Dependency Based Automatic Service Composition using Directed Graph

Abrehet M.Omer
Chair for Computer Networks
TU Dresden
Dresden, Germany
Abrehet_Mohammed.omer@mailbox.tu-dresden.de

Alexander Schill
Chair for Computer Networks
TU Dresden
Dresden, Germany
alexander.schill@tu-dresden.de

***Abstract*—In this paper a method of automatic composition plan creation that relies on automatic extraction of dependencies among services is investigated. For automatic dependency extraction our approach makes use of semantic similarities between I/O parameters of services. Extracted I/O dependencies are represented using a directed graph. The approach recognizes when cyclic dependencies exist and proposes a way of dealing with it. Modified topological sorting algorithm is used for the execution plan generation showing execution order of candidate services. A case study is used to explain the proposed approach.**

***Keywords*-Service dependency; Directed graph; cyclic dependency; Composition Plan.**

I. INTRODUCTION

Web services(WSs) are self-contained, modular units of application logic, which provide business functionality to other applications/users via an Internet connection. WSs are not dependent on the context or state of other web services. The development process of web services has become sufficiently mature. At present more and more small and simple applications are being developed and made available in the form of WS. As a result developers/researchers start working towards other potential usage of web services like developing applications making use of existing web services. Such ways of application development lead to the emerging application development architecture service-oriented architecture (SOA). The building blocks of SOA-based applications are web services that can be reused across various applications. Consequently, composition of web services has received increased interest with SOA.

WS composition is a mechanism of combining two or more basic services into a possibly composite service/application. Composite service/applications are expected to satisfy requirements that cannot be satisfied by individual services. Thus service composition enhances reuse of web services and decreases the effort to develop new applications from scratch.

Service composition could be done either statically, or (semi/fully) dynamically. Dynamic composition techniques are getting more preference due to their potential for handling unpredictable changes of runtime environment that cannot be handled using static composition techniques. However, dynamic composition techniques are not extensively practiced because there are still weak links in its implementation approaches. Realizing dynamic service composition requires not only runtime service binding but it also demands ability for automatic creation of execution plan for the composite service [1]. Techniques for doing the latter are still not rudimentary. In this regard automation of execution plan model creation process is one of the core problems hindering the transition towards dynamic service composition and that needs to be solved. Our approach focuses on this problem.

In dynamic Web services composition software agents are responsible for automatic creation of execution plans. The candidate services for the composition possibly made available by different providers that use different naming systems. This necessitates for semantic descriptions of services to assist IOPE matching of the software agent. Among the challenges for success of semantic WSs are making automatic service discovery and composition techniques efficient. In this regard [2] present how services can be discovered by their capability using semantic knowledge of services. Moreover, to make the discovery process more flexible, they propose four different degrees of match functions to ensure the match between input/output of service advertisement and input/output of the user request. Our approach assumes the existence of such match making techniques.

Individual services forming a composite service have to coordinate and communicate to accomplish an intended task. Such kind of communication brings with it dependencies between services. Thus one key challenge of web service composition is managing such kind of collaboration among WSs.

Our approach tries to forecast and extract potential dependency between candidate services and then use the dependency information to generate an execution plan automatically. We assume the existence of a local repository with abstract semantic description of web services. The user is expected to describe his or her request in terms of goals and candidate service descriptions will be discovered from local repository using goal based discovery mechanism. We will not discuss the candidate service discovery process because it is beyond the scope of this paper. To extract I/O dependency our approach uses the concept of finding the

	Web services	Inputs	Outputs
WS1	PurchaseOrderCreator	Item Quantity Part_number Description Req. date Item_location Price	PurchaseOrder {Item Quantity Part_ number Description Req. date Item_location Price}
WS2	ItemAvailabilitychecker	Item Quantity	In stock or out of stock
WS3	Payment	Bank_detail Purchaseorder	Receipt
WS4	GetItem	Item_detail Availabil- ity_in_stock	Item_location num- ber_of_item part_number price
WS5	ShippingArrangement	Item location Customer detail payment	Delivery date Tracking number

Table I
CASE STUDY INPUT/OUTPUT DESCRIPTION

semantic similarity between service inputs and outputs. This approach utilizes existing graph traversal based algorithms to extract cyclic dependency and generate the execution plan.

