Type-based composition of information services in large scale environments

Ion Constantinescu Boi Faltings Walter Binder
Artificial Intelligence Laboratory
Swiss Federal Institute of Technology
IN (Ecublens), CH–1015 Lausanne (Switzerland)
{ion.constantinescu, boi.faltings, walter.binder}@epfl.ch
http://liawww.epfl.ch

Abstract

Service matchmaking and composition has recently drawn increasing attention in the research community. Most existing algorithms construct chains of services based on exact matches of input/output types. However, this does not work when the available services only cover a part of the range of the input type. We present an algorithm that also allows partial matches and composes them using switches *that decide on the required service at runtime based on the actual data type. We report experiments on randomly generated composition problems that show that using partial matches can decrease the failure rate of the integration algorithm using only complete matches by up to* **7 times** *with no increase in the number of directory accesses required. This shows that composition with partial matches is an essential and useful element of web service composition.*

1. Introduction

Service composition is an exciting area which has recently received a significant amount of interest. Initial approaches to web service composition [11] used a simple forward chaining technique which can result in the discovery of large numbers of services.

There is a good body of work which tries to address the service composition problem by using planning techniques based either on theorem proving (e.g., Golog [7, 8] and SWORD [10]) or on hierarchical task planning (e.g., SHOP-2 [15]). The advantage of this kind of approaches is that complex constructs like loops (Golog) or processes (SHOP-2) can be handled. All these approaches assume that the relevant service descriptions are initially loaded into the reasoning engine and that no discovery is performed during composition.

Recently, Lassila and Dixit [5] have addressed the problem of interleaving discovery and integration in more detail, but have considered only simple workflows where services have one input and one output.

In this paper we are concerned by a particular combination of issues that is specific and unique to the web services context:

1. **discovery in large scale directories** - we assume that a large number of available web services will be stored in (possibly distributed) directories. How can we discover exactly the services that are relevant at each step of the composition process?
2. **runtime non-determinism** - when discovered services match only partially but not completely[1], the reasoning engine has to aggregate several services as switches in order to fulfill the required functionality. The actual flow of messages will be routed based on runtime values on the appropriate paths. How can we discover and create those switches and how can we make sure that they correctly handle all possible combinations of parameter values?

This paper presents solutions to these two issues: as a first contribution we present an algorithm that interleaves the discovery and composition process. Our second contribution is a technique for discovering and composing services with partial type compatibility. In our approach we discretize the space of possible parameter values and we incrementally reduce the space of values that cannot be handled. Then partially matching components are assembled into switches that route the flow of messages on the appropriate paths based on runtime values.

1 We consider as partial matches the **subsume** match type identified by Paolluci [9] and the **intersection** or **overlap** match type identified by Li [6] and Constantinescu [2].

Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence (WI'04)
0-7695-2100-2/04 $ 20.00 IEEE

IEEE COMPUTER SOCIETY

Some other challenging issues are not addressed in this paper but are considered for future work: behavior-based integration, dealing with side-effects and changes of the world that are not under the control of the composition engine, as well as knowledge engineering issues, to enumerate only a few.

This paper is structured as follows: in Section 2 we present the formalism and assumptions used in the rest of the paper and some details regarding type based service composition. In Section 3 we describe the machinery and algorithms that we actually use for computing service compositions. Section 4 describes the testbed that we use for analyzing our algorithms and evaluates some experimental results on randomly generated problems. Finally Section 5 concludes the paper.

2. Service composition with partial type matches

Our approach to service composition is based on the idea of chaining services together either in a forward way, starting from the initial conditions. Forward chaining techniques are used by different types of reasoning systems, in particular for planning [1] and more recently for service integration [11]. We describe next the formalism that we use to model, match, and chain services.

2.1. Formalism and assumptions

We represent services and queries in the standard way [13] as two sets of parameters (inputs and outputs). A parameter is defined through its name and a type that can be primitive [14] (e.g., a decimal in the range [10,12] or [14,16]) or a class/ontological type [12]. Both primitive and class types are represented as sets of numeric intervals. For instance, the generic type $Color$ may be encoded as the interval [1,3], whereas the specific colors (subtypes) Red, $Green$, and $Blue$ may be represented as the single-point subintervals [1,1], [2,2], and [3,3]. For more details on the encoding of classes/ontologies as numeric intervals see [2].

