Web Services Composition using Input/Output Dependency

Abrehet Mohammed Omer
TU Dresden, Chair for Computer Networks
01062 Dresden, Germany
+49 (0) 351 46338263
Abrehet_mohammed.omer@mailbox.tu-dresden.de

Alexander Schill
TU Dresden, Chair for Computer Networks
01062 Dresden, Germany
+49 (0) 351 463-38002
alexander.schill@tu-dresden.de

ABSTRACT
Composition of web services has received increased interest with emerging application development architecture-Service Oriented Architecture (SOA). Doing composition (semi-) automatically is a crucial aspect in overcoming runtime problems that arise due to dynamic nature of runtime environment. In SOA, applications are created as combinations of independently developed Web services. This leads to emergence of different dependencies among the component services forming the composite service. Given a set of candidate web services and a user's request description in terms of (I,O,P,E,G), the proposed method can find a composite service that would satisfy user's requirements in two steps. First, it anticipates the potential direct and indirect dependency between abstract services, and second, it generates process model (PM) automatically using the dependency information. The architecture and application of this method and its application are discussed using a case study. Moreover, a summary of existing techniques and their shortcomings are presented. This approach takes advantages of a sorting algorithm and semantic I/O matching techniques.

Categories and Subject Descriptors
H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *Web-based services*; F.2.2 [**Analysis Of Algorithms And Problem Complexity**]: Non-numerical Algorithms and Problems-*Sequencing and scheduling*-*Sorting* and searching; D.2.m [**Software Engineering**]: Miscellaneous.

General Terms
Algorithms, Design, Theory

Keywords
Automatic service composition, Service dependency.

1. INTRODUCTION
Service Oriented Architecture (SOA) is an emerging application development architecture. It uses individual software services to build composite applications. This is possible because smaller and simpler applications can be developed and availed in the form of Web Services (WS). These individual applications can be published, located, and invoked across the web. The ability to invoke and compose services using multiple individual services allows meeting larger & single user requirements that could not otherwise be met with any of the available smaller services. Thus, complex service based applications can be created in an SOA environment by composing individual services. This newly emerging application development architecture (SOA) has increased the demand for web services. And it has called for researches in the area of WS composition.

The service composition process comprises of three major activities: 1) Process model creation: Process model is a model that simplifies the representation of activities and their enactment. It is used to specify task control-flow and data-flow among different subtask activities. It can be done manually (by developer at design time), semi-automatically (with the help of template) or automatically (via software). 2) Concrete service discovery and binding: this activity involves finding and binding smaller individual services that accomplish sub-tasks of a composite service. It can be done either at design time or run time. 3) Availing composite service: this refers to availing the composite service to clients and its management.

Service composition can be done either statically, or (semi/fully) dynamically. These different levels of automation are determined by how (and who) the process model is created as well as by when the service discovery and binding is done (i.e. at design versus run time). In static composition the process model is created manually and service binding is done at design time. In contrast, dynamic composition process model is created automatically and service binding is done at runtime. All methods between these two extremes are categorized as semi-dynamic [3].

Static service composition has shortcomings in automatically adapting to unpredictable changes in a dynamic run time environment. Unpredictable changes happen, for example, because new services could become available and old services could be made inaccessible on a daily basis. Due to such adaptability shortcomings of static composition methods, nowadays, there is a growing tendency for shifting to dynamic service composition methods. The process of implementing dynamic service composition or tackling problems with static composition mechanisms are not only limited to runtime service binding but it also demands ability for doing process model automatically. Consequently, automation of process model creation is one of the core problems hindering the transition towards automatic service composition and it needs to be solved.

Investigation of activities in process model creation shows that, while trying to create composite services, all methods attempt to extract dependencies (relationships). For example, in graph-based and chaining mechanisms of service composition, algorithms mainly search for direct explicit input/output relationships between services [11, 10]]. In workflow-based techniques of service composition the programmer identifies sub-task dependencies manually.

