Web Services Discovery and Constraints Composition*

Debmalya Biswas

IRISA-INRIA, Campus Universitaire de Beaulieu,
35042 Rennes, France
dbiswas@irisa.fr

Abstract. The most promising feature of the Web services platform is its ability to form new (composite) services by combining the capabilities of already existing (component) services. The existing services may themselves be composite leading to a hierarchical composition. In this work, we focus on the discovery aspect. We generalize the characteristics of a service, which need to be considered for successful execution of the service, as constraints. We present a predicate logic model to specify the corresponding constraints. Further, composite services are also published in a registry and available for discovery (hierarchical composition). Towards this end, we show how the constraints of a composite service can be derived from the constraints of its component services in a consistent manner. Finally, we present an incremental matchmaking algorithm which allows bounded inconsistency.

Keywords: Web Services, Composition, Discovery, Constraints, Matchmaking.

1 Introduction

Web services, also known in a broader context as Service Oriented Architecture (SOA) based applications, are based on the assumption that the functionality provided by an enterprise (provider) are exposed as services. The World Wide Web Consortium (W3C) defines Web Services as "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols". The most promising aspect of the Web services platform is the composability aspect, that is, its ability to form new services (hereafter, referred to as composite services) by combining the capabilities of already existing services (hereafter, referred to as component services). The existing services may themselves be composite leading to a hierarchical composition. The services which do not depend on any other services for their execution are referred to as primitive services.

There are mainly two approaches to composing a service: dynamic and static. In the dynamic approach [1], given a complex user request, the system comes up with a plan to fulfill the request depending on the capabilities of available Web services at run-time. In the static approach [2], given a set of Web services, composite services are defined manually at design-time combining their capabilities. In this paper, we

* This work is supported by the ANR DOCFLOW and CREATE ACTIVEDOC projects.

M. Marchiori, J.Z. Pan, and C. de Sainte Marie (Eds.): RR 2007, LNCS 4524, pp. 73–87, 2007.
© Springer-Verlag Berlin Heidelberg 2007

consider a mix [3] of the two approaches where the composite services are defined statically, but the matchmaking with providers is performed dynamically depending on the user request. The above approach is typical of a group of organizations collaborating to provide recurring general services, usually, requested by users. Thus, we assume that the organizations (providers) agree on some of the compositional aspects, such as, ontology used to describe their services, underlying state transition model, logging format, etc.

As mentioned earlier, the main focus of this paper is on the discovery aspect for Web services composition. The current industry standard, Universal Description, Discovery and Integration (UDDI) [4], only supports classification (keyword) based-search and does not capture the semantics of Web services functionality. To overcome this, work has already been initiated towards a semantic description specification for Web services, especially, the Web Ontology Language for Services (OWL-S) [5] specification. The OWL-S specification allows a service to be specified in terms of its IOPE: Inputs, Outputs (required input and expected output values of the service parameters, respectively), Pre-conditions (the state of the world as it should be before execution), and Effects (the state of the world as it would be after execution). We generalize the above as *constraints*, that is, *characteristics of a service which need to be considered for successful execution of the service.* For example, let us consider a house painting contractor C whose services can be reserved online (via credit card). Given this, the fact that the user requires a valid credit card is a pre-condition; and the fact that the user's house will be painted along with the painting charges deducted from his/her account, are the effects. In addition, we also need to consider any limitations of C during the actual execution phase, e.g., the fact that C works only on weekdays (and not on weekends). The above restriction might be a problem if the user would like to get the work done during weekends. In general, pre-conditions refer to the conditions required to initiate an execution and effects reflect the expected conditions after the execution terminates. Constraints attempt to capture the conditions necessary for the entire execution lifecycle (initiate-terminate).

A significant contribution of this paper is the aspect of constraint composition and its impact on service discovery. This aspect has been mostly overlooked till now as, according to most specifications, the description of a composite service resembles that of a primitive service externally (or at an abstract level). However, determining the description of a complex composite service, by itself, is non-trivial. Given their inherent non-determinism (allowed by the "choice" operators within a composition schema), it is impossible to statically determine the subset of component services which would be invoked at run-time. The above implies the difficulty in selecting the component services, whose constraints should be considered, while defining the constraints of the composite service. Basically, the constraints of a composite service should be consistent with the constraints of its component services. In this paper, we take the bottom-up approach and discuss how the constraints of a composite service can be consistently derived from the constraints of its component services. Towards this end, we consider four approaches: optimistic, pessimistic, probabilistic and relative. Finally, we discuss how matchmaking can be performed based on the constraints model. Current matchmaking algorithms focus on "exact" matches (or the most optimum match). They do not consider the scenario where a match does not exist. We try

to overcome the above by allowing inconsistencies during the matchmaking process (does not have to be an exact match) up to a “bounded” limit.

