Web Service Composition as Planning

Mark Carman and **Luciano Serafini** and **Paolo Traverso**
ITC-IRST, Via Sommarive 18 - Loc. Pantè, I-38050 Povo, Trento, Italy
{carman,serafini,traverso}@irst.itc.it

Abstract

We show how the service composition problem can be viewed as a planning problem in which state descriptions are ambiguous and operator definitions are incomplete. We then discuss the problem of interpreting documents (which describe the world state), and introduce a semantic type matching algorithm. The matching algorithm together with an interleaved search and execution algorithm allow for basic automated service composition.

Introduction

The ability to perform automated or semi-automated service composition would revolutionise many application areas for web service technology including e-commerce and systems integration. A first step toward such automated composition is to map the problem into that of planning. By doing so we can see what planning techniques (such as the various backtracking search algorithms) are already available to tackle this problem, as well as the new techniques that will need to be developed.

In this paper we attempt to view the service composition problem as a planning problem. We first describe what an abstract planning problem is, and then describe how abstract planning constructs map to the domain of the web services. We discuss ways of planning and executing in this domain despite the problem of having an ambiguous and incomplete domain description. Our approach is based on datatype matching and interleaved search and execution for service composition. We then also discuss the problem of handling services with internal state.

The Planning Problem and Web Services

A planning problem can be described as a five-tuple $\langle S, s_0, G, A, \Gamma \rangle$, where S is the set of all possible states of the world, $s_0 \in S$ denotes the initial state of the planner, $G \subset S$ denotes the set of goal states the planning system should attempt to reach, A is the set of (ground) actions the planner can perform in attempting to reach a goal state, and the transition relation $\Gamma \subseteq S \times A \times S$ defines the semantics of each action by describing the state (or set of possible states if the operation is non-deterministic) that results when a particular action is executed in a given world state.

Planning states can be described by a set of numeric variables, or more generally by a set of typed objects with attributes, where attributes can be numeric variables or other objects. Whatever the representation, a "state" is simply an assignment of values to all of the variables describing the domain.

Similarly, actions are not described individually, but are grouped into logically equivalent operators, which need to be instantiated (grounded) using parameters when executed. For example, a robot might have available to it a generic operator *pick-up-object*, which can be grounded to a particular action *pick-up-ball-1*, by assigning the parameter *object* to the particular object *ball-1* in the domain.

The semantics of an action, which above we defined by the domain relation Γ, is normally described as part of the operator description in terms of preconditions and effects. Preconditions describe the applicability of the operator (and thus its ground instances) in various states of the world. The preconditions can be seen as describing a set of states, $S_{prec} \subseteq S$ in which a grounded action can be executed. Operator preconditions can be described as a formula over state variables and the operator parameters. Meanwhile, the effects of an action describe the way in which the state of the world is changed after action execution and may depend on the value of operator parameters. Action effects can be described in terms of changes (additions, subtractions, inversions, etc.) to the state variables describing the domain.

Documents and States

Since any interaction with a web service involves sending and receiving messages, one way to describe the state of a system which is interacting with such services is in terms of the messages it has sent and (more importantly) received. One can interpret the information contained in each message as a description of the current world. Thus a document returned by a weather service which states that the weather temperature in Melbourne is $15°C$, can be understood to describe a world (the current state of affairs) in which the temperature in Melbourne really is $15°C$.

A state is then described by an ordered set of documents, where the most recently generated documents can be assumed to carry the most accurate information regarding the current state of the world. One can draw a distinction here between the actual state of the world, which is only partially described by the documents in possession of the planner, and the state of the planner (the history of its interactions with services) which is fully known, but which only approximates the "real world".

We see that already in describing the state of the world we a faced with two problems, not "normally" encountered in planning systems:

1. partial observability of state
2. ambiguity in state description

The first problem describes the fact that we can only know as much about the current state of the world, as is described in the small set of documents in the possession of the planner. Unless we have a document which states what the current temperature in Melbourne is, we do not know that piece of information. Thus information gathering is an important part of the planning process.