The concept of dependency is explored initially for the purpose of managing component-based systems [7]. The work by

[2] looks for service dependencies from composite service management point of view. In their approach, it was demonstrated that dependencies could be tracked from log files, which normally are available in SOA audit files. [14] discusses the possibility of using service dependency for deploying and reusing composite services. [1],[4]&[13]used service dependency to create composite service. However they created the composition plan using design time (pre-computed) generated dependency. [6] has proposed that service composition method that utilizes Casual Link Matrix to store semantic I/O link between candidate services.

In this paper, we introduce a simplified I/O dependency based automatic process model creation. In order to extract I/O dependency our approach uses the concept of finding the semantic similarity between service input and outputs. Then it utilizes the dependency information for the purpose of automatic process model creation. In our approach process model (execution plan) is generated using sorting algorithms.

This paper is organized as follows: following this introduction, section 2 presents a case study which will be used throughout the paper; in section 3 the process of identifying, representing, analyzing I/O dependency and its application is presented and section 4 gives discussion of proposed approach. Finally in section 5 conclusions and planned further works are presented.

2. CASE STUDY

As a case study an example of e-health scenario that is taken from [6] is considered. This scenario assumes the existing medical applications and devices interfaced by web services. So by creating composition of devices (composition of wrapped web services) one can enable online patient follow-up, to reduce time-consuming consultation and medical checkups. For this scenario the following web services are considered: WS1 returns the blood pressure (BP) of a patient given his PatientID (PID) and DeviceAddress (Add); WS2 returns the supervisor (Person) given a medical of an organization (Org) for example: Emergency department ; WS3 returns a Warning level (WL) given a blood pressure; WS4 returns the Emergency department given a level of Warning; WS5 returns the Organization given a Warning level. Table 1 shows the input and output of each service. The shaded column shows from where a service gets its inputs.

3. PROPOSED APPROACH

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 application 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

AUPC'09, July 13–17, 2009, London, United Kingdom.

Table 1: Case study Input/ output description

Web services	Inputs	Source web service	Outputs
WS1	PID;ADD	User request	BP
WS2	Org	WS5	Person
WS3	BP	WS1	WL
WS4	WL	WS3	ED
WS5	WL	WS3	Org

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 do not directly expressed in service descriptions.

Generally, managing dependencies are considered to be the basis for defining process (services) coordination mechanisms [5] Sequential, alternative, iterative and concurrent coordination mechanisms are considered as the basic coordination mechanism in any business process or dependency management. These are the coordination mechanisms used during process model creation for composite services. Though the research final target is to extract all kind of dependency and use them, in this paper we present extraction and usage of explicit direct and indirect I/O dependency for automatic PM creation.

In the following sub-sections first the notion of abstract service description, service request description dependency representation and a procedure to create PM automatically using I/O dependencies will be presented.

3.1 Composite Service Request and abstract service specification

Web service and user requests have to be described in a suitable way so that dependencies among candidate services can be extracted for composition. The proposed approach relies on a formal description both from 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 request. For our intention, the abstract description of web services and request includes tuple with (I, O, P, E, G) as they are defined on OWL [12] where I:list of inputs; O:list of output parameters; P:precondition which describes logical expression that must be satisfied in order to invoke composite service; and E: effect which describes the changes to the current state resulting from the invocation of composite service.

In this approach we assume the availability of local repository that stores abstract service description in the above format. Such abstract description includes only a single description for all web services with the same functionality regardless of their quality. Thus candidate services will be discovered from the local repository based on user requirement goal definition. Then dependency between those abstract services will be extracted for PM generation. The process of discovering candidate abstract services is out of the scope of our work.

Note that the concrete service binding for the actual service composition will be done based on abstract description and additional non-functional property after process model creation.