Before proceeding, we would like to mention that the work in this paper is part of ongoing work to provide a lightweight discovery mechanism for ActiveXML (AXML) [6] systems. AXML systems provide an elegant way to combine the power of XML, Web services and Peer to Peer (P2P) paradigms by allowing (active) Web service calls to be embedded in XML documents. An AXML system consists of the following main components:

- AXML documents: XML documents with embedded Web service calls. The embedded services may be AXML services (defined below) or generic Web services.
- AXML Services: Web services defined as queries/updates over AXML documents. An AXML service is also exposed as a regular Web service (with a WSDL description file).
- AXML peers: Nodes where the AXML documents and services are hosted.

Currently, the provider for an embedded service call is hard coded in the AXML document. The objective is to let AXML systems also benefit from the additional flexibility offered by dynamic selection (among the available AXML peers). As obvious, this can be achieved by replacing the hard coding with a query to select the provider at run-time. Given this, we needed a mechanism for discovery in an environment, which is more homogeneous as compared to dynamic Web services compositions (and allows us to assume the presence of a shared ontology, state transition model, etc.). As the proposed concepts are valid for Web services compositions in general, we present them in a Web services context (in the sequel); and only mention their usage with respect to AXML to show their practical relevance.

The rest of the paper is organized as follows: Section 2 deals with the constraints aspect in detail, starting with a predicate logic specification of constraints (sub-section 2.1) followed by the constraints composition model (sub-section 2.2). The incremental matchmaking algorithm is presented in section 3. Sections 4 and 5 discuss related works and conclude the paper, respectively.

2 Constraints

As mentioned earlier, constraints refer to the characteristics of a service which need to be considered for a successful execution of the service. Before proceeding, we would like to discuss some heuristics to decide if a characteristic should (or should not) be considered as a constraint. If we consider constraints as limitations, then the fact that an Airline ABC cannot provide booking for a particular date is also a limitation (and hence, a constraint). However, we do not expect such characteristics to be expressed as constraints as they keep changing frequently. Similarly, we do not expect characteristics which depend on internal business rules (sensitive or confidential information) to be exposed as constraints. Thus, what should (or should not) be expressed as constraints is very much context-specific, and we simply consider constraints as a level of filtering during the discovery process.

2.1 Constraint Specification

Constraints are specified as first order predicates associated with the service definitions. For example, the fact that an airline ABC provides vegetarian meals and has facilities for handicapped people on only some of its flights (to selected destinations) can be represented as follows:

```
flight(Airlines,X,Y):-
        veg_meals(Airlines,Destination_List), member(X,Destination_List),
        hnd_facilities(Airlines,Destination_List), member(Y,Destination_List).
veg_meals('ABC',['Paris','Rennes']).
hnd_facilities('ABC',['Paris','Grenoble']).
```

In the above snippet, 'member(X,Y)' is a system defined predicate which holds if X is an element of the set Y. Now, let us consider "related" constraints or scenarios where there exists a relationship among the constraints. By default, the above example assumes an AND relation among the constraints (both veg_meals and hnd_facilities predicates have to be satisfied). The operators studied in literature for the composition of logic programs are: AND, OR, ONE-OR-MORE, ZERO-OR-MORE and any nesting of the above. We only consider the operators AND, OR and any level of nesting of both to keep the framework simple (ONE-OR-MORE and ZERO-OR-MORE can be expressed in terms of OR). An example of an OR relation among the constraints is as follows: Airline ABC allows airport lounge access at intermediate stopovers only if the passenger holds a business class ticket or is a member of their frequent flier programme. The above scenario can be represented as follows:

```
lounge_access(Airlines,X):-
        ticket_type('ABC',X,'Business').
lounge_access(Airlines,Y):-
        frequent_flier(Airlines,FF_List), member(Y,FF_List).
```

We briefly consider the following qualifiers which may be specified in conjunction with the constraints:

– Validity period: Period until when the constraints are valid. The validity period qualifier can be used to optimize matchmaking. Basically, there is no need to repeat the entire matchmaking process for each and every request. Once a service provider is found suitable, it remains so till the validity period of at least one of its "relevant" constraints expires.
– Commitment: The commitment of a provider towards providing a specific service (levels of commitment [7]). For example, a provider may be willing to accept the responsibility of providing its advertised services under any circumstance; or that it is capable of providing the services, but not willing to accept responsibility if something goes wrong.
– Non-functional: Qualifiers related to non-functional aspects, such as, transactions, security, monitoring (performance), etc. From a transactional point of view, we need to know the protocols supported for concurrency control (e.g., 2PL), atomic commit (e.g., 2PC), and the following attributes required for recovery: idempotent (the effect of executing a service once is the same as executing it more than once), compensatable (its effects can be semantically canceled), pivot (non-compensatable).

