Generalized Semantics-based Service Composition

Srividya Kona and Gopal Gupta

Department of Computer Science,
The University of Texas at Dallas
Richardson, TX 75083

Abstract. Web services and Service-oriented computing is being widely used and accepted. In order to effectively reuse existing services, we need to automatically compose Web services. The automatic composition techniques need to be semantics-based rather than syntax-based, since only semantics can accurately capture the task a service performs. In this paper we present general semantics-based techniques for automatic service composition for Web services. Our general method takes as input (i) a repository, R, of semantic descriptions of Web services, (ii) a semantic description, Q, of the service that is desired, and produces a composite service consisting of services taken from R that realizes Q. The composite service may combine services linearly as a chain or as a directed acyclic graph, where in the most general case the combination may be conditional. The composite service generated by our automatic composition method can be coded as an OWL-S document. We present details of our initial implementation along with performance results.

1 Introduction

Service-oriented computing is changing the way software applications are being designed, developed, delivered and consumed. A Web service is an autonomous, platform-independent program accessible over the web that may effect some action or change in the world (i.e., causes a side-effect). Examples of such side-effects include a web-base being updated because of a plane reservation made over the Internet, a device being controlled, etc. As automation increases, these services will be accessed directly by applications rather than by humans [7]. In this context, 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 side-effect(s) it may cause. 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 [1]. Several efforts are underway to build such an infrastructure [10–13].

Service Composition involves effectively combining and reusing independently developed component services. 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 [20]. 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 inorder 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.

Note that one must make clear distinction between automatic service composition and manual service composition. Services can be composed manually and the manually specified composition encoded as a program/document in a language such as OWL-S [2] or BPEL4WS [11]. *Automatic service composition can be thought of as generating this OWL-S or BPEL4WS description automatically*. Thus, languages such as OWL-S and BPEL4WS do not solve the automatic service composition problem—they merely provide notations for specifying service composition.

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. In our method, after a solution is obtained, the semantic description of the new composite service is generated using the composition language OWL-S [2]. This description document can be registered in the 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.

Our research makes the following novel contributions: (i) We present a generalized composition algorithm based on generating conditional directed acyclic graphs; (ii) we present an 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; (iii) we automatically generate OWL-S descriptions of the new composite service obtained; and, (iv) we present a prototype implementation based on constraint logic programming that works efficiently on large repositories.

The rest of the paper is organized as follows. In section 2 we present the related work in the area of service composition and discuss their limitations. In section 3, we present the service composition problem, the different kinds of composition with examples, and our technique for automatic composition. In section 4, we discuss automatic generation of OWL-S service descriptions. Section 5 presents our prototype implementation and performance results. The last section presents the conclusions and future work.

2 Related Work

Composition of Web services has been active area of research recently [19, 20]. Most of these approaches present techniques to solve one or more phases listed in section 1.

There are many approaches [14–16] 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 [14] present a composition technique by applying logical inferencing on pre-defined 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 objective of our work is to come up with a technique that can automatically produce the composition without the need for any manual intervention.

There are industry solutions based on WSDL and BPEL4WS 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 [16]. 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 find the services that are suitable for composition based on the query requirements for the new composed service.

There are solutions such as [18] 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 criterion are measured using fuzzy numbers.

There has been a lot of work on composition languages such as WS-BPEL, WSML, AO4BPEL, etc. which are useful during the execution phase. FuseJ is also once such description language for unifying aspects and components. Though this language was not designed for Web services, the authors present in [17] that it can be used for service composition as well. It uses connectors to interconnect services. There is no centralized process description, but instead is spread across the connectors. With FuseJ, the planning phase has to be performed manually wherein the connectors have to be written. Similarly, OWL-S can also describe a composite service which is actually an abstract

service. Service grounding of OWL-S maps the abstract service to the concrete WSDL specification. These languages are useful after the planning, discovery, and selection is done. The new composite service can be described using one of these languages.

