SOCA (2016) 10:111–133
DOI 10.1007/s11761-014-0167-5

ORIGINAL RESEARCH PAPER

Generalized semantic Web service composition

Srividya Bansal · Ajay Bansal · Gopal Gupta · M. Brian Blake

Received: 29 January 2014 / Revised: 31 August 2014 / Accepted: 23 October 2014 / Published online: 8 November 2014

Abstract With the increasing popularity of Web Services and Service-Oriented Architecture, we need infrastructure to discover and compose Web services. In this paper, we present a generalized semantics-based technique for automatic service composition that combines the rigor of process-oriented composition with the descriptiveness of semantics. Our generalized approach presented in this paper introduces the use of a conditional directed acyclic graph where complex interactions, containing control flow, information flow, and pre-/post-conditions are effectively represented. Composition solution obtained is represented semantically as OWL-S documents. Web service composition will gain wider acceptance only when users know that the solutions obtained are comprised of trustworthy services. We present a framework that not only uses functional and non-functional attributes provided by the Web service description document but also filters and ranks solutions based on their trust rating that is computed using Centrality Measure of Social Networks. Our contributions are applied for automatic workflow generation in context of the currently important bioinformatics domain. We evaluate our engine for automatic workflow generation of a phylogenetic inference task. We also evaluate our engine for automated discovery and composition on repositories of different sizes and present the results.

S. Bansal (✉) · A. Bansal
Arizona State University, Mesa, AZ, USA
e-mail: srividya.bansal@asu.edu

A. Bansal
e-mail: ajay.bansal@asu.edu

G. Gupta
The University of Texas at Dallas, Richardson, TX, USA
e-mail: gupta@utdallas.edu

M. B. Blake
University of Miami, Miami, FL, USA
e-mail: m.brian.blake@miami.edu

Keywords Service composition · Service discovery · Semantic Web · Ontology · Workflow generation

1 Introduction

The next milestone in the evolution of the World Wide Web is making services ubiquitously available. As automation increases, Web services will be accessed directly by the applications themselves rather than by humans [1,2]. In this context, a Web service can be regarded as a "programmatic interface" that makes application-to-application communication possible. To make services ubiquitously available, we need infrastructure that applications can use to automatically discover, deploy, compose, and synthesize services. A Web service is an autonomous, platform-independent program accessible over the web that may affect some action or change in the world. Sample of Web services include common plane, hotel, rental car reservation services or device controls like sensors or satellites. A Web service can be regarded as a "programmatic interface" that makes application-to-application communication possible. Informally, a service is characterized by its input parameters, the outputs it produces, and the actions that it initiates. The input parameter may be further subject to some pre-conditions, and likewise, the outputs produced may have to satisfy certain post-conditions. In order to make Web services more practical, we need an infrastructure that allows users to discover, deploy, synthesize, and compose services automatically. To make services ubiquitously available, we need a semantics-based approach such that applications can reason about a service's capability to a level of detail that permits their discovery, composition, deployment, and synthesis [3]. Several efforts are underway to build such an infrastructure [4–6].

With regard to service composition, a composite service is a collection of services combined together in some way to achieve a desired effect. Traditionally, the task of automatic service composition has been split into four phases: (i) Planning, (ii) Discovery, (iii) Selection, and (iv) Execution [7]. Most efforts reported in the literature focus on one or more of these four phases. The first phase involves generating a plan, i.e., all the services and the order in which they are to be composed in order to obtain the composition. The plan may be generated manually, semi-automatically, or automatically. The second phase involves discovering services as per the plan. Depending on the approach, often planning and discovery are combined into one step. After all the appropriate services are discovered, the selection phase involves selecting the optimal solution from the available potential solutions based on non-functional properties like QoS properties. The last phase involves executing the services as per the plan and in case any of them are not available, an alternate solution has to be used.

In this paper, we present a general approach for automatic service composition. Our composition algorithm performs planning, discovery, and selection automatically, all at once, in one single process. This is in contrast to most methods in the literature where one of the phases (most frequently planning) is performed manually. Additionally, our method generates most general compositions based on (conditional) directed acyclic graphs (DAG). Note that service discovery is a special case of composition of n services, i.e., when $n = 1$. Thus, we mainly study the general problem of automatically composing n services to satisfy the demand for a particular service, posed as a query by the user. In our framework, the DAG representation of the composite service is reified as an OWL-S description. This description document can be registered in a repository and is thus available for future searches. The composite service can now be discovered as a direct match instead of having to look through the entire repository and build the composition solution again. We show how service composition can be applied to a Bioinformatics analysis application, for automatic workflow generation in the field of Phylogenetics [8].

One of the current challenges in automatic composition of Web services also includes finding a composite Web service that can be trusted by consumers before using it. Our approach uses analysis of Social Networks to calculate a trust rating for each Web service involved in the composition and further prune results based on this rating. Web-based Social Networks have become increasingly popular these days. Social Network Analysis is the process of mapping and measuring the relationships between connected nodes. These nodes could represent people, groups, organizations, computers, or any knowledge entity. We propose to measure the trust factor of a service by measuring the centrality of a service provider and/or a service provider organization in a well-known Social Network. The three level indices that can be applied to measure centrality are degree, betweenness, and closeness [9]. We adopt our idea of computing trust using centrality measure based on the notion of centrality and prestige being key in the study of social networks [9,10]. The role of central people (nodes with high centrality) in a network seems to be fundamental as they adopt the innovation and help in transportation and diffusion of information throughout the rest of the network. So our rationale is that these central figures who play a fundamental role in the network are trusted by others in the network who are connected (directly or indirectly) to them.

A simple use case scenario Jane is a researcher in the field of Evolutionary Genetics. One evening she is examining the evolution of crab species and needs to build a Phylogenetic tree for various crab species using protein sequence data. In order to complete this task, she will have to go to her lab and access the computer with necessary software and perform multiple computations using various algorithms. She uses the well-known Molecular Evolutionary Genetics Analysis software program (MEGA5) [11] that is an integrated tool for conducting automatic and manual sequence alignment, inferring phylogenetic trees, mining web-based databases, estimating rates of molecular evolution, inferring ancestral sequences, and testing evolutionary hypotheses. Jane has to use this software to first align the sequence data using one of several algorithms (such as Clustal W [12], MUSCLE [13], etc.) provided by MEGA5 for this purpose. Next, she wants to compute and compare the evolutionary distances for sequences from crab species using various algorithms. In order to do this she must first compute relevant models for crab species. The next step is to compute evolutionary distances using Jukes–Cantor model followed by Tamura–Nei model and compare them. She is also interested in the evolutionary distance computed based on proportion of amino acid differences. Finally, she is interested in building a Phylogenetic tree from the aligned sequence data. She will have to pick one of the algorithms/methods provided by MEGA5 that include Maximum Likelihood, Minimum Evolution, Maximum Parsimony, Neighbor-Joining, etc. Currently, there is no easy way to perform this analysis and she will have to use the software tools manually and go through this step-by-step process and wait while computation for each of the steps is being performed. Jane will have to go through this labor-intensive process in order to find an answer to her research question.

Imagine a Software-as-a-Service (SaaS) platform [14] available on the cloud and Jane has access to it from any computer or mobile device. With a few simple clicks, Jane provides a query request that includes input parameters to this workflow process and expected final outputs. The software platform built upon our composition engine produces multiple possible workflows using different combinations of algorithms/methods (available as Web services) for each of the tasks such as sequence alignment, model computation, distance computation, and generation of phylogeny. Jane picks a workflow that is most suited for her research analysis and possibly even edits the workflow by adding a service to compute the diversity in the subpopulation of crabs. She saves off this workflow to her profile for future use. She initiates the workflow execution and on the following day, analyses the output results that were produced and saved off in her account. This software platform is able to save Jane a significant amount of time—not only in performing the computations for analysis, but also with configuring workflows and using interesting workflows already created by her colleagues. This is just one simple example of the potential of Web service discovery and composition in various disciplines. This paper presents the underlying composition engine that is needed in order to build such a software platform.

This paper extends our previous work in the area of Web service composition [15] by conducting a case study on automatic workflow generation for Phylogenetic Inference tasks in Bioinformatics using our composition engine and introducing the computation of a trust rating of each Web service, based on Centrality measure in Social Network analysis, and using this trust rating in filtering and ranking services. This work would support the development of a SaaS platform that supports domain-specific workflow generation. Our research makes the following novel contributions:

(i) Formalization of the generalized composition problem based on our conditional directed acyclic graph representation; (ii) Computation of trust rating of composition solutions based on individual ratings of service providers obtained using the Centrality measure of Social Networks; (iii) Efficient and scalable algorithm for solving the composition problem that takes semantics of services into account; our algorithm automatically discovers and selects the individual services involved in composition for a given query, without the need for manual intervention; (iv) Automatic generation of OWL-S descriptions of the new composite service obtained; (v) Case study of our generalized composition engine to automatically generate workflows in the field of Bioinformatics for Phylogenetic Inference tasks.

The rest of the paper is organized as follows. In Sect. 2, we present the related work in the area of Web service discovery and composition and discuss their limitations. In Sect. 3, we formalize the generalized Web service composition problem. We present our multi-step narrowing technique for automatic Web service composition and automatic generation of OWL-S service description in Sect. 4. We present the implementation and experimental results in Sect. 5. Section 6 presents an application of our generalized composition engine to automatically generate workflows for Bioinformatics analysis tasks. The last section presents conclusions and future work.

2 Related work

Composition of Web services has been active area of research [7,16,17]. Most of these approaches present techniques to solve one or more phases of composition as listed in Sect. 1. There are many approaches [6,18,19] that solve the first two phases of composition namely planning and discovery. These are based on capturing the formal semantics of the service using action description languages or some kind of logic (e.g., description logic). The service composition problem is reduced to a planning problem where the sub-services constitute atomic actions and the overall service desired is represented by the goal to be achieved using some combination of atomic actions. A planner is then used to determine the combination of actions needed to reach the goal. With this approach an explicit goal definition has to be provided, whereas such explicit goals are usually not available. To the best of our knowledge, most of these approaches that use planning are restricted to sequential compositions, rather than a directed acyclic graph. In this paper, we present a technique to automatically select atomic services from a repository and produce compositions that are not only sequential but also non-sequential that can be represented in the form of a directed acyclic graph. The authors in [18] present a composition technique by applying logical inferencing on predefined plan templates. Given a goal description, they use the logic programming language Golog to instantiate the appropriate plan for composing Web services. This approach also relies on a user-defined plan template, which is created manually. One of the main objectives of our work is to come up with a technique that can automatically produce composition without the need for any manual intervention. Boustil et al. [20] present an approach that uses an intermediate ontology built using OWL-DL and SWRL rules to define the affected object and their relationships. Their selection strategy considers relationships between services by looking at object values of affected objects. They use a custom intermediate ontology that is built within their framework using OWL-DL. Our approach focuses on the semantics of the parameters as well as constraints represented as pre- and post-conditions. Also our approach is generic and can be used with any domain ontology to provide semantics.

There are industry solutions based on WSDL and BPEL4 WS where the composition flow is obtained manually. BPEL4WS can be used to define a new Web service by composing a set of existing ones. It does not assemble complex flows of atomic services based on a search process. They select appropriate services using a planner when an explicit flow is provided. In contrast, our technique automatically determines these complex flows using semantic descriptions of atomic services. A process-level composition solution based on OWL-S is proposed in [19]. In this work, the authors assume that they already have the appropriate individual services involved in the composition, i.e., they are not automatically discovered. They use the descriptions of these individual services to produce a process-level description of the composite service. They do not automatically discover/select the services involved in the composition, but instead assume that they already have the list of atomic services. In contrast, we present a technique that automatically finds the services that are suitable for composition based on the query requirements for the new composed service. There are solutions such as [21] that solve the selection phase of composition. This work uses pre-defined plans and discovered services provided in a matrix representation. Then, the best composition plans are selected and ranked based on QoS parameters like cost, time, and reputation. These criterions are measured using fuzzy numbers.