A case study which will be used through out the paper is presented in section II. Section III presents the proposed methodology from identifying dependency to execution plan generation. Section IV gives concise review of related works and is followed by discussion in section V. Finally, conclusions and further works are given in section VI.

II. CASE STUDY

A case study an online shopping is considered as an example. In online shopping venture consumers browse through online catalogs select their preference item, purchase and get it delivered.

For this scenario the following web services are considered: WS1 (PurchaseOrderCreator) returns purchase order of all items selected by user for purchase given all details about the items; WS2 (ItemAvailabilitychecker) returns the availability of item in stock given item detail and; WS3 (Payment) returns payment confirmation given credit card detail and purchase order; WS4(GetItem) returns Item location, part number and price given item name; WS5 (ShippingArrangement) returns Delivery date and Tracking number given Item location, customer detail and payment confirmation(Receipt).Figure I shows the input and output description of each service.

III. PROPOSED APPROACH

A. Overview of dependency

Primarily services that are created by same or different providers are meant to be accessed and work independent to each other. But, establishment of composite services based applications necessitates interaction, communication, cooperation and coordination of services. This leads to emergence of different types of dependency among services involved in composite services, such as:

1) *Input/Output dependency*: occurs when a service requires/or provides data from/to another service.
2) *Constraint dependency*: occurs due to user constraints.
3) *Cause and Effect dependency*: occurs when a service has preconditions to be satisfied based on the effect of other services.

Such dependencies could occur between two services directly which we call it *direct dependency* or indirectly between two services through an intermediate service(s) which we call it *indirect dependency*. Service dependency can also occur in explicit or implicit manner. *Explicit dependency* can be readily visible and extractable from service descriptions. *Implicit dependency* are not directly expressed in service descriptions.

We claim that by utilizing the direct dependency between abstract service descriptions, one can generate an execution plan for the composite service. To prove this claim we extract explicit direct I/O dependencies automatically and represent it using directed graph and then utilize it for composite execution plan creation.

B. Composite Service Request and abstract service specification

Extracting dependencies from candidate services for composition requires suitable ways of describing web services and user requests. The proposed approach bases on formal descriptions from both the user and service side. Currently we are working on conceptual implementation of the proposed approach and our interest is conceptual description of services and user requests. For our intention, the description of web services and request includes tuple (I, O, P, E, G) [3]:

- I: list of inputs.
- O: list of output parameters
- P: the precondition. It describes a logical expression that must be satisfied in order to invoke the composite service.
- E: the effect. It describes the changes to the current state resulting from the invocation of composite service.
- G: Goal. It describes the goals that will be achieved by the composite service.

We assume the availability of a local repository that stores abstract service description in the above format. Such abstract description includes only a single description for all web services with equivalent functionality regardless of their quality. Thus candidate abstract services will be discovered from the local repository based on user requirement goal definition. Then dependency between those abstract services

will be extracted for execution plan generation. We defined same structure of description for both user request and services because it is one requirement for candidate abstract service discovery.

Concrete service binding for the actual service composition will be done after execution plan creation. It will be done based on abstract description and additional non-functional property.

C. Execution plan generation procedure

There are three components in our proposed architecture: dependency graph generator, dependency analyzer and execution plan generator. We summarize the tasks of these components as follows:

1) Identify explicit direct dependencies from input and output parameters of WSs and construct dependency graph.
2) Find out all cyclic dependencies if there are any using algorithm developed by [4] to find cycle in directed graph.
3) Regenerate the graph by making each cyclic sub graph as one compound node.
4) Calculate the number of services dependent on a particular service by counting incoming edges from the graph found in step 3.
5) Calculate the number of other services dependent on a particular service by counting the outgoing edges from the graph found in step 3.
6) Use graph traversal algorithm(modified topological sorting) to generate an execution plan based on calculated values in step 2 and step 3.

