Generalized Semantics-based Service Composition

Srividya Kona, Ajay Bansal, M. Brian Blake
Department of Computer Science,
Georgetown University
Washington, DC 20057

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

Abstract

Service-oriented computing (SOC) has emerged as the eminent market environment for sharing and reusing service-centric capabilities. The underpinning for an organization's use of SOC techniques is the ability to discover and compose Web services. Although industry approaches to composition have a strong notion of business processes, these approaches largely use syntactic descriptions. As such composition is limited since the true functionality of ambiguous service operations cannot be inferred. Alternatively, academia uses semantic approaches to disambiguate services, but, at the same time, most of these approaches neglect the process rigor needed for complex compositions. 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 extends the common practice of linearly linked services by introducing the use of a conditional directed acyclic graph (DAG) where complex interactions, containing control flow, information flow and pre/post conditions, are effectively represented. Furthermore, the composition can be represented semantically as OWL-S documents. Our contributions are applied for automatic workflow generation in context of the currently important bioinformatics domain.

1. Introduction

Service-oriented computing is changing the way software applications are being designed, developed, delivered, and consumed [1]. 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. As automation increases, these services will be accessed directly by applications rather than by humans [10]. 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 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 [5]. Several efforts are underway to build such an infrastructure [13, 14, 18].

With regards 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 [22]. 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 Web service composition can be applied to a Bioinformatics analysis application, for automatic workflow generation in the field of Phylogenetics [4].

Our research makes the following novel contributions: (i) We formalize the generalized composition problem based on our conditional directed acyclic graph representation; (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 apply our generalized composition engine to automatically generate workflows in the field of Bioinformatics.

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 formalize the generalized service composition problem followed by a discussion of our technique for automatic Web service composition and automatic generation of OWL-S service descriptions in Section 4. Section 5 presents an application of our generalized composition engine to automatically generate workflows for Bioinformatics analysis tasks. The last section presents the conclusions and future work.

2. Related Work

Composition of Web services has been active area of research recently [21, 22]. Most of these approaches present techniques to solve one or more phases listed in Section 1. There are many approaches [15, 16, 18] 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 [15] 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 finds the services that are suitable for composition based on the query requirements for the new composed service.

There are solutions such as [20] 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, FuseJ, AO4BPEL, etc. which are useful only during the execution phase. FuseJ is a description language for unifying aspects and components [19]. 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

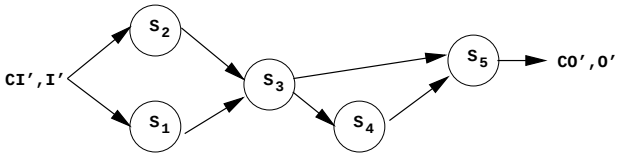

Figure 1. Example of a Composite Service as a Directed Acyclic Graph

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 the described abstract service to the concrete WSDL specification which 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 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.

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 $\mathcal{S}_j$. We next formalize the generalized composition problem. In this generalization, we extend our previous notion of composition [9] to handle non-sequential conditional composition (which we believe is the most general case of composition).

Definition (Repository of Services): Repository ($\mathcal{R}$) 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. The pre- and post-conditions are ground logical predicates.

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 list of 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 list of post-conditions. These are all the parameters of the requested service.

Definition (Generalized Composition): The generalized 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 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:

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 $\mathcal{S}_i$ represents a service and 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 $I_i \sqsubseteq \bigcup_k \mathcal{O}_{ik} \cup I'$, $CI_i \Leftarrow (\mathcal{CO}_{i1} \wedge \mathcal{CO}_{i2} ... \wedge \mathcal{CO}_{im} \wedge CI')$.
4. $\forall i\ \mathcal{S}_i \in \mathcal{V}$ where $\mathcal{S}_i$ represents a condition that is evaluated at run-time and has exactly one incoming edge, let $\mathcal{S}_j$ be its immediate predecessor node such that there is a directed edge from $\mathcal{S}_j$ to $\mathcal{S}_i$. Then the inputs and pre-conditions at node $\mathcal{S}_i$ are $I_i = \mathcal{O}_j \cup I'$, $CI_i = \mathcal{CO}_j$. The outgoing edges from $\mathcal{S}_i$ 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 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* $\mathcal{S}_1$;

With the above formalism in place, we next present our generalized composition algorithm.

4. Automatic Generation of Composite Services

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

4.1. Generalized 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 2 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 2, CI, I are the preconditions 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.