From a security perspective, it is important to know the protocols supported for message exchange (e.g., X.509), and if any part of the interaction or service description needs to be kept confidential. The relevant qualifiers, from a monitoring point of view, would be the time interval between successive snapshots of the system state, snapshot format, etc. It is obviously possible to have qualifiers which overlap between the aspects, e.g., it may be required to specify if part of the monitored data (snapshot) cannot be exposed due to security issues.

2.2 Constraints Composition

2.2.1 Broker

The composite provider aggregates services offered by different providers and provides a unique interface to them (without any modification to the functionality of the services, as such). In other words, the composite provider acts as a broker for the aggregated set of services [8]. The accumulated services may have different functionalities or the same functionality with different constraints (as shown by the following example scenario). Scenario: Provider XYZ composing the flight services offered by Airlines ABC and DEF.

Airlines ABC:

```
flight(Airlines,X):-
        hnd_facilities(Airlines,Destination_List), member(X,Destination_List).
hnd_facilities('ABC',['Marseilles','Grenoble']).
```

Airlines DEF:

```
flight(Airlines,X):-
        hnd_facilities(Airlines,Destination_List), member(X,Destination_List).
hnd_facilities('DEF',['Rennes','Paris']).
```

Composite provider XYZ:

```
flight(Airlines,X):-
        hnd_facilities(Airlines,Destination_List), member(X,Destination_List),
        Airlines:= 'XYZ'.
hnd_facilities('ABC',['Marseilles','Grenoble']).
flight(Airlines,X):-
        hnd_facilities(Airlines,Destination_List), member(X,Destination_List),
        Airlines:= 'XYZ'.
hnd_facilities('DEF',['Rennes','Paris']).
```

The addition of the clauses *Airlines:= 'XYZ'* in the above code snippet ensures that the binding returned to the outside world is provider XYZ while the provider XYZ internally delegates the actual processing to the providers ABC/DEF. Another point highlighted by the above example is that composition may lead to *relaxation* of constraints, e.g., the composite provider XYZ can offer flights with facilities for handicapped people to more destinations (Marseilles, Grenoble, Rennes and Paris) than offered by either of the component providers ABC (Marseilles, Grenoble)/DEF (Rennes, Paris).

2.2.2 Mediator

Two or more services offered by (the same or) different providers are composed to form a new composite service with some additional logic (if required) [8]. We assume that the composition schema is specified using some conversation language, e.g., Business Process Execution Language for Web Services (BPEL) [2], OWL-S Service Model [5], etc. We show how the constraints of component services, composed in sequence or parallel, can be composed. Given an Airline ABC with facilities for handicapped people on its flights to selected destinations,

```
flight(Airlines,X):-
        hnd_facilities(Airlines,Destination_List), member(X,Destination_List).
hnd_facilities('ABC',['Marseilles','Grenoble']).
```

and a transport company DEF which has facilities for handicapped people on its local bus networks in selected cities,

```
bus(Transport_C,X):-
        hnd_facilities(Transport_C,Cities_List), member(X, Cities_List).
hnd_facilities('DEF',['Marseilles','Rennes']).
```

the constraints of the composite service provider Travel Agent XYZ can be defined as follows:

```
flight_bus(Agent,X):-
        sequence(_flight(Agent1,X),_bus(Agent2,X)),
        Agent:= XYZ.
_flight(Airlines,X):-
        hnd_facilities(Airlines,Destination_List), member(X,Destination_List).
hnd_facilities('ABC',['Marseilles','Grenoble']).
_bus(Transport_C,X):-
        hnd_facilities(Transport_C,Cities_List), member(X, Cities_List).
hnd_facilities('DEF',['Marseilles','Rennes']).
```

The point to note in the above code snippet is the *flight_bus* predicate representing the newly formed composite service. Also, the original predicates of the primitive services are prefixed with _ to indicate that those services are no longer available (exposed) for direct invocation. The above scenario highlights the *restrictive* nature of constraint composition. For example, the newly composed service flight_bus can provide both flight and bus booking with facilities for handicapped people to fewer destinations (Marseilles) as compared to the destinations covered by the component services separately: flight (Marseilles, Grenoble) and bus (Marseilles, Rennes). Finally, we discuss the usage of the *sequence* predicate (in the above code snippet). For a group of constraints, the sequence relationship implies that all the constraints in the group need to hold (analogous to AND), however, they do not need to hold simultaneously, and it is sufficient if they hold in the specified sequence. For example, let us assume that the premium (constraint) of an insurance policy is €10,000, payable over a period of 10 years. The above constraint is, in reality, equivalent to a sequence of €1000 payments each year (the user does not have to pay €10,000 upfront). The sequential relationship among the constraints can be derived from the ordering of

their respective services in the composition schema. Note that we do not consider the "parallel" relationship explicitly as it is equivalent to AND.

