies, such as ontologies, can leveraged to describe these two properties of Web services. Examples of Web services include online reservation, ticket purchase, stock trading, and auction. Standards are key enablers of Web services [26] and are led by major industry players. This has greatly facilitated the adoption and deployment of Web services [12]. Three key standards have been defined: SOAP [28], WSDL [30], and

UDDI [29]. SOAP defines a communication protocol for Web services. WSDL enables XML description of Web services. UDDI offers a service registry to advertise and discover Web services.

The Web is poised to take a new giant step to be a central repository for the ever increasing number of Web services [23]. This will have the effect of transforming the Web from a *data-oriented* repository to a *service-oriented* repository. In this new framework, existing business logic would be wrapped as Web services to be accessible on the Web via a Web services middleware [27]. Web services would work as a self-contained entity to fulfill users' requests. Interoperation among multiple Web services would additionally improve the *quality* of answers by providing *value-added* services.

The ability to *efficiently* access Web services is necessary, in light of the large and widely geographically disparate space of services. Using Web services generally consists of invoking operations by sending and receiving messages. However, for complex applications accessing diverse Web services (e.g., a travel package), there is a need for an *integrated* and *efficient* approach to manipulate and access Web services' functionalities. This should also be performed in a *user-transparent* manner. In addition, as the number of Web services is expected to substantially increase, this would have the effect of introducing competition among Web services that offer "similar" functionalities. Another major challenge is devising optimization strategies for finding the "*best*" Web services and/or their combinations with respect to the user expected *quality*, such as price, response time, and reputation.

Existing Web service technologies only provide partial solutions for the above issues. For example, service composition enables the combination of multiple simple services to generate a value-added service package [2,11,18–20]. Composability rules and automatic service composition techniques have been intensively investigated in [18,19]. However, users are required to specify the composite services in advance by using a high level Composite Service Specification Language (CSSL). Some service quality aspects are defined in the composability model in [18]. However, the composition process mainly focuses on the "functional" correctness of the composition instead of the "non-functional" properties. The Web Service Level Agreement (WSLA) language defines the obligations of service providers on the service quality when delivering their corresponding services [13]. It mainly focuses on specifying the measures to be take when there is deviation or failure to the asserted quality guarantees instead of how to leverage the quality information to select the best services. Some quality-aware service optimization techniques have been investigated [6,32,33]. However, most of these approaches assume that a feasible composition plan is already available and the optimization is to select providers that result in a plan with the best quality. In addition, none of these approaches address how to optimize multiple feasible composition plans simultaneously.

We propose an integrated *query framework* that offers comprehensive query and optimization facilities over Web services. The proposed framework provides a *complete and integrated* solution for the above issues. A first step in enabling such a comprehensive service query framework is defining a query model that facilitates the formulation and submission of queries and their transformation into actual invocations of Web service operations. We propose a three-level query model, including *query level, community level*, and *service level*. It defines a set of mapping rules and uses service communities to locate useful services. Users formulate declarative queries at the query level. Queries are then processed throughout the three levels resulting in a *Service Execution Plan (SEP)* where Web service operations are *composed* and *invoked* in a transparent manner. A two-phase optimization strategy is integrated into the query framework and works interactively with the query model to optimize the service execution plan. The first phase is to generate a set of feasible *conceptual choreographies*. A conceptual choreography includes a list of abstract operations defined at the community level and their invocation order. The second phase instantiates the conceptual choreographies using the concrete service operations to generate SEPs. The SEPs with the best *Quality of Web Service* (*QoWS*) will be selected. The concept of *QoWS* is considered as a key feature in distinguishing between competing Web services [27]. *QoWS* encompasses different quality parameters that characterize the behavior of a Web service in delivering its functionalities. We present an evolutionary algorithm coupled with a co-evoluation strategy that optimizes multiple feasible choreographies simultaneously. We also implement a greedy algorithm for comparison purpose.

The remainder of this paper is organized as follows. In Sect. 2, we describe a scenario to motivate the proposed service query framework. In Sect. 3, we propose a three-level Web service query model. In Sect. 4, we present our optimization strategies. In Sect. 5, we describe the analytical model. In Sect. 6, we present the simulation experiments and interpretation of the results. We overview the related work in Sect. 7 and conclude in Sect. 8.

2 Scenario: Travel planning

As a way to motivate this work, assume a university professor, say John, planning to attend an international conference in Paris. This professor will naturally want to devise a strategy whereby the time and cost of the travel will be optimal. Assume that John has access to a Web service infrastructure where the different entities playing a role in John's travel plan are represented by Web services. Typical Web services

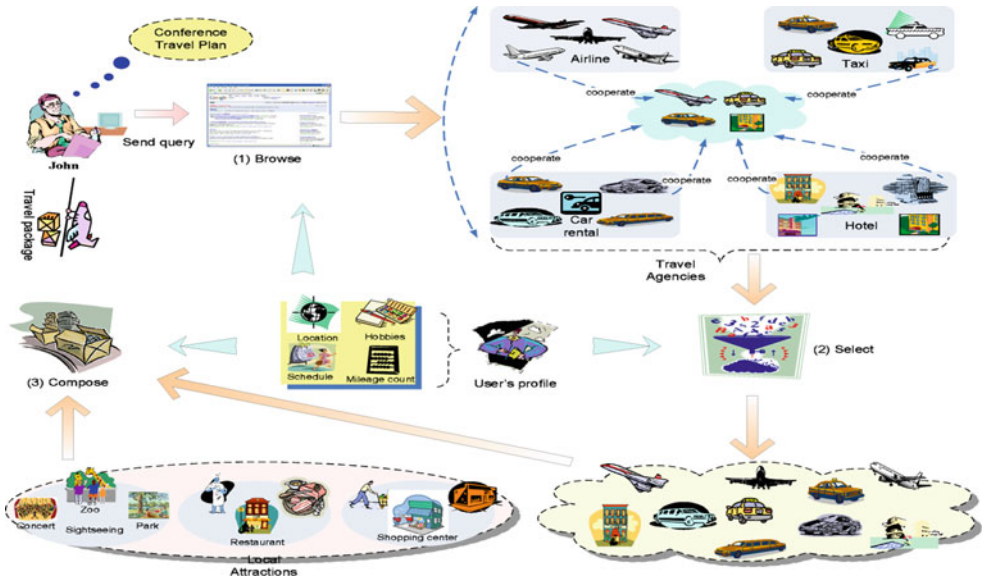


Fig. 1 The travel planning scenario

that need to be accessed will include airlines, taxi services, hotels, car rental, map, weather, etc. In order to get the travel package, the user may also need to access some functionalities, like payment, that is common to all the services. Since Web services are stand-alone, each service may offer its own payment procedure (in terms of service operations). In this case, users need to follow service-specific rules for the payment. A more convenient way is to use a common service that can handle the payment from all services so that s/he just needs to login and provide credit card information once. It is worth to note that some of these common functionalities are already available as Web services. For example, hundreds of online stores are using google checkout[1] as one of their payment services.

Due to the expected large number of available services, we anticipate that there will be many competitors to provide each of these services mentioned above to this professor. In addition, it is important that the users' preferences could be reflected as another criterion for service selection. Any query optimization strategy would need to be aware of these requirements to respond to the user with maximized convenience and minimized cost. Using this scenario as an illustration, we divide the decision-making process for the purpose of selecting an optimal travel plan into three steps (see Fig. 1):

2.1 Step 1: Browsing services

John starts his travel planning process by browsing the services required by his travel package. He may use currently available Web registries or search engines to search these services, including airlines, taxi companies, hotels, and car rental companies (Step 1 in Fig. 1). First, John wants to get the information about airlines that provide scheduled flight to Paris. He types keywords "airlines, Paris" and submits the request. The search engines may return hundreds of options where some of them might only be trivial advertisement, or even dead links that are not reachable. John needs to manually filter these answers and look into the useful ones. Since the process is tedious, John may not go through all the options, which means he could probably miss some good deals. Similar problems would occur for taxi, hotel, and car rental services.

2.2 Step 2: Selecting services

After locating the useful answers from search engines, the next step is to select the appropriate services. John would send further queries to get more information about the airline, taxi, hotel, and car rental services (Step 2 in Fig. 1). The challenge John needs to face is to make a selection from a large space of options combining his own situation. Manually selecting the more suitable services would be a painstaking process.

2.3 Step 3: Composing services

The service selection process may become more complicated since these services may be related to each other. Selection of one service may affect the selection of another. Services need to be composed and considered collectively (Step 3 in Fig. 1). For example, the arrival date, time, and location of

[1] http://www.google.com/checkout/.

the airline service will determine the date, time, and location of rental car pickup and hotel check-in. Therefore, feasible combinations of these service need to be identified. It is obvious that the large number of combinations makes any ad hoc approach infeasible. These pose additional challenges for the service selection problem.

Because of a lack of an efficient query mechanism, the above scenario points to the difficulties in devising an optimal strategy for Web service selection. To summarize, in this hypothetical scenario, John would have to first select services from a large number of candidates that provide similar functionalities. Second, the selected services may also affect one another. This makes the decision-making process more painstaking and time-consuming. Third, John may still miss some better plans because his manual analysis is performed in an ad hoc manner.