In this paper, we present a technique for automatically planning, discovering and selecting services that are suitable for obtaining a composite service, based on the user query requirements. As far as we know, all the related approaches to this problem assume that they either already have information about the services involved or use human input on what services would be suitable for composition. We also show different kinds of compositions such as, non-sequential compositions (i.e., composition where there can be more than one service involved at any stage, represented as a directed acyclic graph of services), sequential composition (i.e, a linear chain of services) and composition with if-then-else conditions. We also automatically generate OWL-S descriptions for the new composite service obtained.

3 Web service Composition

Informally, the Web service Composition problem can be defined as follows: 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 1 shows an example composite service made up of five services $\mathcal{S}_1$ to $\mathcal{S}_5$. In the 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 $\mathcal{S}_i$ and $\mathcal{S}_j$ indicates that outputs of $\mathcal{S}_i$ constitute (some of) the inputs of $\mathcal{S}_j$.

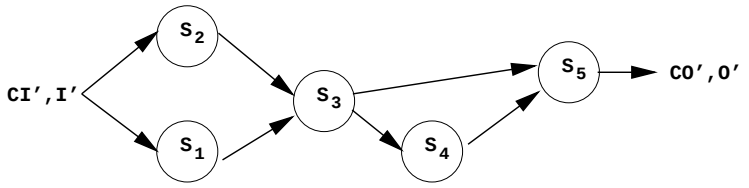

Fig. 1. Example of a Composite Service as a Directed Acyclic Graph

As mentioned in section 1, composition can be seen as four phases which include (i) Planning, (ii) Discovery, (iii) Selection, and (iv) Execution. Most of the related work presents techniques to solve one of these phases or a combination of them. We need a technique that can automatically determine the services involved in composition without the need for any manual intervention or user-defined plans. In this section we present the different kind of compositions and our technique for automatic composition which performs the first three phases - planning, discovery, and selection simultaneously as one phase.

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

Definition (Service): A service is a 6-tuple of its pre-conditions, inputs, side-effect, affected object, outputs and post-conditions. $\mathcal{S}$ = ($\mathcal{CI}$, $\mathcal{I}$, $\mathcal{A}$, $\mathcal{AO}$, $\mathcal{O}$, $\mathcal{CO}$) is the representation of a service where $\mathcal{CI}$ is the list of pre-conditions, $\mathcal{I}$ is the input list, $\mathcal{A}$ is the

service's side-effect, $\mathcal{AO}$ is the affected object, $\mathcal{O}$ is the output list, and $\mathcal{CO}$ is the list of post-conditions.

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

3.1 Sequential Composition

A sequential composition is one which has a linear chain of services that form the composite service. When all nodes in the directed acyclic graph of figure 1 have not more than one incoming edge and not more than one outgoing edge, the problem reduces to a sequential composition problem.

Example 1: 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 1 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 2 shows this example sequential composition as a directed acyclic graph.

Service	*Input Parameters*	*Output Parameters*
Query	*PassengerName, OriginAirport,Start Date,DestinationAirport,ReturnDate*	*HotelConfirmationNum, CarConfirmationNum*
ReserveFlight	*PassengerName, OriginAirport,Start Date, DestinationAirport,ReturnDate*	*FlightConfirmationNum, ArrivalFlightNum*
ReserveHotel	*PassengerName, ArrivalFlightNum, StartDate, ReturnDate*	*HotelConfirmationNum, HotelAddress*
ReserveCar	*PassengerName,ArrivalDate ArrivalFlightNum, HotelAddress*	*CarConfirmationNum*

Table 1. Example Scenario for Sequential Composition

Definition (Sequential Composition): More generally, the Composition problem can be defined as automatically finding a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of services from repository $\mathcal{R}$, given query $\mathcal{Q} = (\mathcal{CI}', \mathcal{I}', \mathcal{A}', \mathcal{AO}', \mathcal{O}', \mathcal{CO}')$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ 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: $\forall i\ \mathcal{S}_i \in \mathcal{V}$, $\mathcal{S}_i \in \mathcal{R}$, $\mathcal{S}_i = (\mathcal{CI}_i, \mathcal{I}_i, \mathcal{A}_i, \mathcal{AO}_i, \mathcal{O}_i, \mathcal{CO}_i)$

1. $\mathcal{I}' \sqsupseteq \mathcal{I}_1$, $\mathcal{O}_1 \sqsupseteq \mathcal{I}_2$, ..., $\mathcal{O}_n \sqsupseteq \mathcal{O}'$
2. $\mathcal{CI}' \Rightarrow \mathcal{CI}_1$, $\mathcal{CO}_1 \Rightarrow \mathcal{CI}_2$, ..., $\mathcal{CO}_n \Rightarrow \mathcal{CO}'$

The meaning of the $\sqsubseteq$ 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.