There has been a lot of work on composition languages such as WS-BPEL, FuseJ, AO4BPEL, etc. which are useful only during the execution phase. FuseJ is a description language for unifying aspects and components [22]. Though this language was not designed for Web services, the authors contend that it can be used for service composition as well. It uses connectors to interconnect services. We believe that there is no centralized process description, but instead information about services is spread across the connectors. With FuseJ, the planning phase has to be performed manually that is the connectors have to be written by the developer. Similarly, OWL-S also describes a composite service but does not automatically find the services involved in the composition. So these languages are only useful for execution which happens after the planning, discovery, and selection of services is done. Service grounding of OWL-S maps that describe abstract services to the concrete WSDL specification helps in executing the service. In contrast, our approach automatically generates the composite service. This new composite service generated can then be described using one of these composition languages.

QoS-aware composition has also been active area of research [6,21]. Research on a QoS-aware composition [23–25] consider applying SLA's to workflow compositions or Web service compositions, although they do not perform dynamic composition. They use one of the existing composition languages to create the composite service manually or create a template that is later used to select appropriate services for each stage of composition. After obtaining composition solutions manually or semi-automatically, these approaches present a QoS model and apply the non-functional attributes on the potential solutions to confirm that they comply with the pre-defined agreements. Thus, the solutions are pruned based on SLA compliance. Work on workflow Composition of service-level agreements [26] presents a set of SLA measures and principles that best support QoS-based Composition. A model and representation of SLA attributes were introduced and an approach to compose SLA's associated with a workflow of Web services was presented. The research on creating a QoS-Aware middleware for Web service Composition in [27] is similar to our work as they identify services that can fit into a useful composition based on QoS measures. They use two approaches for selection: one based on local (task-level) selection of services and the second is based on a global allocation of tasks to services. They also use a template for composition; in this case, a state chart that has the generic service tasks defined. Finding a composite service involves finding concrete services that fit into the template. In contrast, we do not use any template but instead find the composition solution automatically. The work presented in [28] combine semantic annotations and SLA's thereby providing better approach to specification of SLA's.

Researchers have looked into a fuzzy linguistic preference model to provide preference relations on various QoS dimensions [29]. They use a specific weighting procedure to provide numeric weights to preference relations, and then use a hybrid evolutionary algorithm to find skyline solutions efficiently. Their algorithm is designed on the basis of Pareto-dominance and weighted Tchebycheff distance. In this approach, the authors assume that they have candidate services for composition. Their algorithm helps identify best solution based on their SLA's.

Feng's research group proposed an approach to composition that associated QoS attributes to service dependencies and showed their approach could model real-life services and perform effective QoS constraint satisfaction and optimization [30]. The attributes taken into consideration by this study are Response time, Cost, Reliability, Availability, and Reputation. They consider that QoS values of a service may be dependent not only on the service itself but also some other services in the workflow. They propose 3 types of QoS for each attribute namely: default QoS, partially dependent QoS, and totally dependent QoS. Default QoS applies no matter what the preceding service is in a workflow, just like the conventional QoS. Partially dependent QoS applies if and only if some of the inputs of a service are provided by the outputs of another service. Totally dependent QoS applies if and only if all inputs of a service are provided by the outputs of another service. Formal modeling of QoS attributes is provided in OWL-S [27]. Work by Wen et al. [32] presents an approach to obtaining probabilistic top-K dominating services with uncertain QoS. QoS values tend to fluctuate at run-time and hence this approach uses probabilistic characteristics of service instances to identify dominating service abilities for better selection. A detailed survey of approaches for a reliable dynamic Web service composition is presented by Immonen and Pakkala [33]. They discuss various approaches that use Reliability ontology to manage and achieve reliable composition. They address the lack of formalization to handle reliability of composition, whereas the focus of our approach is the formalization of a generalized composition that uses functional attributes to compose a solution and non-functional attributes help in further filtering and ranking solutions.

A number approaches focus on trust and reputation QoS criteria for service selection. Mehdi et al.'s [34] approach assigns trust scores to Web services and only services with highest scores are selected for composition. They use Bayesian networks to learn the structure of composition. The approach presented by Kutler et al. [35] considers social trust in Web service composition. They compute trust based on similarity measures over ratings of users added into a system. They use the correlation between trust and overall similarity measures in online communities. On the contrary, we use the centrality measure in a Web-based Social Network.

In this paper, we present a technique for automatically planning, discovering, and selecting services that are suitable for obtaining a composite service based on user-query requirements. As far as we know, all the related approaches to this problem assume that they either already have information about services involved or use human input on what services would be suitable for composition. This work is an extension of our earlier work [15] that introduced a generalized Web service composition engine. In this paper, we use the trust rating of a Web service in addition to the functional and non-functional attributes of a service in filtering and ranking solutions. In this paper, we evaluate our composition the engine using a case study from the bioinformatics domain for Phylogenetic inference tasks to show that this engine can be used for automatic workflow generation. The case study uses example workflows that would be generated as a sequential composition, non-sequential composition as well as non-sequential conditional composition.

3 Automated Web service discovery and composition

Discovery and composition are two important tasks related to Web services. In this section, we formally describe these tasks and develop the requirements of an ideal discovery/composition engine.

3.1 The discovery problem

Given a repository of Web services, and a query requesting a service (hereafter *query service*), automatically finding a service from the repository that matches these requirements is the Web service discovery problem. Only those services that produce at least the requested output parameters that satisfy the post-conditions and use only from the provided input parameters that satisfy the pre-conditions and produce the same side effects can be valid solutions to the query. Some of the solutions may be over-qualified, but they are still considered valid as long as they fulfill input and output parameters, pre-/post-conditions, and side effect requirements. This activity is best illustrated using an example:

Example (*Discovery*) A buyer is looking for a service to buy a book and the directory of services contains services S_1 and S_2. Table 1 shows the input/output parameters of the query and services S_1 and S_2. In this example service S_2 satisfies the query; however, S_1 does not as it requires *BookISBN* as an input and it is not provided by the query. Our query requires *ConfirmationNumber* as the output and S_2 produces *ConfirmationNumber* and *TrackingNumber*. The extra output produced can be ignored. Also the semantic descriptions of service input/output parameters should be same as the query parameters or satisfy the subsumption relation. The discovery engine should be able to infer that the query parameter *BookTitle* and input parameter *BookName* of service S_2 are semantically the same concepts. This can be inferred using semantics from the annotation of the service and the ontology (e.g., OWL WordNet ontology) provided. The query also has a pre-condition that the *CreditCardNumber* is numeric, which should logically imply pre-conditions of the discovered service.

Definition (*Service*) A service is a 6-tuple of its pre-conditions, inputs, side effect, affected object, outputs and post-conditions.

$S = (CI, I, A, AO, O, CO)$ is the representation of a service where *CI* is the list of pre-conditions, *I* is the input list,

Table 1 Discovery—example

Service	Input parameters	Pre-conditions	Output parameters	Post-conditions
Query	BookTitle, CreditCardNumber, AuthorName, CreditCardType	lsNumeric(CreditCard Number)	ConfirmationNumber	
S_1	BookName, AuthorName, BooklSBN, CreditCardNumber		ConfirmationNumber	
S_2	BookName, CreditCardNumber	lsNumeric(CreditCard Number)	ConfirmationNumber, TrackingNumber	

A is the service's side effect, AO is the affected object, O is the output list, and CO is the list of post-conditions. The pre- and post-conditions are ground logical predicates.

Definition (*Repository of Services*) Repository (R) is a set of Web services.

Definition (*Query*) The *query service* is defined as $Q = (CI', I', A', AO', O', CO')$ where CI' is the list of pre-conditions, I' is the input list, A' is the service affect, AO' is the affected object, O' is the output list, and CO' is the list of post-conditions. These are all the parameters of the requested service.

Definition (*Discovery*) Given a repository R and a query Q, the discovery problem can be defined as automatically finding a set S of services from R such that $S = \{s \mid s = (CI, I, A, AO, O, CO), s \in R, CI' \Rightarrow CI, I \sqsubseteq I', A = A', AO = AO', CO \Rightarrow CO', O \sqsupseteq O'\}$. The meaning of $\sqsubseteq$ is the subsumption (subsumes) relation and $\Rightarrow$ is the implication relation. For example, say x and y are input and output parameters, respectively, of a service. If a query has $(x > 5)$ as a pre-condition and $(y > -x)$ as post-condition, then a service with pre-condition $(x > 0)$ and post-condition $(y > x)$ can satisfy the query as $(x > 5) \Rightarrow (x > 0)$ and $(y > x) \Rightarrow (y > -x)$ since $(x > 0)$. Figure 1 shows the substitution rules for the discovery problem.

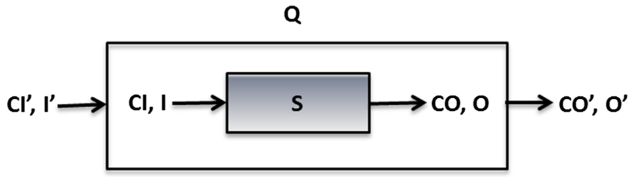


Fig. 1 Substitutable service

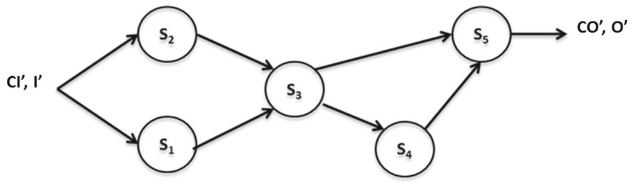


Fig. 2 Composite service represented as a directed acyclic graph

3.2 The composition problem

Given a repository of service descriptions, and a query with the requirements of the requested service, in case a matching service is not found, the composition problem involves automatically finding a directed acyclic graph of services that can be composed to obtain the desired service. Figure 2 shows an example composite service made up of five services S_1 to S_5. In figure, I' and CI' are the query input parameters and pre-conditions, respectively. O' and CO' are the query output parameters and post-conditions respectively. Informally, the directed arc between nodes S_i and S_j indicates that outputs of S_i constitute (some of) the inputs of S_j.

Example (*Sequential Composition*) Suppose we are looking for a service to make travel arrangements, i.e., flight, hotel, and rental car reservations. The directory of services contains *ReserveFlight*, *ReserveHotel*, and *ReserveCar* services. Table 2 shows the input/output parameters of the user query and the three services *ReserveFlight*, *ReserveHotel*, and *ReserveCar*. For the sake of simplicity, the query and services have fewer input/output parameters than the real-world services. In this example, service *ReserveFlight* has to be executed first so that its output *ArrivalFlightNum* can be used as input by *ReserveHotel* followed by the service *ReserveCar* which uses the output *HotelAddress* of *ReserveHotel* as its input. The semantic descriptions of the service input/output parameters should be the same as the query parameters or have the subsumption relation. This can be inferred using semantics from the ontology provided. Figure 3 shows this example sequential composition as a directed acyclic graph.

Definition (*Sequential Composition*) The sequential Composition problem can be defined as automatically finding a directed acyclic graph $G = (V, E)$ of services from repository R, given query $Q = (CI', I', A', AO', O', CO')$, where V is the set of vertices and E is the set of edges

Table 2 Sequential composition example

Service	Input parameters	Pre-conditions	Output parameters	Post-conditions
Query	PassengerName, OriginAirport, StartDate, DestinationAirport, ReturnDate		HotelConfirmationNum, CarConfirmationNum	
ReserveFlight	PassengerName, OriginAirport, StartDate, DestinationAirport, ReturnDate		FlightConfirmationNum, ArrivalFlightNum	
ReserveHotel	PassengerName, ArrivalFlightNum, StartDate, ReturnDate		HotelConfirmationNum, HotelAddress	
ReserveCar	PassengerName, ArrivalDate, ArrivalFlightNum, HotelAddress		CarConfirmationNum	

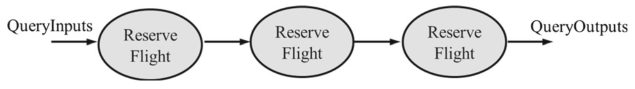


Fig. 3 Sequential composition example

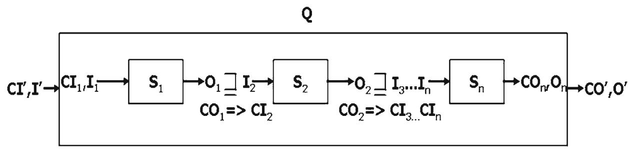


Fig. 4 Sequential composition

of the graph. Each vertex in the graph represents a service in the composition. Each outgoing edge of a node (service) represents the outputs and post-conditions produced by the service. Each incoming edge of a node represents the inputs and pre-conditions of the service. The following conditions should hold on the nodes of the graph: $\forall i \quad S_i\ V, S_i\ R, S_i = (CI_i, I_i, A_i, AO_i, O_i, CO_i)$

1. $I' \sqsupseteq I_1, O_1 \sqsupseteq I_2, \ldots, O_n \sqsupseteq O'$
2. $CI' \Rightarrow CI_1, CO_1 \Rightarrow CI_2, \ldots, CO_n \Rightarrow CO'$

The meaning of the $\sqsupseteq$ is the subsumption (subsumes) relation, and $\Rightarrow$ is the implication relation. In other words, we are deriving a possible sequence of services where only the provided input parameters are used for the services and at least the required output parameters are provided as an output by the chain of services. The goal is to derive a solution with minimal number of services. Also, the post-conditions of a service in the chain should imply the pre-conditions of the next service in the chain. Figure 4 depicts an instance of sequential composition.

