DISTRIBUTED DYNAMIC WEB SERVICE COMPOSITION

Snehit Prabhu
Autonomic Computing
IBM Software Labs
Embassy Golf Links, Bangalore-17
Phone : +91 832 2461112
snehit.prabhu@gmail.com

Abstract*:* The creation of a custom-built composite web service given a set of available services involves identifying which services to use, the order in which they should be executed, and how to orchestrate the flow of control and data through the sequence(s). Automating such a task is a computationally intensive effort better distributed between multiple agents. We present a client-server framework within which the efforts of composition creation and management are shared by several agents. Additionally, we present a composition algorithm that addresses a part of the task.

Keywords*:* Service Composition, Hypergraphs, Shortest Path algorithms, Dynamic Composition, Distributed Composition.

1. Introduction

A web service is a software platform designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL) [1]. With web services, tremendous reuse of applications based on functionality and the dynamic provision of computing resources may be achieved. Yet despite growing interest and recent efforts, Web services middleware is still rather primitive in terms of offered functionality : far from what enterprise application integration (EAI) middleware can provide for intra-enterprise applications. As a result, Web service development remains a fairly complex activity [2]. While implementing and deploying web services within the Service Oriented Architecture is an established procedure, standards to publish these are woefully inadequate for composition. Locating and integrating existing services across networks to provide a larger custom made solution remains an arduous task, accomplished only with significant developer effort. The area of automated web service composition, as a result, has received a lot of research attention in recent years.

Current WS discovery standards like UDDI solely use the static specification of a service's programming interface [5], requiring a programmer to search for a function by its name. Engineers and academicians generally agree that a more powerful WS publication model is needed if we expect *autonomic agents* to find, examine, and aggregate atomic services into high level compositions with little or no help from human programmers. Web Service Composition addresses the situation when a client request cannot be satisfied by a single available web service, but needs a custom-made composite service created by combining parts of available web services [3]. Several powerful formalisms have been used to model the different parts of the problem, like Finite State Machines [3,4], Petri Nets [8], State Charts [10,11] and Logical Reasoning and Rule Sets [7,9].

We break the Dynamic (also called Automatic) Service Composition problem into three primary subtasks – *(i) Semantic Reconciliation, (ii) Composition Synthesis* and *(iii) Runtime Orchestration*. Because web services are deployed by various organizations across the world, in reality it is highly unlikely that they will all name their interface variables consistently, or share an understanding about each of their functional scopes and responsibilities. In such an unstructured environment, composition approaches are reduced to non determinism and heuristics due to lack of conformance to a framework. In order to address the problem of using arbitrary semantics during specification, and in order to facilitate understanding between services, ontologies are used to define concepts (variables) and relations they share [11,12]. If web services conform to these ontologies in their service definition, then irreconcilable misunderstandings do not arise. Ontologies help to create a structured search space, which can then be approached by deterministic composition engines.

Preprint. To Appear in Proceedings of The 8th IEEE International Symposium on Autonomous Decentralized Systems.

Semantic Reconciliation deals with connecting services that may conform to different ontologies in a composite workflow. Composition Synthesis deals with the search problem and its combinatorial aspects. The creation of a composite web service given a set of available implemented services involves matching service interface data types, identifying the order in which the services should be executed, how the flow of control and data progresses through the sequence(s), etc. We must consider the possibility of large service repositories, and the resulting massive combinatorial search space that arises. Viable algorithms must attempt to optimize themselves accordingly. Lastly, Runtime Orchestration involves service invocation, passing data between services, temporal ordering of component services and maintaining state of the composite.

This work briefly addresses the first and third aspects of the problem, while focusing on the second aspect : Composition Synthesis. In accordance with prevalent terminology, we refer to the inputs and outputs of a web service definition as *concepts* belonging to an ontology [10,12]. Our abstraction treats concepts as independent atomic entities wlog, while relationships between concepts (e.g. concept aggregation into a larger concept) are ignored in the interest of clarity.

2. Motivation

A concomitant side effect of using powerful models to increase the semantic richness of a service specifications, is an increased computational burden on composition finding agents: in particular the agents managing service providers and service registries which may be swamped with client requests. In principle, the requirement of transparency to the end user implies that a composite service should be no different to handle than an atomic one: the end user should be unaware that the application he is using is actually composed of several heterogeneous web services. This would imply composite services should be created and made available at the registry like any other service.

Yet, we recognize that behind the scenes, composite services are very different to manage than atomic single point implementations. If an architecture assigns the entire responsibility of managing such services to its central agents, then we expect (i) Service Composition itself to become an expensive, possibly fee based service : since the provider is required to manage a composite service (at least partly) on behalf of a client, throughout its lifespan. (ii) Consequently, for service providers to avoid looking beyond pre-cached or locally available services, since their expected involvement is not a one time search effort, but rather a commitment to manage (interleave, execute, rollback, orchestrate, etc.) these third party services throughout the period of their usage by the client. Indeed, the result might be a new Service Oriented Architecture whose design impedes some fundamental goals of SOC like fostering the widespread reuse of e-Enabled Business Processes [1].