The problem of ambiguity arises from the use of documents to describe a world state. Since the schemas defining these documents are written by different people from those building the planning system, the planner may in some cases misunderstand the meaning of a particular document. I.e. there is a problem of matching and interpreting instance level data. For example, the message about the temperature in Melbourne may have been referring to the Australian city or could (perhaps less likely) have been referring to Melbourne, Florida, USA. One way to avoid this problem is to assume that there are standards which define the particular domain within which the planner will be working (such as standard documents for defining e-commerce in a particular vertical industry). In most domains, however, there will be no global standard for describing the world, and the ambiguity describing the state of the world will have to be taken into account. This problem of heterogeneous domain descriptions is new to the planning community, which is accustomed to handling unambiguous and well specified planning domains.

Operations and Actions

In general, a web service operation is specified by its name, its input and output message types, and a set of fault message types, i.e. $o := \langle name, t_{in}, t_{out}, T_{fault} \rangle$. We can view service operations as the operators available to the planning system. Since we interpret documents as describing states, we can say that the output message type t_{out}, in some way describes the effects of operator execution. The input message type t_{in}, can be seen as describing the parameters for the operator described by o. For instance one could have an operation *getWeather* that takes as input a string *Location* and returns a document containing a *WeatherForecast*. Here the operator parameter is the string variable *Location* and the effect is to know/have the object *WeatherForecast*. Moreover, the input schema can also be seen as describing some of the preconditions of operator execution. Given that the input string to this operation is labelled with "Location", it would be nonsensical to give it as input something which is not a location, such as "computer" for instance. I.e. there is a precondition that the planner should know/have a string of type location, such as "Melbourne".

Unfortunately, unlike in "normal planning domains" where operator semantics are fully specified, here we are missing four types of information:

1. preconditions on input parameter values
2. preconditions on prior operation invocation
3. knowledge of instance values in the output document
4. real world preconditions and effects

The first problem states that we do not know a-priori which input parameter values will cause an operation to execute successfully and which will cause it to return fault messages. For example when submitting a *purchaseOrder* to a vendor service, if the *productID* number refers to an item which is out-of-stock, then the operation will return some sort of fault message. In this case, it was impossible to know a-priori that the operation would fail on that input, and trial-and-error execution is the only way to proceed.

The second problem has to do with the fact that services generally contain more than one operation, and in some cases the successful invocation of an operation is dependent on the previous execution of another. For example, one may need to *Login* prior to accessing the *getWeather* operation. We discuss the problem of services with state in section 4.1.

The third problem is due to the fact that the output of service operations is described by a data-type, not a value. Prior to invoking a particular operation, we cannot know what its exact output will be. Normally in a planning domain the effects of action execution are either constant, conditional on some execution parameters (if the action is not grounded), or are dependent on (/ can be derived from) the state of the planning system itself. Some planning systems can handle actions with non-deterministic effects, but the number of successor states is usually low, which is not the case here, as each possible value of the output document represents a different state.

In the case of a web service operation, the effect (the document being produced) may be relatively constant (e.g a weather notification service for Singapore), may depend only on the values in the input document (e.g. a weather service based on Zip-Code), may depend on the state of the planning system (e.g. a *purchaseItem* operation where the product name was specified in a previous *selectItem* operation), or may even be non-deterministic (e.g. a random number generating service). In all of these cases, there is nothing in the operation description which tells us how to predict the output of the service based on its input. The fact that there is no direct relationship between input and output parameters is not the end of the world, however! One can infer connection based on type matches between the input and the output. For example, a *purchaseItem* operation might take as input a *purchaseOrder* document within which the "product" field is described by a *productID* type, and produce a *confirmation* document with the same "product" field, in which case a planning system would do well to assume that both types refer to the same instance of a *productID* number. Such an assumption is not always valid, however. Take for instance the case of a book recommendation service, where the operation *recommendBook* takes a book-title as input and returns as output a different title from the same genre likely also to interest the reader. Again, trial-and-error execution with learning and a the use of a clever heuristic may be the only way to execute an operation in such a way as to produce the desired output.