In the rest of this section each of the above steps will be described in detail along with illustrative case study.

1) Construction of dependency graph: An explicit direct I/O dependency between two services exists if at least one output of a service is taken as input by the other service. During service composition all inputs of web services are either from user request or are output of another web service. For the purpose of explaining the proposed approach we use an example that has almost perfect match between I/O parameters. However, in real case scenario we do not get services where their interface shows a perfect match. Thus, here the extraction of explicit direct I/O dependency is done using semantically enabled I/O matching a technique, which is adopted from [2]. It uses the following three semantic I/O matching functions proposed by [2] and intersection proposed by [5].

1) Exact : If the output parameter of WS_1 and the input parameter WS_2 are equivalent concepts;
2) Plug in : If output of WS_1 is sub-concept of input WS_2;
3) Fail : if all the above conditions are not satisfied
4) Intersection $\bigcap$: If the intersection of output of WS_1 and input WS_2 is satisfiable.

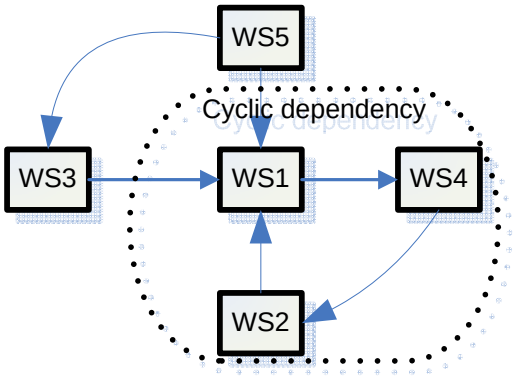

Figure 1. Direct Dependency Graph(DDG)

The dependency graph generator checks the intersection between the whole set of input parameters of one service with the whole set of output parameters of the other service. To do the intersection operation each input parameter should be checked with the output parameter using exact or plug in function. i.e. In $(WS_1) \cap$ Out $(WS_2) \neq \emptyset$ if and only if at least one pair of parameter set (each from Input(WS_1) and Output(WS_2)) has either exact or plug in relationship. This is done because our main aim is to find out from which services a particular service gets its inputs, i.e. on which services it is dependent. We do not consider the degree of match between I/O parameters of services.

Dependency can be represented as graph or matrix based model. In this approach, directed graph which can be equivalently represented by adjacency matrix is used to represent I/O dependencies between services. It is also used in [6]–[8] to represent dependencies between service. The graph that models the dependency will have n nodes where n equals available services to form the composite service and edges represent the dependency link. The edge direction indicates the service dependency flow. i.e. if i^{th} service is dependent on j^{th} service then there will be directed edge from WS_i to WS_j. Figure 1 shows explicit direct input/output dependencies for the on line shopping scenario.

2) Finding Cyclic Dependency: The dependency graph shows either unidirectional or bidirectional communication between services. In unidirectional communication one service gives its outputs and the other receives. As a result there will be one way dependency between a service input provider and receiver. When all dependencies are unidirectional the dependency graph also will be direct acyclic graph. In cases of bidirectional communication a service starts execution and gives partial output to another service and waits for reply to finish execution. Or service(s) may be required to be invoked and exchange data a number of times. In such cases the dependency graph will include cyclic dependency. In such case most of the existing approaches that use graph traversal algorithm to find execution plan

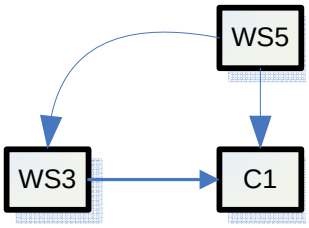

Figure 2. Direct Dependency Acyclic Graph(DDAG)

fails because their bottom line assumption is non existence of cyclic dependency. As a result finding and extracting dependency has become compulsory during execution plan generation. We propose a way to extract cyclic dependency and regeneration of acyclic dependency graph as a first step of execution plan generation.