Input and output parameters of service descriptions have the following semantics:

- In order for the service to be invokable, a value must be known for each of the service input parameters and it has to be consistent with the respective parameter type. For primitive data types, the invocation value must be in the range of allowed values or in the case of classes the invocation value must be subsumed by the parameter type.
- Upon successful invocation, the service will provide a value for each of the output parameters and each of these values will be consistent with the respective parameter type.

Service composition queries are represented in a similar manner but have different semantics:

- The query inputs are the parameters available to the integration (e.g., provided by the user). Each of these input parameters may be either a concrete value of a given type, or just the type information. In the second case the integration solution has to be able to handle all the possible values for the given input parameter type.
- The query outputs are the parameters that a successful integration must provide and the parameter types define what ranges of values can be handled. The integration solution must be able to provide a value for each of the parameters in the problem output and the value must be in the range defined by the respective problem output parameter type.

For manipulating service or query descriptions we will make use of the following helper functions:

- $in(X)$, $out(X)$ – return the set of input or output parameter names of a service or query description X.
- $type(P, X)$ – returns the type of a parameter named P in the frame of a service or query description X as the set of intervals of all possible values for P. The $\subseteq$ operator in conjunction with this function will represent a range inclusion in the case that P has a primitive data type or subsumption in case P is defined through a class or concept description [12]. The operator $\cap$ in conjunction with this function will represent a range intersection in the case that P has a primitive data type or in the case of a class/concept description it will represent the sub-class (possibly the bottom class Nothing) common to both the arguments of the operator.

We assume that both service and query descriptions (X) are well formed in that they cannot have the same parameter both as input and output: $in(X) \cap out(X) = \emptyset$. The rationale behind this assumption is that if a description had an overlap between input and output parameters this would only lead to two equally undesirable cases: either the two parameters would have the same type, in which case the output parameter is redundant, or they would have different types, in which case the service description is inconsistent.

Parameter names (properties in the case of DAML-S [4] or strings in the case of WSDL [13]) attach also some semantic information to the parameters[2]. Thus, in our com-

2 For WSDL this is not explicitly specified by the standard, but we assume that two parameters with the same name are semantically equivalent.

IEEE COMPUTER SOCIETY

position algorithm we not only consider type compatibility between parameters but also semantic compatibility.

2.2. Composing services

Informally, the idea of forward chaining is to iteratively apply a possible service S to a set of input parameters provided by a query Q (i.e., all inputs required by S have to be available). If applying S does not solve the problem (i.e., still not all the outputs required by the query Q are available) then a new query Q' can be computed from Q and S and the whole process is iterated. This part of our framework corresponds to the planning techniques currently used for service composition [11].

Now we consider the conditions needed for a service S to be applied to the inputs available from a query Q using forward chaining: for all of the inputs required by the service S, there has to be a compatible parameter in the inputs provided by the query Q. Compatibility has to be achieved both for names (that have to be semantically equivalent) and for types, where the range provided by the query Q has to be more specific ($\subseteq$) than the one accepted by the service S:

$$(\forall P \in in(S)) \quad (P \in in(Q) \wedge type(P,Q) \subseteq type(P,S))$$

This kind of matching between the inputs of query Q and of service S corresponds to the **plugIn** match identified by Paolluci [9].

Forward complete matching of types is too restrictive and might not always work, because the types accepted by the available services may partially overlap the type specified in the query. For example, a query for restaurant recommendation services across all Switzerland could specify that the integer parameter zip code could be in the range [1000,9999] while an existing service providing recommendations for the french speaking part of Switzerland could accept only integers in the range [1000-2999] for the zip code parameter.

A major contribution of this paper is an approach where the above condition for forward chaining is modified such that services with *partial type matches* can be supported. For doing that we relax the type inclusion to a simple overlap:

$$(\forall P \in in(S)) \quad (P \in in(Q) \wedge type(P,Q) \cap type(P,S) \neq \emptyset)$$

This kind of matching between the inputs of query Q and of service S corresponds to the **overlap** or **intersection** match identified by Li [6] and Constantinescu [2].