Fig. 2. Example of Sequential Composition as a Directed Acyclic Graph

3.2 Non-Sequential Composition

Let us consider an example where a non-sequential composition can be obtained using the given repository of services. A non-sequential composition can have more than one service involved at any stage of the composition task.

Example 2: Suppose we are looking for a service to make international travel arrangements and the directory of services contains *ReserveFlight*, *ReserveHotel*, *ReserveCar*, and *ProcessVisa* services. In this scenario, we first need to apply for a visa and then make the flight, hotel, and car reservations. Table 2 shows the input/output parameters of the user-query and the four services.

In this example, service *ProcessVisa* has a post-condition *VisaApproved*. The services *ReserveFlight* and *ReserveHotel* have the pre-condition that the visa must be approved before making reservations. *ProcessVisa* has to be executed first followed by *ReserveFlight* and *ReserveHotel*. The post-conditions of *ProcessVisa* must imply the pre-conditions of *ReserveFlight* and similarly must imply the pre-conditions of *ReserveHotel*. The *ReserveCar* service needs inputs *HotelAddress* and *ArrivalFlightNum*. Hence it is executed after both *ReserveFlight* and *ReserveHotel* are executed so that their outputs can be used as inputs to *ReserveCar*. 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 non-sequential composition example as a directed acyclic graph.

Definition (Non-Sequential Composition): More generally, the Composition problem can be defined as automatically finding a directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of services from repository $\mathcal{R}$, given query $\mathcal{Q} = (\mathcal{CI}', \mathcal{I}', \mathcal{A}', \mathcal{AO}', \mathcal{O}', \mathcal{CO}')$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ 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. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ has exactly one incoming edge that represents the query inputs and pre-conditions, $\mathcal{I}' \sqsupseteq \bigcup_i \mathcal{I}_i$, $\mathcal{CI}' \Rightarrow \wedge_i \mathcal{CI}_i$.

2. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ has exactly one outgoing edge that represents the query outputs and post-conditions, $\mathcal{O}' \sqsubseteq \bigcup_i \mathcal{O}_i$, $\mathcal{CO}' \Leftarrow \wedge_i \mathcal{CO}_i$.
3. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ has at least one incoming edge, let $\mathcal{S}_{i1}$, $\mathcal{S}_{i2}$, ..., $\mathcal{S}_{im}$ be the nodes such that there is a directed edge from each of these nodes to $\mathcal{S}_i$. Then $I_i \sqsubseteq \bigcup_k \mathcal{O}_{ik} \cup I'$, $CI_i \Leftarrow (\mathcal{CO}_{i1} \wedge \mathcal{CO}_{i2} \ldots \wedge \mathcal{CO}_{im} \wedge CI')$.

Service	*Pre-Conditions*	*Input Parameters*	*Output Parameters*	*Post-Conditions*
Query		*PassengerName, OriginAirport, DestinationAirport, StartDate, ReturnDate*	*FlightConfirmationNum, HotelConfirmationNum, CarConfirmationNum*	
Process Visa		*PassengerName, VisaType, StartDate, ReturnDate*	*ConfirmationNum*	*Visa-Approved*
Reserve Flight	*Visa-Approved*	*PassengerName, OriginAirport, DestinationAirport, StartDate, ReturnDate*	*FlightConfirmationNum, ArrivalFlightNum*	
Reserve Hotel	*Visa-Approved*	*PassengerName, StartDate, ReturnDate*	*HotelConfirmationNum, HotelAddress*	
Reserve Car		*PassengerName, ArrivalDate, ArrivalFlightNum, HotelAddress*	*CarConfirmationNum*	

Table 2. Example Scenario for Non-Sequential Composition

The meaning of $\sqsubseteq$ 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. Further details on the formal description of the composition problem are in [5].