The last problem has to do with the fact that some "real world" preconditions and effects of operation invocation cannot be deduced automatically from the input and output document schemas, but rather require background knowledge regarding the semantics of the operations themselves. For example, the aforementioned *purchaseItem* operation may require a *creditCardNumber* as input and when executing the operation one would expect to later receive a debit for the cost of the item on one's credit-card statement. This sort of "extra semantics" is of course not described within the input and output documents of the *purchaseItem* operation. In general, there are three approaches for the planner to gain such "background" knowledge:

1. require standardisation
2. server side semantic mark-up
3. client side semantic "interpretation"

In some domains it might be possible to get all of the members of the domain (service creators and service consumers) to come together and agree on a common conceptualisation and formal semantics for the domain. In that case, processes such as "credit card payment" could be formalised.

The second approach is that advocated by the Semantic Web community. The idea is that the service provider should formally mark-up the service operations they provide with explicit descriptions of their preconditions and effects. This then leaves then the non-trivial task of mapping between the ontology used by the service provider, and that used by the planner. At the moment few service providers have taken up the opportunity to mark-up their services, for the simple reason that they don't envisage the use of their services by an automated planning system.

A third alternative is to place the burden of semantic understanding back on the client planning system. The idea would be to match operation names and meta-data of a new service with another service for which the formal semantics are already known by the planner, thus attempting to discover automatically the semantics of operations in the new service at run time. Another somewhat less involving approach to client-side semantic interpretation would be to simply restrict the use of sensitive information such as a *creditCardNumber*, and add "extra semantics" to the planner regarding the effects of its usage.

Initial Conditions and the Goal

If states are described by a set of documents, then the initial state could be viewed in the same way, as a set of documents available to the planner at the outset. This "local information" would be available to the system to use as input when executing service operations. For example, if the goal were to "buy a particular book", then the local information might be the name, address, and credit card details of the person requesting the goal.

Along a similar line, the planning goal (which is simply a set of desired states for the planner to reach) can also be described in terms of documents. That is to say, goals can be expressed as information requirements - as a type of XML document that the system needs to create. By placing restrictions on the values of fields within the requested document, one can express a goal such as *find the temperature in Melbourne*, as *create an instance of the document*:

```
<Weather>
 <Temperature type="decimal"/>
 <Location type="string"/>
</Weather>
```

where the value of the "Location" field is equal to "Melbourne". Thus a goal is a datatype and a set of restrictions on the values of that datatype. In order to satisfy the goal, the system needs to provide a document (an output from a service), which is of similar type to the goal and adheres to the given constraints. These constraints could be equalities such as "bookName" equals "Harry Potter ..." or numerical inequalities such as that the "price" field has value "$<$ 20 Euro". Describing goals of this way has some similarity to performing "query by example" in database systems.

One could also allow for goals containing multiple documents and constraints across them. In that way it would be possible to handle more complex situations, such as that of booking a flight and a hotel, where the flight and hotel dates need to coincide. In that situation, the goal would comprise of two documents, one describing the flight reservation, the other the hotel reservation. Furthermore, we would require that the flight destination matched with the hotel location, and the flight arrived on the same day as the hotel booking commenced. The goal documents could look as follows, (note the constraint variables @*date* and @*city*):

```
<Flight_Reservation>
 <Departure ....
 <Arrival>
  <Date type="date" value="@date"/>
  <City type="string" value="@city"/>
 </Arrival>
</Flight_Reservation>
```

```
<Hotel_Reservation>
 ....
 <City type="string" value="@city"/>
 <In_Date type="date" value="@date"/>
 <Out_Date ....
</Hotel_Reservation>
```

Interpreting and Matching Datatypes

As mentioned previously, a datatype has a set of values that can be considered as representing different possible states of the world. For example when receiving a message of type *Person* with the field "name" equal to "Peter" and "age" equal to "21", we can interpret the message as saying that there exists a person called Peter who is 21 years of age. And if another message arrives, this time of type *UniversityStudent* with name and age as before, we can interpret it as saying that there exists a person Peter who's 21 and goes to university. The second message describes a smaller set of possible worlds (interpretations) than the first, i.e. the cases where Peter is not a school student nor a worker, but a university student.

