

# A Planning Approach for Message-Oriented Semantic Web Service Composition

Zhen Liu, Anand Ranganathan and Anton Riabov

IBM T.J. Watson Research Center, Hawthorne, NY, USA

{zhenl, arangana, riabov}@us.ibm.com

## Abstract

In this paper, we consider the problem of composing a set of web services, where the requirements are specified in terms of the input and output messages of the composite workflow. We propose a semantic model of messages using RDF graphs that encode OWL ABox assertions. We also propose a model of web service operations where the input message requirements and output message characteristics are modeled using RDF graph patterns. We formulate the message-oriented semantic web service composition problem and show how it can be translated into a planning problem. There are, however, significant challenges in scalably doing planning in this domain, especially since DL reasoning may be performed to check if an operation can be given a certain input message. We propose a two-phase planning algorithm that incorporates DLP reasoning and evaluate the performance of this planning algorithm.

An important class of web services consists of those that either do data processing or provide information, i.e. they take in messages containing input data, process them in some manner, and produce messages containing output data or results. In this paper we propose a novel way of associating rich semantic information with messages and web service operations. Our model describes messages using RDF graphs that encode OWL ABox assertions. It also describes the input message requirement and the output message description of each operation using RDF graph patterns. The terms used in these patterns are defined in OWL ontologies that describe the application domain.

The main motivation behind this model is to allow automatic composition of workflows that process, transform or analyze data to produce some desired information. In such workflows, it is necessary to have expressive models of messages and of the data processing capabilities of services so as to compose services that are semantically compatible and to create workflows that produce the desired information. We formulate the message oriented service composition problem as one of producing messages that satisfy certain semantic conditions, from certain initial input messages. We show how this problem can be cast as a planning problem.

Many existing service models (like OWL-S (Martin *et al* 2004)) do not allow expressions with variables in describing inputs and outputs. OWL-S describes inputs and outputs using concepts in an ontology. Similarly, SA-WSDL (Akkiraju *et al* 2005), which allows linking semantic annotations to WSDL files, is also typically used to associate inputs and outputs with concepts in an ontology. Our model describes inputs and outputs using logical expressions with variables, and can relate the semantics of outputs to the semantics of inputs. These expressions describe constraints on data instances present in a message. In this sense, it is similar to WSML (de Bruijn 2005) which also allows specifying axioms with variables in the pre- and post-conditions of a web service capability. Our model also has a semantic description of messages. It explicitly defines the elements in a message and describes the semantics of these elements using an RDF graph. 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 a web service and how it can construct these messages from other messages that have been produced by other services in a partial plan.

There are many differences between our message-oriented composition problem and the problem tackled by many existing works, e.g. (Sirin & Parsia 2004; Traverso & Pistore 2004; Narayanan & McIlraith 2002), where the requirements are specified in terms of the initial and goal states of the world. The difference is that in our model, the world consists of a number of mutually independent messages, and new messages may be produced when operations are invoked. In this paper we explain further differences.

One challenge with planning concerns the use of description logic reasoning for checking if two components can be connected. Since a planner needs to perform a large number of such checks while building a plan, this results in a large number of calls to a reasoner during plan-building. We describe a planner that overcomes this challenge by using a two-phase approach where pre-reasoning is performed once offline, and plan-building is performed online for each new request, without the use of reasoning. The pre-reasoning is done based on DLP (Description Logic Programs) (Grosz *et al.* 2003), and it produces additional inferred facts based on the descriptions of operations. The plan-building process makes use of the original and inferred facts when checking compatibility of operations and messages.

The key contributions of this paper are the investigation of message-oriented model of services and service composition, the identification of some of the challenges that arise in using planning for composition, and the design of a two-phase planner that tackles these challenges. We also present evaluation results of the planner.

## Workflow and Service 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  $e(u, v)$  represents a logical flow of messages from  $u$  to  $v$ . If there is an edge  $e(u, v)$ , then it means that an output message produced by the operation  $u$  is used to create an input message to the operation  $v$ . If  $v$  has many incoming edges of the form  $e_1(u_1, v), e_2(u_2, v), \dots, e_n(u_n, v)$ , it means that the output messages of  $u_1, u_2, \dots, u_n$  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 to  $v$ .