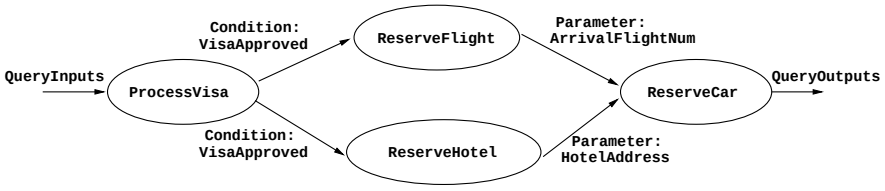

Fig. 3. Example of Non-Sequential Composition as a Directed Acyclic Graph

3.3 Non-Sequential Conditional Composition

Let us now consider an example of a non-sequential composition with if-then-else conditions, i.e., the composition flow varies depending on the result of the post-conditions of a service.

Example 3: This example is similar to Example 2 but with more constraints. 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 3 shows the input/output parameters of the user-query and the services.

Service	*Pre-Conditions*	*Input Parameters*	*Output Parameters*	*Post-Conditions*
Query		*PasngrName,OriginArprt,Dest-, Arpt,StartDate,ReturnDate*	*FlightConfNum,Hotel-ConfNum,CarConfNum*	
Reserve Flight		*PasngrName,OriginArprt,Dest-Arpt,StartDate,ReturnDate*	*FlightConfNum*	
Reserve Hotel		*PasngrName, StartDate, ReturnDate*	*HotelConfNum*	
Process Visa		*PasngrName,VisaType, FlightConfNum,HotelConfNum*	*ConfirmationNum*	*VisaApproved* $\vee$ *VisaDenied*
ConfirmFlight	*VisaApproved*	*FlightConfNum,CreditCardNum*	*ArrivalFlightNum*	
ConfirmHotel	*VisaApproved*	*HotelConfNum,CreditCardNum*	*ConfirmationNum*	
CancelFlight	*VisaDenied*	*PasngrName,FlightConfNum*	*CancelCode*	
CancelHotel	*VisaDenied*	*PasngrName,HotelConfNum*	*CancelCode*	
Reserve Car		*PasngrName,ArrivalDate ArrivalFlightNum*	*CarConfirmNum*	

Table 3. Example Scenario for Non-Sequential Conditional Composition

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 if 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 an OR node with the outgoing edges representing the generated conditions and the outputs. In this case conditions *(VisaApproved* $\vee$ *VisaDenied)* $\Rightarrow$ *VisaApproved* and *(VisaApproved* $\vee$ *VisaDenied)* $\Rightarrow$ *VisaDenied* are generated. Depending on which condition holds, the corresponding services *ConfirmFlight* or *CancelFlight* are executed. Figure 4 shows this conditional composition example as an AND/OR directed acyclic graph.

Definition (Non-Sequential Conditional Composition): More generally, the Composition problem can be defined as automatically finding an AND/OR directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of services from repository $\mathcal{R}$, given query $\mathcal{Q} = (\mathcal{CI}', \mathcal{I}', \mathcal{A}', \mathcal{AO}', O', CO')$, where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges of the graph. Each vertex in the graph is either an *OR* or *AND* vertex and represents a service in the composi-

tion. 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 outgoing edges of an *AND* node have an arc around them to differentiate from the *OR* node. The following conditions should hold on the nodes of the graph:

1. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ has exactly one incoming edge that represents the query inputs and pre-conditions, $\mathcal{I}' \sqsupseteq \bigcup_i \mathcal{I}_i$, $\mathcal{CI}' \Rightarrow \wedge_i \mathcal{CI}_i$.
2. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ has exactly one outgoing edge that represents the query outputs and post-conditions, $\mathcal{O}' \sqsubseteq \bigcup_i \mathcal{O}_i$, $\mathcal{CO}' \Leftarrow \wedge_i \mathcal{CO}_i$.
3. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ is an *AND* vertex, let $\mathcal{S}_{i1}$, $\mathcal{S}_{i2}$, ..., $\mathcal{S}_{im}$ be the nodes such that there is a directed edge from $\mathcal{S}_i$ to each of these nodes. Then $(O_i \cup I') \sqsupseteq \bigcup_k \mathcal{I}_{ik}$, $CO_i \Rightarrow (\mathcal{CI}_{i1} \wedge \mathcal{CI}_{i2} \ldots \wedge \mathcal{CI}_{im})$.
4. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ is an *OR* vertex, let $\mathcal{S}_{i1}$, $\mathcal{S}_{i2}$, ..., $\mathcal{S}_{im}$ be the nodes such that there is a directed edge from $\mathcal{S}_i$ to each of these nodes. Then $(O_i \cup I') \sqsupseteq \bigcup_k \mathcal{I}_{ik}$, $CO_i \Rightarrow (\mathcal{CI}_{i1} \vee \mathcal{CI}_{i2} \ldots \vee \mathcal{CI}_{im})$.

