

Modeling Web Services using Semantic Graph Transformations to aid Automatic Composition

Zhen Liu, Anand Ranganathan, Anton Riabov
IBM T.J. Watson Research Center, Hawthorne, NY 10532
{zhenl, arangana, riabov}@us.ibm.com

Abstract

In this paper, we propose a novel way of modeling web services using semantic graph transformations. Each operation supported by a web service is associated with a semantic annotation that describes the input and output messages using RDF graph patterns. The terms used in these patterns are defined in OWL ontologies that describe the application domain. A key difference between our model and existing semantic web service models like OWL-S is that it describes the inputs and outputs in terms of instance-based graph patterns, rather than in terms of concepts. This allows associating a rich set of constraints on the input and output data in terms of relations between instances. We also propose a composition model for web service operations, that describes the conditions for composing services into workflows. The composition model includes the notion of semantic propagation, i.e. the semantic description of the output message of an operation depends on the semantics of the input message. We have developed a planner that uses this model to compose services, automatically. The planner uses DLP reasoning to aid plan search. We present performance results for the planner.

1 Introduction

An important class of web services consists of those that primarily do data processing, i.e. they take in input data, process or transform them in some manner, and produce output data. In this paper we propose a novel way of associating rich semantic information with the messages consumed and produced by such web services. Our model describes the input message requirements and the output message specification of each web service operation using RDF graph patterns. The terms used in these patterns are defined in OWL ontologies that describe the application domain.

The main motivation for developing this model is to allow automatic composition of workflows that process, transform or analyze data to produce some desired infor-

mation. Our model provides rich descriptions of the inputs and outputs of web service operations. This allows us to compare the semantics of the data produced as output by one or more operations and the semantics of the data required as input by another operation and decide if the two are compatible. The semantics-based comparison is better than a plain syntactic check and can help verify the semantic composability of two services.

Our model provides both a workflow-independent and a workflow-dependent description of a web service operation. The workflow-independent description provides the minimal conditions that must be satisfied by the data provided as input to the operation, and the minimal conditions that will be satisfied by the data produced as output. In a given workflow, however, the actual data provided as input may have additional semantic constraints depending on the semantics of the data produced by previous services in the workflow. The actual data produced as output may similarly have additional constraints. Hence, the workflow-dependent description is a specialization of the workflow-independent description taking into account the actual semantics of the data given as input to the operation.

A key aspect of our model is the notion of semantic propagation, i.e. the semantic description of the output message of an operation depends on the semantics of the input message, which in turn depends on the semantics of the previous messages in the workflow, and so on. In order to model semantic propagation, we model operations using *graph transformations* [2]. The graph transformations allow replacing variables in the RDF graph pattern describing the output message with either single values or entire sub-graphs based on the actual semantics of the input message.

There is an important difference between our model and other semantic web service models. OWL-S [8] describes inputs and outputs using concepts in an ontology. Similarly, SAWSDL [1], which allows linking semantic annotations to WSDL files, is typically used to associate inputs and outputs with concepts in an ontology. Our model describes input and output messages in terms of RDF graph patterns based on data instances that appear in these messages. Our

approach allows associating constraints on these instances based on both the concepts (or classes) they belong to and their relationship to other instances. Such constraints are more difficult to express in pure concept-based representations and often require the creation of a large number of additional concepts corresponding to different combinations of constraints. In addition, the instance-based approach also allows describing messages of complex types such as sequences, and associating rich semantics with the elements within the complex type. As a result, our model allows associating rich semantic information about services, which aids the automatic composition of workflows that involve data processing and data transformation.

Another difference between our model and existing models (like OWL-S and SAWSDL) is in the use of variables. Existing models do not allow expressions with variables for describing inputs and outputs. Our model describes inputs and outputs using graph patterns, which can be considered as logical expressions with variables. The use of variables allows our model to relate the semantics of outputs to the semantics of inputs. WSMML [4] also allows specifying axioms with variables in the pre- and post-conditions of a service capability. However, it does not have an explicit model of messages, which is necessary to determine if certain messages can be given as input to an operation. Our model explicitly defines the elements in a message and describes the semantics of these elements using an RDF graph consisting of OWL ABox assertions. This feature is useful during composition by AI planning, when the planner needs to decide what kind of messages can be given as input to an operation and how it can construct these messages from other messages that have been produced so far by other operations in a partial plan.