To extract the cyclic dependency we used an algorithm [4] to find cycles in directed graphs. This algorithm enumerates all cycles in the by taking the DDG in the form of adjacency list. By considering each cyclic component subgraph identified as one compound node a new acyclic graph is generated. Figure 2 shows the acyclic dependency graph for the considered scenario.

3) Dependency analysis: From the dependency graph which is free of cyclic dependency we get two straightforward but important indicators that will be used during execution plan generation. First, the number of other services that is dependent on a given service (C_A) which can be found by counting incoming edges. Second, the number of services a given service is dependent on (C_B), which can be found by counting the number of outgoing edges from the dependency graph.

In this approach C_A is used to get which services can be executed first. And C_B value is used in a service(s) selection criteria that can be included in execution plan. Thus the two values have a key role in the execution plan generation algorithm.

4) Composition Algorithm: The execution plan is generated using a topological sorting algorithm. Topological sorting is often used in scheduling jobs or task given precedence constraints. In our case the precedence constraint is the dependency graph. It takes acyclic graph and outputs a linear ordering tasks (node/services). We adopt modified topological sorting that is used to sort threads that can be executed concurrently [9](see the above algorithm). The execution plan generated by this algorithm for the on line shopping scenario is given in figure 3. This execution plan includes a compound node since its input is the regenerated acyclic graph that also has a compound node.

To get the final execution plan the compound node has to be replaced by their execution plan that involves loop

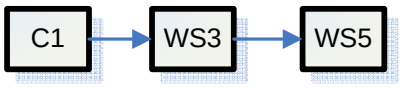

Figure 3. Execution plan with compound node (DDG)

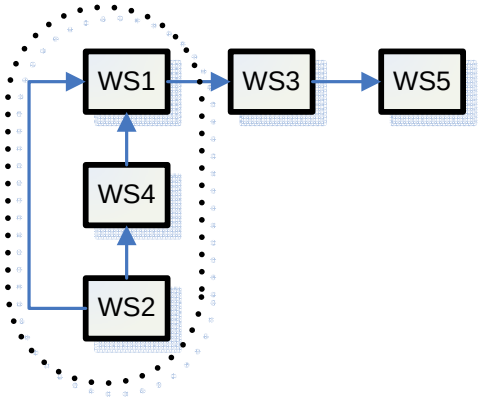

Figure 4. Final execution plan

control flow. To do this it is only required to get the starting node of the cycle. Then by traversing the dependency graph in backward direction the order of execution of the WSs that are involved in the loop can be determined. This simplified approach that create execution sub-plan for cyclic component assumes service execution inside the loop is only sequential which occurs in most cases. However, in case of other control flows nested within the loop recursive and repetitive use of Topological sorting algorithm is required. The final execution plan generated for the on line shopping scenario is shown in figure 4.

{MODIFIED TOPOLOGICAL SORTING ALGORITHM}
$INPUT : Dependency\ graph\ G(V, E)$
$OUTPUT : Path(C_0, C_1 ... C_N)$
{ path contains a sequence of group of services }
$L \leftarrow 0$
$C_0 = C_1 = ... = C_N = Empty$
{Ci contains a service or services that can be executed concurrently }
while V is Non-Empty **do**
  $C_L \leftarrow all\ v\ in\ V\ \ without\ outgoing\ edge$
  $E \leftarrow E - \{all\ E\ that\ start\ from\ v\ in\ C_L\}$
  $Path \leftarrow Path + C_L$
  $L \leftarrow L + 1$
end while

IV. Related work

In this section we present a brief overview of web service composition techniques. We consider techniques that use service dependency information, graph structure, and seman-

tics. The concept of dependency is explored initially for the purpose of managing component based systems [10]. The work by [11] looks at service dependencies from composite service management point of view. In their approach it is shown that dependencies could be tracked from log files that normally are available in SOA audit files. [12] discusses the possibility of using service dependency for deploying and reusing composite services.