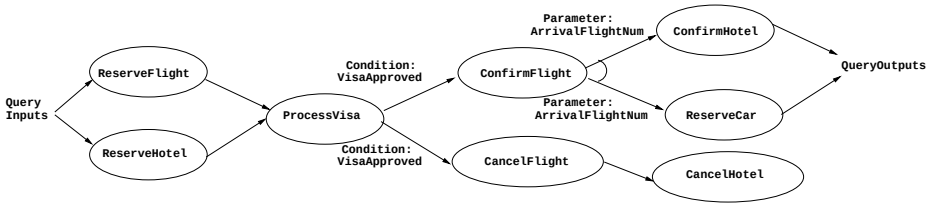

Fig. 4. Example of Conditional Composition as an AND/OR Directed Acyclic Graph

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, an *OR* node is created in the graph. Each outgoing edges represent the possible conditions which will be evaluated at run-time. Depending on the condition that holds, the corresponding services are executed. That is, if a subservice $\mathcal{S}_1$ is composed with subservice $\mathcal{S}_2$, then the postconditions $\mathcal{CO}_1$ of $\mathcal{S}_1$ must imply the preconditions $\mathcal{CI}_2$ of $\mathcal{S}_2$. The following conditions are evaluated at run-time:
if ($\mathcal{CO}_1 \Rightarrow \mathcal{CI}_2$) *then execute* $\mathcal{S}_1$;
else if ($\mathcal{CO}_1 \Rightarrow \neg\ \mathcal{CI}_2$) *then no-op*;
else if ($\mathcal{CI}_2$) *then execute* $\mathcal{S}_1$;

3.4 Automatic Composition Algorithm

In order to produce the composite service which is the graph, as shown in the example figure 1, we filter out services that are not useful for the composition at multiple stages. Figure 5 shows the filtering technique for the particular instance shown in figure 1. The

composition routine starts with the query input parameters. It finds all those services from the repository which require a subset of the query input parameters. In figure 5, CI, I are the pre-conditions and the input parameters provided by the query. $\mathcal{S}_1$ and $\mathcal{S}_2$ are the services found after step 1. $\mathcal{O}_1$ 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., $\mathcal{I}_2 = \mathcal{O}_1 \cup I$. $\mathcal{I}_2$ is used to find services at the next stage, i.e., all those services that require a subset of $\mathcal{I}_2$. In order to make sure we do not end up in cycles, we get only those services which 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 which do not directly or indirectly contribute to the query output parameters. This is done starting with the output parameters working our way backwards.

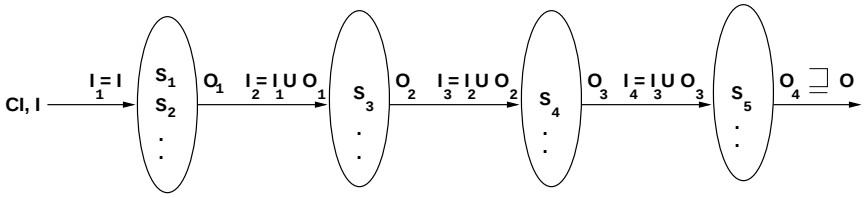

Fig. 5. Composite Service

4 Automatic Generation of OWL-S descriptions

After we have obtained a composition solution (sequential, non-sequential, or conditional), the next step is to produce a semantic description document for this new composite service. Then 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 [2] 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 the kind of composition. OWL-S also provides a property called *composedBy* using which the services involved in the composition can be specified. Below is the algorithm for generation of the OWL-S document when the composition solution in the form of a graph is provided as the input.

```
Algorithm: GenerateServiceDescription (Input: G - Solution Graph)
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 more than one outgoing edge
              GenerateSplitJoinConstruct
            If V is OR vertex with one outgoing edge
              GenerateAtomicService
            If V is OR vertex with more than one outgoing edge
              GenerateConditionalConstruct
          End For
7.    End SequenceConstruct
8.    End Composite Service element
9.    Generate generic footer constructs
```