3 Web service query model

The preceding scenario shows that efficient service selection and combination is a challenging task. Users may need to consider a large pool of candidates, interpret their function and quality, and consider their *composition* with other services. We propose a three-level query model to address the above challenges: *query level*, *community level*, and *service level*. Users can submit declarative queries to express the tasks and the constraints that need to be satisfied. The query model provides a framework for the selection of those services that meet users' preferences. Queries are processed across the three levels to result in a *service execution plan* where Web services operations are invoked and their results combined. A two-phase optimization strategy will be integrated and work interactively with the three-level query model. The optimization strategy optimizes the service execution plan to best achieve users' quality requirements.

Figure 2 illustrates how the proposed query model could facilitate John in his travel planning. John can specify his travel tasks through a form-based interface provided by the *query level*, which is located at the top of the query model. He would initially be prompted to give his preferences and constraints as part of his user profile. For example, he is a frequent flyer with `NorthAmerican Airlines`. The user profile may also include the identity, location, schedule, and hobbies of a given user. *The community level* represents in a generic way the space of Web services within a given application domain. *Abstract operations* generalize the functionalities of a *service community*. They are implemented by the Web services that belong to the corresponding service community. Therefore, we use abstract operations to connect users' queries with actual Web services from the *service level*. This enables John to submit declarative queries without knowing about the details of any actual Web services.

3.1 Query level

At the query level, users can formulate and submit declarative queries. A query consists of tasks that users want to achieve. The query system provides input forms for users to specify their tasks. For example, a travel schedule may include taking a flight, reserving a hotel room, and getting a local map,

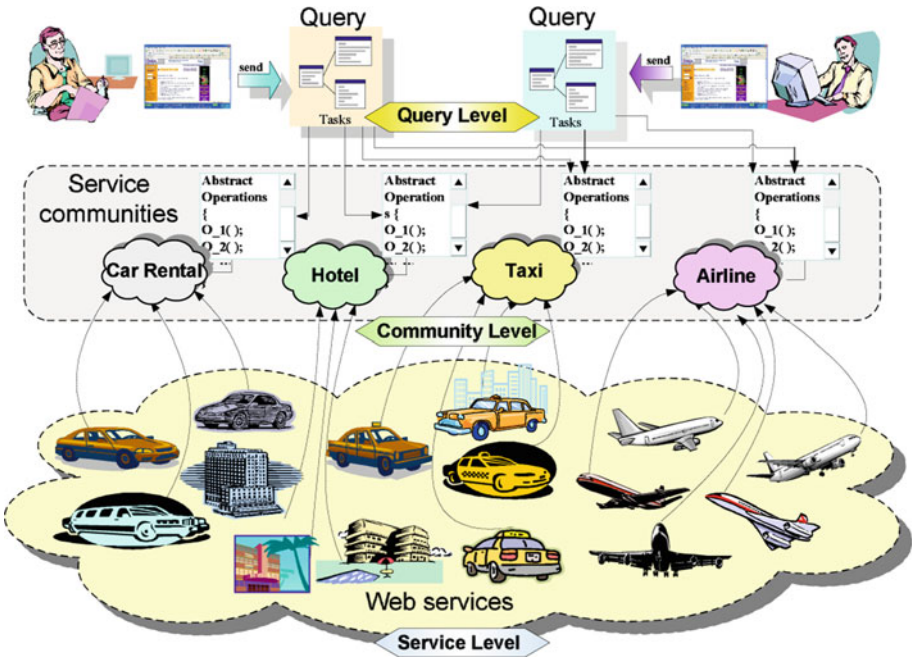


Fig. 2 Three level query model

etc. Tasks can be mapped to a set of abstract operations that provide the required functionalities.

We represent tasks as conjunctive queries over abstract operations (defined at the community level). More precisely, let $\mathcal{T}$ be the set of tasks specified at the query level and op_{ik} the kth abstract operation from service community *SCi*.

Definition 3.1 For any tasks $\mathrm{T_j} \in \mathcal{T}$,

$$\mathrm{T_j}(x_1, x_2, \ldots, x_n) : - \bigwedge_{i,k} op_{ik}(y_{ik}^1, y_{ik}^2, \ldots, y_{ik}^m) \bigwedge_{l} C_l$$

where x_j are the attributes of T_j, and y_{ik} are the corresponding operation's input variables. C_l's represent conditions on variables appearing in the different abstract operations op_{ik}. Their form is $C_l = x\ op\ c$, where x is an input or output variable from any op_{ik}, c a constant, and $op \in \{=, \neq, <, >, \geq, \leq\}$. $\diamond$

Let's use our travel planning example to illustrate the above definition. Consider the task of getting a local map, which be mapped to two operations:

$$Map : -geoCode(address) \wedge getMap(geoLocation)$$

The query level defines the mapping between tasks and abstract operations that enable users to use a declarative way to query Web services. Users only need to specify their desired task, and the corresponding abstract operations will be automatically identified. It is worth to note that the query is submitted against a given service community. We assume that the service community assigns unique operation signatures (including operation name and input/output parameters) to avoid problematic mapping between tasks and abstract operations. The mapping is then defined at the query level by considering the business logics of a given application domain. A detailed description of service communities is given in the following section.

3.2 Community level

The issue of discovering appropriate services to answer a query is particularly challenging because of the large and dynamic Web service space. Services required by a query need to be searched from an *exploratory* service space. Exploratory refers to the non-deterministic process of identifying the necessary Web services. To tackle this issue, we use *ontologies* to organize the large and dynamic Web service space [3]. Ontologies help describe *service communities*, which are collections of Web services that provide relevant functionalities in a given domain. More specifically, a service community is described by two ontologies: *Functionality Ontology* and *Quality Ontology*. *Functionality* is collectively represented by a list of *abstract services*. *Quality* defines a set of quality parameters, which can serve as the metrics to evaluate the service operations. The ontological descriptions of service communities are published in a service registry. Both service providers and consumers may discover a service community through a registry.

3.2.1 Functionality ontology

The functionality ontology organizes the abstract services in a hierarchical structure. Services on a higher layer refers to a more general functionality, while services on a lower layer refers to a more concrete functionality. The parent service and its child services have an inheritance relationship. Users can directly use the abstract services in the ontology to specify their tasks. The hierarchical structure gives users further flexibility to specify their tasks. For example, if a user has not determined the way for ground transportation, s/he can just specify ground transportation as the task. In this case, the optimization process will automatically select the best service plan for the user. On the other hand, if the user prefers to use rental car, s/he can directly specify car rental as the task in the query. Figure 3 shows the functional ontology for the Travel community. It is worth to note that the functional ontology is designed in an extensible manner where new abstract services can be easily added to the structure.

Each abstract service consists of a set of abstract operations. Abstract operation is the central concept of the service community because the abstract operations collectively reflect the functionality of an abstract service or an entire service community. An abstract operation is described by category (denoted as *Cate*), purpose (denoted as *Purp*), and invocation information (denoted as *Invo*). *Cate* describes the domain of interest of the operation or the service community.

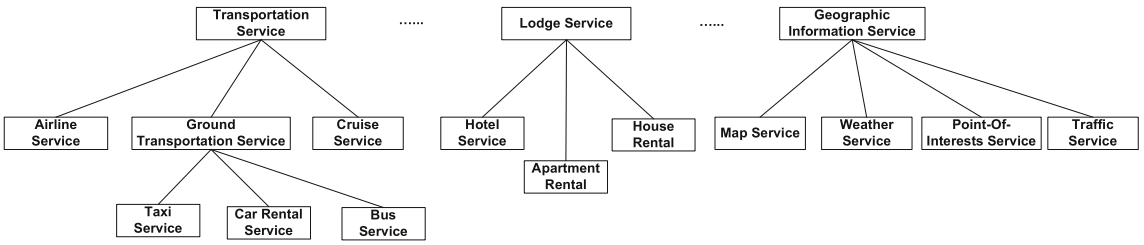


Fig. 3 Functionality ontology for the travel community

An operation may belong to multiple communities, and therefore it may have multiple categories. For example, the categories of the `getMap` operation may include travel and GIS. *Purp* specifies the business functionality of the operation. The functionality attributes can be defined by following the existing taxonomies such as RosettaNet, cXML (Commerce XML), and EDI (Electronic Data Interchange) [17]. *Invo* specifies how the operation can be invoked. The operations consume the service input (denoted as $\mathcal{I}$) and generate the output (denoted as $\mathcal{O}$) of the service. Since a service could offer multiple operations, there may be dependency constraints between different service operations. For example, in a map service, the `getMap` operation depends on the `geoCode` operation to transform an address or a zip code to the corresponding geo-location. There may be dependency between operations from different services as well. For example, if you reserve a package that includes both airline booking and car rental, the pickup time and location of the rental car are automatically determined by the air ticket. These dependencies can be usually derived from the business logics commonly accepted in the given domain. The dependencies can be formally specified using *preconditions (denoted as* $\mathcal{P}$) and *effects (denoted as* $\mathcal{E}$) of operations. In case that there are multiple preconditions and effects, we assume that the set of preconditions and effects is interpreted in a conjunctive manner. The definition of *Invo* can be formalized as follows:

Definition 3.2 The invocation information of an operation *op* is defined as a quadruple $(\mathcal{I}, \mathcal{O}, \mathcal{P}, \mathcal{E})$, where

- $\mathcal{I}$ is a set of input data that must be provided to invoke *op*;
- $\mathcal{O}$ is a set of output data generated by invoking *op*;
- $\mathcal{P}$ is a set of preconditions that need to be satisfied to invoke *op*;
- $\mathcal{E}$ is a set of effects generated by invoking *op*. $\diamond$

For example, the *Invo* of the `rentCar` operation can be defined as follows:

- $\mathcal{I}$ = {pickupDate, pickupTime, pickupLocation, dropOffDate, dropOffTime, dropOffLocation, userInfo, carType};
- $\mathcal{O}$ = {carReservation};
- $\mathcal{P}$ = {userAuthenticated = true $\wedge$ ((flightReserved = true $\wedge$ flightServiceInvoked = true) $\vee$ flightServiceInvoked = false)}
- $\mathcal{E}$ = {carReserved}.