We propose instead that the shared, centrally located agents of WS registries and providers work on a minimal model (services modeled as simple input-output rules), identify a subset of services capable of meeting the requirements by calculating a shortest path flow, and return these along with their exported models to the requesting client. The onus of state maintenance, runtime sequencing and orchestration of the small set of identified services – activities that require an agent to exploit the fine grained model of a service – are left to the client. Dividing the computational effort of dynamic composition thus, dramatically reduces the burden on the central agents of the resulting architecture.

The proposed composition approach divides the work into a static, one time effort performed at the central agents (WS Composition Agent), and a possibly more compelling runtime effort invested through the lifespan of the composite service, left to the client agents (Travel Agent or Mobile User). Our central agent WS Composition algorithm uses a bottom-up approach to identify a shortest path sequence (or sequences) of web services from the WS Registry, which map user inputs to outputs. These are then returned to the client, which may then decide to use them in the order identified, or alternatively, identify which services to engage multiple times, in what order, how to sequence/parallelize/interleave their execution for optimal performance, and how to orchestrate the same.

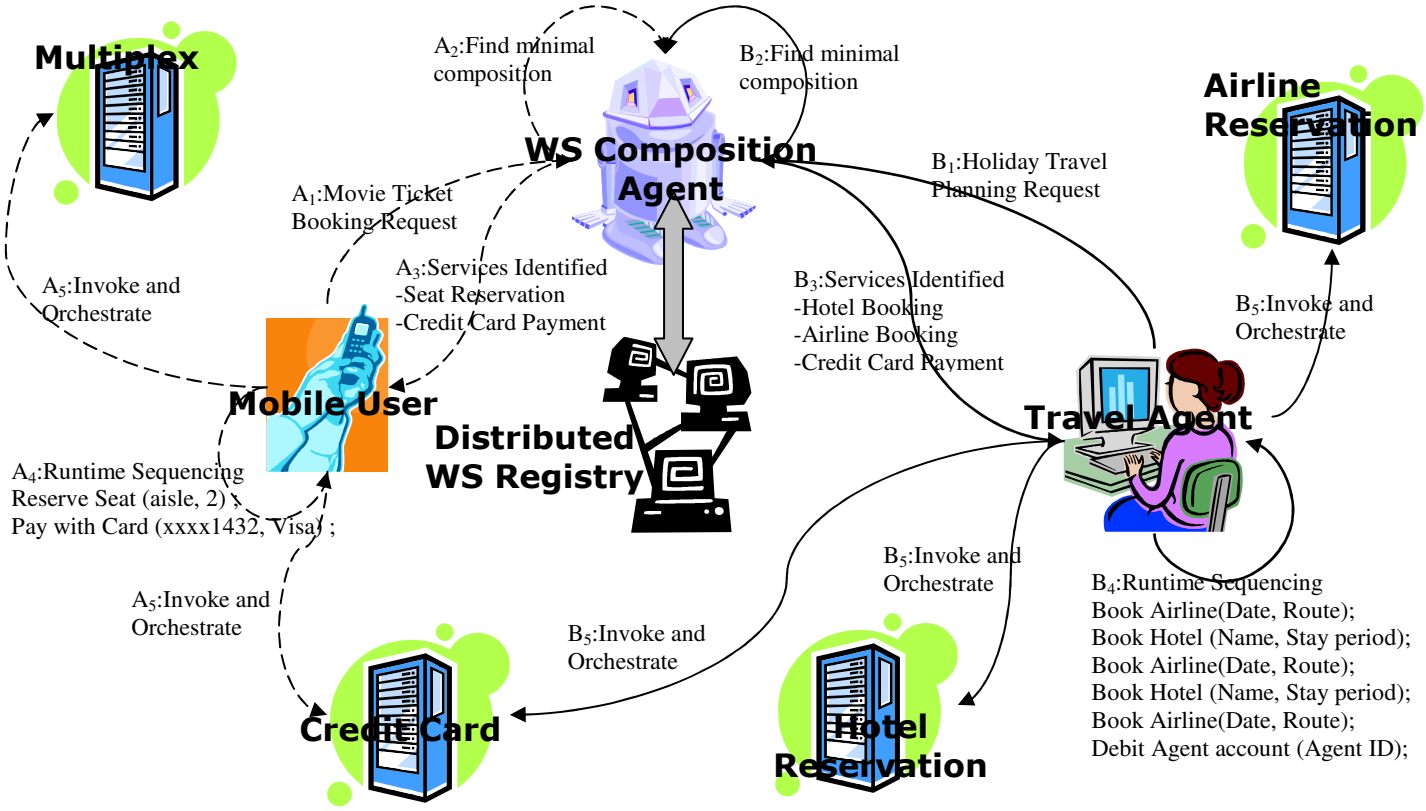

Figure 1 – Central agents in the architecture (WS Composition Agent) work on a graph model of the WS Registry as briskly as possible. The onus of managing the runtime of the composite service is left to client agents (Mobile user and Travel Agent). Note that distribution of the computational workload among multiple agents still allows us to maintain end user transparency.