Example (*Non-sequential composition*) Suppose we are looking for a service to buy a book and the directory of services contains services *GetISBN*, *GetAvailability*, *AuthorizeCreditCard*, and *PurchaseBook*. Table 3 shows the input/output parameters of the query and the four services in the repository. Suppose a single matching service is not found in the repository, a solution is synthesized from among the set of services available in the repository. Figure 5 shows this composite service. The post-conditions of the service *GetAvailability* should logically imply the pre-conditions of service *PurchaseBook*.

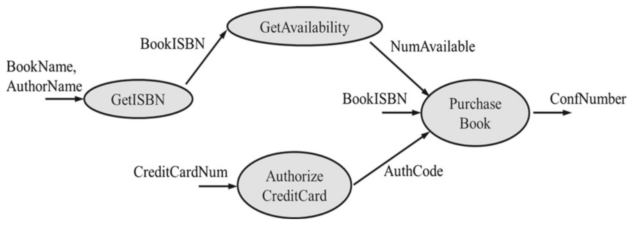


Fig. 5 Non-sequential composition example

Definition (*Non-sequential composition*) More generally, the Composition problem can be defined as automatically finding a directed acyclic graph $G = (V, E)$ of services from repository R, given query $Q = (CI', I', A', AO', O', CO')$, where V is the set of vertices and E is the set of edges of the graph. Each vertex in the graph represents a service in the composition. Each outgoing edge of a node (service) represents the outputs and post-conditions produced by the service. Each incoming edge of a node represents the inputs and pre-conditions of the service. The following conditions should hold on the nodes of the graph:

1. $\forall i \quad S_i \in V$ where S_i has exactly one incoming edge that represents the query inputs and pre-conditions, $I' \sqsupseteq \cup_i I_i,\ CI' \Rightarrow \wedge_i CI_i$.
2. $\forall i\, S_i \in V$ where S_i has exactly one outgoing edge that represents the query outputs and post-conditions, $O' \sqsubseteq \cup_i O_i,\ CO' \Leftarrow \wedge_i CO_i$.
3. $\forall i\, S_i \in V$ where S_i has at least one incoming edge, let $S_{i1}, S_{i2}, \ldots, S_{im}$ be the nodes such that there is a directed edge from each of these nodes to S_i. Then, $I_i \sqsubseteq \cup_k O_{ik} \cup I',\ CI_i \Leftarrow (CO_{i1} \wedge CO_{i2} \ldots \wedge CO_{im} \wedge CI')$.

Figure 6 depicts an instance of non sequential composition.

Table 3 Non-sequential composition example

Service	Input parameters	Pre-conditions	Output parameters	Post-conditions
Query	BookTitle, CreditCardNum, AuthorName, CardType		ConfNumber	
GetISBN	BookName, AuthorName		ConfNumber	
GetAvailability	BookISBN		NumAvailable	NumAvailable > 0
Authorize CreditCard	CreditCardNum		AuthCode	AuthCode > 99 ∧ AuthCode < 1000
PurchaseBook	BookISBN, NumAvailable, AuthCode	NumAvailable > 0	ConfNumber	

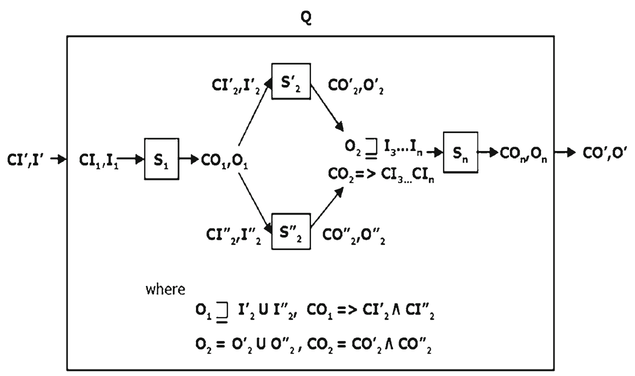


Fig. 6 Non-sequential composition

Example (*Non-sequential conditional composition*) Non-sequential conditional composition consists of if-then-else conditions, i.e., the composition flow varies depending on the result of the post-conditions of a service. Suppose we are looking for a service to make international travel arrangements. We first need to make a tentative flight and hotel reservation and then apply for a visa. If the visa is approved, we can buy the flight ticket and confirm the hotel reservation, else we will have to cancel both the reservations. Also, if the visa is approved, we need to make a car reservation. The repository contains services *ReserveFlight*, *ReserveHotel*, *ProcessVisa*, *ConfirmFlight*, *ConfirmHotel*, *ReserveCar*, *CancelFlight*, and *CancelHotel*. Table 4 shows the input/output parameters of the user query and services. In this example, service *ProcessVisa* produces the post-condition *VisaApproved* $\vee$ *VisaDenied*. The services *ConfirmFlight* and *ConfirmHotel* have the pre-condition *VisaApproved*. In this case, one cannot determine whether the post-conditions of service *ProcessVisa* implies the pre-conditions of services *ConfirmFlight* and *ConfirmHotel* until the services are actually executed. In such a case, a condition can be generated which will be evaluated at runtime and depending on the outcome of the condition, the corresponding services will be executed. The vertex for service *ProcessVisa* in the graph is followed by a condition node which represents the post-condition of service *ProcessVisa*. This node has two outgoing edges one representing the case if the condition is satisfied at run-time and other edge for the case where the condition is not satisfied. In other words, these edges represent the generated conditions which in this case are, (*VisaApproved* $\vee$ *VisaDenied*) $\Rightarrow$ *VisaApproved* and *VisaApproved* $\vee$ $\underline{\text{V}}$isaDenied) $\Rightarrow$ *VisaDenied*. Depending on which condition holds, the corresponding services *ConfirmFlight* or *CancelFlight* are executed. Figure 7 shows this conditional composition example as a directed acyclic graph.

Definition (*Generalized Composition*) The generalized Composition problem can be defined as automatically finding a directed acyclic graph $G = (V, E)$ of services from repository R, given query $Q = (CI', I', A', AO', O', CO')$, where V is the set of vertices and E is the set of edges of the graph. Each vertex in the graph either represents a service involved in the composition or post-condition of the immediate predecessor service in the graph, whose outcome can be determined only after the execution of the service. Each outgoing edge of a node (service) represents the outputs and post-conditions produced by the service. Each incoming edge of a node represents the inputs and pre-conditions of the service. The following conditions should hold on the nodes of the graph:

Table 4 Non sequential conditional composition example

Service	Pre-conditions	Input parameters	Output parameter	Post-conditions
Query		PassengerName, OriginAirport, StartDate, DestinationAirport, ReturnDate	FlightConfirmationNum, HotelConfirmationNum, CarConfirmationNum	
ReserveFlight		PassengerName, OriginAirport, StartDate, DestinationAirport, ReturnDate	FlightConfirmationNum, ArrivalFlightNum	
ReserveHotel		PassengerName, ArrivalFlightNum, StartDate, ReturnDate	HotelConfirmationNum, HotelAddress	
ProcessVisa		PassengerName, VisaType, FlightConfirmationNum, HotelConfirmationNum	ConfirmationNum	VisaApproved *V* VisaDenied
ConfirmFlight	VisaApproved	FlightConfirmationNum, CreditCardNum	FlightConfirmationNum	
ConfirmHotel	VisaApproved	HotelConfirmationNum, CreditCardNum	HotelConfirmationNum	
CancelFlight	VisaDenied	FlightConfirmationNum, PassengerName	CancelCode	
Cancel Hotel	VisaDenied	HotelConfirmationNum, PassengerName	CancelCode	
ReserveCar		PassengerName, ArrivalDate, HotelAddress, ArrivalFlightNum	CarConfirmationNum	

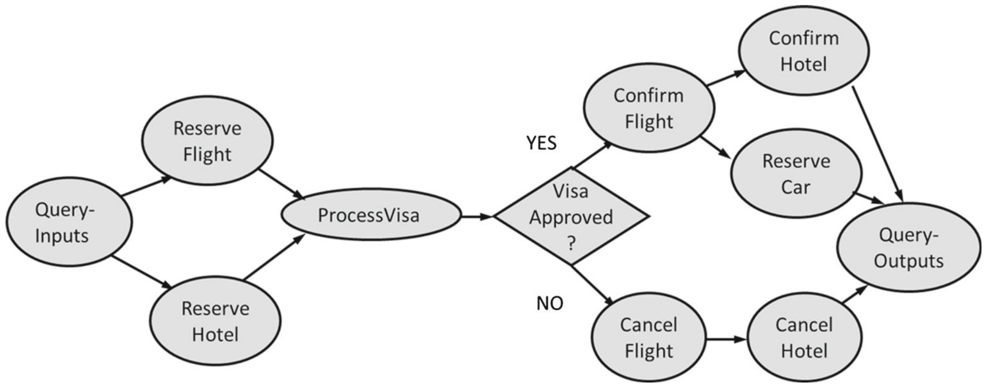


Fig. 7 Non-sequential conditional composition

1. $\forall i \quad S_i \in V$ where S_i has exactly one incoming edge that represents the query inputs and pre-conditions, $I' \sqsupseteq \cup_i I_i,\ CI' \Rightarrow \wedge_i\ CI_i$.
2. $\forall i\, S_i \in V$ where S_i has exactly one outgoing edge that represents the query outputs and post-conditions, $O' \sqsubseteq \cup_i O_i,\ CO' \Leftarrow \wedge_i\ CO_i$.
3. $\forall i\, S_i \in V$ where S_i represents a service and has at least one incoming edge, let $S_{i1}, S_{i2}, ..., S_{im}$ be the nodes such that there is a directed edge from each of these nodes to S_i. Then, $I_i \sqsubseteq \cup_k O_{ik} \cup I',\ CI_i \Leftarrow (CO_{i1} \wedge CO_{i2} \ldots \wedge CO_{im} \wedge CI')$.
4. $\forall i\, S_i \in V$ where S_i represents a condition that is evaluated at run-time and has exactly one incoming edge, let S_j be its immediate predecessor node such that there is a directed edge from S_j to S_i. Then, the inputs and pre-conditions at node S_i are $I_i = O_j \cup I'$; $CI_i = CO_j$. The outgoing edges from S_i represent the outputs that are same as the inputs I_i and the post-conditions that are the result of the condition evaluation at run-time.

The meaning of the $\sqsubseteq$ is the subsumption (subsumes) relation and $\Rightarrow$ is the implication relation. In other words, a service at any stage in the composition can potentially have as its inputs all the outputs from its predecessors as well as the query inputs. The services in the first stage of composition can only use the query inputs. The union of the outputs produced by the services in the last stage of composition should contain all the outputs that the query requires to be produced. Also the post-conditions of services at any stage in composition should imply the pre-conditions of services in the next stage. When it cannot be determined at compile time whether the post-conditions imply the pre-conditions or not, a *conditional* node is created in the graph. The outgoing edges of the conditional node represent the possible conditions that will be evaluated at run-time. Depending on the condition that holds, the corresponding services are executed. That is, if a subservice S_1 is composed with subservice S_2, then the post-conditions CO_1 of S_1 must imply the preconditions CI_2 of S_2. The following conditions are evaluated at run-time:

if $(CO_1 \Rightarrow CI_2)$ *then execute* S_1;
else if $(CO_1 \Rightarrow \neg\, CI_2$ *then no-op*$\}$;
else if $(CI_2$ *then execute* $S_1\}$;

When the number of nodes in the graph is equal to one, the composition problem reduces to the discovery problem. When all nodes in the graph have not more than one incoming edge and not more than one outgoing edge, the problem reduces to a sequential composition problem. Further details and examples are available in our prior work [15].