Several semantic service languages, like OWL-S [22] and WSMO [31], can be directly used to describe the functionality of a community. For example, the definition of an operation can be mapped to the IOPE language constructs in OWL-S and thus can be precisely defined.

3.2.2 *Quality of a community*

Due to the large space of competing Web services, a query could be potentially solved by several service execution plans using different Web services. Thus, it is necessary to set appropriate criteria to select the "best" service execution plan. Recent literature shows that QoWS of individual Web services is crucial for their competitiveness [8]. In addition, there is an increasing need to provide acceptable QoWS over Web applications. The challenge is to define appropriate metrics to characterize QoWS and devise techniques to use it in optimizing service-based queries. In our approach, QoWS encompasses a number of quantitative and qualitative parameters (non-functional properties) that measure the Web service performance in delivering its functionalities.

We define a comprehensive ontological framework for QoWS in the context of Web services. Figure 4 shows the QoWS ontology including the different QoWS classes and their relationships. The root class QoWS has three subclasses: *runtime* quality, *business* quality, and *security* quality.

Runtime quality—represents the measurement of properties that are related to the execution of an operation op_{ik}. We identify three runtime quality classes: *response time*, *reliability*, and *availability*. The *response time* measures the expected delay between the moment when op_{ik} is initiated and the time op_{ik} sends the results. The *reliability* of op_{ik} is the ability of the operation to be executed within the maximum expected time frame. The *availability* is the probability that the operation is accessible.

Business quality—allows the assessment of an operation op_{ik} from a business perspective. We identify two business

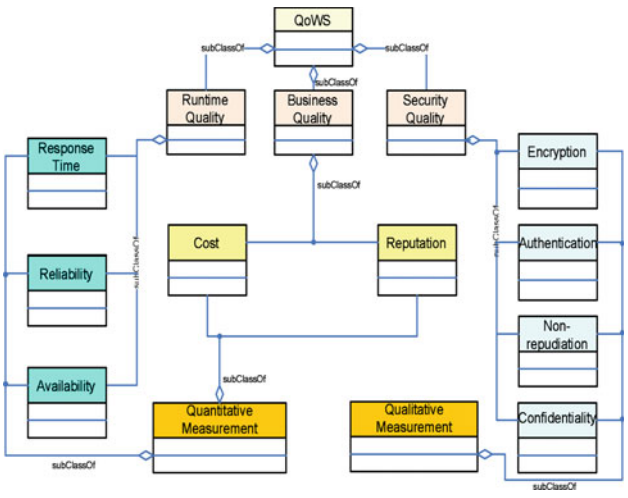


Fig. 4 QoWS ontology

Table 1 Instantiation of QoWS classes

QoWS class	QoWS subclass	Definition
Run-time	Response time	$\text{Time}_{process}(op_{ik}) + \text{Time}_{results}(op_{ik})$ where $\text{Time}_{process}$ is the time to process op_{ik} and $\text{Time}_{results}$ is the time to transmit/receive the results
	Reliability	$\text{N}_{success}(op_{ik})/\text{N}_{invoked}(op_{ik})$ where $\text{N}_{success}$ is the number of times that op_{ik} has been successfully executed and $\text{N}_{invoked}$ is the total number of invocations
	Availability	$\text{UpTime}(op_{ik})/\text{TotalTime}(op_{ik})$ where UpTime is the time op_{ik} was accessible during the total measurement time TotalTime
Business	Cost	Dollar amount to execute the operation
	Reputation	$\sum_{u=1}^{n} Ranking_u(op_{ik})/n$, $1 \leq \text{Reputation} \leq 10$ where $Ranking_u$ is the ranking by user u and n is the number of the times op_{ik} has been ranked
Security	Encryption	A Boolean equal to *true* iff messages are encrypted
	Authentication	A Boolean equal to *true* iff consumers are authenticated
	Non-repudiation	A Boolean equal to *true* iff participants cannot deny requesting or delivering the service
	Confidentiality	List of parameters that are not divulged to external parties

attributes: *cost* and *reputation*. The *cost* gives the dollar amount required to execute op_{ik}. The *reputation* of op_{ik} is a measure of the operation's trustworthiness. It mainly depends on users' experiences on invoking op_{ik}. Other parameters may be added depending on the target application domain.

Security quality—describes whether the operation op_{ik} is compliant with security requirements. Indeed, service providers collect, store, process, and share information about millions of users who have different preferences regarding security of their information. We identify four quality classes related to security: *encryption*, *authentication*, *non-repudiation*, and *confidentiality*. *Encryption* indicates whether op_{ik}'s message are securely exchanged (using encryption techniques) between servers and clients. *Authentication* states whether op_{ik}'s consumers (users and other services) are authenticated (e.g., through passwords). *Non-repudiation* specifies whether participants (consumers and providers) can deny requesting or delivering the service after the fact. *Confidentiality* indicates which parties are authorized to access the operation's input and output parameters. Confidentiality (op_{ik}) contains op_{ik}'s input and output parameters that should not be divulged to external entities (i.e., other than the service provider).

Response time, reliability, availability, cost, and reputation give quantitative measurements of Web services. For example, $\text{Response Time} = \text{Time}_{process}(op_{ik}) + \text{Time}_{results}(op_{ik})$, where $\text{Time}_{process}$ is the time to process op_{ik} and $\text{Time}_{results}$ is the time to transmit/receive the results. In contrast, encryption, authentication, non-repudiation, and confidentiality give qualitative measurements of Web services. For example, encryption is a Boolean, which is true if and only if messages are encrypted. Table 1 lists the instantiation of quality classes. This will be used to evaluate competing Web services and their compositions.

3.3 Service level

The service level represents the space of Web services offered on the Web—the potential candidates for answering queries. A Web service can register with a service community if it provides some operations in the community's operation list. During the registration, the Web service operations need to follow the specification of the abstract operations defined by the service community. For example, the number and types of parameters from the Web service operations need to be same as specified by the abstract operations. In addition, each Web service operation needs to be assigned with a set of quality parameters.

4 The two-phase optimization strategy

The QoWS model sets the appropriate criteria for selecting the best Web services among all possible competitors. We present our optimization strategy in this section. The optimization strategy contains two phases: constructing conceptual choreographies and generating service execution plans.

Figure 5 gives an overview of the two-phase optimization process. In the first phase, the optimization process takes as input the abstract operations obtained from the query level. It selects appropriate abstract operations and their combination and outputs feasible conceptual choreographies. For example, after John has rented a hotel, he needs to find a means to go to the conference venue. He could take a taxi, rent a car, or take the bus. Therefore, the `rentHotel` operation could be orchestrated with three possible operations, `takeTaxi`, `rentCar`, and `takeBus`. This would result in three conceptual choreographies. As the participant abstract operations increase, the possible conceptual choreographies will also increase accordingly. We

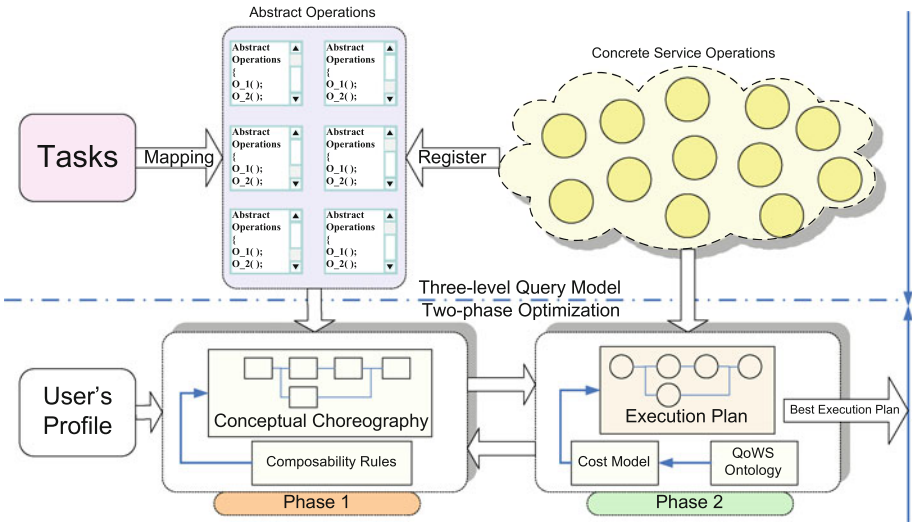


Fig. 5 Two-phase optimization strategy

define a set of rules to check the compatibility of abstract operations. This helps ensure that the participants of an choreography can be actually combined, hence avoiding unexpected failures at runtime. The second phase instantiates the abstract operation in the conceptual choreographies by selecting the registered concrete service operations to generate an optimal service execution plan. We define a cost model based on our QoWS framework to evaluate service execution plans.