3. Approach

We represent the set of user given input concepts by ***I*** and the set of user expected output concepts by the set ***O***. These are made available to the central agent. Let ***W*** be the set of existing web services that are available to the central agent, from which it has to identify services that are required by the user. Each web service in ***W*** is modeled as a rule (e.g. I_1 & I_2 & $I_3 \rightarrow O_1$) or a set of such rules [7], with the body of the rule representing the inputs, and the head representing the outputs of the service. The available web services in ***W*** may therefore be modeled by a directed graph as illustrated in Fig 3. The graph **G** is the search space, composed of vertices **V** representing concepts, and directed edges **E** representing web services in ***W***. Henceforth, we shall refer to vertices and concepts, as well as edges and web services interchangeably, as applicable in the discussion context.

The central composition algorithm works both forwards from seed ***I*** and backwards from seed ***O*** *simultaneously*, performing a breadth first search from each end (albeit differently) on **G**. At the end of each stage (BFS tree layer), it checks to see if there is a collision between the two trees. When a collision is detected, it signifies the shortest path calculation. An abstract representation of the algorithm's progress is given in Fig. 2. The illustration is a simplified visual aid : it does not capture how the algorithm deals with concept tuples, or multiple parallel searches. We consider this the optimal solution, though in practice it may not be the case. For

the time being we ignore aspects like service availability, reliability, security etc. though these may be accommodated with minor variations or add-ons to the algorithm. Search is then terminated and all services on the solution path are communicated to the client.

3.1 Central Algorithm

To accurately model a web service as an edge connecting a set of concepts, we use directed hypergraphs. Hypergraphs allow us to model web services more accurately than regular graphs: if there exists a web service that maps an input concept tuple to an output tuple, then we represent it by hypergraph edges. For example, a service with signature (City, Date)→(Weather, IsHoliday) may be represented by 2 edges (City, Date)→(Weather) and (City, Date)→(IsHoliday). Also, two distinct edges terminating in a vertex represent two