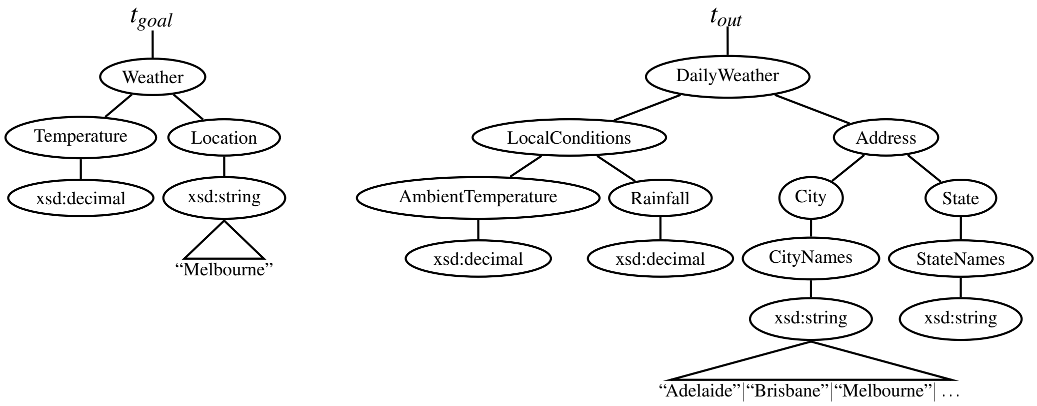

Figure 1: Matching schema types

Now if we needed to fulfill a goal (or provide an input to a service) of type *Person*, then an instance of the message *UniversityStudent* can be used to provide the required information. If however we require an input of type *UniversityStudent* and have a message of type *Person* the reverse is not possible as we do not know whether or not the instance to which the message refers is a university student or a school student, and so on. Thus in order to be able to map from one datatype to another we require that the latter describes a superset of the possible worlds that can be described by the former. I.e. that the target type is a more generic version of the source (under a given mapping).

In order to compose services based on their service descriptions we need to be able to do exactly this type of data mapping, i.e. to map service outputs to other service inputs and so on. The ability to map data between identical types is not sufficient for our purposes as we cannot guarantee (indeed it almost certainly not the case) that the required services will input and output types from a common schema. Thus we need to tackle the problem of data heterogeneity, which is to decide if under some mapping the data described by one datatype can be substituted for that described by another. I.e. if we can take the output produced by one service, map it, and use it as input for another service.

We have developed a type matching algorithm based on the idea that the target type needs to be shown to be a more general version of the source. In our type matching algorithm, when we compare the goal (target) type t_{goal}, to a particular service output (source) type t_{out}, we require that $t_{goal} \supseteq_M t_{out}$, which is to say that all documents conforming to the output type also conform (form a subset of those conforming) to the goal type after a certain mapping M has been applied to them. For example, the goal to find the current temperature in “Melbourne” (as given in section 2.3) should match against a schema such as:

```
<DailyWeather>
 <LocalConditions>
  <AmbientTemperature type="decimal"/>
  <Rainfall type="decimal"/>
 </LocalConditions>
 <Address>
  <City type="CityNames"/>
  <State type="StateNames"/>
 </Address>
</DailyWeather>
```

where:

```
<simpleType name="CityNames">
 <restriction base="string">
  <enumeration value="Adelaide"/>
  <enumeration value="Brisbane"/>
  <enumeration value="Melbourne"/>
  ....
 </restriction>
</simpleType>
```

because the information required by the first can be found from within the second. I.e. the values for “AmbientTemperature” and “City” in the second can be mapped to “Temperature” and “Location” in the first. Note also the fact that the value “Melbourne” (which is a restriction on the field “Location” in the goal), is one of the possible values of the type “CityNames” in the output type. I.e. there are instances of the output type which adhere to the value restrictions given in the goal.

