Planning For Web Services the Hard Way

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

Abstract

In this paper we outline a framework for performing automated discovery, composition and execution of web services based solely on the information available in interface descriptions and service directories. In our framework we do not rely on the existence of semantic mark-up in service descriptions, thus making our approach "hard". (From a planning perspective, we tackle the problem of planning in a semantically heterogeneous domain.) The framework consists of a domain model, a planning and execution algorithm, and learning procedures. Preliminary work on implementing a service matching system is also discussed.

1. Introduction

The wealth of information available on the internet is currently being complemented by an ever-increasing number of services. These services offer the possibility not only to gain more specific types of information but also to interact with the sources of the information, changing the state of these systems and causing real world processes to occur. Our aim is to plan for the automatic discovery, composition and execution of such services in order to achieve user specified goals. The ability to perform automated service composition would revolutionise a number of application areas including e-commerce and systems integration. In e-commerce a planning system capable of discovering and interacting with flight and accomodation booking services could automatically arrange business trips based on user preferences. In systems integration a planning system capable of discovering and integrating new partners automatically, could bring the dream of a dynamic virtual organisations closer to a reality.

In the next section we describe the problem and our approach to solving it. We then present a model of the web services planning domain, and describe an algorithm and learning framework for planning in this domain. Finally we discuss an implementation of a service matching system required in the framework.

2. Problem

Services are made up of operations, which can be seen as actions that a planning system can perform on the world. Unfortunately, from a planning point of view, these actions are not fully specified: their real world effects are unknown, as are the preconditions on their input variables, and the conditional effects on their output variables. The problem of planning for web services is further complicated by the fact that the information describing the domain is distributed (and possibly infinite), heterogeneous (two actions with the same effects may have different names) and incomplete.

One approach to solve this problem is to explicitly describe the semantics of service operations in the interface documentation. In this case the service provider annotates each service interface document with "real world" preconditions and effects, (described using semantic web standards such as OWL[1] and DAML-S[2]). For example, she could state that the effect of the "buyBook" operation is to make the $own(book)$ predicate true. Thus a planner, which has the goal of arriving in a state in which this predicate holds, can conclude that it needs to execute the operation "buyBook" by giving it the relevant inputs. In the work presented here we do not assume that such semantic mark-up is present, because in the short term at least, there is little economic justification for the providers of web services to create such mark-up.

Another approach is to attempt to use the information that is already available in the service interface definitions:

1. input/output data signatures
2. process descriptions
3. labels in natural language
4. meta-data such as business taxonomies

The first type of information is found in WSDL documents and is described in terms of XML Schema. The i/o signature of an operation in a service tells us what type of document needs be provided in order to execute it, as well as the types of documents that will be returned in the case of successful and unsuccessful execution. The signature gives us

[1] see http://www.w3.org/TR/owl-ref/

[2] see http://www.daml.org/services/

information on possible compositions of services. For example, if a particular service has an operation "buyBook", which takes as input an "isdnCode", and another operation "getISDN" (from a different service) outputs values of the same type, then the planner may try to execute the latter in order to generate input for the former.

The i/o characteristics of service operations, give a static description of a service, without any representation of its state. One can state for example that "selectItem" and "purchaseItem" are both operations of the service, but not that the former should proceed the latter when executing the service. Service process definitions (described using choreography standards such as WSCL[3], WSCI[4] and BPEL[5]) can provide such procedural information to the planner.

Natural language tags used to describe operations in a service (such as "buyBook") or parts of a document (such as "billingAddress") can provide further information to the planner when combined with lexical resources such as WordNet [3]. This information can be used to classify services and the discover documents containing relevant data.

Service classification information is found in UDDI-type service registries, in which services are classified according to industry-segment, provider, geographic location, and so on. This information can be used when searching for relevant services and for inferring similarity between them.

The four sources of information described above are not sufficient to guarantee that a plan generated by the planner, will execute without failure and produce the desired output. The planner will need, therefore, to learn from experience (past execution attempts). Experience can also be gathered off-line by monitoring a user interacting with the service.