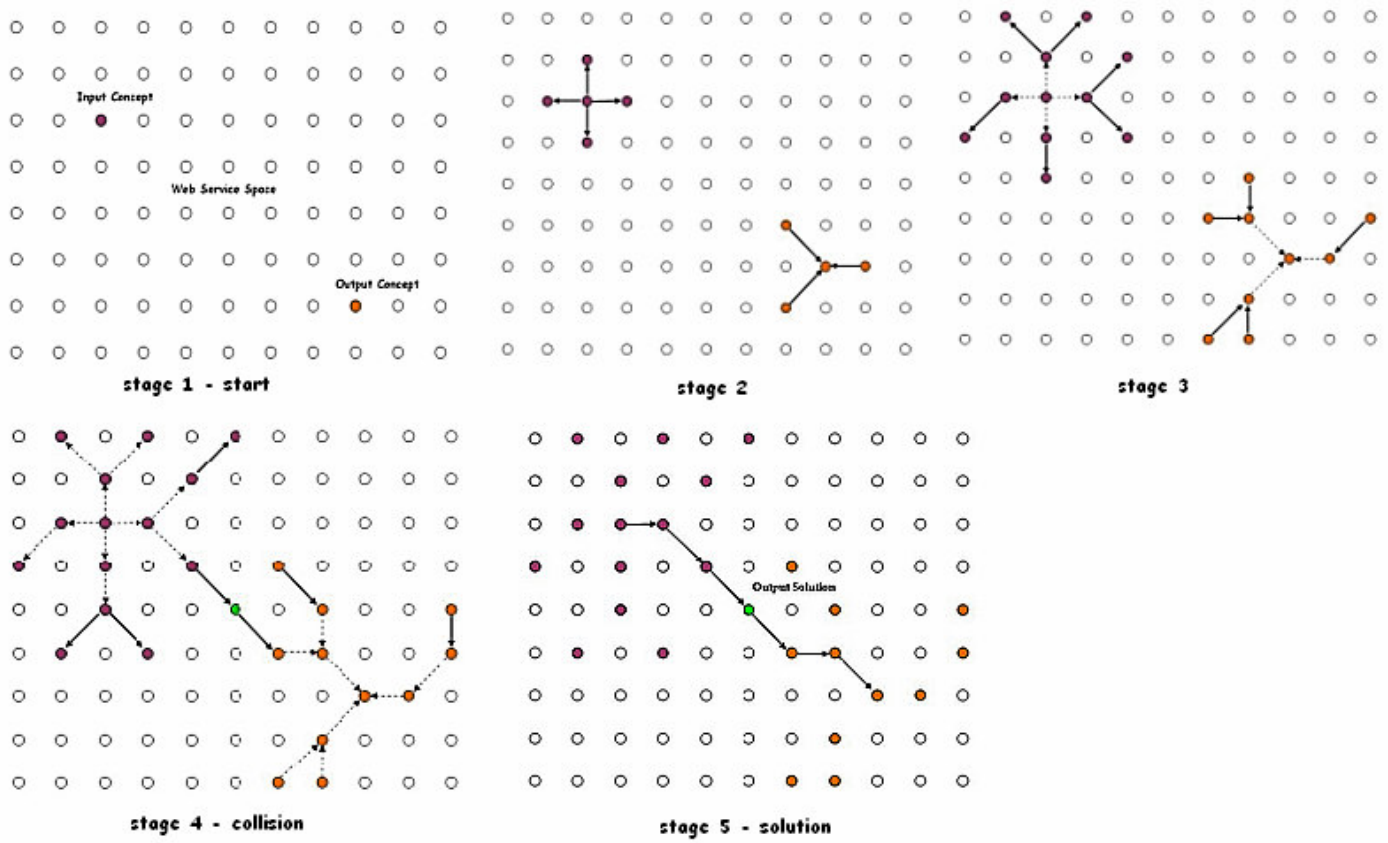

Figure 2 – Note the progress of the algorithm over the stages. When the two search trees collide at stage 4, search terminates and a back trace gives us a composition.

separate web services in ***W*** which output that concept, possibly independent of each other (Fig. 3).

The limited composition problem at the central agent may now be specified a : "Is there a subset of web services of ***W***, whose execution in some order, can produce the user expected outputs from the user given inputs? In other words, is there at least one path in graph **G**, from some subset of vertices in ***I***, to every output vertex in ***O***". A sample hypergraph modeling a set of web services is given in Fig. 3. Note how different edges in the graph may represent the same web service (if the service outputs more than one concept).

For a better intuitive understanding of the algorithm, we model using production rules akin to those used by finite automata grammars (e.g. S → aTb , S and T are variables, a and b are string constants). We may think of the solution in terms of reducing the start variable (that captures some hidden path from inputs to outputs) using a

series of substitutions, to a string of constants (edges) representing the solution. The algorithm progresses in stages. Let $\mathbf{s}^{\mathbf{fwd}}$ be the set of available inputs to the algorithm at any stage, while $\mathbf{s}^{\mathbf{bkd}}$ be a set of permissible outputs sets : any one set of which must be completely reached by the input. At the start of the algorithm $\mathbf{s}^{\mathbf{fwd}} = \boldsymbol{I}$ (user given input), $\mathbf{s}^{\mathbf{bkd}} = \{\boldsymbol{O}\}$ is a singleton of user expected output, and $\boldsymbol{R_{IO}}$ represents start variable. At each stage, the search for a solution progresses incrementally, working on results offered by the past stage(s). This increment may be modeled as the sequential application of productions: replacing the unsolved left hand side, with a partially solved right hand side of the production. A generalized production rule like the following may be imagined :

$$R_{IO} \rightarrow s^{fwd} R_{AB} s^{bkd} ;$$

where A is a vertex set defined as $A = \{ y \in V \mid \exists x \in s^{fwd}, (x \rightarrow y) \in E,\ y \notin s^{fwd} \}$

Assuming in general $s^{bkd} = \{ n_i \}$ at the previous stage, then $B = \{ m_j \}$ is a set of vertex sets, where each member set $m_j \in B$ *covers* some set $n_i \in \mathbf{s}^{\mathbf{bkd}}$ as follows:

$$m_j = \{ a \mid \forall b \in n_i, \exists a \in m_j \ where\ (a \rightarrow b) \in E, n_i \in s^{bkd} \}$$

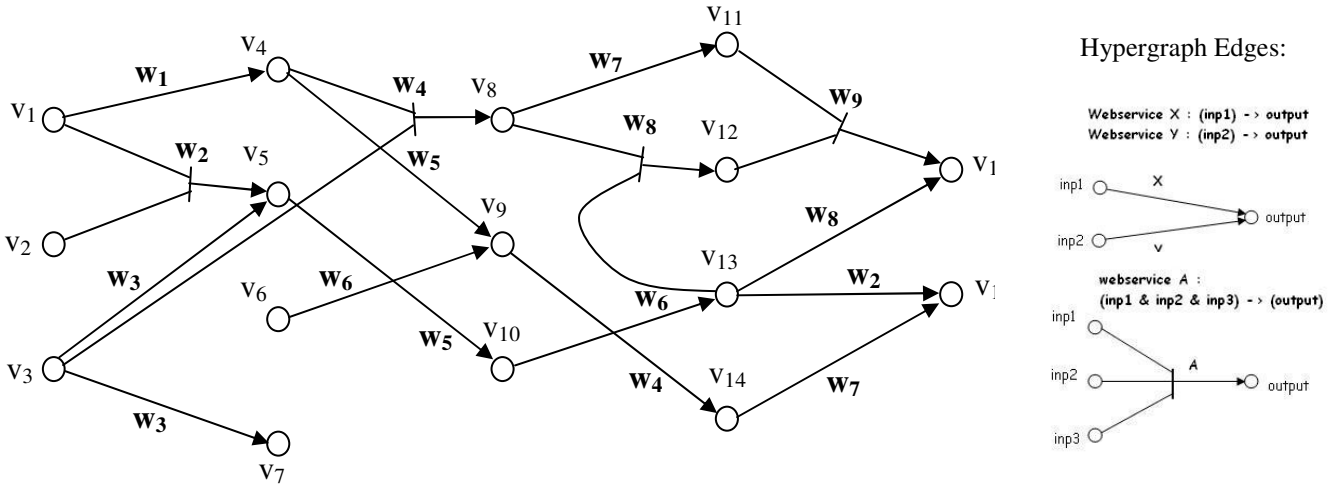

Figure 3 - Graph **G** of a sample Web service repository ***W*.** Note how web service w2 is represented by 2 edges : (v_1 &v_2)→v_5, and (v_{13})→v_{16}.