Our model is particularly suited for describing services that process data or that provide information, and that do not have any preconditions or effects on the state of the world. It is currently limited to describing constraints on the input and output data, although we are working on extending it to describe state-based preconditions and effects (which is supported by other models like OWL-S and WSMML).

Based on our model, we define the conditions under which web services can be connected to one another in a workflow. We have also developed a planner that can automatically build workflows given a user requirement in terms of a high-level web service whose inputs and outputs are also expressed as RDF graph patterns. The resulting workflow is represented in BPEL and describes a DAG of services that can be invoked to perform the tasks required. The planner incorporates reasoning based on Description Logic Programs (DLP) [5] in building plans. One of the features of the planner is the use of a two-phase approach, where pre-reasoning is performed on component descriptions and the results of reasoning are then reused when generating

plans for different user goals.

In summary, the key contributions of our paper are the proposal of a web service model which is more expressive than OWL-S in describing inputs and outputs, and which includes the notions of semantic propagation, and workflow-independent and workflow-dependent descriptions of services. Another contribution is a planning algorithm that composes workflows with aid of DLP reasoning given a user goal. In Sections 2 and Section 3, we describe the service and composition models. In Sections 4 and 5, we present our planner and evaluate it. Sections 6 and 7 describe related work and conclusions.

2 Model of Web Services

In our model, web service operations are described by the kinds of messages they require as input and the kinds of messages they produce as output. The model describes the data elements contained in the input and output messages. In addition, it describes the semantics of these data elements using RDF graph patterns. Our model provides a blackbox description of the component; it only describes the inputs and outputs, and doesn't model the internal state of the component.

The WSDL description of a web service describes the inputs and outputs of an operation using types defined in XML Schema. Our semantic model builds upon the types and attaches additional semantics to the inputs and output messages. In this paper, we restrict our discussion to simple atomic xml schema types or complex types that are sequences. However, the model can be extended to accommodate other type definitions as well.

For example, consider a `getStockPrice` operation. It takes in a ticker symbol of a company and returns a price with a timestamp. Relevant snippets of the WSDL 2.0 description of the web service are shown below:

```
<types> <xs:schema>
  <xs:element name="tickerSymbol" type="xs:String"/>
  <xs:complexType name="stockPriceInfo">
    <xs:sequence>
      <xs:element name="price" type="xs:float"/>
      <xs:element name="timeStamp" type="xs:dateTime"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema> </types>

<interface name="StockInterface" >
  <operation name="getStockPrice" >
    <input messageLabel="In" element="tickerSymbol" />
    <output messageLabel="Out" element="stockPriceInfo"/>
  </operation> </interface>
```

Figure 1 shows the semantic description of this operation. It captures many of the semantic conditions that must be satisfied by the input and output messages, which

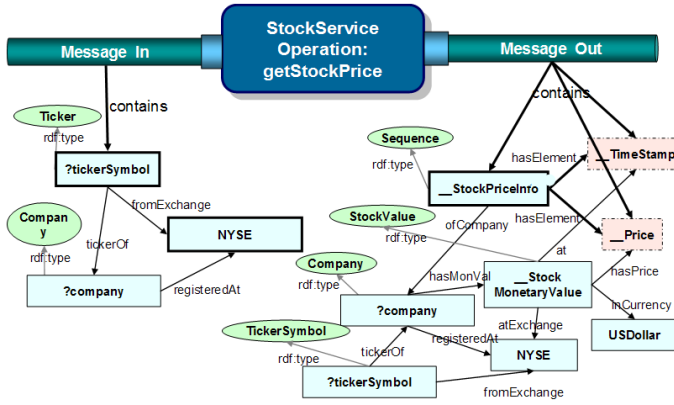


Figure 1. Description of `getStockPrice` operation that takes in a message containing a ticker symbol of a company registered at NYSE and produces a message containing `StockPriceInfo` which is a sequence with two elements: a price and a timestamp for the same company. Rectangle boxes represent OWL individuals or variables, ovals represent OWL classes and dashed rectangles represent literals

are missing from the WSDL declaration. The input message is described as containing one element, `?tickerSymbol`, of some `?company` registered at the NYSE (New York Stock Exchange). The output message contains a sequence (`__StockPriceInfo`) with two elements (`__Price` and `__TimeStamp`). The semantics of these elements are described in terms of an instance called `__StockMonetaryValue` of the same `?company` from the input, with a price `__Price` in `USDollar` at `__TimeStamp`. The exact meaning of the notations “?” and “__” will be described later.