3.2 Dependency representation

Dependency can be representation as graph or matrix based model. In this approach, matrix is used to represent I/O dependencies between services, which are also used in [7] to represent dependencies between components. The matrix that models the dependency will be a square matrix (nxn) where n equals available services[1] to form the composite service. Each row and column represents candidate web services for the composite web service (WS_i). And if a service on i^{th} column is dependent on a service on j^{th} row then the C_{ij} value of the matrix will be 1 otherwise it will be zero.

Let the composite service to be created require n web services: WS_1, WS_2,…WS_n. Then the dependency matrix (DM) can be defined as follows:

$$DM=\begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & .. & .. & C_{2n} \\ \vdots & .. & .. & \vdots \\ C_{n1} & C_{n2} & .. & C_{nn} \end{bmatrix} \quad \text{Where} \quad C_{ij}=\begin{cases} 1 & \textit{if ws i is dependent on ws j} \\ 0 & \textit{otherwise} \end{cases}$$

3.3 Automatic PM Creation

Here a detailed explanation of the proposed architecture is provided. As it is mentioned before a list of abstract services and the formal user request description are the two inputs for the system. The following steps are performed in order to create the PM automatically:

1. Identify explicit direct dependencies from input and output parameters of web services and construct dependency matrix.
2. Identify explicit indirect I/O dependencies by recursively exploring the explicit direct dependencies and construct indirect DM.
3. Merge the explicit direct and indirect dependencies and form one I/O DM.
4. Calculate the number of services dependent on a particular service by adding each row of the matrix found in step 3.
5. Calculate the number of other services dependent on a particular service by adding each column of the matrix found in step 3.
6. Use simple sorting algorithm to generate a PM based on calculated values in step 3 and step 4.

This is a simplified stepwise description of PM creation. However, each part dependency identification (steps 1, 2 and 3), analysis (step 4 and 5) and PM generation (step 6)) will be done by different component as it is shown in figure 1.

In the next section using the above case study the automatic process model creation will be elaborated.

[1] We used services and abstract services interchangeably

3.4 Service dependency generator

Explicit Input/Output dependencies between services occur when

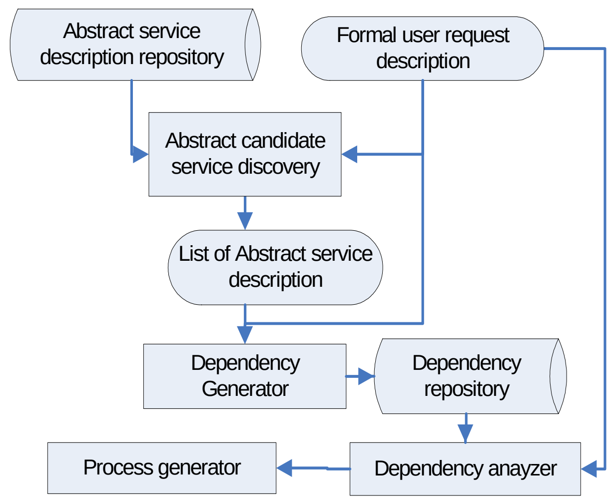

Figure 1. Architecture

a service requires/or provides data from/to another service

The approach extracts I/O dependency in two steps. First it extracts explicit direct dependency and then extracts explicit indirect dependency from the direct dependency. And then by summing up the two DM's it makes ready the full I/O dependency.

3.4.1 Construction of explicit direct DM

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 output of another web service. For the purpose of explaining the proposed approach we show 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, the extraction of explicit direct I/O dependency is done using semantically enabled I/O matching techniques which is adopted from [9]. It uses the following four semantic I/O matching functions proposed by [9],[8].