Since according to this alternative approach, we do not have any description of the real-world effects of operations, we need some way to express goals in the system that is independent of these effects. We do this by describing goals as information requirements - as a type of document that needs to be discovered or created. By placing restrictions on values of fields within the requested document, we can express goals such as $own(book)$ as: create a "purchaseOrderConfirmation" document within which the value of the "bookName" field is the name of the desired book.

3. Related Work

Some relavent work using Golog to interact with DAML-S services is described in [5], where "precompiled" plans are available prior to execution, and are "instantiated" at run-time. In a second work targeted at DAML-S, the planning domain description language PDDL was extended to describe interactions with services, thus allowing regression planning systems to reason over them [4]. Along a similar line (assuming the full semantics of operations is known) was a proposal to use a combination of a non-deterministic planning language with extended-goals and constraint satisfaction to model the web services planning problem [1]. A different approach was taken by the authors of [8] in which automated service composition is achieved by modeling services as web information sources for which a common data model was already known. There has also been some interesting work done on the equivalence/replaceability of different web services in [6].

None of these works have tackled the heterogeneity problem directly. In this paper we do so by proposing a framework for planning and executing web services using only the information available in service descriptions.

[3]see http://www.w3.org/TR/wscl10/

[4]see http://www.w3.org/TR/wsci/

[5]see http://www.ibm.com/developerworks/library/ws-bpel/

4. Formal Description of the Domain

In this section we describe our model of the web services domain, which consists of an XML data model, an extended finite state machine representation of services, and definitions for schema and service matching. We also define the planning problem in terms of the model. In the following, upper-case letters are used to represent sets, while lower-case letters represent corresponding elements, (eg. $t \in T$).

4.1. Simplified XML Data Model

When interacting with web services all data is formated in XML and typed using XML Schema. Thus we need to model XML data structures in our formalism. The data model we introduce is simplified to reduce the complexity of our problem, while still maintaining the essential aspects of XML.

The set of all XML datatypes, denoted T, can be divided into sets of primitive types, PT, simple types, ST, and complex types, CT. Each primitive type is defined by a unique name, uri, and the set of possible values, VAL. 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.

$pt := \langle uri, VAL \rangle$

$st := \langle uri, pt, res() \rangle;\ res : \{pt.VAL\} \rightarrow \{true, false\}$

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.

$ct := \langle uri, struct \rangle$

$struct := [\ t \mid \langle (label, struct), (label, struct)+ \rangle\ ]$

An XML data element is a tuple of a datatype and a value for that type. The set of values a variable of a given type can take is calculated using the second equation below.

$e := \langle uri, t, val \rangle \mid val \in val(t)$

$$val(s) \equiv \begin{cases} s.VAL & if\ s \in PT \\ s.res(s.pt.VAL) & if\ s \in ST \\ val(s.struct) & if\ s \in CT \\ \langle val(s_1), .., val(s_n) \rangle & if\ s = \langle s_1, .., s_n \rangle \end{cases}$$

In order to refer to parts of a document or a schema, we introduce a syntax for path expressions, and a function for discovering the node represented by a path. (Note that s_n represents the n^{th} child-node or sub-tree of s.)

$path := [n(/n)*] \mid n \in \mathbb{N}$

$$s[path] \equiv \begin{cases} s_n & if\ path = n \\ s_n[subpath] & if\ path = n/subpath \end{cases}$$

4.2. Model of Web Services

We now introduce our representation for web services. A web service interface, i, is defined by a set of procedures, P, and a state machine, m. The set of procedures represent all of the operations that can be performed on (an implementation of) the service, while the state machine limits the ordering of these operations to conform to a given high-level protocol. A procedure is given by its name, its input and output message types, and a set of fault message types.