Note that an edge only represents a logical flow of messages, and not necessarily an actual flow. In the case of a BPEL workflow, the flow may occur through the coordinating workflow process, which gets the output message from the first operation, and copies some or all of the data elements from this output message into the input message to the second web service operation.

Typically, a whole workflow can be viewed as one request-response operation on a WSDL portType defined for the workflow, 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. All the remaining vertices in the DAG represent other web services that the workflow may call using a request-response paradigm. One of the main reasons why we restrict the workflow model to a DAG is that it is extremely difficult to create plans with cycles. Most AI planners create partially ordered plans, which can be represented as DAGs.

For now, we assume that the definition of a message in the WSDL description of a web service includes only simple atomic XML types or complex types that are sequences. This restriction allows us to model the collection of elements in a message as a set.

We now describe the model of messages. WSDL describes elements in a message using types in XML Schema. We extend this model to describe additional semantics associated with the elements. The semantics of the elements are described as an RDF graph consisting of OWL facts.

The semantics describe a message that may be produced as output by a web service operation in a workflow. This message is of the form  $M(E, R)$  where

- $E$  is the set of elements in the message.
- $R$  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).

One challenge with such a description is that different messages produced as output by a web service operation may have different values of the elements. Hence, we provide a layer of abstraction over the actual values and define the data elements of an exemplar message that contains exemplar individuals or exemplar literals. 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`.

Exemplar individuals and literals act as existentially quantified variables. In a particular message, they may be substituted by a value that belongs to the set of non-exemplar individuals (i.e. an OWL individual not belonging to the concept Exemplar) or non-exemplar literals (i.e. a literal that is not of type `xs:exemplar`). In this paper, we represent all exemplar individuals and literals with a preceding “\_”.

For example, consider a message described by the complex type `corpProfile`:

```
<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>
```

While the XML schema definition provides the general format of a message, messages produced by an operation in a workflow have more specific semantics. For example, the corporation profile may be of a specific corporation, XYZ, and the ticker from the New York Stock Exchange. An example description, in an N3-based syntax is below.

```
import http://www.example.com/corporation.owl;
Sequence _corpProfile
  ContainsElements _name, _address, _ticker
  WithSemanticDescription {
    _corpProfile a Exemplar; a CorporationRecord;
    ofCompany XYZ.
    XYZ a Corporation; hasRegName _name; hasTicker _ticker;
    hasAddress _address; registeredAt NYSE.
    _ticker a Exemplar; a StockTicker; fromExchange NYSE. }
```

The description imports definitions of different concepts, properties and individuals from OWL ontologies. Note that the model of a message is an instance-based description, i.e. it describes the semantics based on exemplar individuals and literals. This allows us to specify relationships between the different instances in a message. It is more difficult to describe such relationships using a pure class-based description, that only describes messages based on the classes of data they contain.

Based on this model of a message, we now describe the semantics of a web service operation. A web service operation requires an input message that satisfies certain properties, processes it in some way and produces an output message that satisfies certain other properties. These input requirements and output characteristics are specified in terms of *message patterns*.

A *message pattern* is of the form  $P(VS, GP)$  where

- $VS$  is a set of variables and exemplars that correspond to the elements that must be present in a message;

- $GP$  is an RDF graph pattern that describes the semantics of the variables and exemplars, i.e. it describes constraints on their values and the relationships between them.

A *Web Service Operation* is of the form  $W(P_i, P_o)$  :

- $P_i$  is a message pattern that describes the kind of messages required as *input* by the operation;
- $P_o$  is a message pattern that describes the kind of messages produced as *output* by the operation;
- The set of variables in  $P_o$  is a subset of the set of variables that are described in  $P_i$ . This helps ensure that no free variables exist in the output description, an essential requirement for the planning process.