3.3 Requirements of an ideal engine

The features of an ideal discovery/composition engine are as follows:

Correctness One of the most important requirements for an ideal engine is to produce correct results, i.e., the services discovered and composed by it should satisfy all the requirements of the query. Also, the engine should be able to find all services that satisfy the query requirements.

Minimal query execution time Querying a repository of services for a requested service should take a reasonable amount of (minimal) time, i.e., a few milliseconds. Here, we assume that the repository of services may be pre-processed (indexing, change in format, etc.) and is ready for querying. In case services are not added incrementally, then time for pre-processing a service repository is a one-time effort that takes considerable amount of time, but gets amortized over a large number of queries.

Incremental updates Adding or updating a service to an existing repository of services should take minimal time. An ideal discovery and composition engine should not pre-process the entire repository again; rather incrementally update pre-processed data (indexes, etc.) with data for the new service.

Cost function If there are costs associated with every service in the repository, then an ideal discovery and composition engine should be able to provide results based on requirements (minimize, maximize, etc.) over the costs. We can extend this to services having an associated attribute vector, and the engine should be able to provide results based on maximizing or minimizing functions over the attribute vector. These requirements have driven the design of our semantics-based discovery and composition engine described in this paper.

3.4 Centrality measure in social networks

Social Network Analysis focuses on the structure of relationships ranging from casual acquaintance to close bonds. It involves measuring the formal and informal relationships to understand information/knowledge flow that binds the interacting units that could be a person, group, organization, or any knowledge entity. Social Network Analysis has an increasing application in social sciences that has been applied to diverse areas such as psychology, health, electronic communications, and business organization. In order to understand social networks and their participants, the location of an actor in a network is evaluated. The network location is measured in terms of centrality of a node that gives an insight into the various roles and groupings in a network. Centrality gives a rough indication of the social power of a node based on how well they "connect" the network. There has been extensive discussion in the Social Network community regarding the meaning of the term centrality when it is applied to Social Networks. One view stems directly from graph theory [9]. The graph-theoretic conception of compactness has been extended to the study of Social Networks and simply renamed "graph centrality". Their measures are all based upon distances between points, and all define graphs as centralized to the degree that their points are all close together. The alternative view emerged from substantive research on communication in Social Networks. From this perspective, the centrality of an entire network should index the tendency of a single point to be more central than all other points in the network. Measures of a graph centrality of this type are based on differences between the centrality of the most central point and that of all others. Thus, they are indexes of the centralization of the network [36]. The three most popular individual centrality measures are *Degree*, *Betweenness*, and *Closeness* Centrality.

- *Degree centrality* The network activity of a node can be measured using the concept of degrees, i.e., the number of direct connections a node has. In the example, network shown in Fig. 10 and Table 5, Provider D has the most direct connections in the network, making it the most active node in the network. In personal Social Networks, the common thought is that "the more connections, the better".
- *Betweenness centrality* Though Provider D has many direct ties, Provider H has fewer direct connections (close to the average in the network). Yet, in many ways, Provider H has one of the best locations in the network by playing the role of a "broker" between two important components. A node with high betweenness has greater influence over what flows and does not in the network.
- *Closeness centrality* Provider F and G have fewer connections than Provider D, yet the pattern of their direct and indirect ties allow them to access all the nodes in the network more quickly than anyone else. They have the shortest paths to all others, i.e., they are close to everyone else. They are in an excellent position and have the best visibility into what is happening in the network.

Table 5 Degree centrality of nodes in Fig. 10

Service provider	Degree
Provider A	2
Provider B	3
Provider C	1
Provider D	8
Provider E	3
Provider F	4
Provider G	5
Provider H	3
Provider I	1
Provider J	2
Provider K	4
Provider L	1
Provider M	

Individual network centralities provide insight into the individual's location in the network. The relationship between the centralities of all nodes can reveal much about the overall network structure.

3.5 Trust rating of a service and trust threshold

The trust rating of each service in the repository is computed as a measure of the degree centrality (**CD**) of the social network to which the service provider belongs. It is calculated as the degree or count of the number of adjacencies for a node, s_k:

$$C_D(s_k) = \sum_{i-0}^{n} a(s_i, s_k)$$

where

$a(s_i s_k) = 1$ iff s_i and s_k are connected by a line
0 otherwise

As such it is a straightforward index of the extent to which s_k is a focus of activity [9]. $C_D(s_k)$ is large if service provider s_k is adjacent to, or in direct contact with, a large number of other service providers, and small if s_k tends to be cut off from such direct contact. $C_D(s_k) = 0$ for a service provider that is totally isolated from any other point. Our algorithm filters out any services whose provider has a zero degree centrality in a social network, i.e., such services will not be used in building composition solutions. Trust rating of the entire composite service is computed as an average of the individual trust ratings of the services involved in the composition. We also need to set a Trust Threshold and any service with a Trust rating that is below this threshold is not used while generating composition solutions. In our initial prototype implementation, we set the Trust threshold to zero, i.e., degree centrality of the service provider in the network is zero. A service provider or service provider organization that is not connected to any other nodes in the Social network is not known to anyone else and is an immediate reason to be pruned out from composition solutions, as the service cannot be trusted. Composition solutions can be ranked such that solutions with highest trust rating appear on top of the list.

4 Dynamic Web service composition: methodology

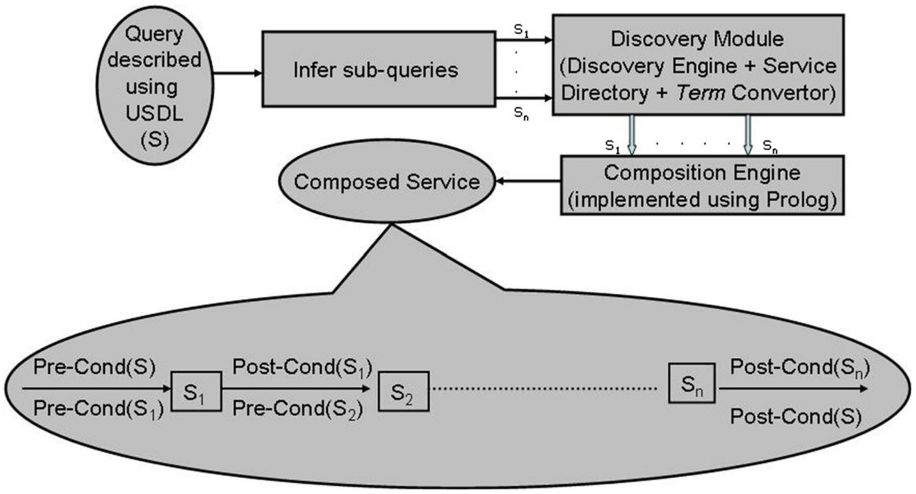
In this section, we describe our methodology for automatic Web service composition that produces a general directed acyclic graph. The composition solution produced is for the generalized composition problem presented in Sect. 3. We also present our algorithm for automatic generation of OWL-S descriptions for the new composite service produced.

4.1 Algorithms for Web service discovery and composition

Our approach is based on a multi-step narrowing of the list of candidate services using various constraints at each step. As mentioned earlier, discovery is a simple case of Composition. When the number of services involved in the composition is exactly equal to one, the problem reduces to a discovery problem. Hence, we use the same engine for both discovery and composition. We assume that a directory of services has already been compiled, and that this directory includes semantic descriptions for each service. In our implementation, we use semantic descriptions written in USDL [37], although the algorithms are general enough that they will work with any semantic annotation language. The repository of services contains one USDL description document for each service. However, we still need a query language to search this directory, i.e., we need a language to frame the requirements of the service that an application developer is seeking. USDL itself can be used as such as a query language. A USDL description of the desired service can be written (with tool assistance), a query processor can then search the service directory for a “matching” service. For service composition, the first step is finding the set of composable services. USDL itself is used to specify the requirements of the composed service that an application developer is seeking. Using the discovery engine, individual services that make up the composed service can be selected. Part substitution techniques [38] can be used to find the different parts of a whole task and the selected services can be composed into one by applying the correct sequence of their execution. The correct sequence of execution can be determined by the pre-conditions and post-conditions of the individual services. That is, if a subservice S_1 is composed with subservice S_2, then the post-conditions of S_1 must imply the pre-conditions of S_2. The goal is to derive a single solution, which is a directed acyclic graph of services that can be composed together to produce the requested service in the query. Figure 8 shows a pictorial representation of our composition engine.

4.2 Multi-step narrowing solution

To produce a composite service, as shown in the example Fig. 2, our algorithm filters services that are not useful for the composition at multiple stages. Figure 9 shows the filtering technique for the particular instance graph represented in Fig. 2. The composition routine begins with the query input parameters and finds all those services from the repository that require a subset of the query input parameters. In Fig. 9, CI, I are the pre-conditions and the input parameters provided by the query. S_1 and S_2 are the services found after step 1. O_1 is the union of all outputs produced by the services at the first stage. For the next stage, the inputs available are the query input parameters and all the outputs produced by the previous stage, i.e., $I_2 = O_1 \cup$ I. I_2 is used to find services at the next stage, i.e., all those services that require a subset of I_2. To make sure we do not end up in cycles, we get only those services that require at least one parameter from the outputs produced in the previous stage. This filtering continues until all the query output parameters are produced. At this point, we make another pass in the reverse direction to remove redundant services that do not directly or indirectly contribute to the query output parameters. This is done starting with the output parameters and working our way backwards. Next, another level of filtering is performed using the trust ratings of services. Table 6 shows the algorithm and we have prototype implementation of this algorithm implemented using Prolog [39] with Constraint Logic Programming over finite domain (CLP(FD)) [40]. The rationale behind the choice of



Query described using USDL (S)
Infer sub-queries
S1
Sn
Discovery Module (Discovery Engine + Service Directory + *Term* Convertor)
S1
Sn
Composition Engine (implemented using Prolog)
Composed Service
Pre-Cond(S)
Pre-Cond(S1)
S1
Post-Cond(S1)
Pre-Cond(S2)
S2
Sn
Post-Cond(Sn)
Post-Cond(S)

Fig. 8 Composition engine—design

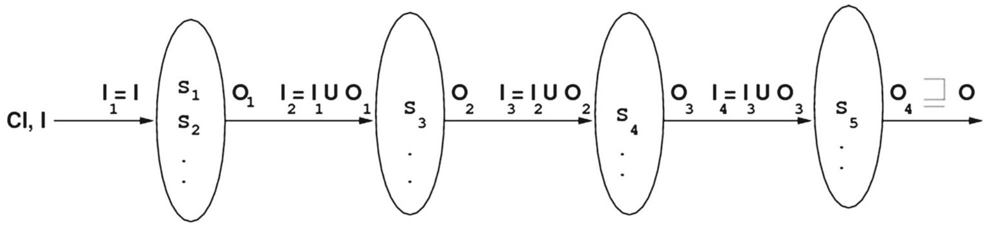


Fig. 9 Multi-step narrowing solution

Table 6 Algorithm for multi-step narrowing

Multi-step Narrowing Algorithm
Algorithm: Composition
(Input QI - QueryInputs, QO - QueryOutputs, QCI - Pre-Cond, QCO - Post-Cond, T - TrustThreshold)
(Output: Result - ListOfServices)
1. L $\Leftarrow$ NarrowServiceList(QI, QCI);
2. O $\Leftarrow$ GetAllOutputParameters(L);
3. CO $\Leftarrow$ GetAllPostConditions(L);
4. While Not (O $\sqsupseteq$ QO)
5. I = QI $\cup$ O; CI $\Leftarrow$ QCI $\wedge$ CO;
6. L$'$ $\Leftarrow$ NarrowServiceList(I, CI);
7. End While;
8. IntResult $\Leftarrow$ RemoveRedundantServices(QO, QCO);
9. Result $\Leftarrow$ RemoveRedundantServices(T, IntResult);
10. Return Result;

CLP(FD), implementation details, and experimental results are available [15].