$i := \langle uri, P, m \rangle;\ p := \langle qname, t_{in}, t_{out}, T_{fault} \rangle$

A web service state machine, m, meanwhile, is defined by a set of states, Ψ, a set of transitions possible between these states, Λ, and the initial state, ψ_0. An individual transition is described by an initial state, a final state, a procedure (which causes the triggering of 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. For example, it might require that the "price" element has value "$\geq$ \$200". A datalink requires equality of a field in this input with a field in the output of a previous transition. It is described by a transition and two path expressions. Datalinks are used to maintain references between documents (this document refers to "trackingNumber" XYZ) or to provide extra "data-flow" information, (that the output "productID-Number" should be used as input to a subsequent operation "makePurchase"). The service state machine can be created automatically from process descriptions written in WSCL, WSCI or BPEL.

$m := \langle \Psi, \Lambda, \psi_0 \rangle;\ \lambda := \langle \psi_i, \psi_f, p, guard(), DLINK \rangle$

$guard : \{val(p.t_{in})\} \rightarrow \{true, false\}$

$dlink := \langle \lambda_i, path_1, path_2 \rangle$

$where\ \lambda_i.p.t_{out}[path_1] = \lambda.p.t_{in}[path_2]$

Through industry standardisation efforts, the same web service interface may come to be implemented by different providers. Thus we need to define the concept of a web service implementation, denoted w. An implementation is described by a web location, uri, a service interface, and a set of instantiation (end-point) parameters. The parameters, (containing information like "geographicLocation"), can be used by the planner to choose between implementations.

$w := \langle uri, i, PAR \rangle$

4.3. Schema & Service Matching

In order for the planner to select which pieces of data to use as input to a given procedure, it needs heuristics which tell it what documents are most likely a good match for the target input. These heuristics are based on schema and service matching. Schema mapping is the process by which two schemas are compared for similarity and the matching sub-elements are discovered. Schema matching systems use structural information, linguistic information, and instance data to automatically create matches, (see [7]). We define a schema mapping, map, between two (complex) datatypes, t_a and t_b, as a set of links connecting equivalent sub-elements, as well as a value quantifying the similarity between the two datatypes.

$map := \langle t_a, t_b, LINK, sim \rangle;\ link := \langle path_a, path_b \rangle$

Service matching can be seen in some way as the dual of schema matching. The idea is to discover semantically equivalent states in the state machines representing different services. Service matching information can then be used in inferences such as "executing these two services will likely produce similar effects" and "these services are likely to execute properly using the same input data". We define a service mapping, i_map, between two service interfaces, i_a and i_b, as a set of links between equivalent states in their respective state machines, and a metric estimating the overall similarity between them.

$i\_map := \langle i_a, i_b, EQUIV, sim \rangle;\ equiv := \langle \psi_a, \psi_b \rangle$

4.4. Problem Definition

In this section we formalise the input to the system, namely the goal that has to be achieved, and the problem that must be solved. As mentioned previously, we describe goals in terms of information requirements, i.e. a piece of information that must be discovered by the system. A goal, g, is defined as a datatype, t_g and a set of restrictions, R, on the values of fields within the desired document. In order to satisfy the goal, the system needs to provide a document (an output from a service), which has type similar to t_g and values adhering to the given constraints. An example of a goal could be to have a "purchaseOrderConfirmation" document, in which the "booktitle" field has the value "Harry Potter", and the "price" field has value "$<$ 20 Euro".

$g := \langle t_g, R \rangle$

$r := \langle path, op, e_{val} \rangle \mid t_g[path] = e_{val}.t \in ST \cup PT$

$op \in \{=, \leq, <, >, \geq\}$

A problem is a combination of a goal and a set of local documents available to the planner to use as input when executing services. An example of such local information could be the name, address, and credit card details of the person requesting the goal.

$prob := \langle g, E_{local} \rangle$

5. Framework

In this section we outline a basic algorithm for the discovery, planning and execution of services to achieve user specified goals. We then describe how learning techniques can be used to improve the domain model, thus improving execution over time.