In [8], the authors propose dependency graph based web service composition. They use backward chaining in combination with depth first search to get required services for a composite task.Their solution is rather abstract and does not clearly discuss execution plan generation algorithm.

[7] used dependency graph to store information about existing web services in repository. In the graph nodes represent I/O parameters and edges represent web services. Web services are modeled using I/O description and dependency information to other WSs through its I/O. They utilized graph search algorithm to find set candidate services for the composite service as sub graph. They also used interface automata tool to create execution path by taking discovered services. They did not discuss a way to stop search of candidate services. This possibly makes search complicate in case of more than one set of candidate service exists. They also did not put a way to choose best solution from alternatives if it exists.

[6] proposes to pre-compute and store network of services that are linked by their I/O parameter. The link is built by using semantic similarity functions that basis ontology. They represent the service network using graph structure. Their approach utilizes back ward chaining and depth-first search algorithms to find sub-graph that contains services to accomplish the requested task. Unlike [7] they propose a way to select optimal plan in case of more than one plan found.

However, [6]–[8], generates (pre-computed) dependency graph between all services in repository that complicated and makes the graph size very big when there is high number of services. They do selection of candidate services based on pre-computed dependency graph. They all assume the dependency graph to be acyclic which is not always true in reality.

In [13], a service composition technique that utilizes Casual Link Matrix(CLM) is presented. CLM is used to store semantic I/O link between candidate services. The casual link matrix is built based on semantic similarity functions that provide the degree of similarity between input and output parameters of web services.To generate the composition plan they used a regression-based search, AI planning technique. Such an approach brings with it scalability problems due to the inherent computational complexity.

Comparing with the method in [13] which uses CLM matrix our approach uses a simple algorithm to generate the process model, which we deem, makes it more efficient especially when the numbers of candidate services are high. Moreover, unlike CLM based technique our approach offer a means to identify concurrent and iterative control flow.

Contrary to other proposed approaches this method explicitly shows which service is dependent on which service in its dependency graph. For example: CLM only shows the degree of similarity between Input and output parameters, graph based composition techniques proposed by [7] shows the dependency between services implicitly however the dependency graph is generated at design time.

V. Discussion and Contributions

We tested the applicability of our approach using case studies taken from [7], [14] and other related papers. In all cases our approach gave process models that are similar to the ones in the papers reviewed. This has been of support to empirically prove the correctness of the process model generated using the proposed method. In the future we will develop an evaluation mechanism to guarantee the correctness and completeness of the output solution.

Unlike all other methods that construct dependency between all services in repository we generated dependency between candidate services automatically. We believe, pre-computing all possible semantic links (dependency) between services(even services with same functionality) might lead to extended graph that increases the complexity of plan creation. To tackle the potential complexity problem due to complex dependency graph in existing approaches, our approach assumes goal based candidate service discovery upon receiving of user request. Then takes those discovered candidate services, extracts dependency, represents it using directed graph and generates composition execution plan.

The proposed approach finds execution plan in three steps using three different algorithms. The whole process complexity is dependent on the three algorithm complexity. The first step is dependency graph generation algorithm that has complexity is O (#(Input parameters) * #(Output parameters)) in worst case scenario. The second step is cyclic dependency extraction which has the same complexity as the algorithm, that is of linear in the number of edges (E), vertices (V)and number of cycles(C) of the dependency graph (O(#(V)+#(E)+#(C))). The third step is composition plan generation algorithm complexity is equivalent to the complexity of topological sorting algorithm which (O(#(V)+#(E))). Therefore, the overall running time is equivalent to the dominating complexity which is complexity of dependency graph generation.

To summarize, among many, the proposed approach's main contributions are:

1. To the best of our knowledge this approach is the first to deal about cyclic dependency in detail.

2. We propose the use of topological sorting algorithm for generating a composition plan. We trust this solves the

scalability problems that occur in many composition plan generation algorithms.