2.2.3 Mediator with Non-determinism

Till now, we have only considered deterministic operators in the composition schema, that is, sequential and parallel composition. With *non-deterministic operators*, the situation is slightly more complicated. Some of the component services, composed via non-deterministic operators, may never be invoked during an execution instance. As such, we need some logic to determine if the constraints of a component service should (or should not) be considered while defining the constraints of the composite service. For example, let us consider the e-shopping scenario illustrated in Fig. 1. There are two non-deterministic operators (choices) in the composition schema: Check Credit and Delivery Mode. The choice "Delivery Mode" indicates that the user can either pick-up the order directly from the store or have it shipped to his/her address. Given this, shipping is a non-deterministic choice and may not be invoked during the actual execution. As such, the question arises "if the constraints of the shipping service, that is, the fact that it can only ship to certain countries, be projected as constraints of the composite e-shopping service (or not)". Note that even component services composed using deterministic operators (Payment and Shipping) are not guaranteed to be invoked if they are preceded by a choice. We consider some approaches to overcome the above issue:

- *Optimistic:* Consider the constraints of only those services, which are guaranteed to be invoked in any execution, while defining the constraints of the composite service. The set of such services (hereafter, referred to as the strong set) can be determined by computing all the possible execution paths and selecting services which occur in all the paths. For example, with reference to the e-shopping scenario in Fig. 1, the strong set = {Browse, Order}. We call this approach optimistic as it assumes that the services in the strong set are sufficient to represent the constraints of the composite service. The concept of a strong set is analogous to the notion of strong unstable predicates [9] or predicates which will "definitely" hold [10] in literature. Strong unstable predicates are true if and only if the predicate is true for all total orders. For example, strong unstable predicates can be used to check if there was a point in the execution of a commit protocol when all the processes were ready to commit. Intuitively, strong unstable predicates allow us to verify that a desirable state will always occur.
- *Pessimistic:* In this approach, we take the pessimistic view and consider the constraints of all those services which are in at least one of the possible execution paths (while defining the constraints of the composite service). We refer to such a set of component services as the weak set. Note that the weak set would consist of all the component services if there are no "unreachable" services in the composition schema. Again, with reference to the e-shopping scenario in Fig. 1, the weak set = {Browse, Order, Cancel Order & Notify Customer, Arrange for Pick-up, Payment, Shipping}. We refer to this approach as pessimistic as it considers the constraints of those services also which may not even be invoked during the actual execution. The corresponding notion in literature is weak unstable predicates [11] or predicates which will "possibly" occur [10]. A weak unstable predicate is true if

and only if there exists a total order in which the predicate is true. For example, weak unstable predicates can be used to verify if a distributed mutual exclusion algorithm allows more than one process to be in the critical region simultaneously. Intuitively, weak unstable predicates can be used to check if an undesirable state will ever occur.

– *Probabilistic:* Another option would be to consider the most frequently invoked component services (or the component services in the most frequently used execution path) as the representative set of the composite service. Such a set can be determined statically from the execution logs or dynamically with the help of some mathematical model (such as, Markov Decision Processes [12]) to assign probabilities to the component services based on previous executions. Again, with reference to the e-shopping scenario in Fig. 1, a probable set of most frequently used component services would be {Browse, Order, Arrange for Pick-up}. While this option appears the most attractive at first sight, developing and solving a Markovian model is non-trivial for a complex composition schema (especially, if it involves a lot of choices).