In this section, we first illustrate the process of constructing the conceptual choreographies and service execution plans. We then present our cost model based on the QoWS parameters. Finally, we propose two optimization algorithms: greedy and evolutionary algorithms.

4.1 Constructing conceptual choreographies

When a user's task maps to more than one abstract operations, these operations need to form a *composable* conceptual choreography to perform the task. In this section, we first propose a set of rules, called *composability rules*. The proposed composability rules check whether two operations op_{ik} and op_{jl} are composable based on message and behavior. Message includes the input and output parameters of an operation. Behavior specifies an operation's business logics. We then introduce the basic choreography process, which uses the composability rules to combine different abstract operations. The optimization algorithms will extend this basic process by adding different selection mechanisms to generate the best service execution plan.

4.1.1 Composability rules

Message composability rule compares the properties of the input and output messages of two operations.

Definition 4.1 (*M-composable*). Let op_{ik} and op_{jl} be two operations. We say that op_{ik} is *M-composable* with op_{jl} if the data type of each output parameter of op_{ik} is a subtype of the data type for corresponding input parameter of op_{jl}. $\diamond$

Other rules for comparing message parameters (e.g., parameters' units such as US Dollar, AU Dollars, and Euro) may also be defined. Due to space limitation, we consider only data types and the number of parameters in this paper. Such rules may be simply defined similar to M-composability.

Behavioral composability rule (*B-Composability*) is constructed based on the business logics of Web services. We adopt Plan Domain Description Language (PDDL) to describe behavior composability rules [9]. PDDL helps specify the *precondition*, *effect*, and *parameters* of executing an operation op_{ik}:

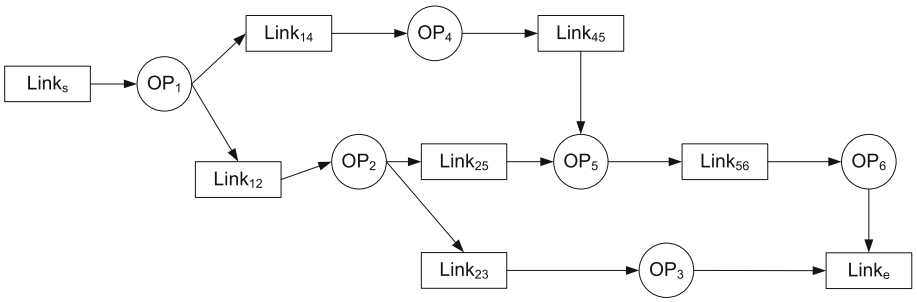
Action op_{ik}
$\rightarrow$ **Parameters[Input, Output]**
$\rightarrow$ **Precondition** $\{\text{PC}\}_1^n$
$\rightarrow$ **Effect** $\{\text{ET}\}_1^m$ $\diamond$

Definition 4.2 (*B-composable*) An operation op_{ik} is *B-composable* with op_{jl} if the following two conditions are true:

(1) op_{ik} is message composable with op_{jl}
(2) $\text{PC}(\text{op}_{jl}) \bigcap \text{ET}(\text{op}_{ik}) \neq \phi$, if $\text{PC}(\text{op}_{jl}) \neq \phi\diamond$

B-composability determines the dependencies between different operations when they are composed together in a service package. For example, if the service package includes both flight service and car rental service, car rental service needs to be invoked after flight service, because based on the

Fig. 6 The flow graph



B-composable rules, the effect of `flightReservation`, which is flightReserved, is among the preconditions of the `rentalCar` operation.

4.1.2 Choreography of abstract operations

We introduce the notion of *flow graph* to represent the conceptual choreography. The *flow graph* links together all abstract operations in the choreography. In essence, the flow graph is a bi-partite graph that consists of two types of nodes: *operation* and *compound* nodes (Fig. 6). Arrows in this graph relate nodes of different types. Each compound node *Link* in the flow graph has one predecessor op_i and one successor op_j (e.g., $Link_{23}$). This means that op_j can be invoked after the execution of op_i. There are two special compound nodes, Link$_s$ and Link$_e$, which represent the starting and ending points of the choreography.

The flow graph is built in an incremental fashion (Fig. 7). We assume that users provide the initial conditions and final objective of their tasks. For example, John needs to give his departure date and place as the initial conditions. Similarly, he needs also to specify the date and place of the conference as his objective. The abstract operations obtained from the mapping rules (see Definition 3.1) will be used to build the flow graph. We apply the behavior composability rule to construct the flow graph in the backward direction. The process starts by finding the abstract operation whose effects match users' objectives. It then continues to find abstract operations whose effects match some preconditions of an existing operation in the graph. If there is a match, it will check the message composability of these two operations. If these two operations are also message composable, which means they are behavior composable, the new operation will be inserted into the graph. For instance, if op_k is behavior composable with an existing operation op_i, then a new operation node is created for op_k. A new compound node $Link_{ki}$ is also inserted in the graph. The nodes op_k, $Link_{ki}$, and op_i are then linked by the following edges: $op_k \rightarrow Link_{ki} \rightarrow op_i$. The process will stop until all the preconditions in the graph can be fulfilled by the initial conditions or by the existing operations in the graph.

BuildConceptualChoreography
Input: A set of abstract operations $\mathcal{OP}$ from the mapping rules;
The starting and ending conditions of users' tasks $\mathcal{S}, \mathcal{E}$;
Output: A conceptual choreography $\mathcal{F}$;
(01) $\mathcal{PC}=\phi$; /*$\mathcal{PC}$ is the unmatched preconditions in $\mathcal{F}$*/
(02) for each $op_i \in \mathcal{OP}$
(03) if $(ET(op_i) \subseteq \mathcal{E})$
(04) $\mathcal{OP} = \mathcal{OP} - op_i$;
(05) $\mathcal{F}$.add_arrow(op_i, $Link_e$);
(06) $\mathcal{F}$.add_node(op_i);
(07) $\mathcal{PC} = \mathcal{PC} + PC(op_i)$;
(08) break;
(09) Endif
(10) Endfor
(11) **Choreograph**($\mathcal{PC}, \mathcal{F}, \mathcal{OP}, \mathcal{S}$);
(12) return $\mathcal{F}$;

Choreograph
Input: $\mathcal{PC}, \mathcal{F}, \mathcal{OP}, \mathcal{S}$;
(01) if $(\mathcal{PC} \subseteq \mathcal{S})$
(02) return;
(03) Endif
(04) else
(05) for each $op_i \in \mathcal{F}$
(06) for each $op_k \in \mathcal{OP}$
(07) if (is_B_composable(op_k, op_i))
(08) $\mathcal{F}$.add_node($Link_{ik}$);
(09) $\mathcal{F}$.add_arrow($op_i, Link_{ik}$);
(10) $\mathcal{F}$.add_node(op_k);
(11) $\mathcal{F}$.add_arrow($Link_{ik}, op_k$);
(12) $\mathcal{OP} = \mathcal{OP} - op_k$;
(13) $\mathcal{PC} = \mathcal{PC} - (ET(op_k) \cap PC(op_i))$;
(14) $\mathcal{PC} = \mathcal{PC} + PC(op_k)$;
(15) break;
(16) Endif
(17) Endfor
(18) Endfor
(19) **Choreograph**($\mathcal{PC}, \mathcal{F}, \mathcal{OP}, \mathcal{S}$);

is_B_composable
Input: op_k, op_i;
(01) if $((ET(op_k) \cap PC(op_i) == \phi) \wedge (PC(op_i) \neq \phi))$
(02) return false;
(03) Endif
(04) if (is_M_composable(op_k, op_i)==false)
(05) return false;
(06) Endif
(07) return true;

Fig. 7 Building A conceptual choreography

Figure 7 illustrates the basic steps of constructing a conceptual choreography. In the choreography process, operations are randomly selected from the abstract operation set and inserted into the flow graph as long as they can fulfill the B-composability rules. The optimization algorithms will rely

on this process and add some selection mechanisms, which help generate the best execution plan.

4.2 Generating service execution plans

Answering a query requires accessing actual Web services, which provide concrete operations. We need to replace the abstract operations in the conceptual choreography with concrete operations to build a *service execution plan*. The only difference between a conceptual choreography with a service execution plan is whether the operations are abstract or concrete. Since several Web services may register with the same abstract operation, the optimization strategy needs to have some criterion to evaluate multiple service execution plans and select the best one. In the next section, we propose a cost model based on our QoWS framework, which is used to assess the quality of execution plans.