The power of this semantic description is that it explicitly describes all the semantic relationships between the different data elements in a message as well as between the input and output messages. It describes the inputs and outputs in terms of instance-based graph patterns. This is in contrast to class-based descriptions that are commonly used in various interface description languages. The instance-based descriptions allow associating rich semantics to the web service, by specifying complex inter-relationships between different instances (for example, the relations between `?tickerSymbol`, `?company` and `NYSE`). Such relationships are more difficult to capture using class-based descriptions without having to create a large number of new classes for different combinations of the relationship constraints.

The semantic descriptions are based on one or more domain ontologies in OWL that define various concepts, properties and instances in a domain of interest. A domain ontology in this scenario may define concepts like `Company` and `Ticker`, properties like `tickerOf` and `fromExchange`, and instances like `NYSE` and `IRS`.

We now describe the semantic model formally. Some of the elements of this model are adapted from SPARQL [12],

a standard RDF query language.

Let U be the set of all URIs. Let RDF_L be the set of all RDF Literals. The set of RDF Terms, RDF_T , is $U \cup RDF_L$. RDF also defines blank nodes, which we do not include in our model. An RDF Triple is a member of the set $U \times U \times RDF_T$. An RDF Graph is a set of RDF triples.

A *variable* is a member of the set V where V is infinite and disjoint from RDF_T . A variable is represented with a preceding “?”.

A *triple pattern* is a member of the set $(RDF_T \cup V) \times U \times (RDF_T \cup V)$. An example is $(?tickerSymbol \text{ ofCompany } ?company)$.

A *graph pattern* is a set of triple patterns.

An *input message pattern* describes the properties of messages required by an operation. It is of the form $IMP(VS, GP)$:

- VS is a set of variables representing the data elements that must be contained in the message. $VS \in 2^V$. If an element is a sequence, it recursively includes all elements in the sequence.
- GP is a graph pattern that describes the semantics of the data elements in the message.

In an output message, an operation may create new objects that did not appear in the input message. In the output message pattern description, these new objects are represented explicitly as *exemplars*. Exemplars act as existentially quantified variables, and are meant to represent example individuals and literals in an example output message. In a specific output message, these exemplars are replaced by RDF Terms. An exemplar individual is represented in OWL as belonging to a special concept called `Exemplar`. An exemplar literal is of a user defined type called `xs:exemplar`, which is derived from `xs:string`. Exemplars are represented with a preceding “__”.

Let E represent the set of all exemplars. An *output message pattern* is of the form $OMP(OS, GP)$:

- OS is a set of variables and exemplars, that represent the data elements that are contained in the output message. $OS \in 2^{V \cup E}$. If an element is a sequence, it recursively includes all elements in the sequence.
- GP is a graph pattern that describes the semantics of the data elements in the output message.

The output message description has a combination of variables and exemplars. Variables represent those entities that were carried forward from the input message description; and exemplars represent those entities that were created by the component in the output message description.

A *web service operation* is described as taking an input graph pattern and transforming into an output graph pattern. It is of the form $W(IMP, OMP)$:

- IMP is an input message pattern that describes the input requirements of the operation.

- *OMP* is an output message pattern, that describes the output of the operation.
- The set of variables in *OMP* is a subset of the set of variables in *IMP*. This helps ensure that no free variables exist in the output description, an essential requirement for the planning process.

The semantics described so far is a *workflow-independent description* of the operation. In a specific workflow, this operation may get certain kinds of messages from another web service and hence, the semantics of its operation will differ.

3 Composing Services into Workflows

The semantic model of web service operations allows us to determine if it is possible to use the output messages of one or more operations and construct a message that can be sent as input to another operation. We first illustrate the composition check through an example, and then provide a formal model for the composition.