1. Exact : If the output parameter of WS1 and the input parameter WS2 are equivalent concepts;
2. Plug in : If output of WS1 is sub-concept of input WS2;
3. Intersection: If the intersection of output of WS1 and input WS2 is satisfiable.
4. Fail : if all the above conditions are not satisfied

The dependency matrix 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 (WS1) $\cap$Out (WS2) $\neq\varnothing$ if and only if at least one pair of parameter set (each from Input(WS1) and Output(WS2)) has either exact or plug in relationship. This is done because our main aim is to find out from which services gets a particular service its inputs i.e. on which services it is dependent on. Table 2 shows explicit direct I/O DM for the E-health scenario.

Table 2: Explicit direct DM

Web service	WS1	WS2	WS3	WS4	WS5
WS1	0	0	1	0	0
WS2	0	0	0	0	0
WS3	0	0	0	1	1
WS4	0	0	0	0	0
WS5	0	1	0	0	0

3.4.2 Construction of explicit indirect DM

Since dependency holds transitivity property one can extract indirect I/O dependencies between services from explicit direct I/O dependency. For example if service B has an explicit direct dependency on service A and service C again has explicit direct dependency on service B then service C will have explicit indirect dependency on service A. Thus, one should traverse all possible explicit direct service dependency chains to extract explicit indirect dependencies. This dependency chain is a linked list of services that starts from a service in focus and terminates with a service that doesn't have explicit direct dependency with any service. The link between individual services in a chain represents the explicit direct dependency between services.

For example one possible dependency chain for WS2 in a case study is: WS2→ WS5 → WS3→ WS1→ none

Thus, an explicit indirect I/O dependency exists if and only if:

-a service has explicit direct dependency to at least one service,

-there exists service in a chain of explicit direct dependency that does not have explicit direct dependency with a particular service in focus. From above chain, since WS5 has explicit direct dependency with WS2 only explicit indirect dependency with WS3 and WS1 are counted. (While representing implicit dependency all explicit dependencies should be excluded to control redundant counting of dependency).

Thus, the following algorithm is developed to generate the explicit indirect DM from explicit direct DM. It takes explicit direct dependency as input & delivers an explicit indirect DM that does not include any explicit direct dependency (see table 4).

```
n=number of services
i=1
while (i<=n){
 Function(i,i)
 i=i+1 }
//The recursive function definition
Function(k,m) {
for (j=1 to  n)
{ if DM1[j][k]=1// the jth service  is dependent on kth service
{  if(DM1[j][m]!=1) // there is no explicit direct dependency
                    between jth and mth service
 {  DM2[j][m]=1 // assign 1 on  the jth row and mth column of
                    indirect dependency matrix     }
F(j,m)   // call the function with new parameters to get the
         chain of dependent matrices    }   }
 return 0 }
```

3.4.3 Explicit direct and indirect DM

By simply adding the explicit direct and indirect dependency matrices full input/output dependencies can be found. In table 5 complete I/O dependencies are shown,

Table 3: Explicit indirect DM

Web services	WS1	WS2	WS3	WS4	WS5
WS1	0	1	0	1	1
WS2	0	0	0	0	0
WS3	0	1	0	0	0
WS4	0	0	0	0	0
WS5	0	0	0	0	0

Table 4: Explicit direct and indirect DM

Web service	WS 1	WS 2	WS3	WS4	WS5	C_A=Σ column
WS1	0	1	1	1	1	4
WS2	0	0	0	0	0	0
WS3	0	1	0	1	1	3
WS4	0	0	0	0	0	0
WS5	0	1	0	0	0	1
C_B=Σ row	0	3	1	2	2	

3.5 Dependency Matrix Analysis

The dependency matrix 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 a single control flow passing from input provider to receiver. In case 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. Such kind of communication requires iterative control flow(1..n). Thus, the first step of DM analysis is finding out bidirectional communication between services if it exists. Cyclic dependency is the indicator of bidirectional communication. It can be identified by comparing the symmetrical elements or by checking its diagonal elements value of the DM. Thus cyclic dependency exists:

1. When symmetrical elements of the DM are equal to 1. For example: if DM[i][j]=DM[j][i]=1 then ith & jth element has bidirectional communication.
2. When diagonal element of DM is 1. This implies a service is dependent on itself. This implies a service needs to execute more than once to accomplish the composite task so loop control flow should be attached it.