4.3 Cost model

Since a service execution plan contains multiple operations, we need to aggregate the QoWS parameters from different operations. Table 2 lists the aggregation function for each of these QoWS parameters. Since users may have preferences over how their queries are answered, they may specify as part of their profile which and how important *QoWS* parameters are. We assign *weights*, ranging from 0 to 1, to each *QoWS* parameter to reflect the level of importance. Default values are otherwise used.

We are now ready to state the targeted optimization problem. Given a query, find operations from the Web service space which form a feasible service execution plan that maximizes the objective function F:

$$F = \left(\sum_{Q_i \in neg} W_i \frac{Q_i^{\max} - Q_i}{Q_i^{\max} - Q_i^{\min}} + \sum_{Q_i \in pos} W_i \frac{Q_i - Q_i^{\min}}{Q_i^{\max} - Q_i^{\min}} \right)$$

Table 2 QoWS for a service execution plan

QoWS parameter	Aggregation function
Response time	$\sum_{ws_i \in \max} time(op_i)$
Availability	$\prod_{i=1}^{N} av(op_i)$
Reliability	$\prod_{i=1}^{N} rel(op_i)$
Cost	$\sum_{i=1}^{N} cost(op_i)$
Reputation	$\min(rep(op\_i))$
Encryption	$\frac{1}{N}\sum_{i=1}^{N} enc(op_i)$ *or* $\prod_{i=1}^{N} enc(op_i)$
Authentication	$\frac{1}{N}\sum_{i=1}^{N} aut(op_i)$ *or* $\prod_{i=1}^{N} aut(op_i)$
Non-repudiation	$\frac{1}{N}\sum_{i=1}^{N} nrep(op_i)$ *or* $\prod_{i=1}^{N} nrep(op_i)$
Confidentiality	$\frac{1}{N}\sum_{i=1}^{N} con(op_i)$ *or* $\prod_{i=1}^{N} con(op_i)$

Where *neg* and *pos* are the sets of negative and positive *QoWS* respectively. In negative (resp. positive) parameters, the higher (resp. lower) the value, the worse is the quality. W_i are weights assigned by users to each parameter. Q_i is the value of the ith *QoWS* of the service execution plan obtained through the aggregation functions from Table 2. $Q_i^{\max}$ is the maximum value for the ith *QoWS* parameter for all potential service execution plans, and $Q_i^{\min}$ is the minimum. Assume that a SEP consists of m operations. These two values can be computed as follows.

$$Q_i^{\max} = agg_{j=1}^{m}(Q_i^{\max}(op_j))$$
$$Q_i^{\min} = agg_{j=1}^{m}(Q_i^{\min}(op_j))$$

where agg is the aggregation function for the ith QoWS attribute. From the above definitions, it is obvious that the purpose of including $Q_i^{\max}$ and $Q_i^{\min}$ is to normalize different QoWS attributes, so that they are comparable to each other.

4.4 Optimization algorithms

In this section, we propose two optimization algorithms: greedy algorithm and evolutionary algorithm. These two algorithms are based on the basic choreography process but adopt different mechanisms to select abstract operations and concrete operations. Both algorithms are designed to select the SEPs with the best QoWS but using different optimization strategies.

4.4.1 Greedy algorithm

The greedy algorithm conducts a local selection. When an abstract operation fulfills the B-composability rule, the algorithm takes each concrete operation that registers with this abstract operation as a subexecution plan and uses it to calculate the objective function F. The abstract operation takes the highest F value among all these concrete operations as its own F value. The concrete operation with the best F value will be temporarily stored. The greedy algorithm goes through all possible abstract operations that compete for the same operation node in the flow graph. It chooses the abstract operation that has the highest F value and inserts it into the graph. The corresponding concrete operation will be used to build the service execution plan. The selected concrete operations can be invoked based on their orders in the flow graph (Fig. 8).

The greedy algorithm is expected to be efficient in terms of response time because it only performs local search. A major drawback of this algorithm is that it does not optimize the service execution plan as a whole. In addition, the greedy algorithm can only ensure that single services comply to users' constraints. It provides no guarantee for the entire execution plan to comply with the constraints.

Greedy Algorithm
Input: A set of abstract operations $\mathcal{OP}$ from the mapping rules;
The starting and ending conditions of users' tasks $\mathcal{S}, \mathcal{E}$;
Output: A Local-optimal service execution plan $\mathcal{EP}$;
(01) $\mathcal{COP}_{\mathcal{F}} = \phi$; /* $\mathcal{COP}_{\mathcal{F}}$ is the selected concrete operations*/
(02) $\mathcal{EP} = \phi$;
(03) $\mathcal{F}$=**BuildConceptualChoreography**($\mathcal{OP}, \mathcal{S}, \mathcal{E}, \mathcal{COP}_{\mathcal{F}}$);
(04) $\mathcal{EP}$= buildPlan($\mathcal{F}, \mathcal{COP}$);
(05) return $\mathcal{EP}$;

BuildConceptualChoreography
Input: $\mathcal{PC}, \mathcal{F}, \mathcal{OP}, \mathcal{S}, \mathcal{COP}_{\mathcal{F}}$;
(01) if ($\mathcal{PC} \subseteq \mathcal{S}$)
(02) return;
(03) Endif
(04) else
(05) for each $op_i \in \mathcal{F}$
(06) op_k = **LocalSelection**($op_i, \mathcal{OP}, \mathcal{COP}_{\mathcal{F}}$);
(07) $\mathcal{F}$.add_node($Link_{ik}$);
(08) $\mathcal{F}$.add_arrow($op_i, Link_{ik}$);
(09) $\mathcal{F}$.add_node(op_k);
(10) $\mathcal{F}$.add_arrow($Link_{ik}, op_k$);
(11) $\mathcal{OP} = \overline{\mathcal{OP}} - op_k$;
(12) $\mathcal{PC} = \mathcal{PC} - (ET(op_k) \cap PC(op_i))$;
(13) $\mathcal{PC} = \mathcal{PC} + PC(op_k)$;
(14) Endfor
(15) Endif
(16) **BuildConceptualChoreography**($\mathcal{PC}, \mathcal{F}, \mathcal{OP}, \mathcal{S}, \mathcal{COP}_{\mathcal{F}}$);

LocalSelection
Input: $op_i, \mathcal{OP}, \mathcal{COP}_{\mathcal{F}}$;
(01) for each $op_k \in \mathcal{OP}$
(02) if (is_B_composable(op_k, op_i))
(03) for each $cop_k^j \in \mathcal{COP}_{\mathcal{K}}$ /*$\mathcal{COP}_{\mathcal{K}}$ is the list of concrete operations of op_k */
(04) $cop_k^j.F = compute\_F(cop_k^j)$;
(05) if ($cop_k^j.F > best_k.F$)
(06) $best_k = cop_k^j$;
(07) $best_k.F = cop_k^j.F$;
(08) Endif
(09) Endfor
(10) $op_k.F = best_k.F$;
(11) $op_k.cop = best_k$;
(12) Endif;
(13) if ($op_k.F > best.F$)
(14) $best = op_k$;
(15) Endif
(16) Endfor
(17) $\mathcal{COP} = \mathcal{COP} + best$;
(18) return best;

Fig. 8 Greedy algorithm

4.4.2 Evolutionary algorithm

Evolutionary algorithms (EAs) perform a global search to optimize the overall service execution plan. Therefore, they tackle the issue not addressed by the greedy algorithm. The selection of the concrete operations is deferred until the optimal execution plan has been found. EAs have become an effective approach in solving complex optimization problems [10]. They conduct a global search by simultaneously evaluating performances at multiple points in the solution space. In the very beginning, an initial population of multiple coded chromosomes representing random candidate solutions is created. Each chromosome is assigned a fitness value. At each generation of search, multiple candidates are evaluated, and the search will be directed intelligently according to the "survival-of-the-fittest" principle. This evolution cycle will be repeated until some stop criterion is satisfied. Although simplistic from a biologist's viewpoint, these algorithms are sufficiently complex to provide robust and powerful mechanisms for global search and optimization [10].