We use WordNet (Fellbaum 1998) as a lexical resource for performing matching of labels within the type structures. We first use it to match synonyms like “car” and “automobile”, as most documents which refer to an instance of car are also referring to an instance of automobile (we ignore for the moment the problem of sense multiplicity). We then try to use more of the semantic information available in WordNet. Generally speaking, there are two ways to use this information - either qualitatively (explore “isa”, “part of” hierarchies to find precise semantic relationship between concepts) or quantitatively (by using similarity measure algorithms as described in (Budanitsky and Hirst 2000), to discover the approximate “similarity” between concepts). In the first case, the semantic relationships are used to find semantic equivalence between substructures. In the second the similarity values can be combined in algorithms to calculate the overall similarity between two data structures. In the following we will take the qualitative approach, using the WordNet noun hierarchy to find generalisation relationships such as "car"⊇"vehicle" (car is a type of vehicle) or "city"⊇"location".

Type Matching Algorithm

In order to define a type matching algorithm we first introduce a simple abstraction on the XML Schema type model. (We simplify the model for the purposes of brevity in our description.) Types in XML Schema can be either primitive, simple or complex. Primitive types such as "string" and "decimal" are defined by a qualified name (a namespace URI and a local label) and the set of possible values (denoted *VAL*) that an instance of the datatype can take.

$pt := \langle qname, VAL\rangle$; $qname := \langle namespace, label\rangle$

A simple type, meanwhile, is defined in terms of a primitive type and a restriction function, which limits the range of values of the type. An example of a simple type would be the set of decimal values less than 50 or the set of Australian city names shown in the figure.

$st := \langle qname, pt, res()\rangle$; $res : \{pt.VAL\} \rightarrow \{\top, \bot\}$

Complex types are defined recursively as tree structures. Nodes in the tree structure may contain references to other primitive, simple or complex types, and can have natural language labels associated with them. (Below we denote a single node by n, and a set of nodes by N.)

$ct := \langle qname, N\rangle$; $|N| \geq 2$; $n := [\, t \mid \langle label, N'\rangle \,]$

We can now outline three rules used to decide whether one type (the goal document) is a generalisation of another type (the output document). In the following, n is used to represent a node, which could be a simple or complex type, or any node within a complex type.

Rule 1: let $n_1 = \langle l_1, pt_1\rangle$ and $n_2 = \langle l_2, pt_2\rangle$,
if $l_1 \supseteq l_2$ and $pt_1 = pt_2$ then $n_1 \supseteq_M n_2$

Rule 1 states that if two nodes made up of a label and a primitive type are such that the label associated with the first is a generalisation of that associated with the second, and both nodes have the same primitive type, then the first node is a generalisation of the second. For example if l_1 is "Temperature" and l_2 "AmbientTemperature" then $l_1 \supseteq l_2$ as ambient temperature is a type of temperature. And since both nodes have the same primitive type xsd:decimal the first node is found to be a generalisation of the second.

Rule 2: let $n_1 \in N$ and $n_2 = \langle l_2, N_2\rangle$,
if $\exists n_i \in N_2 | n_1 \supseteq_M n_i$ then $n_1 \supseteq_M n_2$

Rule 2 states that if one of the children associated with a node is a specialisation of another node, then so is the parent node. This rule allows us to "skip" intermediate nodes in the hierarchy of the more specific type. For example since the node "AmbientTemperature" was found to be a specialisation of the node "Temperature" then so is its parent node "LocalConditions". This is because, under the same transformation (one that maps the value of AmbientTemperature to that of Temperature and Rainfall to nothing) the set of documents will be the same as before.