After finding the cyclic dependency the necessary control structures should be attached to the respective services. And then the bi-directional communication indicators should be eliminated from the matrix for the next step (i.e to find the sequential and concurrent control flows).

From the DM which is free of cyclic dependency we get two straight forward but important indicators to decide the execution priority of services. They are described as follows:

1. The number of other services that dependent on a given service (C_A): This number can be found by counting services the

number of taking input directly from output of a service (explicit direct dependency) plus the number of services that has explicit indirect I/O dependencies on it. From the full I/O DM one can get this value by adding each row of the matrix. In Table 5 summarizes the result of full (direct DM plus indirect DM) DM. The second column (C A) shows the number of services dependent on ith service. For example: there are 4 services dependent on WS1. From this indicator we can reach the partial conclusion that the more services are dependent on a service, the higher priority that service has. Because when m services are dependent on that service definitely that particular service should be executed before all services dependent on it.

2. The number of services a given service is dependent on (C_B): In similar manner as first indicator this number can also be found by counting services from which a service takes input directly (direct dependency) plus the number of services a service indirectly depends on. From the full I/O DM one can get this value by summing up each column of the matrix. In Table 5 the third column shows the number of services the jth service depends on (C_B). For example, WS1 is dependent on only one service. From this indicator we can also reach to another partial conclusion that the more services a service depends on the lesser priority that service has. Because when a service is dependent on m services this indicates that these m services that service depends on should be executed before it.

Therefore, from a straight forward analysis of input/output dependency we got the first two indicators which could provide valuable information to create the process model.

3.6 Application

Here, we discuss an application of our dependency analysis in generating simple a process model with sequential and concurrent coordination mechanisms which is the task of a PM generator based on our architecture. Moreover the interpretation of the results will be given.

The possible two process models are generated based on the two numbers described in section 3 by using a simple sorting algorithm starting from the initial random order given by Table 1. These possible process models (sequential execution paths) will be explained as follows:

1. Sorted based on C_A: this sorting is based on the number of services depended on a particular service in descending order. (See table 5 column 1 & 2) This is because a service with higher number of services dependent on it should logically have a higher priority.
2. Sorted based on C_B : this sorting is done based on how many other services a particular service depends on in ascending order.(see table 5 column 4 & 5) This is because a service that depends on many services logically should have lower priority compared to service dependent on a smaller number of services.

Table 5: Sorted based C_A and C_B

Web Services	C_A	Web Services	C_B
WS1	4	WS1	0
WS3	3	WS3	1
WS5	1	WS4	2
WS2	0	WS5	2
WS4	0	WS2	3

From observation we have seen services with equal value of C_A or C_B can be executed concurrently. In first case WS2 and WS4

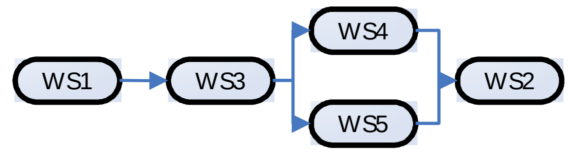

Figure 3. PM generated using C_B.

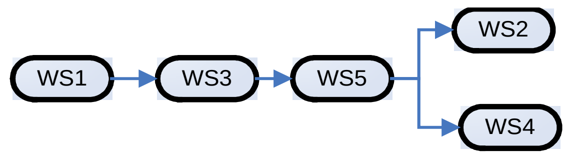

Figure 2. PM generated using C_A

can be executed concurrently. In the second case WS4 and WS5 can be executed concurrently. As a result the output process model is given in Fig2 and 3

4. Discussion

The DM generation algorithm complexity is O (#(Input parameters) × #(Output parameters)) in worst case scenario. The composition plan generation algorithm complexity is equivalent to the sorting algorithm used which is O (n*n) n being the number of services. Consequently the overall approach complexity is equivalent to the DM generation algorithm, which is of quadratic time. As number of services increases the search space for DM matrix generator will increase. To overcome this limitation in the future we intend to provide a user query interface to receive intermediate inputs and hints to dependency generator.