Evolutionary Algorithm
Input: A set of abstract operations $\mathcal{OP}$ from the mapping rules;
The starting and ending conditions of users' tasks $\mathcal{S}, \mathcal{E}$;
Output: Optimal or semi-optimal service execution plan.
(01) For g=1 to GenerationSize
(02) For i=1 to NumberOfChoreographies
(03) randomize($\mathcal{OP}$);
(04) $\mathcal{F}$=**BuildConceptualChoreography**($\mathcal{OP}, \mathcal{S}, \mathcal{E}$);
(05) $\{\mathcal{F}\} = \{\mathcal{F}\} + \mathcal{F}$;
(06) Endfor
(07) For each $\mathcal{F}_{\mathcal{K}} \in \{\mathcal{F}\}$
(08) For i=1 to SubpopulationSize
(09) for each $op_k^i \in \mathcal{F}_{\mathcal{K}}$;
(10) $op_k^i \leftarrow$ RandomSelect(Cop_k^i);
(11) Endfor
(12) Endfor
(13) newPopulation = $\bigcup_i$ subPopulation(i);
(14) if (g == 1)
(15) oldPopulation = newPopulation;
(16) else
(17) CaculateQuality(newPopulation);
(18) oldPopulation=Selection(oldPopulation,newPopulation);
(19) Crossover(oldPopulation);
(20) Mutation(oldPopulation);
(21) CaculateQuality(oldPopulation);
(22) EndFor
(23) $\mathcal{EP}_{BEST} \leftarrow$getBest(oldPopulation);
(24) return $\mathcal{EP}_{BEST}$;

Fig. 9 Evolutionary algorithm

The evolutionary algorithm for query optimization is specified in Fig. 9. Since the choreography process can generate multiple conceptual choreographies, different types of execution plans (generated from different conceptual choreographies) may co-exist. Due to its intrinsic parallel processing capacity, the evolutionary algorithm can co-evolve all these execution plans. For each type of execution plan, there is a corresponding subpopulation. All these subpopulations can go through their evolution process in parallel. Finally, they compete with each other and output the best execution plan. Thus, both the conceptual choreography and the service execution plan can be optimized.

The `CaculateQuality` operator uses the objective function to compute the quality value of each service execution plan in the population. The `Crossover` operator is a standard single-point crossover where two parents exchange their genes from a random position to reproduce the offspring [10]. Crossover is only applied to the execution plans from the same subpopulation to avoid breaking the composability rules. To improve the efficiency of EAs, we propose an enhanced `Mutation` operator. The new mutation operator aims to accelerate the convergence to the optimal solutions. Typically, the mutation operator would randomly replace an operation in the execution plan with another feasible operation. In the enhanced mutation operator, it will try a fixed number of times (say k) to look for an operation from the service space that can *dominate* the existing operation in the current population. If such an operation is found within k trials, the operator will replace the original operation with the newly selected one. Otherwise, the randomly selected operation from the $(k+1)^{th}$ trail will be used to replace the existing operation. The dominance concept has been introduced for computing database skylines [4]. A point $\vec{p}$ $(p_1, \ldots, p_d)$

Table 3 Symbols and parameters

Variables	
N_{op}	Number of operations in the registry
N_C	Number of communities
N_a	Number of abstract operations per community
N_m	Number of abstract operations obtained from the mapping rule
N_o	Number of operations per choreography
G_s	Maximum generation size of EA
O_s	Number of choreographies under evolution
P_s	Population size of EA
C_p	Crossover probability of EA
M_p	Mutation probability of EA
Performance measurement parameters and functions	
B_t	Time to check B-composability
CM_t	Time to compute cost model
I_t	Time to initialize a new population
S_t	Time to perform the selection operation
C_t	Time to perform the crossover operation
M_t	Time to perform the mutation operation
F_t	Time to compute the fitness function

dominates another point $\vec{r}\,(r_1, \ldots, r_d)$ if $\forall\, i \in [1, d]$, $p_i \succeq r_i$ and $\exists\ j \in [1, d]$, $p_j \succ r_j$. We use $\succeq$ to generally represent *better than or equal to* and $\succ$ to represent *better than*. In the context of Web services, operation op_i dominates operation op_j if op_i is as good as op_j in all QoWS attributes and better than op_j in at least one QoWS attribute. More importantly, for any monotone objective function F, given any two execution plans, plan_i and plan_j, where these execution plans have the same set of operations except that plan_i includes op_i and plan_j includes op_j, we have $F_{plan_i} \geq F_{plan_j}$. This nice property helps accelerate the convergence of the EA algorithm. Meanwhile, the mutation operator still keeps the randomness to avoid being trapped to a local optimal.

5 Analytical model

In this section, we present the analytical model for the above optimization algorithms. Table 3 defines the parameters and the symbols used in this section.

5.1 Greedy algorithm

We start by considering the minimum time required by the greedy algorithm. In this case, the algorithm finds a feasible conceptual choreography in a single iteration. This means that the algorithm selects N_o abstract operations from the available operation list, and these N_o operations can form a feasible choreography to fulfill the user's tasks. When an abstract operation is selected, it will be removed from the available operation lists. For the i^{th} abstract operation in the conceptual choreography, the algorithm needs to check the $(N_m - (\mathrm{i} - 1))$ operations for B_composability. On average, there are N_m/N_o abstract operations competing for the same operation node in the choreography. For each of these abstract operations, the algorithm needs to compute the objective function of their registered concrete operations. The average number of concrete operations per abstract operation is $N_{op}/(N_C \times N_a)$. Therefore, the total time for deciding the i^{th} operation in the service execution plan is:

$$Ti = (N_m - (i-1)) \times N_m/N_o \times B_t \times N_{op}/(N_C \times N_a) \times CM_t$$

The lower bound for the greedy algorithm to build a service execution plan is:

$$\Omega\left(\sum_{i=1}^{N_0} T_i\right)$$

The upper bound refers to the case where the algorithm goes through all the possible combinations of N_o operations out of N_m options, which is:

$$O\left(\frac{N_m!}{(N_m - N_o)!} \times \sum_{i=1}^{N_0} T_i\right)$$

5.2 Evolutionary algorithm

The evolutionary algorithm contains two major phases in each generation. The first phase is to construct a set of conceptual choreographies and initialize a subpopulation of execution plans for each conceptual choreography. For each conceptual choreography, we compute the lower and upper bounds of the construction time as follows:

$$\Omega\left(\sum_{i=1}^{N_0}(N_m-(i-1))\times B_t\right)$$
$$O\left(\frac{N_m!}{(N_m-N_o)!}\times\sum_{i=1}^{N_0}(N_m-(i-1))\times B_t\right)$$

Evolutionary algorithm needs to construct O_s conceptual choreographies in the first phase. It also needs to initialize P_s execution plans based on these conceptual choreographies. Therefore, the lower and upper bounds required by the first phase is:

$$\Omega\left(\sum_{i=1}^{N_0}(N_m-(i-1))\times B_t\times O_s+P_s\times I_t\right)$$
$$O\left(\frac{N_m!}{(N_m-N_o)!}\times\sum_{i=1}^{N_0}(N_m-(i-1))\times B_t+P_s\times I_t\right)$$

The second phase applies four operations on the population of execution plans. The population size, P_s, is kept stable. The selection operation needs to be conducted on two populations of execution plans, which are $2\times P_s$ times. The crossover operation is applied to two candidate solutions, and the crossover probability is C_p. Hence, the crossover operation is conducted $P_s\times C_p/2$ times. The mutation operation applies to one candidate solution and has a probability M_p. It will be performed $P_s\times M_p$ times. The objective function needs to be computed for each execution plan in two populations. It will be performed $2\times P_s$ times. Therefore, the time complexity for the second phase is:

$$T_2=P_s\times\left(\frac{1}{2}\times C_p\times C_t+M_p\times M_t+2\times F_t\right)$$

The evolutionary algorithm will run G_s generations. Assume that the time for the first phase is T_1. Thus, the total time for EA is $G_s\times(T_1+T_2)$. Based on the above analysis, the lower and upper bounds of the total computation time are as follows:

$$\Omega\left(\sum_{i=1}^{N_0}(N_m-(i-1))\times B_t\times O_s+P_s\times I_tP_s\times\left(\frac{1}{2}\times C_p\times C_t+M_p\times M_t+2\times F_t\right)\right)$$
$$O\left(\frac{N_m!}{(N_m-N_o)!}\times\sum_{i=1}^{N_0}(N_m-(i-1))\times B_t+P_s\times I_tP_s\times\left(\frac{1}{2}\times C_p\times C_t+M_p\times M_t+2\times F_t\right)\right)$$

6 Experimental study

We conducted a set of experiments to assess the performance of the proposed algorithms. We run our experiments on a *Sun Enterprise Ultra 10* server with a 440-MHz *UltraSPARC-IIi* processor, 1-GB of RAM, and under *Solaris* operating system. Table 4 summarizes the parameter settings of the experiments. In the first parameter setting, the average number of concrete operations that register with an abstract operation is 250. This is derived using the formula $N_{op}/(N_C\times N_a)$. The average number of operations per conceptual choreography varies from 10 to 160. The population size of the evolutionary algorithm is set to 100, and the mutation and crossover probabilities are set to 0.8 and 0.5, respectively. Based on the characteristics of the generation size, we propose two evolutionary algorithms. The first one—*Static EA (SEA)*—has a fixed generation size, and the second one—*Dynamic EA (DEA)*—has a adaptive generation size. The generation size of SEA is set to 200. The generation size of DEA adapts to the complexity of the task. It changes based on the number of operations in the conceptual choreography, it is set to $No\times 5$. In the second setting, the average number of concrete operations that register with an abstract operation is 500. Since the

Table 4 Simulation settings

Parameters	1st setting	2nd setting
N_{op}	50,000	100,000
N_C	10	10
N_a	20	20
No	10–160	10–160
$G_s(SEA)$	200	200
$G_s(DEA)$	$No\times 5$	$No\times 5$
P_s	100	200
M_p	0.8	0.8
C_p	0.5	0.5