Consider an example web service operation that takes as input a ticker symbol and produces as output a message containing a price and a time stamp. The semantics of the input and output message patterns are shown below:

```

import http://www.example.com/corporation.owl;
Operation getStockPrice
RequiresInputMessage ?tickerSymbol
WithSemanticDescription {
  ?tickerSymbol a StockTicker; tickerOf ?company.
  ?company a Company; registeredAt NYSE. }
ProducesOutputMessage
Sequence __stockPriceInfo
ContainsElements _price, _timeStamp
WithSemanticDescription{
  __stockPriceInfo a Exemplar; a StockRecord;
  ofCompany ?company.
  ?company a Company; hasPrice __stockMonetaryValue;
  registeredAt NYSE.
  ?tickerSymbol a StockTicker; tickerOf ?company.
  __stockMonetaryValue atTime _timeStamp;
  fromExchange NYSE; hasPrice _price;
  inCurrency USDollar; a Exemplar; a StockValue. }

```

The variables in the output message description are those that are carried forward from the input message pattern. The exemplars in the description represent elements that are newly created by the operation. In an actual workflow, the variables will be instantiated depending on the messages sent as input to the operation, and the output message pattern becomes the description of an exemplar output message.

This service model only describes the input and output messages of web service operations. It does not describe the internal state of the web service. Also, it does not describe any additional preconditions and effects that the service may have on the state of the world. Our model is well suited for describing services that are either stateless or if they do maintain state, then their state does not influence the semantic description of the input or output messages. Examples of such services include information-providing services and services that do data transformation or analysis.

## Composition Model

We now describe the conditions that must be satisfied by the web service operations in a workflow graph. These conditions determine whether the semantics of the exemplar messages produced by a certain set of operations in the workflow

are compatible with the semantics of the messages required by other operations.

We define that a workflow is valid if the messages that are sent as input to any operation satisfy the conditions described in the input message pattern of that operation. Consider a workflow DAG,  $G(V, E)$ . The input message sent to the operation corresponding to a vertex in the DAG is constructed using the messages received on each of the incoming edges to the vertex, from the previous operations in the workflow. This construction is based on copy statements; i.e. the different elements of the input message are copied from elements of the messages received on the incoming edges.

Consider vertex  $v$  in the DAG. If  $v$  has many incoming edges of the form  $e_1(u_1, v), e_2(u_2, v), \dots, e_n(u_n, v)$ , it means that the output messages of  $u_1, u_2, \dots, u_n$  are all used to create an input message to  $v$ . Let the semantics of the messages on the edge  $e_i(u_i, v)$  be described using the exemplar message  $M_i(E_i, R_i)$ . Let the input requirements of the operation  $v$  be described using the 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  $P$ . We define that there is a *match* between a set of exemplar messages and an input message pattern iff there exists a substitution of variables,  $\theta$  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 substituted graph pattern 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,  $MP$ , with a substitution function  $\theta$ . If such a match exists, then, it is possible to copy the elements corresponding to  $\theta(VS)$  and create an input message to the operation  $v$ .

As an example, the message `__corpProfile` (represented as CP), described in Page 2, matches the input message pattern of `getStockPrice` (represented as GSP). Using a substitution,  $\theta_1$ , 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.

We now formulate the message oriented semantic web service composition problem. A workflow can be viewed as one request-response operation. The requirements of the composition problem are specified in terms of the semantics of the request and response messages. The request message is described using an exemplar message, and the response

message is described using a message pattern.

The input to the composition problem is an exemplar request message  $Req(E, R)$  and a response message pattern  $Res(VS, GP)$ . 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,  $v_{sink}$ , 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 modeled as an action. The preconditions of the action include the constraints on the kinds of input messages the operation accepts, as specified in the input message pattern,  $P_i$ . The effects of the action involve creating a new message, and the properties of this new message are as described in the output message pattern,  $P_o$ .

The initial state in the planning problem contains only the exemplar request message,  $Req(E, R)$ , which is described as containing a set of elements with properties as defined in  $R$ . The goal of the planning problem is to create a message that satisfies the response message pattern,  $Res(VS, GP)$ .

### Differences with AI Planning

While our web services composition problem is, in many respects, similar to AI planning, there are a number of structural properties of this problem that are difficult to express efficiently in existing planning formalisms, such as PDDL. The key differences are:

1. The state of the world is partitioned into a number of mutually independent messages. Each ground predicate describes one and only one message, and no predicate can connect two or more messages. Similarly, the preconditions and effects of actions are based on predicates that are defined in the context of a message.
2. Actions produce new messages. As a result, the set of exemplar messages grows as more and more actions are applied. This again is in contrast to the PDDL model where the set of objects in the world is the same in all states.
3. Actions cannot modify any previously existing messages. One consequence is that actions have no delete effects. This, however, does not make the problem simpler, as it happens in AI planning, due to the increasing state space.
4. DL reasoning (or some subset thereof) is required to match a set of messages to a message pattern. This is in contrast to STRIPS-based planning systems where the reasoning during evaluation of preconditions is limited.