At each stage, after calculating A and B the algorithm updates the values of $\mathbf{s}^{\mathbf{fwd}}$ and $\mathbf{s}^{\mathbf{bkd}}$ as :

$$s^{fwd} = s^{fwd} \cup A$$
$$s^{bkd} = B$$

Note the difference between the two. At each stage $\mathbf{s}^{\mathbf{fwd}}$ *adds to itself* newly reachable vertices (A) of **G,** which are at unit distance from the previous stage vertices**.** $\mathbf{s}^{\mathbf{bkd}}$ on the other hand, *replaces itself* with a set of vertex sets B. Each member set of B contains vertices that suffice *to reach* all the vertices of some $n \in \mathbf{s}^{\mathbf{bkd}}$ of the previous stage. Since the base case is $\mathbf{s}^{\mathbf{bkd}} = \{\boldsymbol{O}\}$, by an inductive argument each member set of $\mathbf{s}^{\mathbf{bkd}}$ at stage k is

sufficient to reach all vertices of **O**. $\mathbf{R_{AB}}$ is a variable representing the remaining sub problem : finding paths in **G** from A to any one member set of B. Refer to the pseudocode for an elaborate treatment.

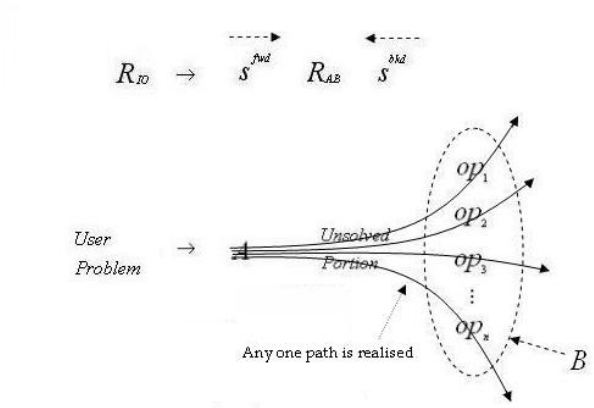

Figure 4 – The start variable $\mathbf{R_{IO}}$ is eaten into from both ends, till a solution is found (or alternatively, till the search space runs out).

The shortest sequence(s) of services identified are then communicated to the client agent (Fig.1). Indeed, the returned set of services are by definition sufficient to satisfy the client's requirement – that of producing output concepts from given inputs. These form a *minimal service set* (analogous to a programming language instruction set), which may then be configured (programmed) to generate a runtime composition (code script). The responsibility of programming this minimal service set is delegated to the client agent. Given such a set, the client agent can generate a script that will wrap these services into a workflow based on other end user inputs like number of orders to be placed (service instances to be engaged). The client agent can also perform other optimizations on the composition like deciding which services to invoke simultaneously, which in sequence, and how to interleave their executions in the generated script.

For example, the minimal service set identified for the Travel Agent in Fig.1 is `HotelBooking`, `AirlineBooking` and `CreditCard`. The client agent armed with additional specifications like the customer's itinerary, may engage these services in any order and as many times as needed (subject to issues like QoS requirements and service availability). The generated script will round off the entire process of booking a customer's business trip:

```
BillableSession sessionID = CreditCard.NewSession(TravelAgent ID)

AirlineBooking(24th Jan, Delhi to NY, sessionID);
HotelBooking(Hilton, 25th Jan and 26th Jan, sessionID);
AirlineBooking(26th Jan, NY to Paris, sessionID);
HotelBooking(Four Seasons, 26th Jan, sessionID);
AirlineBooking(27th Jan, Paris to Delhi, sessionID);

CreditCard.DebitAccount(sessionID);
```

In the absence of explicit co-operation between two services in the set, like `AirlineBooking` and `CreditCard`, the responsibility of data persistence and state maintenance is left to the client agent, not to any central agent of the architecture. Here, `sessionID` is an identifier that enables the `AirlineBooking` and `HotelBooking` services to perform stateful transactions with `the CreditCard` Server in the spirit of SOA (service-service interaction) and **not** a requirement that the client agent act as central memory store for state data.

As an aside, it must be mentioned that provisions must be made to the central algorithm to prevent rework. The Reverse thread works on the results of a set Cartesian product of the previous stage's search. If a vertex is a member of several identified sets in B, then its parents need be calculated only once (Fig.5). There is a surplus of literature on algorithms for digraph BFS, and we won't labor over these details as they do not shed new light on the fundamental idea. The Appendix contains an example illustrating the central algorithm's working, again without some obvious optimizations for purpose of clarity.