Let us consider another web service called *IRS.CorpInfoService*. It supports an operation that takes in an Employer ID Number (EIN) of a Corporation (as granted by the Internal Revenue Service) and returns the profile of the company including its name, address and ticker symbol.

```
<types> <xs:schema>
  <xs:element name="EIN" type="xs:String"/>
  <xs:complexType name="corpProfile">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="address" type="xs:string"/>
      <xs:element name="ticker" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema> </types>

<interface name = "CorpProfileInterface" >
  <operation name="getCorpProfile" >
    <input messageLabel="In" element="EIN" />
    <output messageLabel="Out" element="corpProfile"/>
  </operation> </interface>
```

The semantic description of this service is shown in Figure 2. Now, the question we shall try to answer is whether this service can be composed with the *StockService* in a BPEL workflow that receives an EIN given by *IRS* and replies with the stock price of the corporation that has this EIN. Intuitively, the *getCorpProfile* operation can be used to get the ticker symbol of the corporation, and then the *getStockPrice* operation of the *StockService* can be used to get the stock price. However, an automated workflow composer will not be able to perform this composition just by looking at the WSDL description. The types of the messages in the

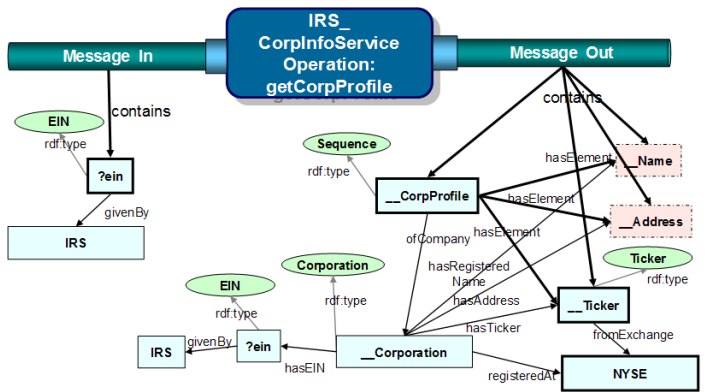


Figure 2. Description of *getCorpProfile* operation that takes in a message containing an ein number and produces a message containing a *CorpProfile* sequence, which has 3 elements: name, address and ticker symbol of a corporation

input and output messages are different for the two operations. In addition, the WSDL description does not describe how exactly the elements of the inputs relate to the outputs.

Our semantic descriptions of the services, however, provide enough information to compose these two services. In addition, it can also be used to generate a copy statement in the BPEL script to copy the ticker element from the output message of the *getCorpProfile* operation to the input message of the *getStockPrice* operation. We shall now describe the necessary conditions for matching the output of one service to the input of another.

Figure 3 shows the composed view of the services and how their semantic descriptions change as a result of the composition. The input to *getCorpProfile* is described using an exemplar that is created by the web service partner that calls the composed service. This exemplar, called *..EIN1*, is described as having been issued by the *IRS*. The message matches the input message pattern requirement of *getCorpProfile*. As a result, the variable *?ein* in the input message pattern is substituted by *..EIN1*, and this substitution is propagated to the output message pattern as well.

The next step is to see whether the output of *getCorpProfile* (or any part of it) can be sent as input to the *getStockPrice* operation. This involves matching the output message with the input message pattern of *getStockPrice*. This matching process, described in detail in Section 3.2, determines that the ticker element of the output message can be extracted and sent as input to *getStockPrice*. The matching process uses description logic reasoning based on ontologies that state, for example, that *Corporation* is a subclass of *Company*. As a result of the matching, the input and output message patterns of *getStockPrice* have their variables replaced by RDF graphs in the workflow dependent description. This description includes additional semantic constraints (like the fact that the corporation has a certain

EIN given by IRS) which are derived from the match on the input and are propagated to the output.

3.1 Workflow Model

A workflow is a graph $G(V, E)$ where G is a DAG (Directed Acyclic Graph). Each vertex $v_i \in V$ is a web service operation. Each edge (u, v) represents a logical flow of messages from u to v . If there is an edge (u, v) , then it means that an output message produced by u is used to create an input message to v . If v has many incoming edges of the form $(u_1, v), \dots, (u_n, v)$, it means that the output messages of u_1, \dots, u_n are all used to create an input message to v . This is done by copying data elements from the output messages of u_1, u_2, \dots, u_n into the input message sent to v .

This model can be represented as a BPEL workflow. A BPEL workflow can be viewed as one request-response operation, i.e. there is one partner web service that makes a request to the workflow and receives a reply from it. We model this partner web service using two vertices in the DAG. One vertex called the source makes the request to the workflow. The source vertex has no incoming edges. Another vertex called the sink receives the reply from the workflow. The sink has no outgoing edges.