Rule 3: let $n_1 = \langle l_1, N_1\rangle$ and $n_2 = \langle l_2, N_2\rangle$,
if $l_1 \supseteq l_2$ and $\forall n_i \in N_1 (\exists n_j \in N_2 | n_i \supseteq_M n_j)$ then $n_1 \supseteq_M n_2$

Rule 3 states that if the label associated with a node is more generic than that associated with another node, and for all of the child nodes of the first there is an equivalent (or more specific) node among the children of the second, then the first is a more generic version of the second. For example, since the label "Weather" is found to match the label "DailyWeather" and the child nodes Temperature and Location match with LocalConditions (due to its child AmbientTemperature) and Address (due to its child City), the node Weather represents a generalisation of the node DailyWeather.

Type Matching Implementation

We have implemented a prototype service discovery system based on type-matching, which when given an information goal, attempts to find all services capable of providing the desired information. To do so, it accesses an online database similar to UDDI, which contains links to WSDL documents. It downloads each service descriptions in turn, analysing it for relevant output. In order to test for a match between two types, the labels in each are tokenised using a regular expression, common abbreviations are expanded, compound words (multi-words with their own prescribed meaning in WordNet such as "fire-fighter") are discovered, a "stop-list" is used to remove words with counter-intuitive meanings in WordNet, and the rest of the words are checked against the noun hierarchy. If any one of the nouns in the label of the target document is found to be a descendent (more specific version) of a noun in a label of the goal document, then the two labels are said to match.

The matching system was tested using 212 valid web service descriptions (WSDL documents). Three different goals were used to test the system, the first being the weather example discussed previously, the second and third relate to gaining stock price and exchange rate information. In the weather example, the system was able to discover seven of the eight services capable of providing weather information. In the other two test cases, the matching system did not perform as well (1/5 and 2/5 respectively). The reason for this was that the set of terms used to describe the service output types varied too much for the WordNet algorithm to discover links between them. In other words, WordNet didn't have enough concepts and relations available to properly describe these domains. For example, there is no relation in WordNet linking the concepts "stock ticker" and "stock symbol"! In order to achieve a reasonable success rate, the type matching system was extended slightly for these cases so that it took into consideration also the name of the service operation along with its input requirements.

Service Composition

Having discussed an algorithm capable of matching and mapping data between heterogeneous type structures, we outline a second algorithm that exploits this capability to compose and execute service operations to achieve an information goal. In the section describing operations, we discussed the fact that the output of an operation invocation is not fully specified in its definition. Whenever we execute an operation within a service we cannot guarantee that it will execute properly and provide the desired output (e.g. where "Location" equals "Melbourne"). So when planning we are forced to interleave search and execution, in order to overcome the problem of incomplete knowledge regarding the actions in the domain.

The algorithm executes as follows. It takes as input the goal to be achieved and searches a UDDI directory for all services which are capable of outputting documents of sufficient similarity to the goal, using the type matching algorithm described previously. (Service meta-data and taxonomy matching could also be used to cut down this search space.) The service interface with the most similar output is selected first. If there is more than one implementation of that interface (e.g. the same weather information service is provided by two different companies), the algorithm will need to select one of them based on meta-data values, (by matching and comparing the available local information with this service meta-data). It then attempts to execute the particular service operation that produces the desired output. Before doing so, it must create the required input document. It starts by using the immediately available information, such as that given in the goal, the local information, and past input and output documents if they exist. If the available information is not sufficient, the algorithm must again search the outputs of other services, i.e. the procedure calls itself recursively. (The search heuristic being used here is based on the assumption that currently available information is more reliable than new service outputs, and that newer information is more relevant than older information.) Generally, not all of the data required to fill the input document will be contained in a single source, thus the process repeats on sub-elements of the input document until a complete document is produced or a search limit is exceeded.