The Web service Composition algorithm presented here has been proven to be sound and complete. Intuitively, we can see that every composite service generated by our algorithm is indeed a service that meets all the query requirements and hence is sound. Also our algorithm generates every composite service possible for the given query and hence is complete. Intuitively, we can see from the algorithm that these theorems hold and can be shown via proofs by contradiction. These soundness and completeness proofs are not included here due to lack of space.

```
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 non-conditional vertex
         with one outgoing edge
           GenerateAtomicService
       If V is non-conditional vertex
         with more than one outgoing edge
           GenerateSplitJoinConstruct
       If V is Conditional vertex
         with one outgoing edge
           GenerateAtomicService
       If V is Conditional vertex with
         more than one outgoing edge
           GenerateConditionalConstruct
     End For
7. End SequenceConstruct
8. End Composite Service element
9. Generate generic footer constructs
```

Table 1. Automatic Generation of OWL-S description of Composite Service

4.2. Automatic Generation of OWL-S Description

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 com-

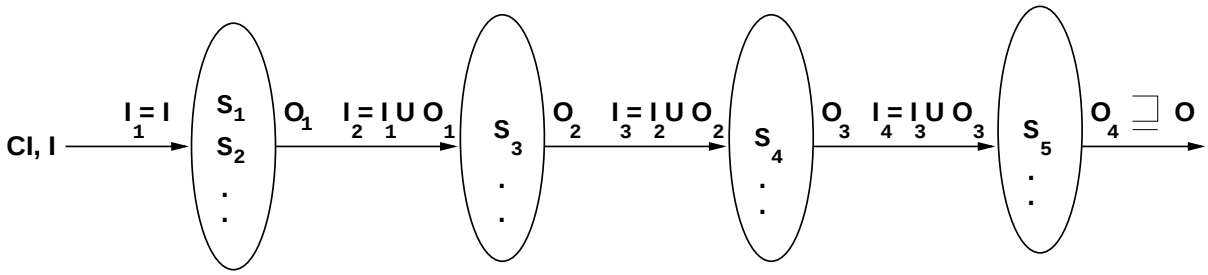

Figure 2. Generalized Composition Technique

posite 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 [6] 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. Table 1 presents the algorithm for automatic generation of the OWL-S description of the composite service.

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.

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.

4.3. Implementation

We implemented a prototype composition engine using Prolog with Constraint Logic Programming over finite domain [12], referred to as CLP(FD) hereafter. In our current implementation, we used semantic descriptions of web services written in the language called USDL [8]. 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. The composition engine consists of 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. 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 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). The next step is to produce a description of the new composite service solution found. *OWL-S Description-Generator* automatically generates the OWL-S description of the composite service using constructs depending on the type of composition.

5. 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 [4] 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.

Phylogenetics

Phylogenetic inferencing involves an attempt to estimate the evolutionary history of a collection of organisms (taxa)

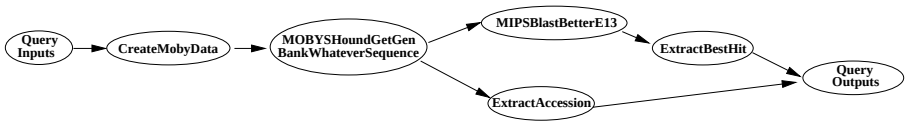

Figure 3. Example of Non-Sequential Composition as a Directed Acyclic Graph

or a family of genes. 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. 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 [11] 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 [7] 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 1: 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*, *MOBYSHoundGetGenBankWhateverSequence*, *MIPSBlastBetterE13*, *ExtractAccession*, and *ExtractBestHit* 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. Figure 3 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 the service *MOBYSHoundGetGenBankWhateverSequence* has a pre-condition that the its input parameter *MobyData* has to be in NCBI format for service execution. The post-condition of *CreateMobyData* should imply the pre-condition of the service *MOBYSHoundGetGenBankWhateverSequence*.
- 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 2: 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. The repository contains services *BLAST*, *CLUSTAL*, *PAUP*, *MEGA*, *PHYLIP*, *BLASTNexus*, and *NexusCLUSTAL*. Table 2 shows the input/output parameters of the user-query and the services. 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

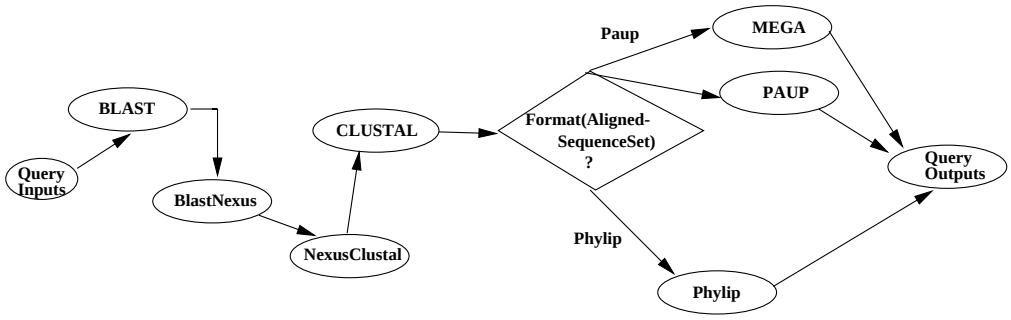

Figure 4. Example of Non-Sequential Conditional Composition as a Directed Acyclic Graph