4.3 Automatic generation of OWL-S descriptions

After obtaining a composition solution (sequential, non-sequential, or conditional), the next step is to produce a semantic description document for this new composite service. This document can be used for execution of the service and to register the service in the repository, thereby allowing subsequent queries to result in a direct match instead of performing the composition process all over again. We used the existing language OWL-S [31] to describe composite services. OWL-S models services as processes and when used to describe composite services, it maintains the state throughout the process. It provides control constructs such as *Sequence*, *Split-Join*, *If-Then-Else* and many more to describe composite services. These control constructs can be used to describe

Table 7 Generation of composite service description

```
Generation of Composite Service Description

Algorithm: GenerateCompositeServiceDescription
(Input: G - CompositionSolutionGraph)
(Output: D - CompositeServiceDescription)
1. Generate generic header constructs
2. Start Composite Service element
3. Start SequenceConstruct
4. If Number(SourceVertices) = 1
     GenerateAtomicService
    Else StartSplitJoinConstruct
     For Each starting/source Vertex V
        GenerateAtomicService
     End For
     EndSplitJoinConstruct
   End If
5. If Number(SinkVertices) = 1
     GenerateAtomicService
    Else StartSplitJoinConstruct
     For Each ending/sink Vertex V
        GenerateAtomicService
     End For
     EndSplitJoinConstruct
   End If
6. For Each remaining vertex V in G
     If V is AND vertex with one outgoing edge
        GenerateAtomicService
     If V is AND vertex with > 1 outgoing edge
        GenerateSplitJoinConstruct
     If V is OR vertex with one outgoing edge
        GenerateAtomicService
qquad If V is OR vertex with > 1 outgoing edge
        GenerateConditionalConstruct
   End For
7. End SequenceConstruct
8. End Composite Service element
9. Generate generic footer constructs
```

the kind of composition. OWL-S also provides a property called *composedBy* using which the services involved in the composition can be specified. Table 7 shows the algorithm for generation of the OWL-S document when the composition solution in the form of a graph is provided as the input.

A sequential composition can be described using the *Sequence* construct that indicates that all the services inside this construct have to be invoked one after the other in the same order. The non-sequential composition can be described in OWL-S using the *Split-Join* construct which indicates that all the services inside this construct can be invoked concurrently. The process completes execution only when all the services in this construct have completed their execution. The non-sequential conditional composition can be described in OWL-S using the *If-Then-Else* construct which specifies the condition and the services that should be executed if the condition holds and also specifies what happens when the condition does not hold. Conditions in OWL-S are described using SWRL. There are other constructs such as looping constructs in OWL-S that can be used to describe composite services with complex looping process flows. We are currently investigating other kinds of compositions with iterations and repeat-until loops and their OWL-S document generation. We are exploring the possibility of unfolding a loop into a linear chain of services that are repeatedly executed. We are also analyzing our choice of the composition language and looking at other possibilities as part of our future work.

5 Implementation and experimental results

This section presents implementation details of the composition engine. We also analyze the performance and present experimental results.

5.1 Implementation

Our discovery and composition engine is implemented using Prolog [39] with Constraint Logic Programming over finite domain [40], referred to as CLP(FD) hereafter. In our current implementation, we used semantic descriptions written in the language called Universal Semantics-Service Description Language (USDL) [37]. The repository of services contains one USDL description document for each service. USDL itself is used to specify the requirements of the service that an application developer is seeking. USDL is a language that service developers can use to specify formal semantics of Web services. In order to provide semantic descriptions of services, we need an ontology that is somewhat coarse-grained yet universal, and at a similar conceptual level to common real-world concepts. USDL uses WordNet [41] which is a sufficiently comprehensive ontology that meets these criteria. Thus, the "meaning" of input parameters, outputs, and the side effect induced by the service is given by mapping these syntactic terms to *concepts* in WordNet [38] for details of the representation. Inclusion of USDL descriptions thus makes services directly "semantically" searchable. However, we still need a query language to search this directory, i.e., we need a language to frame the requirements on the service that an application developer is seeking. USDL itself can be used as such a query language. A USDL description of the desired service can be written, a query processor can then search the service directory for a "matching" service. These algorithms can be used with any other Semantic Web service descrip-

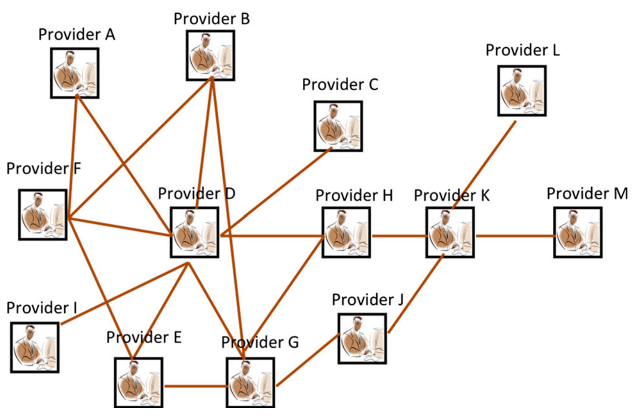


Fig. 10 A social network of Web service providers

tion language as well. It will involve extending our implementation to work for other description formats, and we are looking into that as part of our future work. The parsing of all the USDL description documents and the universal ontology is written in Java. The parsing is done via SAXReader library of Java and after the parsing, prolog engine is instantiated to run the Composition query processor. The complete discovery and composition engine is implemented as a Web service in Java using Apache Tomcat. The Web service in turn invokes Prolog to do all the processing [42–44]. The high-level design of the Discovery and Composition engines is shown in Fig. 10. The software system is made up of the following components:

(i) **Triple generator**

The triple generator module converts each service description into a triple. In this case, USDL descriptions are converted to triples like:

(Pre-Conditions, affect-type(affected-object, I, O), Post-Conditions)

The function symbol *affect-type* is the side effect of the service and *affected object*} is the object that changed due to the side effect. *I* is the list of inputs, and *O* is the list of outputs. *Pre-Conditions* are the conditions on the input parameters, and *Post-Conditions* are the conditions on the output parameters. Services are converted to triples so that they can be treated as terms in first-order logic and specialized unification algorithms can be applied to obtain exact, generic, specific, part and whole substitutions [38]. In case conditions on a service are not provided, the *Pre-Conditions* and *Post-Conditions* in the triple will be null. Similarly, if the affect-type is not available, this module assigns a generic affect to the service.

(ii) **Query reader**

This module reads the query file and passes it on to the Triple Generator. We use USDL itself as the query language. A USDL description of the desired service can be written, which is read by the query reader and converted to a triple. This module can be easily extended to read descriptions written in other languages.

(iii) **Semantic relations generator**

We obtain the semantic relations from the OWL WordNet ontology. OWL WordNet ontology provides a number of useful semantic relations like synonyms, antonyms, hyponyms, hypernyms, meronyms, holonyms and many more. USDL descriptions point to OWL WordNet for the meanings of concepts. A theory of service substitution is described in detail in [38] which uses the semantic relations between basic concepts of WordNet to derive the semantic relations between services. This module extracts all the semantic relations and creates a list of Prolog facts. We can also use any other domain-specific ontology to obtain semantic relations of concepts. We are currently looking into making the parser in this module more generic to handle any other ontology written in OWL.

(iv) **Discovery query processor**

This module compares the discovery query with all the services in the repository. The processor works as follows:

1. On the output parameters of a service, the processor first looks for an *exact* substitutable. If it does not find one, then it looks for a parameter with hyponym relation [38], i.e., a *specific* substitutable.
2. On the input parameters of a service, the processor first looks for an *exact* substitutable. If it does not find one, then it looks for a parameter with hypernym relation [38], i.e., a *generic* substitutable.

The discovery engine, written using Prolog with CLP(FD) library, uses a repository of facts, which contains a list of all services, their input and output parameters and semantic relations between parameters. The code snippet of our engine is shown in Table 8. The query is parsed and converted into a Prolog query that looks as follows:

discovery(sol(queryService, ListOfSolutionServices).

The engine will try to find a list of *SolutionServices* that match the *queryService*.

(v) **Composition engine**

The composition engine is written using Prolog with CLP(FD) library. It uses a repository of facts, which contains all the services, their input and output parameters and the semantic

Table 8 Discovery Algorithm—Code Snippet

Discovery Algorithm
discovery(sol(Qname,A)) :-
dQuery(Qname,I,O), encodeParam(O,OL),
/* Narrow candidate services(S) using output list(OL) */
narrowO(OL,S), fdset(S,FDs), fdsettolist(FDs,SL),
/* Expand InputList(I) using semantic relations */
getExtInpList(I, ExtInpList), encodeParam(ExtInpList,IL),
/* Narrow candidate services(SL) using input list (IL) */
narrowI(IL,SL,SA), decodeS(SA,A).

Table 9 Composition Algorithm—Code Snippet

Composition Algorithm
composition(sol(Qname, Result)) :-
dQuery(Qname, QueryInputs, QueryOutputs),
encodeParam(QueryOutputs, QO),
getExtInpList(QueryInputs, InpList),
encodeParam(InpList, QI),
performForwardTask(QI, QO, LF),
performBackwardTask(LF, QO, LR),
getMinSolution(LR, QI, QO, A), reverse(A, RevA),
confirmSolution(RevA, QI, QO), decodeSL(RevA, Result).

relations between the parameters. A code snippet of our composition engine is shown in Table 9. The query is converted into a Prolog query that looks as follows:

composition(queryService, ListOfServices).

The engine will try to find a *ListOfServices* that can be composed into the requested *queryService*. Our engine uses the built-in, higher order predicate "bagof" to return all possible *ListOfServices* that can be composed to get the requested *queryService*.

(vi) **Output generator**

After the Composition engine finds a matching service, or the list of atomic services for a composed service, the results are sent to the output generator in the form of triples. This module generates the output files in any desired XML format.

5.2 Efficiency and scalability issues

In this section, we discuss the salient features of our system with respect to the efficiency and scalability issues related to Web service discovery and composition problem. It is because of these features that we decided on the multi-step narrowing-based approach to solve these problems and implemented it using constraint logic programming.

(i) **Correctness**

Our system takes into account all the services that can be satisfied by the provided input parameters and pre-conditions at every step of our narrowing algorithm. So our search space has all the possible solutions. Our backward narrowing step, which removes the redundant services, does so taking into account the output parameters and post-conditions. So our algorithm will always find a correct solution (if one exists) in the minimum possible number of steps.

(ii) **Pre-processing**

Our system initially pre-processes the repository and converts all service descriptions into Prolog terms. The semantic relations are also processed and loaded as Prolog terms in memory. Once the pre-processing is done, then discovery or composition queries are run against all these Prolog terms and, hence, we obtain results quickly and efficiently. The built-in indexing scheme and constraints in CLP (FD) facilitate the fast execution of queries. During the pre-processing phase, we use the term representations of services to set up constraints on services and the individual input and output parameters. This further helped us in getting optimal results.

(iii) **Execution efficiency**

The use of CLP (FD) helped significantly in rapidly obtaining answers to the discovery and composition queries. We tabulated processing times for different size repositories, and the results are shown in the next section. As one can see, after pre-processing the repository, our system is quite efficient in processing the query. The query execution time is insignificant.

(iv) **Programming efficiency**

The use of Constraint Logic Programming helped us in coming up with a simple and elegant code. We used a number of built-in features such as indexing, set operations, and constraints and, hence, did not have to spend time coding these ourselves. This made our approach efficient in terms of programming time as well. Not only the whole system is about 200 lines of code, but we also managed to develop it in less than 2 weeks.

(v) **Scalability**

Our system allows for incremental updates on the repository, i.e., once the pre-processing of a repository is done, adding a new service or updating an existing one will not need re-execution of the entire pre-processing phase. Instead, we can easily update the existing list of CLP (FD) terms loaded in the memory and run discovery and composition queries. Our estimate is that this update time will be negligible, perhaps a few milliseconds. With real-world services, it is likely that new services will get added often or updates might be made on existing services. In such a case, avoiding repeated pre-processing of the entire repository will definitely be needed and incremental updates will be of great practical use. The efficiency of the incremental update operation makes our system highly scalable.