Having generated an input document, the algorithm attempts to invoke the operation. If it does not execute properly (or produces an undesired output), the algorithm rolls back certain decisions made when creating the input and tries again. The heuristic guiding this search could be based on the confidence the algorithm had in its decision at each point, i.e. the quality of the match. If after a "reasonable number" of attempts, the operation still can't be executed, then the problem may be the data given as input to the previous (successfully completed) operation. Thus the system either tries to re-execute the previous operation with different inputs, or gives up on the service altogether and searches for a new way of achieving the information goal.

The search tree created by the above algorithm can be seen as a sort of AND-OR tree, where the "OR" branches represent different ways of creating an input, and the "AND" branches represent combinations of service outputs that together produce the input. Leaves in the tree represent data found to be available locally. The execution algorithm described above performs a bounded best-first search through the tree, where the bound sets a limit on the number of failed execution attempts allowed for completing a given sub-tree. The execution bound is decremented for each level of decent in the tree. Viewing the interleaved search and execution as an AND-OR tree allows us to investigate the use of different search techniques such as "limited discrepancy search" to improve execution performance.

Handling Services with State

In the section on operations, we touched on the fact that the output of a particular operation execution may depend on the current state of the planner, i.e. the other operations that have been invoked with that particular service already, and the values given during such invocations. This is because in some cases services are not stateless, but their behaviour depends on the ordering of operation invocation. For example in order to execute a *purchaseItem* operation the system invoking the service might first need to perform a *login* operation. The behaviour of a particular service can be modeled as a non-deterministic finite state transition system or four-tuple $\langle S_{int}, s_{int0}, P, \Delta \rangle$, where S_{int} represents the set of states that the service can find itself in, s_{int0} the initial state, P the set of grounded operations (actions) available, and $\Delta \subseteq S_{int} \times P \times S_{int}$ describes the transitions between different states caused by execution of the grounded operations. We adopt a different notation for state here than used previously, in order to differentiate between the internal state of a particular service and the overall state of the world. The internal state of a service does not take into account all of the other interactions executed by the planner with other services. Moreover, it may be possible to initiate multiple concurrent (yet independent) conversations with the same service, in which case the internal state takes into account only one particular conversation with the service.

Unfortunately, as was the case for the description of states and operations given in section 2, the information describing the domain is incomplete. A full description of the transition system describing a particular service is not available. Instead, there may be some sort of high-level protocol description available to us such as process descriptions written in WSCL, WSCI or BPEL. If available the service will be described as follows:

A web service interface is a four-tuple $\langle \Psi, \psi_0, O, \Lambda \rangle$, where Ψ represents a set of states, (note that each ψ represents an abstraction on actual states of the service, i.e. it maps to a subset of S_{int}), ψ_0 is the initial state, O is the set of operations, and Λ is the set of transitions possible between states. Each transition $\lambda := \langle \psi_i, \psi_f, o, guard(), DLINK \rangle$, is described by an initial state, a final state, an operation (which triggers the transition), a guard condition (which determines whether or not the transition will fire for given input values), and a set of data-links, (which describe any correlation between this input and the outputs of previous transitions). A guard can be any formula over values in the fields of the input document, and can be seen as describing the preconditions of transition execution. For example, a particular guard condition might require that the *price* element has value "$\geq$ \$200", in which case the service can enter a state in which free home delivery is possible. A datalink $dlink := \langle \lambda_i, node_1, node_2 \rangle$, requires equality of a field in this input with a field in the output of a previous transition. Datalinks can be used to maintain references between documents (e.g. this document refers to *trackingNumber* XYZ). Such links can be seen as additional preconditions on transition execution.

The service composition algorithm described in the previous section assumed that all of the operations within each service could be executed independently of one another, i.e. that the service to which they belonged was stateless. In some cases this assumption will be false, and the exact ordering of operations may be critical for the correct execution of services. Moreover the control flow information (transitions and their guard conditions) and data flow information (the datalinks) available in service descriptions could be used to make the task of service composition easier. In (Carman and Serafini 2003), we described a bounded algorithm that takes advantage of the information available in service process descriptions when composing and executing services.