The BPEL workflow has one *receive* and one *reply* activity. Each vertex corresponds to an *invoke* activity. In addition, just before the invoke activity is performed, there may be one or more *assign* activities that take in messages from the incoming edges to the node and creates an input message that will be used as input in the invocation of the web service. Depending on the structure of the graph, a number of *flow* and *sequence* activities are used to describe how the invocations to services proceed. We restrict workflows to be DAGs since it is extremely difficult to create plans with cycles. Most AI planners create partially ordered plans, which can be represented as DAGs.

For every vertex u , we define the semantics of a typical or exemplar message that is produced as output by u and that is sent on all outgoing edges from u . This exemplar message description is based on exemplar individuals and literals that appear in this message. An *exemplar message* is of the form $M(E, R)$ where

- E is the set of elements in the message, represented as exemplars
- R is an RDF graph that describes the semantics of the data elements in the message. The RDF graph consists of a set of OWL facts (ABox assertions).

3.2 Matching a set of messages to an input message pattern

Consider vertex v in the workflow DAG. Let v have n incoming edges: $(u_1, v), \dots, (u_n, v)$, i.e. the output mes-

sages of u_1, \dots, u_n are all used to create an input message to v . Let the semantics of the messages on the edge (u_i, v) be described by the exemplar message $M_i(E_i, R_i)$. Let the input requirements of v be described by the input message pattern $P(VS, GP)$. Let O be a common domain ontology that defines the TBox, on which the descriptions of the messages and message pattern are based. O may also define some ABox assertions on non-exemplar individuals.

We say that v is *validly deployed* iff there is a *match* between the exemplar messages, $\{M_1, \dots, M_n\}$, and the input message pattern P . Such a *match* is possible iff there exists a substitution of variables, θ , defined from the set of variables to the set of OWL individuals or literals, such that:

- $\bigcup_{i=1}^n E_i \supseteq \theta(VS)$. That is, the messages contain at least all the elements required in the message pattern
- $\bigcup_{i=1}^n R_i \cup O \models \theta(GP)$ where \models is an entailment relation defined between RDF graphs. That is, it should be possible to infer the graph pattern, after substituting the variables in it, from the RDF graphs describing the different input messages. The actual entailment relation may be based on RDF, OWL-Lite, OWL-DL, OWL-DLP or other logics.

We represent this match as $\{M_1, \dots, M_n\} \bowtie_{\theta} P$, i.e. exemplar messages M_1, \dots, M_n match the message pattern, P , with a substitution function θ . If such a match exists, then, it is possible to copy some of the elements corresponding to $\theta(VS)$ and create an input message to v .

As an example, `getCorpProfile` produces an output message (represented as CP), which matches `getStockPrice`'s input message pattern (represented as GSP). Using a substitution, θ_1 , defined as $\theta_1(?tickerSymbol) = _ticker$ and $\theta_1(?company) = XYZ$, we can see that $CP \bowtie_{\theta_1} GSP$. The entailment derivation process is based on a suitable flavor of OWL (like OWL-DL or OWL-DLP) and uses TBox definitions in the ontology like `Corporation` is a subclass of `Company`, and `tickerOf` is the inverse of `hasTicker`.

3.3 Semantic Propagation

One of the features of our model is semantic propagation - the actual semantic description of the output messages of an operation in a workflow depends on the semantics of the input messages. This propagation is achieved through the use of common variables in the input and output message patterns. The workflow-dependent description of a web service is a specialization of the workflow-independent description where the variables are replaced by RDF graphs that are obtained from the actual input message given to the service. This is a more expressive model than one where variables are only substituted by single values, and allows the propagation of graphs representing semantic constraints from one service to the next in a workflow.