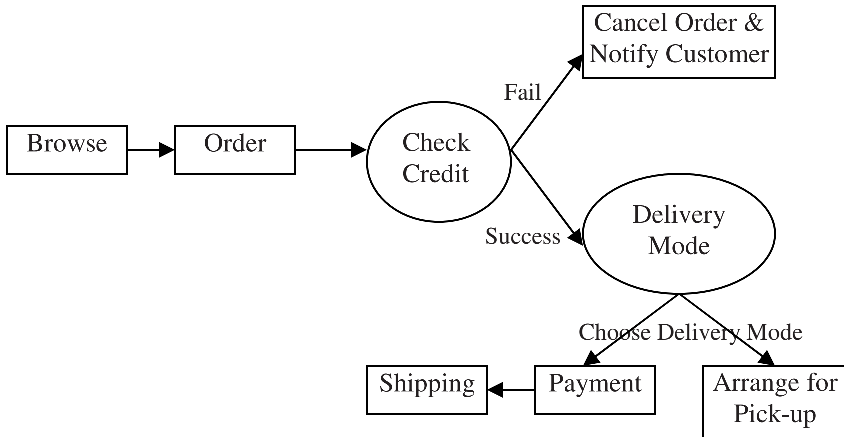

Fig. 1. An e-shopping scenario

The trade-off between the various options (discussed till now) can be summarized as follows: (a) Optimistic: A successful initial match does not guarantee a successful execution (as the constraints of all the component services are not considered initially, it might not be feasible to find a provider for one of the component services at a later stage). Thus, the cost to consider for this case is in terms of failed contractual agreements or simply the loss of user faith. (b) Pessimistic: “Pseudo” constraints may lead to the corresponding composite service becoming ineligible for (an otherwise successful match with) a user request. (c) Probabilistic: For this approach, the cost is in terms of the complexity in finding the adequate probabilities and distribution functions to define the probabilistic model. While the above approaches can be considered as extremes; next, we consider an intermediate, but more practical, approach to determine the representative set of component services (of a composite service).

Relative: In this approach, we consider an incremental construction of the set of component services whose constraints need to be considered (while defining the constraints of the composite service). Basically, we start with the strong set and keep on adding the "related" services as execution progresses. We define related services as follows:

Related services: Let X and Y be component services of a composition schema CS. Given this, X and Y are related if and only if the occurrence of X in an execution path P of CS implies the occurrence of Y in P.

Intuitively, if a component service X of CS is executed then all the component services till the next choice in CS will definitely be executed. For example, with reference to the e-shopping scenario in Fig. 1, services Payment and Shipping are related. As mentioned earlier, the execution of both Payment and Shipping are not guaranteed. However, if Payment is executed, then Shipping is also guaranteed to be executed. The above definition of related services can also be extended to non-invocation of a component service X as follows:

Related services (extended): Let X and Y be component services of a composition schema CS. Given this, X and Y are related if and only if the (non-) occurrence of X in an execution path P of CS implies the (non-) occurrence of Y in P.

Intuitively, if a component service X of CS is not executed then all the component services till the next merge in CS will also not be executed – Fig. 2. The extension is useful if we consider matching for more than one composite service simultaneously (not considered here). Given this, prior knowledge that a component service will not be invoked during a particular execution instance allows better scheduling of the providers among instances.

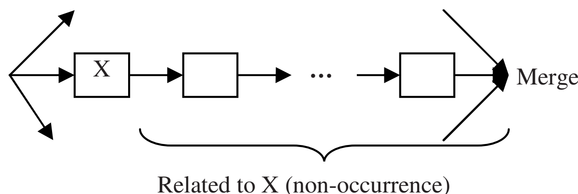

Fig. 2. Related services (based on non-occurrence)

Till now, we have only considered component services related by functional dependencies (as specified by the composition schema). Other relationships between component services can also be (statically) determined based on the application or domain semantics. For example, with reference to an e-shopping scenario, the choice of € as the currency unit implies the (future) need for a shipping provider capable of delivering within countries of the European Union (EU).

AXML application scenario. We discuss an implementation of the "related" approach in the context of query evaluation by AXML systems. Given a query q on an AXML document d, the system returns the subset of nodes of d which satisfy the query criterion. There are two possible modes for query evaluation: lazy and eager. Of the two, lazy evaluation is the preferred mode and implies that only those services are invoked whose results are required for evaluating the query. Now, a query q on a document d may require invoking some of the embedded services in d. The invocation results are

```
<?xml version="1.0" encoding="UTF-8"?>
<ATPList date="18042005">
   <player rank=1>
      <name>
         <firstname>Roger</firstname>
         <lastname>Federer</lastname>
      </name>
      <citizenship>Swiss</citizenship>
      <axml:sc mode="replace" serviceNameSpace="getPoints"
      serviceURL="…" methodName="getPoints">
         <axml:params>
            <axml:param name="name">
            <axml:value>Roger Federer</axml:value>
         </axml:params>
         <points>475</points>
      </axml:sc>
      <axml:sc mode="merge" serviceNameSpace="getGrandSlamsWonbyYear"
      serviceURL="…" methodName="getGrandSlamsWonbyYear">
         <axml:params>
            <axml:param name="name">
            <axml:value>Roger Federer</axml:value>
            <axml:param name="year">
            <axml:value>$year (external value)</axml:value>
         </axml:params>
         <grandslamswon year="2003">W</grandslamswon>
         <grandslamswon year="2004">A, W, U</grandslamswon>
      </axml:sc>
   </player>…
</ATPList>
```