A sequential composition can be described using the *Sequence* construct which indicates that all the services inside this construct have to 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. The OWL-S description documents for the example composite services in section 3 are shown in the Appendix.

There are other constructs such as looping constructs in OWL-S which 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

We implemented a prototype composition engine using Prolog with Constraint Logic Programming over finite domain [8], referred to as CLP(FD) hereafter. In our current implementation, we used semantic descriptions of web services written in the language called USDL [4]. The repository of services contains one description document for each service. USDL itself is used to specify the requirements of the service that an application developer is seeking.

Each service description is converted into a tuple:

(Pre-Conditions, I, A, O, Post-Conditions).

I is the list of inputs and *O* is the list of outputs. *Pre-Conditions* are list of conditions on the input parameters and *Post-Conditions* are the list of conditions on the output parameters. *A* is the list of side-effects represented as *affect-type(affected-object)* where 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. Services are converted to tuples 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 [6]. 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.

The composition engine consists of these modules: (i) Tuple Generator; (ii) Query Reader; (iii) SemanticRelations Generator; (iv) Composition Query Processor; (v) OWL-S Description Generator;

TupleGenerator converts each service in the repository into the tuple format. The *SemanticRelationsGenerator* module extracts all the semantic relations and creates a list of Prolog facts. In our current implementation, we use USDL service descriptions which use OWL Wordnet Ontology [3] to specify the semantics. This module is generic enough to be used with other domain-specific ontology as well to obtain semantic relations of concepts. The *CompositionQueryProcessor* module uses the repository of facts, which contains all the services, their input and output parameters and the semantic relations between the parameters. The following is the code snippet of our composition engine:

```
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).
```

The output of the query processor is the composition solution which is directed acyclic graph of all the services involved in the composition. Our algorithm selects the optimal solution with least composition length (i.e., the number of stages involved in the composition). 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 our constraint based approach. For example, if each service has an associated cost, then the solutions with the minimal cost are returned. The next step is to produce a description of the new composite service solution found. *OWL-S DescriptionGenerator* automatically generates the OWL-S description of the composite service using constructs depending on the type of composition.

We tested our composition algorithm using repositories from WS-Challenge website[9], slightly modified to fit into USDL framework. They provide repositories of various sizes (thousands of services). These repositories contain WSDL descriptions of

services. The queries and solutions are provided in an XML format. The semantic relations between various parameters are provided in an XML Schema file. We evaluated our approach on different size repositories and tabulated Pre-processing and Query Execution 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 4 shows performance results for the different kind of compositions. The preprocessing time remains the same of all three kinds of composition whereas the query execution time varies. The times shown in the 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 results are consistent with our expectations: for a fixed repository size, the preprocessing time increases with the increase in number of input/output parameters. Similarly, for fixed input/output sizes, the preprocessing 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 to 3 msecs) even for complex queries with large repositories.

Repository Size (num of services)	*Number of I/O param--eters*	*Pre-Processing Time (secs)*	*QueryExec Time (msecs)*		
			Sequential Composition	*NonSequential Composition*	*Conditional Composition*
2000	4-8	36.5	1	1	1
2000	16-20	45.8	1	1	2
2000	32-36	57.8	2	2	2
2500	4-8	47.7	1	1	1
2500	16-20	58.7	1	2	2
2500	32-36	71.6	2	2	3
3000	4-8	56.8	1	1	1
3000	16-20	77.1	1	2	3
3000	32-36	88.2	3	3	4

Table 4. Performance of Composition Algorithm

6 Conclusions and Future Work

To make Web services more practical we need an infrastructure that allows users to discover, deploy, synthesize and compose services automatically. 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 criteria like cost of services, number of services in a composition, etc.). The composition flow is determined automatically without the need for any manual intervention. Our engine finds any sequential or non-sequential 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. We are able to apply many optimization techniques to our system so that it works efficiently even on large repositories. Use of Constraint Logic Programming helped greatly in obtaining an efficient implementation of this system.