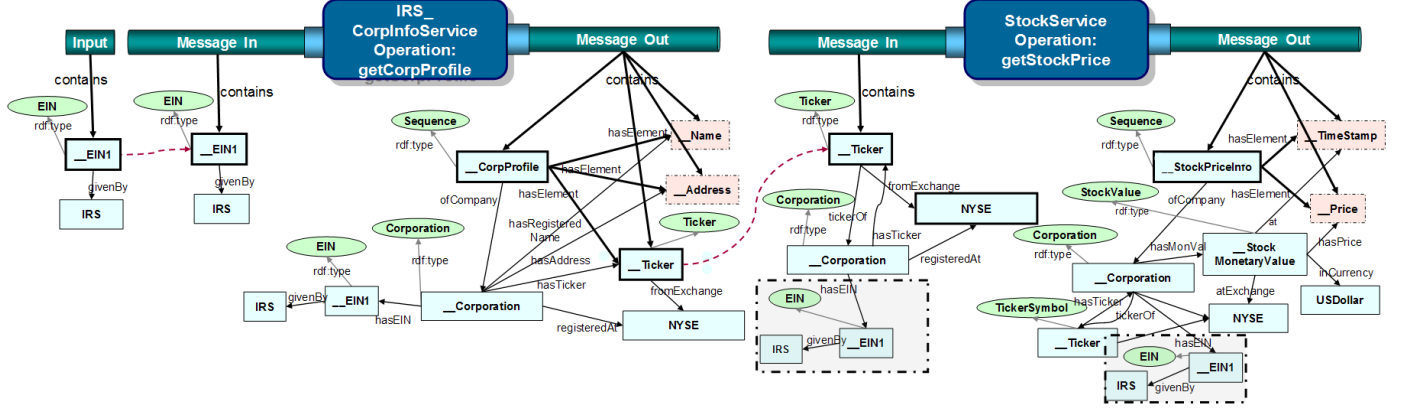


Figure 3. Workflow-dependent semantic descriptions of the services from Figures 1 and 2, where the variables are replaced by graphs based on matches obtained between the messages and the input message patterns. The dashed arrows represent copy statements that extract data elements from one message to initialize the input message to another operation. The shaded rectangle shows a portion of the workflow-dependent description of the *StockService* that does not appear in the workflow-independent description in Figure 1 and has been derived from the previous service. Not all links are shown for sake of readability

Figure 3 has an example where variables like *?ticker*, *?company* and *?ein* that appear in the workflow-independent descriptions in Figures 1 and 2 are replaced by graphs that include the nodes *__Ticker*, *__Corporation* and *__EIN1*. Also, the workflow-dependent description of the *StockService* has additional constraints (represented by the subgraph containing *__EIN1* and *__IRS*) that don't appear in the workflow-independent description.

Formally, the description of the output message is generated using a *graph transformation* operation [2], that combines the description of the input message with the output message pattern. Graph transformations have been used in software engineering to describe the behavior of components in terms of transformations of graphs like UML object diagrams (see [2] for a survey). We apply these ideas to describe the behavior of components based on transformation of RDF graph patterns.

Let L be the input graph-pattern and R the output graph-pattern of the operation. Now, assume that in an actual workflow, L has been matched to a message with an RDF graph, X , with the variable substitution function, θ . Let the output message for this workflow contain the RDF graph, Y . We determine Y using a graph homomorphism, f .

$$f : \theta(L) \cup \theta(R) \rightarrow X \cup Y$$

In our model, f satisfies the following properties:

1. $f(\theta(L)) \subseteq X$. This means that the substituted input graph-pattern is a subgraph of the graph describing the message matched to it. This follows from the entailment relation obtained between the input message-pattern and the message.
2. $f(\theta(R_i)) \subseteq Y_i$. This means that the substituted output graph-pattern is a subgraph of the output message.
3. $f(\theta(L) \setminus \theta(R)) = X \setminus Y$ and $f(\theta(R) \setminus \theta(L)) = Y \setminus X$

where \setminus represents the graph difference operation. This means that exactly that part of X is deleted which is matched by elements of $\theta(L)$ not in $\theta(R)$, and exactly that part of Y is created that is matched by elements new in $\theta(R)$.

Using properties 2 and 3, it is possible to determine the output, Y , of an operation as a result of giving X as input in a certain workflow. This operation is performed in two main steps. In the first step, we remove all the edges and vertices from X that are not present in L , but not present in R . In the second step, we glue in vertices and edges that are newly created in R .

4 The Semantic Planner

We have developed a semantic planner that can compose a BPEL workflow automatically from the semantic descriptions of services with the aid of description logic reasoning. A workflow to be composed is described as a high-level web service that takes in a request and produces a response. The request is described as an exemplar message, $Req(E, R)$, and the response message is described as an input message pattern, $Res(VS, GP)$. For example, the request message may be described as containing one element, which is an EIN given by IRS and the response message must contain one element: the stock price of a corporation that has this EIN. The output of the composition problem is a workflow $G(V, E)$, such that:

- The workflow has a source vertex, v_{src} , and all outgoing edges from v_{src} are associated with the semantics defined in the request message, $Req(E, R)$.
- The workflow has a sink vertex, v_{snk} , such that there is a match between the exemplar messages associated

with the incoming edges and the response message pattern, $Res(VS, GP)$.