One significant challenge comes from the fact that traditional planners are not optimized for the domains where actions produce new objects. The results obtained for the Settlers domain in IPC-3 planner competition have shown that although it is possible to model actions producing new

objects in PDDL (by adding “potential” objects to the initial state), planners tend to perform worse on domains with this structure. When represented in PDDL, these planning problems become very symmetric. In addition, many planners ground actions before planning, and therefore create a large number of unused actions when not all of the potential objects end up being used in the final plan. New approaches to planning are required to address these issues, and in our system we use an extended version of PDDL to represent actions that create new objects.

Combining DL reasoning with planning presents new scalability challenges. The planning process involves a large number of checks to see if different candidate sets of messages can collectively satisfy the input message requirements of actions. For instance, if we implement a forward-search planning approach, as used in many successful planners, we need to look at the current set of messages that have been produced so far, and see which actions have preconditions satisfied. If during the planning process, a partial plan with  $N$  instantiated actions has been created, then at least  $N$  new messages will be created, and  $O(N^2)$  preconditions will have been evaluated (checking each action with each message). Moreover, there could exist up to  $O(N!)$  different partial plans of length  $O(N)$  constructed with those actions, which would require proportionally more precondition evaluations. This simple back-of-the-envelope calculation makes it clear that being able to evaluate preconditions quickly is critical to planner performance.

The use of traditional DL reasoners to evaluate preconditions during planning, apart from the scalability challenge, also presents a practical representation challenge. Since the different messages are independent of one another, and predicates are defined in the context of a message, any reasoner that is used during planning must keep the different message descriptions independent of one another, possibly in separate knowledge bases. This is necessary so that inferences are not made across predicates defined on different messages. It is also possible that combining facts across different knowledge bases may cause inconsistencies; hence, the message descriptions must be kept independent. Since OWL only allows binary predicates, it is not possible to add the message context information to the predicates themselves, without defining a new class for the relation (e.g. as described in (Noy & Rector 2004)). However, this approach doesn't allow reasoning based on the original relations. Hence, the reasoner must be capable of reasoning about a large number of messages that refer to common ontologies, and keep those messages independent.

It is necessary, therefore, either to have a very efficient reasoner that is tightly integrated with the planner and enhanced to support multiple message descriptions, or to reduce the number of reasoner invocations, or even avoid calling the reasoner during planning. In this paper we take the last approach.

### Two-Phase Approach

As a result of these issues, we have developed a planner that employs a two-phase approach to plan building. It consists of two components: a Domain-Generator and a Plan-

Builder. In the first phase, which occurs offline, the Domain-Generator translates descriptions of operations (represented in an N3 syntax as described earlier) into a language called SPPL (Stream Processing Planning Language) (Riabov & Liu 2006). SPPL is a variant of PDDL and is specialized for describing stream-based planning tasks. A stream can be considered to be a generalization of a message. SPPL models the state of the world as a set of messages and interprets different predicates only in the context of a message. During the translation process, the generator also does DLP-reasoning on the output descriptions to generate additional inferred facts about the outputs. The SPPL descriptions of different services are persisted and re-used for multiple requests. The second phase is triggered whenever a composition is required. In this phase, the Plan-Builder translates the request into an SPPL planning goal, and produces a plan.

A key feature of our planning process is that DLP reasoning is performed only once for a service, in an offline manner. During actual plan generation, the Plan-Builder does not do any reasoning. It only checks graph-embedding relationships; it tries to find a substitution of variables so that the input message graph pattern can be embedded in the graph that describes a message. 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 message descriptions during the plan-building process. The reasoner is only invoked when a new service is added to the system.