Our future work includes extending our engine to support an external database to save off pre-processed data. This will be particularly useful when service repositories grow extremely large in size which can easily be the case in future. We are also 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 and exploring other language possibilities is also part of our future work.

Acknowledgments: We are grateful to our co-researcher Ajay Bansal for the helpful discussions and comments.

References

1. S. McIlraith, T.C. Son, H. Zeng. Semantic Web Services. In *IEEE Intelligent Systems Vol. 16, Issue 2, pp. 46-53, March 2001*.
2. OWL-S `www.daml.org/services/owl-s/1.0/owl-s.html`.
3. OWL WordNet: Ontology-based information management system. `http://taurus.unine.ch/knowler/wordnet.html`.
4. A. Bansal, S. Kona, L. Simon, A. Mallya, G. Gupta, and T. Hite. A Universal Service-Semantics Description Language. In *ECOWS, pp. 214-225, 2005*.
5. S. Kona, A. Bansal, and G. Gupta. Automatic Composition of Semantic Web Services. In *ICWS, 2007*.
6. S. Kona, A. Bansal, L. Simon, A. Mallya, G. Gupta, and T. Hite. USDL: A Service-Semantics Description Language for Automatic Service Discovery and Composition. Tech. Report UTDCS-18-06. `www.utdallas.edu/~sxk038200/USDL.pdf`.
7. A. Bansal, K. Patel, G. Gupta, B. Raghavachari, E. Harris, and J. Staves. Towards Intelligent Services: A case study in chemical emergency response. In *ICWS, pp.751-758, 2005*.
8. K. Marriott and P. Stuckey. Prog. with Constraints: An Introduction. *MIT Press, 1998*.
9. WS Challenge 2006. `http://insel.flp.cs.tu-berlin.de/wsc06`.
10. U. Keller, R. Lara, H. Lausen, A. Polleres, and D. Fensel. Automatic Location of Services. In *European Semantic Web Conference, May 2005*.
11. D. Mandell, S. McIlraith Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *ISWC, 2003.*
12. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara Semantic Matching of Web Service Capabilities. In *ISWC, pages 333-347, 2002*.
13. S. Grimm, B. Motik, and C. Preist Variance in e-Business Service Discovery. In *Semantic Web Services Workshop at ISWC, November 2004*.
14. S. McIlraith, T.C. Son Adapting golog for composition of semantic Web services. In *KRR, pages 482–493, 2002*.
15. B. Srivastava. Automatic Web Services Composition using planning. In *KBCS, pp.467-477*.
16. M. Pistore, P. Roberti, and P. Traverso Process-Level Composition of Executable Web Services In *European Semantic Web Conference, pages 62-77, 2005*.
17. D. Suvee, B. Fraine, and M. Cibran Evaluating FuseJ as a Web Service Composition Language In *European Conference on Web Services, 2005*.
18. D. Claro, P. Albers, and J. Hao Selecting Web services for Optimal Compositions In *Workshop on Semantic Web Services and Web Service Composition, 2004*.
19. J. Rao, X. Su. A Survey of Automated Web Service Composition Methods In *Workshop on Semantic Web Services and Web Process Composition(SWSWPC), 2004*.
20. J. Cardoso, A. Sheth. Semantic Web Services, Processes and Applications. *Springer, 2006*.