VI. Conclusions and further works

In this paper we propose an Input/Output dependency based automated composition plan creation method. The I/O dependency is represented as directed graph and the composition plan is created based on graph traversal algorithm, i.e topological sorting. The approach recognizes when cyclic dependencies exists and propose a way of dealing with it. The simplified nature of the proposed methodology increases its applicability in real world scenarios. We have tested the method at a conceptual level making use of scenarios having from 3 to 11 web services. For these scenarios the output process model was valid. Thus, we intend to extend this approach to be able to find complex parameter dependencies, and for exploring other dependencies, for instance Pre-condition/Effect dependencies, and dependencies caused by user constraints. Moreover, further analysis is needed to include alternative control flow in process models. In addition, running extensive experiments to further validate dependencies based process model creation method is suggested.

References

[1] M. Fluegge, I. J. G. Santos, N. P. Tizzo, and E. R. M. Madeira, "Challenges and techniques on the road to dynamically compose web services," in *ICWE '06: Proceedings of the 6th international conference on Web engineering*. New York, NY, USA: ACM, 2006, pp. 40–47. [Online]. Available: http://dx.doi.org/http://doi.acm.org/10.1145/1145581.1145589

[2] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara, "Semantic matching of web services capabilities," in *International Semantic Web Conference*, ser. Lecture Notes in Computer Science, I. Horrocks, J. A. Hendler, I. Horrocks, and J. A. Hendler, Eds., vol. 2342. Springer, 2002, pp. 333–347. [Online]. Available: http://dblp.uni-trier.de/rec/bibtex/conf/semweb/PaoluccciKPS02

[3] M. K. Smith, C. Welty, and D. McGuinness, "Owl web ontology language guide, http://www.w3.org/tr/owl-guide/, accessed," 2004.

[4] R. Tarjan, "Enumeration of the elementary circuits of a directed graph," *J.SIAM*, vol. 2, pp. 211–216, 1973.

[5] L. Li and I. Horrocks, "A software framework for matchmaking based on semantic web technology," in *WWW '03: Proceedings of the 12th international conference on World Wide Web*. New York, NY, USA: ACM Press, 2003, pp. 331–339. [Online]. Available: http://dx.doi.org/10.1145/775152.775199

[6] H. N. Talantikite, D. Aissani, and N. Boudjlida, "Semantic annotations for web services discovery and composition," vol. In Press, Corrected Proof, 2008, pp. –. [Online]. Available: http://www.sciencedirect.com/science/article/B6TYV-4V0MX0B-1/2/ebb454737a30085656ed79bf1d48804a

[7] S. V. Hashemian and F. Mavaddat, "A graph-based approach to web services composition," vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 183–189.

[8] R. Aydogan and H. Zirtiloglu, "A graph-based web service composition technique using ontological information," vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 1154–1155.

[9] Z. Ma, P. Marchal, D. P. Scarpazza, P. Yang, C. Wong, J. I. Gomez, S. Himpe, C. Ykman-Couvreur, and F. Catthoor, *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogeneous Platforms*. Springer, 2007.

[10] B. Li, "Managing dependencies in component-based systems based on matrix model," in *Proc. Of Net.Object.Days 2003*, 2003, pp. 22–25.

[11] S. Basu, F. Casati, and F. Daniel, "Web service dependency discovery tool for soa management," vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 684–685.

[12] J. Zhou, D. Pakkala, J. Perl, and E. Niemel, "Dependency-aware service oriented architecture and service composition," in *IEEE International Conference on Web Services.*, July 2007, pp. 1146–1149.

[13] F. Lecue, E. M. G. da Silva, and L. F. Pires, "A framework for dynamic web services composition," in *2nd ECOWS Workshop on Emerging Web Services Technology (WEWST07), Halle*. Germany: CEUR Workshop Proceedings, November 2007.

[14] F. Lecue and A. Leger, "Semantic web service composition based on a closed world assumption," *Web Services, European Conference on*, vol. 0, pp. 233–242, 2006.