Fig. 3. Sample AXML document ATPList.xml

inserted as children of the embedded service node (modifying d). For example, let us consider the AXML document ATPList.xml in Fig. 3. The document ATPList.xml contains two embedded services “getPoints” and “getGrandSlamsWonbyYear”. Now, let us consider the following query:

Query A:
<action type = “query”>
<location>Select p/citizenship, p/grandslamswon from p in ATPList//player where p/name/lastname = Federer;</location>
</action>

Lazy evaluation of the above query would result in the invocation of the embedded service “getGrandSlamsWonbyYear” *(and not “getPoints”).* However, if the query were defined as follows:

Query B:
<action type = “query”>
<location>Select p/citizenship, p/points from p in ATPList//player where p/name/lastname = Federer;</location>
</action>

Lazy evaluation of query B would result in the invocation of the embedded service call “getPoints” *(and not “getGrandSlamsWonbyYear”).*

Thus, given a q and d, (at a high level) the following three step process is used for query evaluation:

1. Determine the set of relevant embedded services in d to evaluate q.
2. Invoke them and insert their results in d (leading to a modified d).
3. Apply steps 1 and 2 iteratively on the modified d, till step 1 cannot find any relevant calls to evaluate q.

Step 3 is necessary because of the following reasons: (a) The result of a service invocation maybe another service. (b) The invocation results may affect the document d in such a way that formerly non-relevant embedded services become relevant after a certain stage. For more details on the above AXML query evaluation aspects, the interested reader is referred to [13].

To summarize, it is not feasible to statically determine the set of services, which would be invoked during an execution instance (depends on the query and corresponding invocation results). However, for each iteration, we can at least consider the constraints of the relevant (“related”) embedded services determined by step 1 together for discovery.

3 Matchmaking

3.1 Basic Matchmaking

Here, we consider incremental matchmaking, that is, the provider for a service is selected as and when it needs to be executed. For a (composite) service X, let P(X) denote the constraints associated with X. Given this, the required matching for X can be accomplished by posing P(X) as a goal against the providers’ constraints. A logic program execution engine specifies not only if a goal can be satisfied but also all the possible bindings for the unbounded variables in the goal. The bindings correspond to the providers capable of executing X. In case of multiple possible bindings (multiple providers capable of executing the same service), the providers are ranked using some user defined preference criteria or the user may be consulted directly to select the most optimum amongst them.

3.2 Approximate Matchmaking

Now, let us consider the scenario where the matchmaking is unsuccessful, that is, there does not exist a set of providers capable of executing a set of component services. Given this, it makes sense to allow some inconsistency while selecting a

provider. Note that inconsistency is often allowed by real-life systems, e.g., flight reservation system allow flights to be overbooked, but only up to a limited number of seats. Thus, the key here is "bounded" inconsistency. Basically, for a given set of component services SC = {A_{SC1}, A_{SC2}, …, A_{SCn}}, the selected provider for one of the component services A_{SCx} does not have to be a perfect match as long as their accumulated inconsistency is within a specified limit. Again (in the presence of non-determinism), the given set SC of component services implies that all the services in SC will be executed if at least one of the services in SC is executed (related services). Note that the inconsistency induced by a component service A_{SCx} may also have a counter effect on (reduce) the inconsistency induced by another component service A_{SCy}. We use the composition schema CS in Fig. 4 as a running example to illustrate our intuition behind the steps. X enclosed by a rectangle denotes a component service X of CS. Services can be invoked in sequence (D, E) or in parallel (B, C). As before, ovals represent choices.

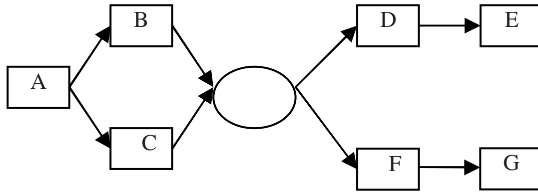

Fig. 4. Sample composition schema CS

For each set of component services SC = {A_{SC1}, A_{SC2}, …, A_{SCn}} considered for matchmaking, perform the following:

1. Determine the common qualifiers: A qualifier q_{SC} is common for SC if a pair of constraints of services A_{SCx} and A_{SCy}, respectively, are based on q_{SC}. For example, if component services D and E need to be completed within 3 and 4 days, respectively; then D and E have constraints based on the common qualifier time. Studies [14] have shown that most constraints in real-life scenarios are based on the qualifiers: price, quantity or time.
2. For each q_{SC}, define a temporary variable C_{qSC} (to keep track of the inconsistency with respect to q_{SC}). *Initially,* $C_{qSC} = 0$.
3. For each A_{SCx} and a common qualifier q_{SC}: Let v_{qSCx} denote the constraint value of A_{SCx} with respect to q_{SC}. For example, $v_{tD} = 3$ denotes the completion time constraint value of D. *Delete the constraint of* A_{SCx}*, based on* q_{SC}*, from the goal.*
4. Perform matchmaking on the reduced goal (as discussed earlier in the previous sub-section).
5. If the matchmaking (above) is successful: [Note that if matchmaking is unsuccessful for the reduced goal then it would definitely have been unsuccessful for the original goal.] Let $p(A_{SCx})$ denote the provider selected to execute A_{SCx}. *For each deleted constraint of* A_{SCx} *based on* q_{SC} *(step 3), get the best possible value* v_{best_qSCx} *of* $p(A_{SCx})$ *with respect to* q_{SC} *and compute* $C_{qSC} = C_{qSC} + (v_{qSCx} - v_{best_qSCx})$. For

example, let us assume that p(D) and p(E) can complete their work in 5 and 1 days, respectively. Given this, $C_t = 0 + (v_{tD} - v_{best_tD}) + (v_{tE} - v_{best_tE}) = (3 - 5) + (4 - 1) = 1$.

6. The selections as a result of the matchmaking in step 4 are valid if and only if, for all q_{SC}, $C_{qSC} \geq 0$. For example, p(D) and p(E) are valid matches for the component services D and E, respectively, as $C_t \geq 0$.

Note that this matching would not have been possible without the above extension as p(D) violates (takes 5 days) the completion time constraint (3 days) of D.

AXML application scenario. We consider a replicated architecture where copies of an AXML document d exist on more than one peer. With respect to each document d, there exists a primary copy of d (and the rest are referred to as secondary copies). Any updates on d occur on the primary copy of d, and are propagated to the secondary copies in a lazy fashion. Let us assume that the system guarantees a maximum propagation delay (of any update to all secondary copies of the affected document) of 1 hour. Given this, a query of the form "List of hotels in Rennes" can be evaluated on any of the secondary copies (that is, inconsistency is allowed). However, a query of the form "What is the current traffic condition on street X?" needs to be evaluated on the primary copy (that is, no inconsistency) or the system needs to be tuned to lower the maximum propagation delay guarantee (that is, inconsistency up to a bounded limit).

4 Related Works

The concept of "constraints" has been there for quite some time now, especially, in the field of Software Engineering as functional and non-functional features associated with components [15]. While studies [16] have identified their need with respect to Web services computing, there hasn't been much work towards trying to integrate them in a model for Web services composition. [17] and [18] describe preliminary works towards integrating the notion of constraints with WSDL/SOAP and OWL-S, respectively. However, their focus is towards trying to represent the operational specifications (e.g., if a service supports the Two Phase Commit protocol, authentication using X.509, etc.) of a Web service using features/constraints in contrast to our approach of trying to capture the functional requirements for a successful execution.

[19] allows each activity to be associated with a constraint c, which is composed of a number of variables ranging over different domains and over which one can express linear constraints. In [20], Vidyasankar et. al. consider "bridging" the incompatibility between providers selected (independently) for the component services of a composite service. In general, the issue of Web services discovery has been studied widely in literature based on different specification formalisms: Hierarchical Task Planning (HTN) [1], Situation Calculus [21], π-Calculus [22], etc. However, none of the above approaches consider composability of the component services' constraints (which is essential to reason about the constraints of the composite service). The notion of bounded inconsistency and its application to matchmaking is also a novel feature of our work.

5 Conclusion and Future Work

In this work, we focused on the discovery aspect of Web services compositions. To enable hierarchical composition, it is required to capture and publish the constraints of the composite services (along with, and in the same manner, as primitive services). We introduced a constraints based model for Web services description. We showed how the constraints of a composite service can be derived and described in a consistent manner with respect to the constraints of its component services. We discussed four approaches: optimistic, pessimistic, probabilistic and relative, to overcome the composition issues introduced by the inherent non-determinism. Finally, we discussed matchmaking for the constraints based description model. We showed how the notion of bounded inconsistency can be exploited to make the matchmaking more efficient.