3.2 Algorithm Pseudocode

Data structures:
User given input concepts : ***I***
User expected output concepts : ***O***
Available web service repository : ***W***(w1, w2, w3 … wi),
where each member service w is specified as
w : (i1, i2 … ij)→(o1, o1, … ok)
Web service graph constructed : **G**(**V**,**E**), where

$$\mathbf{V} = \left\{\bigcup_{W} inputs(w_i)\right\} \bigcup \left\{\bigcup_{W} outputs(w_i)\right\}$$

$$\mathbf{E} = \left\{(x \rightarrow y) \mid \exists w \in W \text{ where } inputs(w) = x \text{ and } outputs(w) = y\right\}$$

Available inputs at some stage : $\mathbf{s}^{\mathbf{fwd}}$
Required output sets (cover any one) at some stage : $\mathbf{s}^{\mathbf{bkd}}$
Vertices at unit distance from $\mathbf{s}^{\mathbf{fwd}}$: A
Sets of vertices covering sets in $\mathbf{s}^{\mathbf{bkd}}$: B

Initialization:
$\mathbf{s}^{\mathbf{fwd}}$ = ***I***; A = Φ; $\mathbf{s}^{\mathbf{bkd}}$ = {***O***}; B = {Φ};

Stage: Process forward and reverse threads in parallel.

Forward thread :

For all subsets s $\subseteq$ $\mathbf{s}^{\mathbf{fwd}}$
{
if($\exists t \in V \mid (s \rightarrow t) \in E$ and $t \notin A$)
Update $A = A \cup t$;
}

if (A==Φ)
Terminate search. No solution exists.

//else
Update $\mathbf{s}^{\mathbf{fwd}}$ = $\mathbf{s}^{\mathbf{fwd}}$ ∪A;

//reset
A = Φ;

Reverse thread :

For each set n ∈ $\mathbf{s}^{\mathbf{bkd}}$
{
For each vertex vi ∈ n {
//Find parent vertices
$p_i = \{w \in V \mid (w \rightarrow v_i) \in E, w \notin n\}$;
}

//Create set of parent sets
Calculate Cartesian
product $P = \mathbf{X}_{v \in n} p_i$;
Update $B = B \cup P$;
} //end for

if (B=={Φ})
No solution exists. Terminate search.

//else
Update $\mathbf{s}^{\mathbf{bkd}}$ = B;

//reset
B = Φ;

Synchronization: Check for vertices at which threads have collided.
For each set n ∈ $\mathbf{s}^{\mathbf{bkd}}$ {

```
        if (∃v∈n | v∈ S^fwd )
                Mark vertex v as reached;
                Update n = n-{v};
        if (cardinality  |n| = 0)
                Add set n to SolutionSet.
}

For each m ∈ SolutionSet {
        Trace back in either direction gives 2 sets of half-solutions (half path(s)).
        Output combinations – solution is any member of the Cartesian product of the
        2 sets.
}

Primary Loop:
main()
{
        Run Initialization code;

        while(no termination signal)
        {
                Run Stage code;
                Run Synchronization code;
        }
}
```

4. Discussion

Identifying the minimal service set requires only an inspection of the web service signature (static effort of composition), as long as the interface data types conform to established ontologies. Computationally heavier operations (dynamic part) that require an agent to consider the internal structure of service by inspecting semantically rich models of the service like FSMs [4] are left to clients. The rationale behind this division of labor is that programming to a small set of identified services (instructions) is computationally far cheaper than programming to a massive (possibly distributed) universe of services. Additionally, the modeling required for service identification is minimal (directed hypergraph), and the internal state transitions and control structures [3,6,7] of only those belonging to the minimal service set need be pulled out and examined.

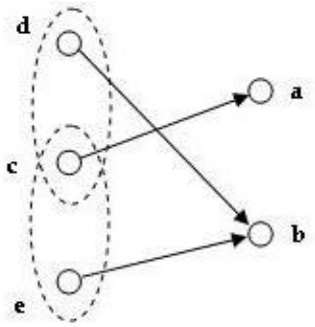

Figure 5 – Rework. If **O** = {a,b} and p_a = {c}, p_b = {d,e}as shown, then P = $p_a \times p_b$ = { {c,d},{c,e} } contains two distinct sets that cover **O**, with a common parent c.

4.1 Advantages

A large number of WS composition algorithms use some manner of graph search to identify paths from input to output [3,6,7,8]. Depth first searches are less preferred since they often rely on heuristics to identify which search routes must be abandoned and which prioritized, and also require considerable amounts of backtracking. Breadth

first search techniques on the other hand, suffer from large space complexities (memory requirements). Consider a large web service graph **G** with an average vertex degree (of outgoing edges) **k**. In the worst case scenario, where the algorithm finds entirely new vertices at each stage, the BFS grows like a tree. The vertices to be remembered by a conventional BFS search after **d** stages is $\sum_{0}^{d} k^{i}$, which is $\Theta(\mathbf{k^d})$. Bidirectional BFS reduces the memory requirements exponentially. If a solution path of length **d** exists then the search in each direction will require $O(\mathbf{k}^{\lceil \mathbf{d}/2 \rceil})$ memory, since they collide at most after $\lceil \mathbf{d}/2 \rceil$ stages. Thus the space requirements of the central algorithm is $2*O(\mathbf{k}^{\lceil \mathbf{d}/2 \rceil}) = O(\mathbf{k}^{\lceil \mathbf{d}/2 \rceil})$. Since both search trees grow synchronously, each covering half the distance, the algorithm in its vanilla version takes not more than half the number of stages of a regular BFS (notwithstanding exponentially larger computational effort with each progressive stage). Once again the actual time complexity improvements noticed are subject to the characteristics like connectivity of the WS graph. If we assume worst case tree like growth then both space and time improvements over a regular BFS are exponential.