5.1. Basic Algorithm

The basic algorithm interleaves planning with execution to overcome the problem of incomplete information (no guaranteed solution to the problem can be discovered at "plan-time"). 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. (Taxonomy information and service matching can be used to cut down the search space, while schema matching is used to assess the similarity between documents.) The service interface with the most similar output is selected first, and the value restrictions given with the goal, are re-expressed in terms of the output of the selected service.

The algorithm then calls the $execService$ procedure (see figure), which attempts to find all implementations of the service interface, and order them based on their parameter values, (using local information such as geographic location to compare them). It then searches the service state machine to discover a plan (an ordered set of transitions) capable of achieving the goal. Choosing the most promising service implementation, it attempts to execute the transitions along the plan until the goal is achieved, or a particular transition becomes unexecutable.

In order to execute a transition, the algorithm needs to create the required input document. It starts by using the datalink information from the state machine. It then attempts to complete the document using other available information, such as the goal, past input and output values, the local information, or the output of other services. Schema matching is used to compare documents, and a certain document is chosen based on a weighted similarity value. If the selected input is another service, then the procedure calls itself recursively. 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.

Figure 1: Recursive "planning and execution algorithm"

$execService(i_{goal}, p_{goal}, R, E_{local}, limit)$ {
$plan \leftarrow shortestPlan(i.m, p_{goal})$;
for $w \in \{w \in W_{uddi} |\ w.i = i\}$ **do**
  $step \leftarrow 1$; $E_{history} \leftarrow \emptyset$; $sLim \leftarrow limit$;
  while $sLim > 0$ **do**
    $\lambda \leftarrow plan[step]$; $j \leftarrow 1$; $E_{tried}[] \leftarrow \emptyset$;
    $e_{out} \leftarrow null$; $tLim \leftarrow sLim$;
    for $dl \in \lambda.DLINK$ **do**
      $e_{in}[dl.path_2] \leftarrow dl.\lambda_i.p.e_{out}[dl.path_1]$;
    while $tLim > 0$ **do**
      $cLim \leftarrow tLim$;
      while $(\exists path | e_{in}[path] = null) \wedge (cLim > 0)$ **do**
        $path_s \leftarrow nextUnassignedNode(e_{in})$;
        $t_s \leftarrow e_{in}[path_s].t$;
        $req(\cdot) := \lambda.guard(\cdot) \wedge (\cdot \notin E_{tried}[j])$;
        $e_1 \leftarrow mostSim(t_s, \{e \in R.e_{val} | req(e)\})$;
        $e_2 \leftarrow mostSim(t_s, \{e \in E_{history} | req(e)\})$;
        $e_3 \leftarrow mostSim(t_s, \{e \in E_{local} | req(e)\})$;
        $\hat{e}_4 \leftarrow mostSim(t_s, \{t \in I_{uddi}.P.t_{out}\})$;
        $e_x \leftarrow weightedMostSim(e_1, e_2, e_3, \hat{e}_4)$;
        if $e_x = \hat{e}_4$ **then**
          $E_l \leftarrow R.e_{val} \cup E_{history} \cup E_{local}$;
          $e_x \leftarrow execService(i_4, p_4, req(), E_l, cLim)$;
        for $l \in map(x, t_s).LINK$ **do**
          $e_{in}[path_s][l.path_a] \leftarrow e_x[l.path_b]$;
        $addTo(E_{tried}[j], e_x)$; j++; $reduce(cLim)$;
      $e_{out} \leftarrow invoke(\lambda, e_{in})$;
      if $e_{out}.t = \lambda.p.t_{out}$ **then** break;
      else $backtrack(e_{in}, j, E_{tried})$; $reduce(tLim)$;
    if $e_{out}.t = \lambda.p.t_{out}$ **then**
      if $step = path.length$ **then**
        if $\forall (r \in R)\ r.op(e_{out}[r.path], r.e_{val})$ **then**
          return e_{out};
        else $backtrack(step, E_{history})$;
      else $step$++; $addTo(E_{history}, \langle e_{in}, e_{out} \rangle)$;
    else $backtrack(step, E_{history})$; $reduce(sLim)$;
return $null$;}