Conclusions and Related Work

We have shown in this paper how the service composition problem can be viewed as a planning problem in which state descriptions are ambiguous and operator definitions are incomplete. We have introduced a semantic type matching algorithm and an interleaved search and execution algorithm that together allow for basic automated service composition. This work is preliminary, but shows promise as a means of performing planning in domains which until now were beyond the reach of state of the art planning systems. In the following we compare the approach and techniques discussed in the paper with the work of others in this area.

The type matching algorithm discussed in this paper can be seen as a type of schema matching algorithm as defined in (Rahm and Bernstein 2001). There are a few differences however between the various techniques cited in this survey paper, and our type matching algorithm. The primary difference is that we take a logical interpretation of the problem of type matching and see it as the problem of deciding whether one type structure is a more generic version of another type structure under a certain set of mappings. Secondly, we use the semantic relations in WordNet as the basis for our algorithm, rather than relying on string matching, synonyms, or "semantic distance values" between words. From a practical point of view we start with two types and ask the question "can type a be seen as a more general version of type b?" I.e. do the two types match at all. If so, we then find the mapping which allows for transforming data from b into a. The assumption in the case of most schema matchers is the opposite, i.e. that the schemas should match in some way, and the job of the schema matcher is to find the best possible mapping. This is because in data-warehousing applications the schemas to be merged can be assumed to contain similar data.

Much work has been performed on designing semantic web standards for adding semantic mark-up to web service descriptions. In terms of planning based on these descriptions there has been some work on the instantiation (based on user preferences and service availability) of precompiled plans in (McIlraith and Son 2002) as well as on extending the planning domain description language PDDL to handle information producing actions (McDermott 2002). In another work which assumes full knowledge of the semantics of operations (Aiello *et al.* 2002), the authors use a non-deterministic planning language with extended-goals and constraint satisfaction to model the web services planning problem. A different approach was taken by the authors of (Thakkar *et al.* 2002) in which automated service composition is achieved by modeling services as web information sources (exposed by automated web-site wrapping software) for which a common data model was already known. A common data model means that database query planning and transformation techniques can be used for plan synthesis and optimisation. In all of these works the authors assume to be interacting with services that are described in a standard and possibly formal manner, i.e. all services which provide the same functionality are called in the same way, require the same inputs and produce the same outputs. By doing so the authors avoid some of the difficulties associated with the heterogeneity of the web services planning domain, and are able to apply techniques from "simpler" (or at least more homogeneous) domains such as database query processing.

References

M. Aiello, M. Papazoglou, J. Yang, M. Carman, M. Pistore, L. Serafini, and P. Traverso. A request language for web-services based on planning and constraint satisfaction. In *VLDB workshop on Technologies for E-Services (TES)*, LNCS, page 10. Springer, 2002.

A. Budanitsky and G. Hirst. Semantic distance in wordnet: An experimental, application-oriented evaluation of five measures. In *Workshop on WordNet and Other Lexical Resources, in the North American Chapter of the Association for Computational Linguistics (NAACL-2000)*, 2000.

Mark Carman and Luciano Serafini. Planning for web services the hard way. In *Workshop on Service Oriented Computing, International Symposium on Applications and the Internet (SAINT-2003)*. IEEE Computer Society Press, 2003.

C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. The MIT Press, 1998.

Drew McDermott. Estimated-regression planning for interactions with web services. In *AI Planning Systems Conference*, 2002.

S. McIlraith and T. Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*. Morgan Kaufmann, 2002.

E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4), Dec 2001.

Snehal Thakkar, Craig A. Knoblock, Jose Luis Ambite, and Cyrus Shahabi. Dynamically composing web services from on-line sources. In *Workshop on Intelligent Service Integration, The Eighteenth National Conference on Artificial Intelligence (AAAI)*, 2002.