Unlike some of its peers [7] , the algorithm does not require the explicit calculation of transitive path closure over **G** : which can be computationally expensive if **G** has a dense edge population (i.e. high vertex connectivity). Computing complete input or output closure can be overkill to find a solution. Also, dependencies of individual output concepts w.r.t. inputs need not be stated (or sometimes guessed) apriori by the user, especially if these dependencies are not known. However, significant improvements in performance of the algorithm may be made (by reduction in search space) if these are available. In a graph of non-uniform edge (web service) density, the forward or reverse search spaces may grow disproportionately w.r.t one another. In such a case, search in the faster direction may be paused, while the other may proceed till collision.

4.2 Drawbacks

The primary drawback of the central algorithm suffers is its inability to detect the absence of a solution without growing through the entire graph: effective to calculating either forward or reverse closures (reverse direction closure, in this case, means an empty parent set B). This also means that closure calculation is only the worst case outcome of the run. The first direction to achieve a closure can terminate the algorithm (the checks A==Φ and B=={Φ}). Applying a heuristic measure to terminate the search after a certain search depth is another way to address the problem. Secondly, synchronization after each stage is a costly procedure (time wise) that reduces effective parallelism that may be achieved.

The discussion hitherto only addresses the computational aspects of a larger WS composition problem. Nevertheless, the algorithm can accommodate salient aspects of web service composition like semantics reconciliation with available extensions [2,10,11]. The problems associated with semantics and reconciliation between services, though significant, are beyond the scope of this discussion.

5. Conclusion

The paper presents an approach to Dynamic WS Composition that requires minimal modifications to the existing SOA architecture. We minimize workload on shared resources by delegating a one-time effort of service identification to them. Fine grained model parameters are exploited at the client end, and the newly generated composite service *need not* itself be modeled and persisted for re-use at the server end. The presented algorithm identifies a minimal set of services needed for composition, based solely on static service signature. The algorithm is purely deterministic, though with extensions/modifications, it can be changed to find the best available solution. Also, any independent approach to address the semantic aspects of WS Composition may be combined with ours.

6. References

[1] D. Booth et al., "Web Services Architecture", http://www.w3.org/TR/ws-arch,W3C working group note.

[2] Benatallah, Casati and Toumani, "Web Service Conversation Modeling – A Cornerstone for E-Business Automation", *IEEE Internet Computing,* Jan-Feb 2004.

[3] D. Berardi, et al., "Automatic Composition of E-services that Export their Behaviour", *Proceedings of the First International Conference on Service Oriented Computing (ICSOC)*, 2003

[4] D. Berardi et al., "Finite State Automata as a conceptual model for e-Services", *Integrated Design and Process Technology*, 2003

[5] D. Berardi et al., "Automatic Composition of Process Based Web Services : A Challenge", url - http://citeseer.ist.psu.edu/734246.html

[6] S. Hashemian and F. Mavaddat, "A Graph-Based Approach to Web Services Composition", *Proceedings of the 2005 Symposium on Applications and the Internet (SAINT'05).*

[7] J. Liu, J.Ciu and N. Gu, "Composing Web Services Dynamically and Semantically", *Proceedings of the IEEE International Conference on E-Commerce technology for Dynamic E-Business (CEC-East'04).*

[8] R. Hamadi and B. Benatallah, "A Petri Net Based Model for Web Service Composition", *Proceedings of the Fourteenth Australasian database conference on Database Technologies 2003*.

[9] J. Rao, P.Küungas and M. Matskin, "Application of Linear Logic to Web Service Composition", *Proceedings of the First International Conference on Web Services (ICWS'03)*, 2003.

[10] Z. Maamar, N. Narendra and P. Thiran, "Towards a Coordination Model for Web Services"

[11] Z. Maamar and N. Narendra, "Ontology-based Context Reconciliation in a Web Services Environment: From OWL-S to OWL-C"

[12] Natalya F. Noy and Deborah L. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology"

Appendix A. Sample run of central algorithm

Consider the WS Hypergraph model in Fig. 3. Given :
User inputs **I** = {v1, v2, v3}, Required outputs **O** = {v15, v16}
Web service repository **W:**(w_1 to w_9) specified by the rules :
$w_1 : (v_1) \rightarrow v_4$
$w_2 : (v_1 \ \& \ v_2) \rightarrow v_5$
$w_3 : (v_3) \rightarrow v_5 , (v_3) \rightarrow v_7$
$w_4 : (v_3 \ \& \ v_4) \rightarrow v_8$
$w_5 : (v_4) \rightarrow v_9 , (v_5) \rightarrow v_{10}$
$w_6 : (v_6) \rightarrow v_9 , (v_{10}) \rightarrow v_{13}$
$w_7 : (v_8) \rightarrow v_{13} , (v_{14}) \rightarrow v_{16}$
$w_8 : (v_8 \ \& \ v_{13}) \rightarrow v_{12} , (v_{13}) \rightarrow v_{15}$
$w_9 : (v_{11} \ \& \ v_{12}) \rightarrow v_{15}$
Here is a trace of the algorithm's progress over **G**:

Initialization :
$\mathbf{s^{fwd}} = \{v_1, v_2, v_3\}$**;** A = Φ; $\mathbf{s^{bkd}} = \{\{v_{15}, v_{16}\}\}$; B = {Φ};

Stage 1:
A = $\{v_4, v_5, v_7\}$ //using w_1, w_2 and w_3 respectively.
$\mathbf{s^{fwd}} = \mathbf{s^{fwd}} \cup A = \{v_1, v_2, v_3, v_4, v_5, v_7\}$;

parents of $v_{15} = \{v_{13} , \{v_{11}, v_{12}\}\}$ //using w_8 and w_9
parents of $v_{16} = \{v_{13} , v_{14}\}$ //using w_2 and w_7

parent set that covers output $\{v_{15}, v_{16}\}$:
P = { $\{v_{13}\}$, $\{v_{13}, v_{14}\}$, $\{\{v_{11}, v_{12}\}, v_{13}\}$, $\{\{v_{11}, v_{12}\}, v_{14}\}$ }
B = P = $\{\{v_{13}\}, \{v_{13}, v_{14}\}, \{\{v_{11}, v_{12}\}, v_{13}\}, \{\{v_{11}, v_{12}\}, v_{14}\}\}$
$\mathbf{s^{bkd}}$ = B = $\{\{v_{13}\}, \{v_{13}, v_{14}\}, \{\{v_{11}, v_{12}\}, v_{13}\}, \{\{v_{11}, v_{12}\}, v_{14}\}\}$

Synchronization after stage 1:
No vertices common to $\mathbf{s^{fwd}}$ and $\mathbf{s^{bkd}}$. Proceed.

Stage 2:
A = $\{v_8, v_9, v_{10}\}$ // using w_4, w_5 and w_5 respectively
$\mathbf{s^{fwd}}$ = $\{v_1, v_2, v_3, v_4, v_5, v_7, v_8, v_9, v_{10}\}$

parents of v_{11} = $\{v_8\}$, parents of v_{12} = $\{\{v_8, v_{13}\}\}$,
parents of v_{13} = $\{v_{10}\}$, parents of v_{14} = $\{v_9\}$

parent cover of set $\{v_{13}\}$ = $\{v_{10}\}$, parent cover of set $\{v_{13}, v_{14}\}$ = $\{v_9, v_{10}\}$, parent cover of set $\{\{v_{11}, v_{12}\}, v_{13}\}$ = $\{\{v_8, v_{13}\}, v_{10}\}$, parent cover of set $\{\{v_{11}, v_{12}\}, v_{14}\}$ = $\{\{v_8, v_{13}\}, v_9\}$

B = { $\{v_{10}\}$, $\{v_9, v_{10}\}$, $\{\{v_8, v_{13}\}, v_{10}\}$, $\{\{v_8, v_{13}\}, v_9\}$ }
$\mathbf{s^{bkd}}$ = { $\{v_{10}\}$, $\{v_9, v_{10}\}$, $\{\{v_8, v_{13}\}, v_{10}\}$, $\{\{v_8, v_{13}\}, v_9\}$ }

Synchronization after stage 2:
Cancel all common vertices and check for null set in $\mathbf{s^{bkd}}$
$\mathbf{s^{fwd}}$ = $\{v_1, v_2, v_3, v_4, v_5, v_7, v_8, v_9, v_{10}\}$
$\mathbf{s^{bkd}}$ = { $\{\cancel{v_{10}}\}$, $\{\cancel{v_9}, \cancel{v_{10}}\}$, $\{\{\cancel{v_8}, v_{13}\}, \cancel{v_{10}}\}$, $\{\{\cancel{v_8}, v_{13}\}, \cancel{v_9}\}$ }

2 empty sets $\{\cancel{v_{10}}\}$and $\{\cancel{v_9}, \cancel{v_{10}}\}$ signify 2 collisions, whose half-solution sets are :
$\{v_{10}\}_{fwd}$ = { w2→w5, w3→w5}, $\{v_{10}\}_{bkd}$ = {(w8,w2)←w6}
$\{v_9, v_{10}\}_{fwd}$ = {(w1,w2)→(w5,w5) , (w1,w3)→(w5,w5)}
$\{v_9, v_{10}\}_{bkd}$ = {(w7,w8)←(w4,w6)}
Minimal service sets = $\{v_{10}\}_{fwd} \times \{v_{10}\}_{bkd}$ union $\{v_9, v_{10}\}_{fwd} \times \{v_9, v_{10}\}_{bkd}$. Any one may be selected.