An obvious extension of the matchmaking algorithm would to consider simultaneous matching for more than one composite service. Doing so, leads to some interesting issues like efficient scheduling of the available providers (touched upon briefly in section 2.2.3). We are already working towards translating the proposed concepts (in this paper) to compose service descriptions specified in OWL-S. In future, we would also like to consider the top-down aspect of constraint composition, that is, to define the constraints of a composite service independently and verifying their consistency against the constraints of its corresponding component services.

Acknowledgments. I would like to thank Krishnamurthy Vidyasankar, Blaise Genest, Holger Lausen and the anonymous referees for their helpful suggestions which helped to improve the paper considerably.

References

1. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D.: HTN planning for Web service composition using SHOP2. Web Semantics 1(4), 377–396 (2004)
2. Business Process Execution Language for Web Services (BPELFWS) Specification v1.1. http://www-128.ibm.com/developerworks/library/ws-bpel/
3. Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., Shan, M.-C.: Adaptive and Dynamic Service Composition in eFlow. HP Technical Report, HPL-2000-39 (March 2000)
4. Universal Description, Discovery and Integration (UDDI) Specification. http://www.uddi.org
5. Web Ontology Language for Services (OWL-S) Specification http://www.daml.org/services/owl-s/
6. Abiteboul, S., Bonifati, A., Cobena, G., Manolescu, I., Milo, T.: Dynamic XML Documents with Distribution and Replication. In: proceedings of 2003 ACM SIGMOD International Conference on Management of Data, pp. 527–538
7. Singh, M.P., Yolum, P.: Commitment Machines. In: Revised Papers from the 8th International Workshop on Intelligent Agents VIII, pp. 235–247 (2001)
8. Hull, R., Benedikt, M., Christophides, V., Su, J.: Eservices: A look behind the curtain. In: proceedings of the 22nd ACM Symposium on Principles of Database Systems (PODS), pp. 1–14 (2003)

9. Garg, V.K., Waldecker, B.: Detection of Strong Unstable Predicates in Distributed Programs. IEEE Transactions on Parallel and Distributed Systems, pp. 1323-1333 (Decemder 1996)
10. Cooper, R., Marzullo, K.: Consistent detection of global predicates. ACM SIGPLAN Notices 26(12) pp. 163–173
11. Garg, V.K., Waldecker, B.: Detection of Weak Unstable Predicates in Distributed Programs. IEEE Transactions on Parallel and Distributed Systems, pp. 299–307 (1994)
12. Doshi, P., Goodwin, R., Akkiraju, R., Verma, K.: Dynamic Workflow Composition: Using Markov Decision Processes. Intl. Journal of Web Services Research 2(1), 1–17 (2005)
13. Benjelloun, O.: Active XML: A data centric perspective on Web services. INRIA PhD dissertation, http://www.activexml.net/reports/omar-thesis.ps (2004)
14. Grosof, B., Labrou, Y., Chan, H.: A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In: proceedings of the 1st ACM International Conference on Electronic Commerce (EC), pp. 68–77 (1999)
15. Chung, L., Nixon, B., Yu, E.: Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design. In: proc. of the 1st International Workshop on Architectures for Software Systems, pp. 31–43 (1995)
16. O'Sullivan, J., Edmond, D., Hofstede, A.: What's in a Service? Towards Accurate Description of Non-Functional Service Properties. In: the Journal of Distributed and Parallel Databases, Vol. 12(2/3) (2002)
17. W3C Position Paper. Constraints and capabilities of Web services agents. In: proc. of the W3C Constraints and Capabilities Workshop, http://www.w3.org/2004/07/12-hh-ccw (2004)
18. OWL-S Coalition. OWL-S Technology for Representing Constraints and Capabilities of Web Services. In: proc. of the W3C Constraints and Capabilities Workshop, http://www.w3.org/2004/08/ws-cc/dmowls-20040904 (2004)
19. Aiello, M., Papzoglou, M., Yang, J., Carman, M., Pistore, M., Serafini, L., Traverso, P.: A Request Language for Web-Services based on Planning and Constraint Satisfaction. In: proc. of the 3rd VLDB Workshop on Technologies for E-Services (TES), pp. 76–85 (2002)
20. Vidyasankar, K., Ananthanarayana, V.S.: Binding and Execution of Web Service Compositions. In: proceedings of 6th International Conference on Web Information Systems Engineering (WISE), pp. 258–272 (2005)
21. Narayanan, S., Mcllraith, S.A.: Simulation, Verification and Automated Composition of Web Services. In: proceedings of the 11th ACM International Conference on the World Wide Web (WWW), pp. 77–88 (2002)
22. Rao, J., Kungas, P., Matskin, M.: Logic Based Web Services Composition: From Service Description to Process Model. In: proceedings of the 2nd IEEE International Conference on Web Services (ICWS), pp. 446–453 (2004)