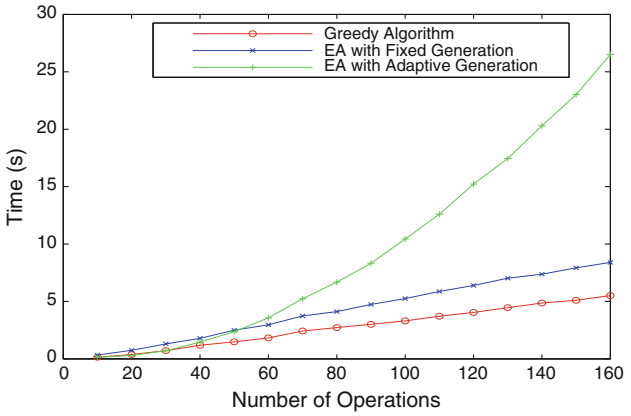


Fig. 10 Processing time with the 1st setting

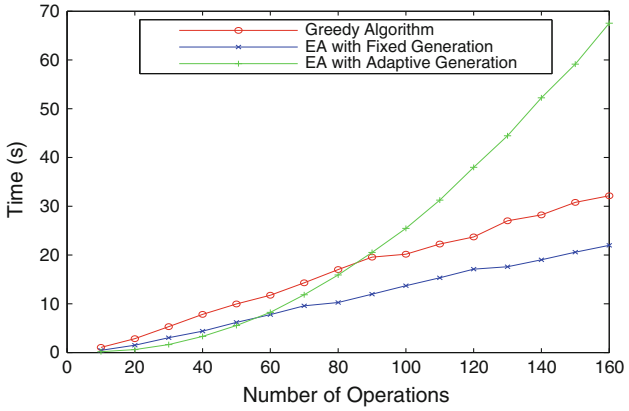


Fig. 11 Processing time with the 2nd setting

search space is made larger, we also increase the population sizes from 100 to 200 for both SEA and DEA. The other parameters remain the same with the first setting.

Figures 10 and 11 compare the computational time of the greedy algorithm, SEA, and DEA. The greedy algorithm has a similar computational time as SEA. When an abstract operation has 250 registered concrete operation (the first parameter setting), the greedy algorithm uses slightly less time for query optimization than SEA. When an abstract operation has 500 registered concrete operations (the second setting), SEA uses slightly less time for query optimization than the greedy algorithm. Since the generation size of SEA is constant, the evolution time of SEA is mainly affected by the population size. It is worth noting that the population size of SEA changes based on the number of concrete operations per abstract operation. When the concrete operation number increases from 250 to 500, the population size also doubles. If the population size is further increased, SEA may need more time than the greedy algorithm in the second setting. DEA uses less time when the task contains a small number of operations. As the number of operation increases, the processing time of DEA increases considerably. In the first setting case, DEA spends more time than the other two algorithms when a task have 50 or more operations. Similarly, in the second setting, DEA spends more time than the other two algorithms when a task has 80 or more operations.

Figures 12, 13, and 14 illustrate the performance behavior of SEA and DEA. We apply the second parameter setting to two different tasks to compare these two algorithms. Figure 12 shows how SEA and DEA work on a complex task, which contains 110 operations. SEA has a fixed generation size and stops evolving at generation 200, whereas DEA has an adaptive generation size and evolves 550 generations. SEA stops evolving when its fitness still has a large increase rate. It does not provide enough generation size to converge to the best execution plan. In contrast, DEA stops evolving when its fitness stops increasing, which means the algorithm has converged to the best solution. The performance difference shows the fitness difference of the best solutions achieved by these two algorithms. The fitness is calculated using the cost model proposed in Sect. 4.3. Due to its early stop, the best execution plan achieved by SEA has a fitness of

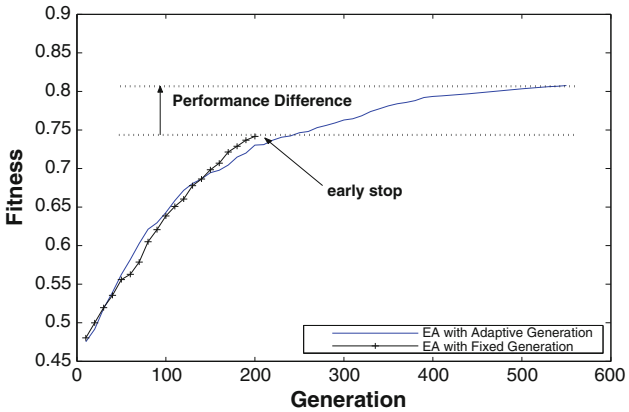


Fig. 12 110 Operations per task

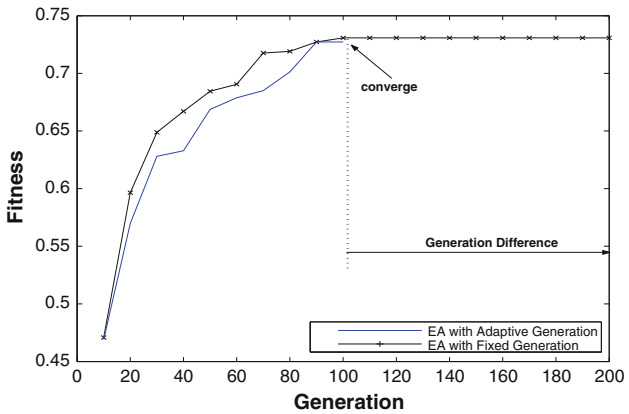


Fig. 13 20 Operations per task

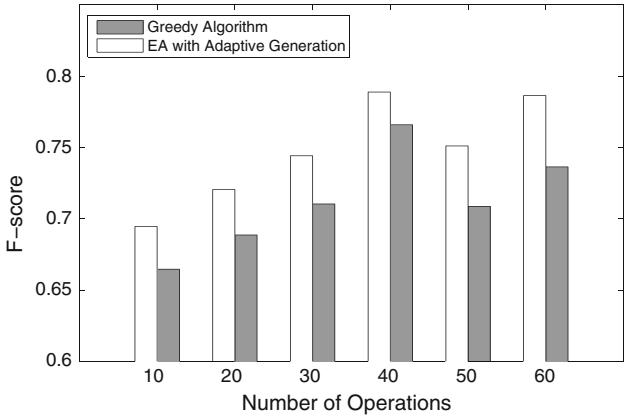


Fig. 14 F-score of best service execution plans

0.742. On the other hand, the best fitness value achieved by DEA is 0.807, which is almost 10% higher than that of SEA. Figure 13 shows the situation when SEA and DEA work on a simple task, which contains 20 operations. SEA still evolves 200 generations, whereas DEA stops at generation 100. In this case, DEA only spends half time as SEA. However, it achieves a fitness value (0.727) which is only 0.4% lower than the one (0.730) achieved by SEA.

The experiment results show the usefulness of an adaptive generation size to the evolutionary algorithm. When dealing with a complicated task that has a large number of operations, the evolutionary algorithm has enough time to achieve convergence. When the task is simple, the evolutionary algorithm spends less time and also guarantees good solutions. The usefulness of the adaptive generation size can also be justified by the relationship between the enhanced mutation operator and the number of tasks in a candidate execution plan. The mutation operator helps guide the evolution process to find the optimal solution. It applies to a candidate execution plan by randomly selecting an operation and replacing it with a better one chosen from the service space. Since a complicated task contains a large number of operations, the mutation operator needs to be applied to the corresponding execution plan several times to reach the optimal solution. Increasing the generation size helps the evolutionary algorithm achieve this effect.

In the final set of experiments, we compare the quality of the SEPs generated by the greedy algorithm and EAs (Fig. 14). Since the last set of experiments justify that DEA is more effective in selecting the best SEPs, we only compare the greedy algorithm with DEA with respect to the quality of the best execution plans. The number of operation per task varies from 10 to 60. As can be seen from the result, DEA is able to select execution plans with better quality than the greedy algorithm. Based on the experimental results on both performance and quality, we can see that when the number of concrete operations per abstract operation is relatively small, say less than 100, DEA is the best choice among the three algorithms, because it is very efficient and also generates the best quality. When the number of concrete operations per abstract operation becomes larger, the processing time with DEA increases dramatically. In this case, the greedy algorithm is a better choice because the user can get much more efficient processing time by only sacrificing moderate quality.

7 Related work

The proliferation of Web services is fostering a very active research area. We examine major research prototypes, standards, and platforms which are most closely related to our work.

7.1 Semantic service description

Techniques have recently been proposed to deal with semantic description of Web services. *OWL-S* (formerly DAML-S) defines a semantic markup for Web services based on the use of ontologies [22]. OWL-S introduces the notions of prerequisites (called Preconditions) and consequences (called Effects) of Web services operations. It supports the specification of composite services. However, it does not include QoWS properties into its service descriptions. *WSMO* (originating from WSMF) defines a Web service modeling ontology for describing semantic Web services [31]. It specifies a set of non-functional properties to describe the quality aspect of a Web service. However, it does not use a formal and extensible ontology to describe these QoWS parameters. Additionally, WSMO does not specify how to aggregate these properties to evaluate Web services.