(vi) **Use of external database**

In case the repository grows extremely large in size, then saving off results from the pre-processing phase into some external database might be useful. This is part of our future work. With extremely large repositories, holding all the results of pre-processing in the main memory may not be feasible. In such a case, we can query a database where all the information is stored. Applying incremental updates to the database is easily possible, thus avoiding recomputation of pre-processed data.

(vii) **Searching for optimal solution**

If there are any properties with respect to which the solutions can be ranked, then setting up global constraints to get the optimal solution is relatively easy with the constraint-based approach. For example, if each service has an associated cost, then the discovery and the composition problem can be redefined to find the solutions with the minimal cost. Our system can be easily extended to take these global constraints into account.

5.3 Performance and experimental results

To conduct our experiments, we looked at various benchmarks and Web services challenge (WSC) datasets [42,43] best suited our needs and fit well into the overall architecture with minimal changes. They provided semantics through XML schema, provided queries and corresponding solutions. We used repositories from WSC website [42,43], slightly modified to fit into USDL framework. They provide repositories of various sizes (thousands of services). These repositories contain WSDL descriptions of services. The input and output messages of the services may contain multiple parameters. Each parameter is annotated with a semantic concept stored in the attribute *type*. Table 10 shows a service *AddressConverter* with one operation named *Convert*. It can be invoked with an input message (*InputName*) and produces a response message (*OutputAddress*). The value of the attribute message represents a reference to a message element. Each message has a set of part elements as children, which represent the service parameters, annotated with concepts referenced by the *type*-attribute. The *Convert* operation in this example requires a parameter of the type *Name* and returns an instance of *US-Address*.

Table 10 Sample service interface description

Sample WSDL description

```
<message name ="InputName">
    <part name = "part0" type = "Name"/>
</message >
<message name ="OutputAddress">
    <part name = "part0" type = "US-Address" />
</message>
<portType name = "AdressConverter">
  <operation name ="Convert" >
    <input message ="InputName" />
    <output message = "OutputAddress"/>
  </operation >
</ portType >
```

The queries and solutions are provided in an XML format. The semantic relations between various parameters are provided in an XML Schema format. Concepts were treated as data types and taxonomies encoded as hierarchies of such data types in XSD schemas. The subsumes relation between two semantic concepts can be compared to the subclass relationship in Object-oriented programming. Table 11 shows a sample XSD schema defining the data types *Address* and *US-Address* inheriting from *Address*. In the context of the WSC, this schema would be interpreted as a taxonomy introducing the concepts Address and US-Address with *subsumes(Address, US-Address)*.

We evaluated our approach on different size repositories and tabulated Pre-processing, Query Execution, and Incremental update time. We noticed that there was a significant difference in pre-processing time between the first and subsequent runs (after deleting all the previous pre-processed data) on the same repository. What we found is that the repository was cached after the first run and that explained the difference in the pre-processing time for subsequent runs. Table 12 shows performance results of our Composition algorithm on discovery queries, and Table 13 shows the results of our algorithm on composition queries. The times shown in tables are the wall clock times. The actual CPU time to pre-process the repository and execute the query should be less than or equal to the wall clock time. The experiments were conducted on different repository sizes as well as varying number of input and output parameters for each service in the repository. The results are plotted in Figs. 11 and 12 where