If the transition does not execute properly (or produce the desired output), the algorithm rolls back certain decisions made when creating the input and tries again. The heuristic guiding this search should be based on the confidence the planner had in its decision at each point, i.e. the weighted similarity measure. If after a "reasonable number" of attempts, the transition still can't be executed, then the problem may be the data given as input to previous transitions. The algorithm backtracks along the trajectory of execution through the service state machine, hopefully to the point at which the "bad input" was given to the service. It does this by spawning a new conversation with the service or by performing compensation actions if the service is

transactional. Eventually, if the planner cannot manage to execute the service as a whole, it will give up on that particular implementation and attempt another, or try a different type of service altogether.

5.2. Learning Procedures

When executing a service, different rules, such as extra $guard$ and $dlink$ constraints, can be learnt and re-used in the future to increase the probability of successful execution. Different types of learning that can be performed include:

1. learning preconditions of procedure execution
2. learning effects of procedure execution
3. learning efficient search policies/heuristics
4. learning inferences based on service similarity

The first type of learning can be performed whenever an attempt is made to execute an operation. Successful execution indicates that the input variables are within the correct range, given the current history in the conversation, failure indicates the contrary. Learning the effects of procedure execution can only be performed when an operation executes correctly. The idea is to discover correspondences between the input variables and the output variables. This kind of information is important, as it tells us how to select input data in order to produce the desired output data. I.e. how to ensure that the "bookName" field in the output document has the desired value. The third type of learning is aimed at optimising the search and execution algorithm itself, by improving the search heuristics it uses. The idea is to learn the most appropriate weight function to use when selecting input data, and to learn how long to "keep trying" to execute transitions/services before giving up and backtracking. The last item on the list refers to learning the ability to generalise between services. The idea is to find services similar to the current one (using service matching), but which have been interacted with before, and to use experience from these interactions when executing the current service.

6. First Step toward Implementation

In this section we briefly describe a first step toward implementing the framework. We have built a system that can perform service matching automatically, and in an unsupervised fashion. The system takes as input WSDL service descriptions, parsing the names of service operations (such as "sendPurchaseOrder"). Verbs, adverbs, adjectives and numbers are discarded, as are words in a "stop-list" (nouns with counterintuitive meaning in WordNet and web services jargon). Common acronyms are expanded, and collocations (double words such as "fire fighter" which are ascribed their own meaning in WordNet) are discovered. Each service is then described by a vector of WordNet word identifiers.

The system uses algorithms based on WordNet, which calculate semantic similarity measures between different words (see [2]), to calculate similarities between service vectors. These similarity values can then be used as an approximation for the $i\_map.sim$ value mentioned in the formalism. The service description vectors are also clustered using a greedy clustering algorithm, classifying them into groups, that can then be exploited when searching for relevant services.

7. Future Work

We are currently implementing the algorithm described in the framework with the intention of testing it on some real world case studies. We want to continue the work on defining the learning procedures and incorporate them along with the service matching system into the algorithm. We would also like to extend the formalism with more complicated goal descriptions. For example we could allow for goals containing multiple documents and constraints across them. In that way it would be possible to handle the situation of booking a flight and a hotel (different "purchaseOrder" documents) where the flight and hotel dates need to coincide.

References

[1] 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.

[2] 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.

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

[4] D. McDermott. Estimated-regression planning for interactions with web services. In *AI Planning Systems Conference*, 2002.

[5] 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.

[6] M. Mecella, B. Pernici, and P. Craca. Compatibility of e-services in a cooperative multi-platform environment. In *2nd VLDB Workshop on Technologies for e-Services (VLDB-TES 2001)*. Springer, 2001.

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

[8] S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically composing web services from on-line sources. In *Workshop on Intelligent Service Integration, The Eighteenth National Conference on Artificial Intelligence (AAAI)*, 2002.