7.2 Web service composition

Web service composition has been the focus of several recent research projects. Some of them are complementary to our work and could help orchestrate service operations. XLANG, WSFL, and BPEL4WS are among the standardization efforts to enable service composition [2,11,20]. They assume that service composers are responsible for checking service and operation compatibility. Semantic and quality aspects of Web services are not considered. Similarly, some other service composition techniques, including WISE, eFlow, and CMI, also require the intervention of service composers. eFlow and CMI enable the selection of services for a single task. However, they do not provide a QoWS framework that helps optimize the entire service execution plan. Composability rules and automatic service composition techniques have been proposed in [18,19]. The focus of these techniques is mainly on the "functional" correctness of the composition instead of

using the "non-functional" properties for service selection. These techniques can be complementary to our framework. For example, they can be used in the first phase of the optimization strategy for building conceptual choreographies. In [33], a composite service optimization approach is proposed based on several quality of service parameters. The authors considered a set of possible parameters as their quality criteria. Composite services are represented as a state-chart. It is not clear, however, how the state-chart would be built. The authors expressed the optimization problem of finding the best Web services to execute a composite service in the form of a linear programming problem. The major difference with our approach is that they only focused on service composition optimization, whereas we take a holistic view of optimization, i.e., from submission to the return of the results. We adopt a two-phase optimization strategy that works interactively with the three-level query model to help achieve users' objectives in an optimal way. In addition, our work defines a formal QoWS ontology and gives an instantiation of QoWS classes defined in the ontology instead of randomly choosing several parameters as the quality criteria. This helps our query infrastructure better capture users' requirements.

7.3 Web service optimization

In [24], an optimization algorithm is proposed to efficiently access Web services. The optimization algorithm takes as input the classical database SPJ like queries over Web services. It uses a cost model to arrange Web services in a query and computes a pipelined execution plan with minimum total running time of the query. In contrast, the optimization framework proposed in this work is both 'user centered' and 'performance-centered'. It focuses on efficiently helping users select the services (or their compositions) with their desired quality. The Web Service Level Agreement (WSLA) language considers other quality attributes in addition to running time. However, the major focus is to ensure that the service providers deliver their services based on the asserted quality guarantees [13]. Quality-aware service optimization techniques have been studied in [32]. However, these approaches assume that a feasible composition plan is already available, and the optimization is to select providers that result in a plan with the best quality. A genetic algorithm has been presented in [6] for selecting the best services. Similarly, the optimization starts from an existing composition plan. In addition, our EA algorithms go beyond this simple GA approach by including a co-evolution strategy that can optimize multiple feasible composition plans simultaneously. In [21], a mutli-level query model is proposed that offers query optimization functionalities for Web services. A user's query can be transformed through these levels and end up with an execution plan. Our approach goes beyond this ad hoc query model by proposing an integrated framework, where service queries can be formulated, processed, and optimized in a disciplined and systematic manner.

7.4 QoS management

QoS management is a major research area in middleware and networking communities [1, 14]. However, research efforts in these communities are mainly focus on the performance of network and devices. Several recent projects in workflow systems also support QoS management to improve the workflow quality of service [7]. However, they do not provide the query mechanisms that enable common users to specify their tasks using declarative queries. In [15], a QoS ontology has been defined for Web services. However, this work does not seem to formulate how to measure the QoS parameters clearly. Additionally, it does not specify how to aggregate all these parameters for service evaluation.

8 Conclusion

We present in this paper a service query framework that enables users to easily access services with their best desired QoWS in an expected large service space. The framework integrates a formal query model and a two-phase optimization strategy. The query model identifies a set of key features of Web services in a given domain and organize these services into service communities. The service communities offer a layer of abstraction that allows users to interact with their desired service without worrying about the underlying technical details. The optimization strategy leverages a QoWS-based cost model to select the best execution plans. Experimental results demonstrate the effectiveness of the optimization algorithms. In future work, we plan to explore the following three important directions:

- The query processing and optimization algorithms rely on the knowledge of the quality information from the service instances. A key extension of this work is to develop quality management mechanisms that can monitor the performance of service providers and precisely report their quality values.
- Our QoWS model and aggregation functions do not take into consideration of missing quality values that may be common in real-world scenarios. Work on fuzzy-set-based querying (e.g., SQL-F [5]) may be relevant to handling the situation of missing values.
- The objective function-based optimization requires users to express their preference over different (and sometimes conflicting) quality parameters as numeric weights. Transforming personal preferences to numeric weights is a rather demanding task for users. It is interesting to incorporate the skyline concept into the service

optimization process. The skylines are computed automatically based on the inherent QoWS features of service providers. Thus, it completely frees service users from the challenging weight assignment task.

References

1. Aurrecoechea C, Campbell AT, Hauw L (1998) A Survey of QoS Architectures. ACM/Springer Verlag Multimed Syst J 6(3):138–151
2. BEA, IBM, and Microsoft (2003) Business process execution language for Web Services (BPEL4WS). http://xml.coverpages.org/bpel4ws.html
3. Berners-Lee T, Hendler J, Lassila O (2001) The semantic web. Sci Am 284(5):34–43
4. Börzsönyi S, Kossmann D, Stocker K (2001) The skyline operator. In: ICDE '01: Proceedings of the 17th International conference on data engineering. IEEE Computer Society, Washington, DC, pp 421–430
5. Bosc P, Pivert O (1995) SQLf: a relational database language for fuzzy querying. IEEE Trans Fuzzy Syst 3(1):1–17
6. Canfora G, Di Penta M, Esposito R, Villani ML (2005) An approach for qos-aware service composition based on genetic algorithms. In: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation. ACM, New York, pp 1069–1075
7. Cardoso J (2002) Quality of service and semantic composition of workflows. Ph.D Thesis, University of Georgia, Athens
8. Conti M, Kumar M, Das SK, Shirazi BA (2002) Quality of service issues in internet Web services. IEEE Trans Comput 51(6):593–594
9. Ghallab M et al (1998) PDDL: The planning domain definition lanaguage, version 1.2. Technical report CVC TR-98-003/DCS TR-1165, Yale center for computational vision and control, Yale University, New Haven
10. Goldberg DE (1989) Genetic algorithms in search, optimization, and machine learning. Addison Wesley, Massachusetts
11. IBM (2003) Web Services Flow Language (WSFL). http://xml.coverpages.org/wsfl.html
12. Langdon CS (2003) The state of Web services. IEEE Comput 36(7):93–94
13. Keller A, Dan A, King RP, Ludwig H, Franck R Web service level agreement (WSLA) language specification
14. Marchetti C, Pernici B, Plebani P (2004) A quality model for multichannel adaptive information. In: WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on alternate track papers & posters. ACM, New York, pp 48–54
15. Maximilien EM, Singh MP (2004) A framework and ontology for dynamic web services selection. IEEE Internet Comput 8(5):84–93
16. Mcllraith SA, Son TC, Zeng H (2001) Semantic Web services. IEEE Intell Syst 16(2):46–53
17. Medjahed B, Benatallah B, Bouguettaya A, Ngu AHH, Elmagarmid AK (2003) Business-to-business interactions: issues and enabling technologies. VLDB J 12(1):59–85
18. Medjahed B, Bouguettaya A (2005) A multilevel composability model for semantic web services. IEEE Trans Knowl Data Eng 17(7):954–968
19. Medjahed B, Bouguettaya A, Elmagarmid AK (2003) Composing web services on the semantic web. VLDB J 12(4):333–351
20. Microsoft (2003) Web Services for Business Process Design (XLANG). http://xml.coverpages.org/xlang.html
21. Ouzzani M, Bouguettaya B (2004) Efficient access to web services. IEEE Internet Comput 37(3):34–44
22. OWL-S (2004) http://www.daml.org/services/owl-s/
23. Petrie C, Bussler C (2003) Service agents and virtual enterprises: a survey. IEEE Internet Comput 7(4):68–78
24. Srivastava U, Munagala K, Widom J, Motwani R (2006) Query optimization over Web services. In: VLDB '06: Proceedings of the 32nd international conference on very large data bases. VLDB Endowment, pp 355–366
25. 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 27th international conference on very large data bases, Roma, pp 633–636, Sept. 2001
26. Vaughan-Nichols SJ (2002) Web services: beyond the hype. IEEE Comput 35(2):18–21
27. Vinoski S (2002) Web services interaction models, part 1: current practice. IEEE Internet Comput 6(3):89–91
28. W3C. Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap
29. W3C. Universal Description, Discovery, and Integration (UDDI). http://www.uddi.org
30. W3C. Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl
31. WSMO (2004) http://www.wsmo.org/
32. Yu T, Zhang Y, Lin K-J (2007) Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans Web 1(1):6
33. Zeng L, Benatallah B, Ngu AHH, Dumas M, Kalagnanam J, Chang H (2004) Qos-aware middleware for web services composition. IEEE Trans Softw Eng 30(5):311–327