Table 11 Sample XSD Schema providing semantics

Sample XSD Schema

```
<complexType name =“ Address ” >
  <sequence >
    <element name = “name” type = “string” minOccurs = “0”/>
    <element name = “street” type = “string” />
    <element name = “city” type = “string” />
  </ sequence>
</complexType>
<complexType name = “US - Address”>
  <complexContent >
    <extension base = “Address” >
      <sequence>
       <element name = “state” type = “US - State”/>
       <element name = “zip” type = “positiveInteger”/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

Table 12 Performance on discovery queries

Repository size	Number of I/O parameters	Pre-processing time (ms)	Query execution time (ms)	Incremental update (ms)
2,000	4–8	36.5	1	18
2,000	16–20	45.8	1	23
2,000	32–36	57.8	2	28
2,500	4–8	47.7	1	19
2,500	16–20	58.7	1	23
2,500	32–36	71.6	2	29
3,000	4–8	56.8	1	19
3,000	16–20	77.1	1	26
3,000	32–36	88.2	3	29

Table 13 Performance on composition queries

Repository size	Number of I/O parameters	Pre-processing I/O time (ms)	Query execution time (ms)	Incremental update (ms)
2,000	4–8	36.1	1	18
2,000	16–20	47.1	1	23
2,000	32–36	60.2	1	30
3,000	4–8	58.4	1	19
3,000	16–20	60.1	1	20
3,000	32–36	102.1	1	34
4,000	4–8	71.2	1	18
4,000	16–20	87.9	1	22
4,000	32–36	129.2	1	32

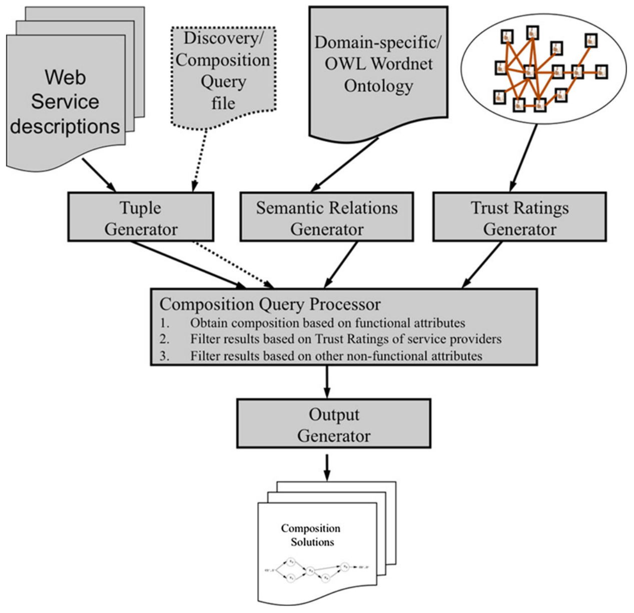

Web Service descriptions
Discovery/ Composition Query file
Domain-specific/ OWL Wordnet Ontology
Tuple Generator
Semantic Relations Generator
Trust Ratings Generator
Composition Query Processor
1. Obtain composition based on functional attributes
2. Filter results based on Trust Ratings of service providers
3. Filter results based on other non-functional attributes
Output Generator
Composition Solutions

Fig. 11 High-level design on composition engine

the numbers of I/O parameters were 4–8, 16–20, and 32–36, respectively. The graphs exhibit behavior consistent with our expectations: for a fixed repository size, the pre-processing time increases with the increase in number of input/output parameters. Similarly, for fixed input/output sizes, the pre-processing time is directly proportional to the size of the repository. However, what is surprising is the efficiency of service query processing, which is negligible (just 1–3 ms) even for complex queries with large repositories.

Discussion

We evaluated our approach for correctness, efficiency, and scalability of the software. The correctness of the algorithm has been described in Sect. 4. In the experiments conducted, discovery and composition solutions obtained were checked against a pre-defined set of solutions produced for the WS-Challenge datasets [43]. This was the first and most important criterion for evaluation of the engine. Our second criterion was query efficiency, i.e., rapidly obtaining answers to discovery and composition queries. The results show that irrespective of the repository size the query execution time was always 1–2 ms. This was because the repositories were pre-processed, and the queries were run against pre-processed data. Different repository sizes were used along with varying the number of input and output parameters of the Web services. The software also scaled well with varying the size of the repositories. These three criterion are important for the successful adoption of the composition engine to produce a Software-as-a-Service (SaaS) platform for automatic workflow generation in a specific domain as described in the sample scenario in Sect. 1. An important part of scalability is the ability to sustained performance even if pre-processed

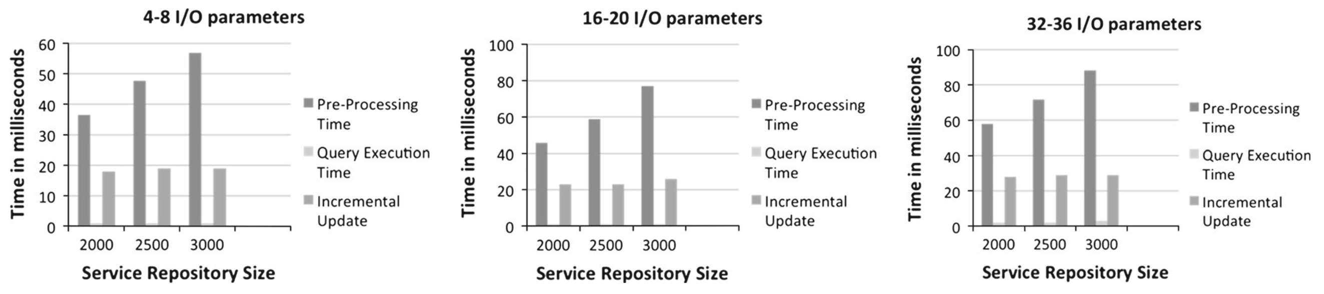


Fig. 12 Performance on discovery queries

repositories change in size, new services are added, existing services are removed, etc. To evaluate this aspect, we studied the time taken for incremental updates to the repository. The results obtained were all less than one second for different repository sizes and well as varying number of input and output parameters. A network of service providers was introduced synthetically into the experimental datasets. Trust rating of services was computed based on this network. Trust ratings generator helps with filtering services based on the trust threshold and helps in ranking. The performance of the engine on discovery and composition queries still remains the same.

6 Application to bioinformatics

We illustrate the practicality of our general framework for automatically composing services by applying it to *phylogenetics*, a subfield of bioinformatics, for automatic generation of *workflows*. In this section, we present a brief description of the field of Phylogenetics [8] followed by an example of a workflow generation problem that can be mapped to a non-sequential conditional composition problem (the most general case of the composition problem) and can be solved using our generalized composition engine.

6.1 Phylogenetics

Phylogenetic inferencing involves an attempt to estimate the evolutionary history of a collection of organisms (taxa) or a family of genes [45]. The two major components of this task are the estimation of the evolutionary tree (branching order), then using the estimated trees (phylogenies) as analytical framework for further evolutionary study and finally performing the traditional role of systematics and classification. Using this study, a number of interesting facts can be discovered, for example, who are the closest living relatives of humans, who are whales related to, etc. Different studies can be conducted, for example, studying dynamics of microbial communities, predicting evolution of influenza viruses and other applications such as Drug Discovery, and vaccine development, etc. In order to perform these tasks, scientists use a number of software tools and programs already available. They use them by putting them together in a particular order (i.e., a workflow) to get their desired results. That is, it involves execution of a sequence of steps using various programs or software tools. These tools use different data formats, and hence translating from one data format to another becomes necessary.

6.2 Automatic workflow generation

The software tools and programs created for specific phylogenetic tasks use different data formats for their input and output parameters. The data description language Nexus [46,47] is used as a universal language for representation of these bioinformatics related data. There are translator programs that convert different formats into Nexus and vice versa. For example, one could use the *BLAST* program to get a sequence set of genes. Once the sequence set is obtained, the sequences can be aligned using the *CLUSTAL* program. But the output from *BLAST* cannot be directly fed to *CLUSTAL*, as their data formats are different. The translator can be used to convert the *BLAST* format to Nexus and then the Nexus format to *CLUSTAL*. In order to perform an inferencing task, one has to manually pick all the appropriate programs and corresponding format translators and put them in the correct order to produce a workflow. We show how Web service composition can be directly applied to automate this task of producing a workflow. *MyGrid* [48] has a wealth of bioinformatics resources and data that provides opportunities for research. It has hundreds of biological Web services and their WSDL descriptions, provided by various research groups around the world. We illustrate our generalized framework for Web service composition by applying it to these services to generate workflows automatically that are practically useful for this field.

Example Workflow Generation (Non-sequential Composition) Suppose we are looking for a service that takes a *GeneInput* and produces its corresponding *AccessionNumbers*, *AGI*, and *GeneIdentifier* as output. The directory of services contains *CreateMobyData*, *MOBYSHoundGetGen-*

Table 14 Bioinformatics application—non-sequential composition example

Service	Pre-conditions	Input parameters	Output parameters	Post-conditions
Query		GeneInput	AccessionN umbers, AGIj GeneIdentity	
CreateMoby Data		GeneInput	MobyData	Format (Moby-Data) = NCBI
MOBYSHoundGet GenBankWbatev erSequence	Format (MobyData) = NCBI	MobyData	GeneSequence	
MIPSBlastBetterE13		GeneSequence	WUGene Sequence	
Extract Accession		GeneSequence	AccessionNumbers	
ExtractBestHit		WUGeneSequence	AGI, GeneIdentity	

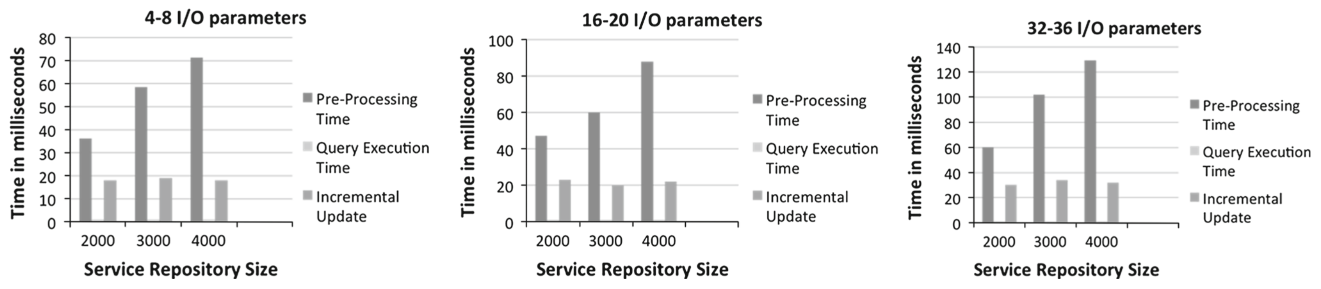


Fig. 13 Performance on composition queries

Bank WhateverSequence, *ExtractAccession, ExtractBestHit, MIPSBlastBetterE13* services. In this scenario, the *GeneInput* first needs to be converted to NCBI data format and then its corresponding *GeneSequence* is further passed to *ExtractAccession* and *ExtractBestHit* to obtain the *AccessionNumbers*, *AGI*, and *GeneIdentity* respectively. Table 14 shows the input/output parameters of the user query and the services. Figure 13 shows this non-sequential composition example as a directed acyclic graph. In this example:

- Service *CreateMobyData* has a post-condition on its output parameter *MobyData* that the format is NCBI and service *MOBYSHoundGetGenBankWhateverSequence* has a pre-condition that its input parameter *MobyData* has to be in NCBI format for service execution. The post-condition of *CreateMobyData* must imply pre-condition of *MOBYSHoundGetGenBankWhateverSequence service*.
- Both services *ExtractAccession* and *ExtractBestHit* have to be executed to obtain the query outputs.
- The semantic descriptions of the service input/output parameters should be the same as the query parameters or have the subsumption relation. This can be inferred using semantics from the ontology provided.

Example Workflow Generation (Non-sequential Conditional Composition) Suppose we are looking for a service that takes a *GeneSequence* and produces an *EvolutionTree* and *EvolutionDistance* after performing a phylogenetic analysis. Also the service should satisfy the post-condition that the *EvolutionTree* produced is in the *Newick* format. This involves producing a sequence set first, followed by aligning the sequence set and then producing the evolution tree and evolution distance. Also any necessary intermediate data format translations have to be performed. Table 15 lists the services in the repository and their corresponding input/output parameters and user query. For the sake of simplicity, the query and services have fewer input/output parameters than the real-world services. In this example, service *BLAST* has to be executed first so that its output *BLASTSequenceSet* can be used as input by *CLUSTAL* after the data format has been translated using *BLASTNexus* and *NexusCLUSTAL*. The service BLASTNexus has a post-condition that the format of the output parameter *NexusSequenceSet* is Nexus which is the pre-condition of the next service *NexusCLUSTAL*. Similarly the service *NexusCLUSTAL* has a post-condition that the format of the output parameter *ClustalSequenceSet* is Clustal that is the pre-condition of the next service *CLUSTAL*. At every step of composition, the post-conditions of a service should imply the pre-conditions of the following service. The post-condition of the service CLUSTAL is that the output parameter *AlignedSequenceSet* has either Paup or Phylip format. Depending on which one of these two conditions hold,

Table 15 Bioinformatics application—non-sequential conditional composition example

Service	Pre-conditions	Input parameter	Outpur parameter	Post-conditions
Query		Sequence, OrganismType, Word Size, DatabaseName	EvolutionTree, EvolutionDistance	Format (EvolutionTree) = Newick
BLAST		Sequence, Organism Type, DatabaseName	BlastSequenceSet	
CLUSTAL	Format (Clustal SequenceSet) = Clustal	Clustal SequenceSet	AlignedSequenceSet	Format (AlignedSequence Set) = Paup ∨ Forrmat(AlignedSequence Set) = Phylip
BLASTNexus		BlastSequenceSet	NexusSequenceSet	Format (NexusSequenceSet) = Nexus
NexusCLUSTAL	Format (Nexus SequenceSet) = Nexus	NexusSeqtienceSet	ClustalSequenceSet	Forma(ClustalSequenceSet) = Clustal
PAUP	Format (Aligned SequenceSet) = Paup	AlignedSequenceSet, Word Size	EvolutionTree	Format (EvolutionTree) = Newick
PHYLIP	Format (Aligned SequenceSet) = Phylip	AlignetBeciuenceSet	EvokiSonTree	Format (EvolutionTree) = Newick
MEGA	Format (Aligned Sequenced) = Paup	AlignedSequenceSet	EvaluationDistance	

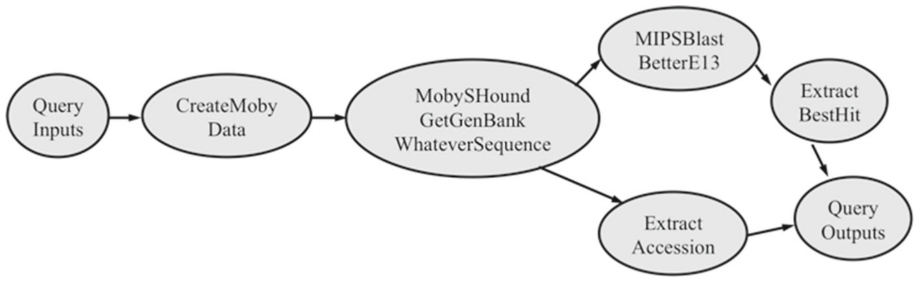


Fig. 14 Bioinformatics application—non sequential composition as a directed acyclic graph

the next service for composition is chosen. In this case, one cannot determine whether the post-conditions of the service *CLUSTAL* imply pre-conditions of *PAUP* or *PHYLIP* until the services are actually executed. In such a case, a condition can be generated which will be evaluated at runtime and depending on the outcome of the condition, corresponding services will be executed.

The vertex for service *CLUSTAL* in Fig. 14 has an outgoing edge to a conditional node. The outgoing edge represents the outputs and post-conditions of the service. The conditional node has multiple outgoing edges that represent the generated conditions that are evaluated at run-time. In this case, the following conditions are generated:

- (Format(AlignedSequenceSet) = Paup ∨ Format(AlignedSequenceSet) = Phylip) ⇒ (Format(AlignedSequenceSet) = Paup)
- (Format(AlignedSequenceSet) = Paup ∨ Format(AlignedSequenceSet) = Phylip) ⇒ (Format(AlignedSequenceSet) = Phylip)

Depending on the condition that holds, the corresponding services *PAUP* and *MEGA* or *PHYLIP* are executed respectively. The outputs *EvolutionTree* and *EvolutionDistance* are produced in both the cases along with the post-condition that the format of the evolution tree is Newick. Figure 14 shows this non-sequential conditional composition example as a conditional directed acyclic graph.

6.3 Implementation

In order to apply service composition to obtain the sequence of tasks automatically, these programs have to be made available as Web services and their descriptions should be provided using one of the Web services description languages like WSDL (Web Services Description Language) or USDL (Universal Service-Semantics Description Language). The translator programs also have to be available as Web services. Then, we can write a query specifying the input parameters provided and the output parameters that have to be obtained. The Composition engine then looks up the repository of available services and finds the solution, i.e., a set

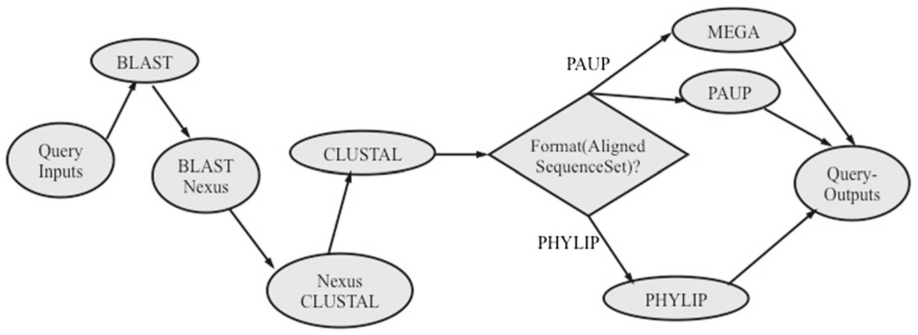


Fig. 15 Bioinformatics application—non sequential conditional composition as a directed acyclic graph

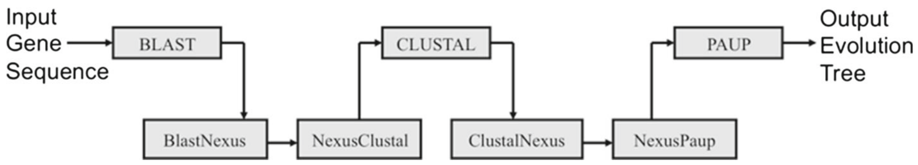


Fig. 16 Composition Solution 1 for query in Table 15