**Pre-reasoning and SPPL Generation.** The Domain-Generator performs DLP-reasoning on the output message graph patterns of different services. Inference on the ABox in DLP can be performed using a set of logic rules. This allows us to take a certain assertion and enumerate all possible assertions that can be inferred from this assertion and an ontology using the rules. The ability to enumerate all inferences is a key reason for the choice of DLP for reasoning. It is not possible to enumerate all inferences in many more expressive DLs.

Since we cannot directly do reasoning on variables, we convert them into OWL individuals that belong to a special concept called Variable. Using this process, a graph pattern can be converted into an OWL/RDF graph, and additional facts about variables and exemplars can be inferred.

Pre-reasoning gives us an expanded message description, which contains an RDF graph that has been expanded with the results of reasoning. Reasoning is done by applying the DLP logic rules (Grosz *et al.* 2003) recursively, in a bottom-up fashion, on the triples in the original graph, and generating additional triples about variables and exemplars, until a fix point is reached. For example, consider the output of the `getStockPrice` operation. The expanded message description includes additional facts like `(?company hasTicker ?tickerSymbol)` (since `hasTicker` is defined to be an inverse of `tickerOf` in the ontology). The complexity of ABox reasoning in DLP is polynomial in the number of facts in the KB. Here, the KB consists of the facts in a message description and the associated ontology.

After pre-reasoning, the expanded descriptions of services

are represented as an SPPL domain, and stored for later use. Concepts used in the descriptions are mapped to SPPL types. Subclass relationships between concepts are also captured in SPPL, which supports multiple inheritance. The set of SPPL predicates includes all OWL properties in the descriptions. The set of SPPL objects include all literals, RDF Terms and exemplars in the descriptions. Finally, each operation is translated into an SPPL action.

**Plan-building** A composition requirement received by the planner is translated into an SPPL problem. The SPPL model yields a recursive formulation of the planning problem, where goals are expressed similarly to service input requirements, and they are matched to streams produced as outputs by services. The Plan-Builder operates in two phases: a presolve phase and a plan search phase (Riabov & Liu 2006). The planner is capable of finding optimal plans for user-defined additive plan quality metrics, such as the number of services included in the plan.

During the presolve phase, the planner analyzes the problem structure and removes services that cannot contribute to the goals, to help restrict the search space. It also translates predicate formulation into a propositional one before searching for the plan. The translation is done by creating all reachable bindings for action parameters, and creating propositional actions with these bindings such that only ground predicates appear in inputs and outputs.

In plan search phase, the planner performs branch-and-bound backward search. The branch-and-bound nodes are expanded backwards, starting from the open goals, leading to the creation of new sub-goals. The sub-goals may be satisfied by existing messages or by outputs of new action instances. A sub-goal is satisfied when all predicates of the sub-goal appear in the connected message. All possible choices for satisfying the goals are explored, unless exploring the goals violates quality bounds. To reduce the time required for finding action candidates for satisfying a sub-goal, during presolve the planner generates an index of compatible action preconditions and effects. All pairs of effects and preconditions that cannot be proven incompatible before planning are added as possibly compatible to the index, and more detailed investigation of compatibility is performed during planning.

Our two phase matching process, consisting of pre-reasoning and sub-graph matching, is sound. If it decides that a message matches an input message pattern, then this match is correct since the message description only contains facts present in the original description or inferred after DLP reasoning, which is sound. However, our matching process is not complete. The planner builds the description of new messages by instantiating the expanded output message pattern based on the substitution of variables obtained when matching the input pattern. Since reasoning is only performed offline on output message patterns, it is possible that the description of the new message may not contain all the facts that can be inferred by DLP reasoning. In our design, we sacrifice completeness for performance. Since we avoid reasoning during plan-building, the matching of messages to services becomes simpler, which helps scalability.

Table 1: Pre-reasoning and planning time (sec).

Experiment 1				Experiment 2		
Srvcs	Msgs	Prereason	Plan	Srvcs	Prereason	Plan
10	15	4.66	0.15	27	13.99	0.24
20	40	10.29	0.22	42	25.01	0.69
30	119	27.49	0.76	72	47.21	1.74
40	190	38.48	1.21	102	67.48	3.06
50	288	59.17	2.26	162	107.88	7.75
100	1,204	233.03	28.73	312	224.85	29.20

## Evaluation

To evaluate planner performance, we measured planning time on increasingly large randomly generated sets of services. Experiments were carried out on a 3GHz Intel Pentium 4 PC with 500 MB memory. We used the DLP reasoner Minerva (Zhou *et al.* 2004) for the offline reasoning.