We tested the applicability of our approach using case studies taken from [4],[6] 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 assistance to empirically prove the aptness of the process model generated by 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 might lead to extended graph that increases the complexity of plan creation.

To generate composition plan those methods often used graph traverse algorithms, this arose O(number of vertex*number of edge) which is fully dependent on number of edge and vertices that in turn dependent on number of services in repository(even services with same functionality).

Therefore, compared to the quadratic complexity of our approach this complexity is much bigger as the number of services in repository increases. To tackle such complexity problem in existing approaches, our approach assumes goal based candidate service discovery upon receival of user request. Then this approach takes those discovered candidate services, extracts their dependency, analyzes it and then generates composition plan .

4.1 Comparison to related work

Comparing with the method in [6] 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. CLM based technique does not offer a means to identify concurrent and iterative control flow. 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.

Contrary to other proposed approaches this method explicitly shows which service is dependent on which service in its DM. For example: CLM only shows the degree of similarity between Input and output parameters, graph based composition techniques proposed by [4] shows the dependency between services implicitly but the dependency graph is generated at design time.

4.2 Contributions

The main contributions, among many, of the proposed approach can be summarized as follows:

1. To the best of our knowledge this approach is the first to show on demand process model creation based on dependency that is extracted automatically from abstract service description. It also shows the use indirect dependencies for composition plan generation.

2. We propose the use of simple sorting algorithm for generating a composition plan in one step. We trust this solves the scalability problems that occurs in many composition plan generation algorithms.

3. Despite most methods that use service dependency for composition plan creation [4], [13], =[6]=[6] we do not pre-computes unnecessary semantic link between all registered services. We believe finding out only the semantic link (dependency) among candidate services for the required composition avoids the unnecessary computation required to create all links between services in the registry. In this approach we managed enlighten what cyclic dependency means, how we use cyclic dependency as an indicator of loop control flow and how to eliminate it to avoid further complexity in further execution plan generation process.

5. Conclusions and further work

In this paper we propose an Input/Output dependency based automated process model creation method for the purpose of service composition. The process model is created based on straightforward analysis of input/output dependency. 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 techniques are needed to incorporate alternative control flow in process models. In addition, running extensive experiments to further validate dependencies based process model creation method is suggested.

6. REFERENCES

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

[2] S. Basu, F. Casati, and F. Daniel. Web service dependency discovery tool for soa management. volume 0, pages 684–685, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[3] 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*, pages 40–47, New York, NY, USA, 2006. ACM.

[4] S. V. Hashemian and F. Mavaddat. A graph-based approach to web services composition. volume 0, pages 183–189, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[5] J. W. Kim and R. Jain. Web services composition with traceability centered on dependency. volume 3, page 89, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

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

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

[8] 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*, pages 331–339, New York, NY, USA, 2003. ACM Press.

[9] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In I. Horrocks, J. A. Hendler, I. Horrocks, and J. A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2002.

[10] S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *Proceedings of the 11th International WWW Conference (WWW2002)*, Honolulu, HI, USA, 2002.

[11] V. Ramasamy. Syntactical & semantical web services discovery and composition. volume 0, page 68, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

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

[13] H. N. Talantikite, D. Aissani, and N. Boudjlida. Semantic annotations for web services discovery and composition. volume In Press, Corrected Proof, pages –, 2008.

[14] J. Zhou, D. Pakkala, J. Perälä, and E. Niemelä. Dependency-aware service oriented architecture and service

composition. In *IEEE International Conference on Web Services.*, pages 1146–1149, July 2007.