of services that can be executed to obtain the requested output parameters. These set of services obtained will need only those input parameters that are provided by the query and their execution produces the output parameters specified by the query. We tested our Composition engine on a repository of services that had descriptions of Web services corresponding to the following programs:

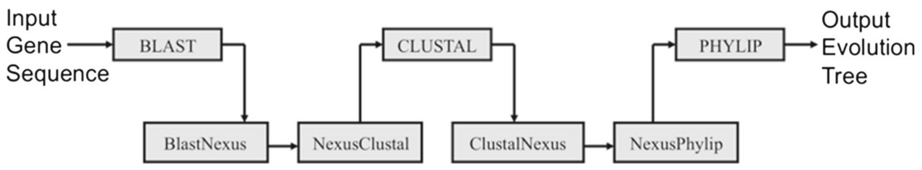
1. BLAST: This service compares protein sequences to sequence databases and calculates the statistical significance of matches. It query's a public database of generic information like GenBank and GSDB and produces a molecular sequence. It takes in *Sequence*, *DatabaseName*, *OrganismType*, *SelectionOptions*, *MaxTargetSequences*, *ExpectedThreshold*, *WordSize* as input parameters and produces *BlastSequenceSet* as the output.
2. CLUSTAL: This service produces multiple sequence alignment for DNA or proteins. It takes in *ClustalSequenceSet* as input parameter and produces *ClustalAlignedSequenceSet* as the output.
3. PHYLIP: This service is used for inferring phylogenies. It analyzes molecular sequences and infers phylogenetic information. It takes in *PhylipAlignedSequenceSet*, *UseThresholdParsimony*, *UseTransversionParsimony* as input parameters and produces *NewickEvolutionTree* as the output.
4. PAUP: This service is used for inferring phylogentic trees. It analyzes molecular sequences and infers phylogenetic information. It takes in *PaupAlignedSequenceSet* as input parameter and produces *NewickEvolutionTree* as the output.
5. BLASTNexus: This service takes input in BLAST format and converts it into Nexus format. It takes in *BLASTSequenceSet* as input and produces *NexusSequence Set*.

Table 16 Bioinformatics application—workflow query

Service	Input parameters	Output parameters
Query1	Sequence, DatabaseName, OrganismType, SelectionOptions, MaxTargetSequences, ExpectedThreshold, UserTransversion Parsimony, UseThresholdParsimony, WordSize	NewickEvolutionTree

6. NexusCLUSTAL: This service takes input in Nexus format and converts it into CLUSTAL format. It takes in *NexusSequenceSet* as input parameter and produces *CLUSTALSequenceSet* as the output.
7. CLUSTALNexus: This service takes input in CLUSTAL format and converts it into Nexus format. It takes in *CLUSTALAlignedSequenceSet* as input parameter and produces *NexusAlignedSequenceSet* as the output.
8. NexusPAUP: This service takes input in Nexus format and converts it into PAUP format. It takes in *NexusAlignedSequenceSet* as input parameter and produces *PaupAlignedSequenceSet* as the output.
9. NexusPHYLIP: This service takes input in Nexus format and converts it into PHYLIP format. It takes in *NexusAlignedSequenceSet* as input parameter and produces *PhylipAlignedSequenceSet*} as the output.
10. MEGA: This service is used for Molecular Evolutionary Genetics Analysis. It takes in *MEGAInp* as input parameter and produces *MEGASequence* as the output.

Fig. 17 Composition Solution 2 for query in Table 15



11. KEPLER: This service provides scientific workflows. It takes in *KEPLERData* as input parameter and produces *KEPLERSequence* as the output.

The composition engine can automatically discover a complex workflow based on the query requirements, from a large repository without having to analyze all the programs manually. Figure 15 shows the non-sequential conditional composition for query specified in Table 16 as a directed cyclic graph. Figures 16 and 17 show the solutions obtained for the query specified in Table 16.

A task that had to be performed manually by biologists whenever they had to make phylogenetic inferences can now be done automatically with Web services Composition. The programs have to be made available as Web services and their descriptions provided. Once we have a repository of such services, the composition engine can be used as shown above to automatically generate workflows.

7 Conclusions and future work

Due to the growing number of services on the Web, we need automatic and dynamic Web service composition in order to utilize and reuse existing services effectively. It is also important that the composition solutions obtained can be trusted. Our semantics-based approach uses semantic description of Web services to find substitutable and composite services that best match the desired service. Given semantic description of Web services, our engine produces optimal results (based on number of services in the composition). The composition flow is determined automatically without the need for any manual intervention. Our engine finds any sequential, non-sequential or non-sequential conditional composition that is possible for a given query and also automatically generates OWL-S description of the composite service. This OWL-S description can be used during the execution phase and subsequent searches for this composite service will yield a direct match. A trust rating is computed for every service in the repository based on the degree centrality of the service provider in a known social network. Currently, we are in the process of testing the trust-based dynamic Web service composition engine in a complete operational setting and running experiments to measure the quality of composition results obtained. We will also explore the other measures of centrality such as betweenness centrality and closeness centrality and analyze the possibility of using a combination of all three measures of centrality to compute trust rating of a service provider. We are able to apply many optimization techniques to our system so that it works efficiently even on large repositories. The strengths of this engine is the minimal query execution time that is achieved through pre-processing of repositories and incremental updates to the pre-processed data whenever a service is added, removed, or modified. Use of Constraint Logic Programming helped greatly in obtaining an efficient implementation of this system and made it easy to incorporate non-functional parameters for ranking of results. The limitations of the engine include trust aspect of the Web services involved in a composition solution. A model that provides a trust rating to services or service providers would improve confidence in the generated solutions. Also, when working with domains such as Bioinformatics where software systems involved in a workflow need to be converted into services, generation of semantics of the inputs and outputs of the Web services is a challenge. They have to be manually assigned semantics by a domain expert.

Our future work includes investigating other kinds of compositions with loops such as repeat-until and iterations and their OWL-S description generation. Analyzing the choice of the composition language (e.g., BioPerl [49] for phylogenetic workflows) and exploring other language possibilities is also part of our future work. We are also exploring combining technologies of automated service composition and *domain-specific languages* to develop a framework for problem solving and software engineering.

References

1. Castagna G, Gesbert N, Padovani L (2008) A theory of contracts for web services. ACM SIGPLAN Not 43:261–272
2. Bansal A, Patel K, Gupta G, Raghavachari B, Harris ED, Staves JC (2005) Towards intelligent services: a case study in chemical emergency response. In: IEEE International conference on web services (ICWS)
3. McIlraith SA, Son TC, Zeng H (2001) Semantic web services. IEEE Intell Syst 16(2):46–53
4. Mandell DJ, McIlraith SA (2003) Adapting BPEL4WS for the semantic web: the bottom-up approach to web service interoperation. In: The semantic web-ISWC. Springer 2003, pp 227–241
5. Paolucci M, Kawamura T, Payne TR, Sycara K (2002) Semantic matching of web services capabilities. In: The semantic Web–ISWC. Springer 2002, pp 333–347
6. Rao J, Dimitrov D, Hofmann P, Sadeh N (2006) A mixed initiative approach to semantic web service discovery and composition: SAP's guided procedures framework. In: International conference on web services. ICWS'06, 2006, pp 401–410

7. Cardoso J, Sheth AP (2006) Semantic web services, processes and applications. Springer, Berlin
8. Edwards AWF, Cavalli-Sforza LL (1964) Reconstruction of evolutionary trees, systematics association publication number 6, No. Phenetic and Phylogenetic Classification, pp 67–76
9. Freeman LC (1979) Centrality in social networks conceptual clarification. Social Netw 1(3):215–239
10. Wasserman SF (1994) Social network analysis: methods and applications. Cambridge University Press, Cambridge
11. Tamura K, Peterson D, Peterson N, Stecher G, Nei M, Kumar S (2011) MEGA5: molecular evolutionary genetics analysis using maximum likelihood, evolutionary distance, and maximum parsimony methods. Mol Biol Evol 28(10):2731–2739
12. Thompson JD, Higgins DG, Gibson TJ (1994) CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res 22(22):4673–4680
13. Edgar RC (2004) MUSCLE: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Res 32(5):1792–1797
14. Badr Y, Caplat G (2010) Software-as-a-service and versionology: towards innovative service differentiation. In: 24th IEEE international conference on advanced information networking and applications (AINA), 2010, pp 237–243
15. Kona S, Bansal A, Blake MB, Gupta G (2008) Generalized semantics-based service composition. In: IEEE international conference on web services (ICWS), pp 219–227
16. Rao J, Su X (2005) A survey of automated web service composition methods. In: Cardoso J, Sheth A (eds) Semantic web services and web process composition. Springer, Berlin, pp 43–54
17. Srivastava B, Koehler J (2003) Web service composition-current solutions and open problems. In: ICAPS 2003 workshop on planning for web services, vol 35, pp 28–35
18. McIlraith S, Son TC (2002) Adapting golog for composition of semantic web services. KR 2:482–493
19. Pistore M, Roberti P, Traverso P (2005) Process-level composition of executable web services: on-the-fly versus once-for-all composition. In: Gomez-Perez A, Euzenat J (eds) The semantic web: research and applications. Springer, Berlin, pp 62–77
20. Boustil A, Maamri R, Sahnoun Z (2013) A semantic selection approach for composite web services using OWL-DL and rules. Serv Oriented Comput Appl 8:1–18
21. Claro DB, Albers P, Hao JK (2005) Selecting web services for optimal composition. In ICWS international workshop on semantic and dynamic web processes, Orlando-USA
22. Suvee D, De Fraine B, Cibrán MA, Verheecke B, Joncheere N, Vanderperren W (2005) Evaluating FuseJ as a web service composition language. In: Third IEEE European conference on web services (ECOWS)
23. Dong W, Jiao L (2008) QoS-aware Web service composition based on SLA. In: Fourth international conference on natural computation (ICNC) vol 5, pp 247–251
24. Yan J, Kowalczyk R, Lin J, Chhetri MB, Goh SK, Zhang J (2007) Autonomous service level agreement negotiation for service composition provision. Future Gener Comput Syst 23(6):748–759
25. Wada H, Champrasert P, Suzuki J, Oba K (2008) Multiobjective optimization of SLA-aware service composition. In: IEEE congress on services-Part I, pp 368–375
26. Blake MB (2007) Decomposing composition: service-oriented software engineers. IEEE Softw 24(6):68–77
27. Zeng L, Benatallah B, Ngu AH, Dumas M, Kalagnanam J, Chang H (2004) QoS-aware middleware for web services composition. IEEE Trans Softw Eng 30(5):311–327
28. Cardoso J, Sheth A, Miller J, Arnold J, Kochut K (2004) Quality of service for workflows and web service processes. Web Semant 1(3):281–308
29. Zhao X, Shen LW, Peng X, Zhao W (2013) Finding preferred skyline solutions for SLA-constrained service composition. In: 2013 IEEE 20th international conference on web services (ICWS), pp 195–202
30. Feng Y, Ngan LD, Kanagasabai R (2013) Dynamic service composition with service-dependent QoS attributes. In: 2013 IEEE 20th international conference on web services (ICWS), pp 10–17
31. “OWL-S”. [Online]. http://www.w3.org/Submission/OWL-S/. (Accessed: 22-Jan-2014)
32. Wen S, Tang C, Li Q, Chiu DK, Liu A, Han X (2014) Probabilistic top-K dominating services composition with uncertain QoS. Serv Oriented Comput Appl 8(1):91–103
33. Immonen A, Pakkala D (2014) A survey of methods and approaches for reliable dynamic service compositions. Serv Oriented Comput Appl 8(2):129–158
34. Mehdi M, Bouguila N, Bentahar J (2013) A QoS-based trust approach for service selection and composition via Bayesian networks. In: 2013 IEEE 20th international conference on web services (ICWS), pp 211–218
35. Kuter U, Golbeck J (2009) Semantic web service composition in social environments. Springer, Berlin
36. Leavitt HJ (1951) Some effects of certain communication patterns on group performance. J Abnorm Soc Psychol 46(1):38
37. Bansal A, Kona S, Simon L, Mallya A, Gupta G, Hite TD (2005) A universal service-semantics description language. In: Third IEEE European conference on web services (ECOWS), pp 214–225
38. Kona S, Bansal A, Simon L, Mallya A, Gupta G (2009) USDL: a service-semantics description language for automatic service discovery and composition. Int J Web Serv Res 6(1):20–48
39. Sterling L, Shapiro EY, Warren DH (1986) The art of Prolog: advanced programming techniques. MIT Press, Cambridge
40. Marriott K, Stuckey PJ (1998) Programming with constraints: an introduction. MIT Press, Cambridge
41. “RDF/OWL Representation of WordNet”. [Online]. http://www.w3.org/TR/wordnet-rdf/. (Accessed: 23-Jan-2014)
42. Blake MB, Cheung W, Jaeger MC, Wombacher A (2006) WSC-06: the web service challenge. In: The 3rd IEEE international conference on E-commerce technology, 2006. The 8th IEEE international conference on enterprise computing, E-commerce, and E-services, pp 62–62
43. Blake MB, Cheung WKW, Jaeger MC, Wombacher A (2007) WSC-07: Evolving the web services challenge. In: The 9th IEEE international conference on E-commerce technology and the 4th IEEE international conference on enterprise computing, E-commerce, and E-services, 2007. CEC/EEE 2007, pp 505–508
44. Kona S, Bansal A, Gupta G, Hite D (2007) Automatic composition of semantic web services. In: International conference on web services (ICWS), vol 7, pp 150–158
45. Felsenstein J (2004) Inferring phylogenies, vol 2. Sinauer Associates, Sunderland
46. Iglesias JR, Gupta G, Pontelli E, Ranjan D, Milligan B (2001) Interoperability between bioinformatics tools: a logic programming approach. In: Practical aspects of declarative languages. Springer, Berlin, pp 153–168
47. Maddison DR, Swofford DL, Maddison WP (1997) NEXUS: an extensible file format for systematic information. Syst Biol 46(4):590–621
48. “myGrid”. [Online]. http://www.mygrid.org.uk/
49. “BioPerl”. [Online]. http://www.bioperl.org/wiki/Main_Page