Service	*Pre-Conditions*	*Input Parameters*	*Output Parameters*	*Post-Conditions*
Query		*Sequence,OrganismType, WordSize,DatabaseName*	*EvolutionTree, EvolutionDistance*	*Format(Evolution-Tree)=Newick*
BLAST		*Sequence,OrganismType, DatabaseName*	*BlastSequenceSet*	
CLUSTAL	*Format(Clustal-SequenceSet)=Clustal*	*ClustalSequenceSet*	*AlignedSequenceSet*	*Format(Aligned-SequenceSet)=Paup* $\vee$ *Format(Aligned-SequenceSet)=Phylip*
BLASTNexus		*BlastSequenceSet*	*NexusSequenceSet*	*Format(Nexus-SequenceSet)=Nexus*
NexusCLUSTAL	*Format(Nexus-SequenceSet)=Nexus*	*NexusSequenceSet*	*ClustalSequenceSet*	*Format(Clustal-SequenceSet)=Clustal*
PAUP	*Format(Aligned-SequenceSet)=Paup*	*AlignedSequenceSet, WordSize*	*EvolutionTree*	*Format(Evolution-Tree)=Newick*
PHYLIP	*Format(Aligned-SequenceSet)=Phylip*	*AlignedSequenceSet*	*EvolutionTree*	*Format(Evolution-Tree)=Newick*
MEGA	*Format(Aligned-SequenceSet)=Paup*	*AlignedSequenceSet*	*EvolutionDistance*	

Table 2. Example Scenario for Non-Sequential Conditional Composition

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 which 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, the next service for the composition is chosen. In this case, one cannot determine if the post-conditions of the service *CLUSTAL* imply the 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 the Figure 4 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 which represent the generated conditions that are evaluated at run-time. In this case the following conditions are generated:

- *(Format(AlignedSequenceSet) = Paup* $\vee$ *Format(AlignedSequenceSet) = Phylip)* $\Rightarrow$ *(Format(AlignedSequenceSet) = Paup*
- *(Format(AlignedSequenceSet) = Paup* $\vee$ *Format(AlignedSequenceSet) = Phylip)* $\Rightarrow$ *(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 4 shows this non-sequential conditional composition example as a conditional directed acyclic graph.

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, 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. 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 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 [3] 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 [2].

References

[1] G. Castagna, N. Gesbert, L. Padovani, et al. A Theory of Contracts for Web Services. In *Symposium on Principles of Programming Languages, January 2008*

[2] G. Gupta, S. Kona, B. Devries, et al. Problem Solving in Evolutionary Analysis with Web Services and DSLs. `https://www.nescent.org/wg_evoinfo/Announcements`.

[3] BioPerl. `http://www.bioperl.org`.

[4] A.W.F. Edwards, L.L. Cavalli-Sforza (1964). *Systematics Assoc. Publ.* No. 6: Phenetic and Phylogenetic Classification: Reconstruction of evolutionary trees, 67-76.

[5] S. McIlraith, T.C. Son, H. Zeng. Semantic Web Services. In *IEEE Intelligent Systems Vol. 16, Issue 2, pp. 46-53, March 2001*.

[6] OWL-S `www.daml.org/services/owl-s/1.0/owl-s.html`.

[7] MyGrid. `http://www.mygrid.org.uk/`.

[8] 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*.

[9] S. Kona, A. Bansal, and G. Gupta. Automatic Composition of Semantic Web Services. In *ICWS, 2007*.

[10] 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*.

[11] J. Iglesias, G. Gupta, E. Pontelli, D. Ranjan, and B. Milligan. Interoperability between Bioinformatics Tools: A Logic Programming Approach. In *Practical Aspects of Declarative Languages (PADL)*, Lecture Notes in Computer Science. Springer Heidelberg, 2001.

[12] K. Marriott and P. Stuckey. Prog. with Constraints: An Introduction. *MIT Press, 1998*.

[13] D. Mandell, S. McIlraith Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *ISWC, 2003*.

[14] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara Semantic Matching of Web Service Capabilities. In *ISWC, pages 333-347, 2002*.

[15] S. McIlraith, T. Son Adapting golog for composition of semantic Web services. In *KRR, pp.482–493, 2002*.

[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] P. Hofmann. SAP AG. Personal Communication.

[18] J. Rao, D. Dimitrov, P. Hofmann, and N. Sadeh A Mixed-Initiative Approach to Semantic Web Service Discovery and Composition In *International Conference on Web Services, 2006*.

[19] D. Suvee, B. Fraine, and M. Cibran Evaluating FuseJ as a Web Service Composition Language In *European Conference on Web Services, 2005*.

[20] D. Claro, P. Albers, and J. Hao Selecting Web services for Optimal Compositions In *Workshop on Semantic Web Services and Web Service Composition, 2004*.

[21] J. Rao, X. Su. A Survey of Automated Web Service Composition Methods In *Workshop on Semantic Web Services and Web Process Composition, 2004*.

[22] J. Cardoso, A. Sheth. Semantic Web Services, Processes and Applications. *Springer, 2006*.