- All the other vertices in the workflow represent web service operations that are validly deployed.

The composition problem can be modeled as a planning problem. Each web service operation, $W(P_i, P_o)$, is an action. The preconditions of the action include the constraints on the input message, as specified in P_i . The effects of the action involve creating a new message, the properties of which are described in P_o . The initial state in the planning problem contains only the request message, $Req(E, R)$. The goal of the planning problem is to create a message that satisfies the response message pattern, $Res(VS, GP)$.

Our planner models the current state as the set of messages created so far. At a high level, the planner works by checking if a set of messages available in the current state can be used to construct an input to an operation, and if so, it generates a new message corresponding to the output. It performs this process recursively and keeps generating new messages until it produces one that matches the goal pattern, or until no new unique messages can be produced.

Description logic reasoning during planning is useful since it allows the planner to match messages to input message patterns even if they are described using different terms and different graph structures. However, this introduces some challenges. One challenge is scalability: invoking a reasoner during planning is expensive. Another challenge is that any reasoner used in this process must keep all facts in the description of one message independent of the facts in the description of other messages; and facts across different messages cannot be combined to infer any additional facts. During the planning process, a very large number of messages may be generated by connecting services into partial workflows. Available reasoners cannot maintain the facts of such a large number of different messages independently and answer queries on individual messages.

To overcome these challenges, the planner employs a two-phase approach to plan building. In the first phase, which occurs offline, it does pre-reasoning on the output descriptions of different operations. During this phase, it does DLP-reasoning [5] on the descriptions of components and translates them into a language called SPPL (Stream Processing Planning Language) [14]. SPPL is a variant of PDDL (Planning Domain Definition Language) and is specialized for describing stream-based planning tasks (a stream can be considered to be a generalization of a message). It models the state of the world as a set of streams and different predicates are interpreted only in the context of a stream. The SPPL descriptions of different components are persisted and re-used for multiple goals. The second phase is triggered whenever a goal is submitted to the system. During this phase, the planner performs branch-and-bound forward search by connecting all compatible operations in a

Table 1. Plan time (sec).

Operations	Messages	Prereason	Plan
10	15	4.66	0.15
20	40	10.29	0.22
30	119	27.49	0.76
40	190	38.48	1.21
50	288	59.17	2.26
100	1,204	233.03	28.73

DAG. It uses some optimizations by analyzing the problem structure and removing operations that cannot contribute to the goal to help restrict the search space. Further details of this plan-search phase can be found in [14].

A key feature of our planning process is that DLP reasoning is performed only once for a component, in an offline manner. During actual plan generation, the planner does not do any reasoning. It only does subgraph matching; it tries to find a substitution of variables so that the input message graph pattern of a component can be matched to the new object graph of a stream. This allows the matching process to be faster than if reasoning was performed during matching. In addition, it eliminates the need for a reasoner that has to maintain and reason about independent stream descriptions during the plan-building process. The reasoner is only invoked when a new component is added to the system. The reason for the choice of DLP (which lies in the intersection of Datalog and DL) for reasoning is that it allows us to enumerate all inferred facts from a given set of facts and is hence suitable for this two-phase planning approach.

5 Evaluation

Since there are no standard datasets for web service composition, we evaluate planner performance by measuring planning time on increasingly large randomly generated sets of operations. Experiments were carried out on a 3GHz Intel Pentium 4 PC with 500 MB memory. For our experiments, we generated random DAG workflows that contained services with randomly constructed semantic descriptions, and then evaluated the time it took to plan these workflows. The DAGs were generated by distributing the nodes randomly inside a unit square, and creating a link from each node to any other node that has strictly higher coordinates in both dimensions with probability 0.4. The resulting connected components are then connected to a single source and sink node. Each link is associated with a randomly generated RDF graph from a financial services ontology in OWL that had about 200 concepts, 80 properties and 6000 individuals. The planner uses the DLP reasoner, Minerva [19], during the pre-reasoning phase. The time taken to plan the DAGs are shown in Table 1. The table has columns for the number of services and exemplar messages in the generated graph, as well as time measurements for the on-line and offline phases of semantic planning.