For our experiments, we generated random DAG plans, with one service for each node in the DAG, and one source and one sink node. The DAGs were generated by distributing the nodes randomly inside a unit square, and creating an arc from each node to any other node that has strictly higher coordinates in both dimensions with probability 0.4. Each link is associated with a randomly generated stream pattern that includes an RDF graph pattern built from a financial services ontology in OWL that had about 200 concepts, 80 properties and 6000 individuals. The time (in seconds) taken to plan the DAGs given a goal that represents the output messages received by the sink node are shown in Table 1. The table has columns for the number of services and messages in the generated graph, as well as time measurements for the online and offline phases of planning.

In Experiment 1 all generated web services were used in the same plan. In practice, however, the processing graphs are likely to be of relatively small sizes, but there could be many services that are not used in the produced processing graph. To model this scenario, we generated a large number of services, with only 2 candidate plans graphs of 6 nodes each. The results are presented in Table 1, Experiment 2.

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 plan graphs involving 100 operations, the planner is able to produce the plan in less than 30 seconds, which is acceptable for the end-user.

An interesting comparison would be between our planner and one that employed a one-phase approach by interleaving planning and reasoning. However, as described earlier, we cannot easily use a DL reasoner during plan-building because of the challenge of maintaining different message descriptions independently, and hence, cannot provide performance comparisons with such a planner at this moment.

## Related Work and Conclusion

Many existing web service composition techniques use planning technologies. Approaches like (Narayanan & McIlraith 2002; Traverso & Pistore 2004; Sirin & Parsia 2004)

work on semantic web service descriptions like OWL-S or DAML-S and some also model the internal state of the services. These approaches model services in terms of their preconditions and effects on the state of the world and inputs and outputs based on concepts. The main novelty of our approach is in modeling the input and output messages as instance-based graph patterns. OWL-S and DAML-S 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. In (Lécué & Léger 2006), the authors use causal link matrices to model input-output matches based on similarity between input and output concepts. We allow more expressive input and output descriptions using logical expressions with variables.

In this paper, we have presented a semantic model for describing operations that allows specifying complex constraints on inputs and outputs. We have also described our planner for composing workflows from components described using this model, and have presented evaluation results. As future work, we are working on removing some of the current restrictions of our workflow model, so as to be able to construct workflows that include conditional branches and cycles. We are also working on extending our model to describe service preconditions and effects on the state of the world.

## References

- Akkiraju *et al.*, R. 2005. Web service semantics - WSDL-S. *W3C Submission*.
- de Bruijn. 2005. The Web Service Modeling Language WSMML. <http://www.wsmo.org/wsml/>
- Grosz, B.; Horrocks, I.; Volz, R.; and Decker, S. 2003. Description logic programs: combining logic programs with description logic. In *WWW'03*.
- Lécué, F., and Léger, A. 2006. A formal model for semantic web service composition. In *ISWC'06*.
- Martin *et al.*, D. 2004. OWL-S: Semantic markup for web services. In *W3C Submission*.
- Narayanan, S., and McIlraith, S. 2002. Simulation, verification and automated composition of web services. In *WWW'02*.
- Noy, N., and Rector, A. 2004. Defining n-ary relations on the Semantic Web. <http://www.w3.org/TR/swbp-n-aryRelations/>
- Riabov, A., and Liu, Z. 2006. Scalable planning for distributed stream processing systems. In *ICAPS'06*.
- Sirin, E., and Parsia, B. 2004. Planning for Semantic Web Services. In *Semantic Web Services Workshop at 3rd ISWC*.
- Traverso, P., and Pistore, M. 2004. Automated composition of semantic web services into executable processes. In *ISWC'04*.
- Zhou, J.; Ma, L.; Liu, Q.; Zhang, L.; Yu, Y.; and Pan, Y. 2004. Minerva: A scalable OWL ontology storage and inference system. In *1st Asian Semantic Web Symp.*