Appendix

The sequential composite service shown in Example 1 and Figure 2 is described in OWL-S as follows:

```
<rdf:RDF ...
  <process:CompositeProcess rdf:ID="TravelReservation">
    ...
    <process:composedOf><process:Sequence>
      <process:components rdf:parseType="Collection">
        <process:AtomicProcess rdf:about="#ReserveFlight"/>
        <process:AtomicProcess rdf:about="#ReserveHotel"/>
        <process:AtomicProcess rdf:about="#ReserveCar"/>
      </process:components></process:Sequence>
    </process:composedOf>
  </process:CompositeProcess></rdf:RDF>
```

The non-sequential composite service shown in Example 2 and Figure 3 is described in OWL-S as follows:

```
<rdf:RDF ...
  <process:CompositeProcess rdf:ID="TravelReservation">
    ...
    <process:composedOf><process:Sequence>
      <process:components rdf:parseType="Collection">
        <process:AtomicProcess rdf:about="#ProcessVisa"/>
        <process:CompositeProcess rdf:about="#Stage1"/>
        <process:AtomicProcess rdf:about="#ReserveCar"/>
      </process:components>
    </process:Sequence>
    </process:composedOf>
  </process:CompositeProcess>
  <process:CompositeProcess rdf:ID="Stage1">
    <process:composedOf><process:Split-Join>
      <process:components rdf:parseType="Collection">
        <process:AtomicProcess rdf:about="#ReserveFlight"/>
        <process:AtomicProcess rdf:about="#ReserveHotel"/>
      </process:components></process:Split-Join>
    </process:composedOf>
  </process:CompositeProcess></rdf:RDF>
```

The conditional composite service shown in Example 3 and Figure 4 is described in OWL-S as follows:

```
<rdf:RDF ...
  <process:CompositeProcess rdf:ID="TravelReservation">
    ...
    <process:composedOf><process:Sequence>
      <process:components rdf:parseType="Collection">
        <process:AtomicProcess rdf:about="#Stage1"/>
        <process:AtomicProcess rdf:about="#ProcessVisa"/>
        <process:CompositeProcess rdf:about="#IfThenElseStage1"/>
      </process:components></process:Sequence>
    </process:composedOf>
  </process:CompositeProcess>
```

```
<process:CompositeProcess rdf:ID="IfThenElseStage1">
  ...
  <process:composedOf>
    <process:If-Then-Else>
      <process:ifCondition>
        <expr:SWRL-Condition rdf:ID="VisaAccepted">
          <expr:expressionLanguage rdf:resource="&expr;#SWRL"/>
          <expr:expressionBody rdf:parseType="Literal">
          <swrl:AtomList>
            <rdf:first>
              <swrl:IndividualPropertyAtom>
                <swrlb:equal rdf:resource="#VisaAccepted"/>
                <swrl:argument1 rdf:resource="#VisaAccepted"/>
                <swrl:argument2 rdf:resource="&rdf;#true"/>
              </swrl:IndividualPropertyAtom>
            </rdf:first>
            <rdf:rest rdf:resource="&rdf;#nil"/>
          </swrl:AtomList>
          </expr:expressionBody>
        </expr:SWRL-Condition>
      </process:ifCondition>
      <process:then>
        <process:Sequence>
          <process:components rdf:parseType="Collection">
            <process:AtomicProcess rdf:about="#ConfirmFlight"/>
            <process:CompositeProcess rdf:about="#Stage2"/>
          </process:components>
        </process:Sequence>
      </process:then>
      <process:else>
        <process:Sequence>
          <process:components rdf:parseType="Collection">
            <process:AtomicProcess rdf:about="#CancelFlight"/>
            <process:AtomicProcess rdf:about="#CancelHotel"/>
          </process:components></process:Sequence>
      </process:else>
  </process:If-Then-Else></process:composedOf>
</process:CompositeProcess>
<process:CompositeProcess rdf:ID="Stage1">
  <process:composedOf><process:Split-Join>
    <process:components rdf:parseType="Collection">
      <process:AtomicProcess rdf:about="#ReserveFlight"/>
      <process:AtomicProcess rdf:about="#ReserveHotel"/>
    </process:components></process:Split-Join>
  </process:composedOf>
</process:CompositeProcess>
<process:CompositeProcess rdf:ID="Stage2">
  <process:composedOf><process:Split-Join>
    <process:components rdf:parseType="Collection">
      <process:AtomicProcess rdf:about="#ConfirmHotel"/>
      <process:AtomicProcess rdf:about="#ReserveCar"/>
    </process:components></process:Split-Join>
  </process:composedOf>
</process:CompositeProcess></rdf:RDF>
```