The experiments show that our pre-reasoning approach

makes semantic planning practical by improving planner scalability. Although pre-reasoning is time consuming, the results of pre-reasoning can be shared between multiple planning requests. Therefore, the observed response time of the planning system in practice is close to planning phase time. We can see that even for workflows involving 100 operations, the planner is able to produce the plan in less than 30 seconds, which is an acceptable performance.

6 Related Work

Existing web service composition works use various techniques including AI planning and reasoning based on different kinds of logics. Sirin et al [16] combine the JShop planner with the Pellet DL reasoner to compose semantic web services described in OWL-S. Heflin et al [6] describe a planner that integrates HTN planning with ontologies written in SHOE, which is based on Datalog. Other composition approaches such as [9, 15, 3, 18] work on semantic web service descriptions like OWL-S or DAML-S. including the process model of a service. These approaches model services in terms of their preconditions and effects on the state of the world and inputs and outputs based on concepts. Still others start with a process model defined in BPEL or as transition systems to compose services [11, 10]. The main novelty of our approach is in modeling the input and output messages as instance-based graph patterns, which are very expressive. OWL-S, DAML-S and BPEL models do not allow expressions with variables in describing inputs and outputs. Our model allows logical expressions with variables in the form of graph patterns to describe inputs and outputs, and also to relate the semantics of outputs to the semantics of inputs and allow semantic propagation.

Akkiraju et al [13] use cues from ontologies and term-matching to compose services. Lécué and Léger [7] use causal link matrices to model input-output matches based on similarity between the input and output concepts. We allow more expressive input and output descriptions using logical expressions with variables. Shivashanmugam et al [17] propose semantic process templates to capture the semantic requirements of a process. Their work requires the existence of a template, while our planning approach only requires a goal specification for composition.

7 Conclusion and Future Work

In this paper, we have presented a semantic graph transformation model for describing web service operations that allows specifying complex constraints on inputs and outputs. The key aspects of this model are a) Instance-based graph pattern description of inputs and outputs, b) Incorporation of notions of semantic propagation and having workflow-independent and workflow-dependent descriptions of services, and c) Determining composability

of operations using description logic reasoning. We have also described our planner that incorporates DLP reasoning while composing workflows.

In future work, we are extending the model to include descriptions of preconditions and effects of an operation on the state of the world. We are also enhancing our planner to allow generating plans with loops.

References

- [1] R. Akkiraju *et al.* Semantic annotations for wsdl and xml schema. *W3C Candidate Recommendation*, 2007.
- [2] L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *1st Int. Conference on Graph Transformation*, 2002.
- [3] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, 2005.
- [4] de Bruijn. The Web Service Modeling Language WSML. <http://www.wsmo.org/wsml/>, 2005.
- [5] B. Grosz, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *WWW'03*.
- [6] J. Heflin and H. Munoz-Avila. LCW-based agent planning for the semantic web. In *Ontologies and the Semantic Web, AAAI Workshop*, 2002.
- [7] F. Lécué and A. Léger. A formal model for semantic web service composition. In *ISWC'06*, 2006.
- [8] D. Martin *et al.* OWL-S: Semantic markup for web services. In *W3C Submission*, 2004.
- [9] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, 2002.
- [10] J. Pathak, S. Basu, and V. Honavar. Modeling web services by iterative reformulation of functional and non-functional requirements. In *ICSOC*, 2006.
- [11] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS web service. In *ICWS*, 2005.
- [12] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. In *W3C Working Draft*, 2006.
- [13] R. Akkiraju *et al.* Semaplan: Combining planning with semantic matching to achieve web service composition. In *ICWS*, 2006.
- [14] A. Riabov and Z. Liu. Planning for stream processing systems. In *AAAI*, 2005.
- [15] M. Sheshagiri, M. desJardins, and T. Finin. A planner for composing services described in DAML-S. In *Web Services and Agent-based Engineering - AAMAS*, 2003.
- [16] E. Sirin and B. Parsia. Planning for Semantic Web Services. In *Semantic Web Services Workshop at 3rd ISWC*, 2004.
- [17] K. Shivashanmugam, J. Miller, A. Sheth, and K. Verma. Framework for semantic web process composition. *Special Issue of the Interl Journal of Electronic Commerce*, 2003.
- [18] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *ISWC'04*.
- [19] J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, and Y. Pan. Minerva: A scalable OWL ontology storage and inference system. In *1st Asian Semantic Web Symp.*, 2004.