2.3. Computing composed types

When services are composed in a forward chaining way new parameters can be added as available or the type of existing available parameters can change.

Let's consider as an example a service integration problem Q that provides the parameter A, where type(A)=[a1]. Let's also consider two functional services S1 and S2 with A as the required parameter, where type(A,S1)=[a0,a1], type(A,S2)=[a1,a2]. The services are functional in the sense that for the same combination of input parameters they will return always the same output values.

Services S1 and S2 forward completley match Q, that is $type(A,Q) \subseteq type(A,S1)$ and $type(A,Q) \subseteq type(A,S2)$. Both services S1 and S2 provide as output parameter X, where type(X,S1)=[x1,x2] and type(X,S2)=[x2,x3]. Upon applying S1 and S2 to the query Q in a forward chaining way, a new composition problem Q' can be generated. Q' will have as available outputs X. The question is what should be type(X,Q'), the type of the new available parameter.

Since S1 and S2 share a common set of inputs, X is semantically equivalent between services (represents in both the context of S1 and S2 exactly the same piece of information) and since we assume that both S1 and S2 are correct, they should produce at runtime the same value for the parameter X. So at composition time we can assume that the only valid outputs that the two services will provide will be those that are common to their types. As such we calculate the type of X in Q' as the intersection of its type in S1 and S2:

$$type(X,Q') = type(X,S1) \cap type(X,S2).$$

This corresponds also to a functional view, where we assume that for computing the value of a given parameter X there is a generic function $f_X : \mathcal{D}_1 \times ... \times \mathcal{D}_k \rightarrow \mathcal{V}$. Then a service S could be (virtually) decomposed to a set of restricted functions, one for each of the service outputs and having as domains one or more of the parameters in the service input. The domains and co-domains of the functions will be expressed as the types of the parameters in the service descriptions, which will be restrictions of the domains of the generic functions:

$$\forall X \in out(S), \exists A_i \in in(S), i = 1..k \text{ and}$$

$$f_X : type(A_1,S) \times ... \times type(A_k,S) \rightarrow type(X,S)$$

where

$$type(A_i,S) \subseteq \mathcal{D}_i, i = 1..k \text{ and } type(X,S) \subseteq \mathcal{V}.$$

IEEE COMPUTER SOCIETY

In our example if $f_X : \mathcal{D} \rightarrow \mathcal{V}$ is the function that computes the value of X and S1 and S2 are restrictions of f_X to the domains $D_1 \subset \mathcal{D}$ respectively $D_2 \subset \mathcal{D}$ and if both S1 and S2 are forward complete matches of the available parameters $D_Q \subseteq D_1 \wedge D_Q \subseteq D2$ then:

$$D_Q \subseteq D_1 \cap D_2 \text{ and}$$

$$f_X(D_Q) \subseteq f_X(D_1 \cap D_2) \subseteq f_X(D_1) \cap f_X(D_2).$$

which means that the values that the two services S1 and S2 can generate from the restricted set of inputs provided by Q will always fall into the intersection of the values that services S1 and S2 would normally provide.

Please note that functional approach presented above relies on the following assumptions:

1. services are implemented and described correctly.
2. services have semantically equivalent parameters (the information produced by different services describes *exactly* the same world state/object).
3. services have to be functional (e.g. the same combination of input values produces always the same set of output values).

Still if any of the three doesn't hold the above approach of computing resulting types as the intersection of service types might not be not efficient but is *correct* since by narrowing the range of available inputs we actually allow *more* services to be matched and to be added to the plan.

Another problem that appears when some of the conditions above don't hold is that two services could provide output values with disjunct data-types (e.g., in the example above $type(X, S1) \cap type(X, S2) = \emptyset$).

We address this issue by introducing a wild-card data-type that we note $\emptyset^*$. Then in order to compute data-type of parameter X in a new query Q' we first compute the intersection of the data-types of X for the composed services providing X as an output parameter. The data-type of the parameter X in the new query Q' ($type(X, Q')$) will then be either the actual type intersection when the intersection is not the empty set or otherwise it will be the special wild-card data-type $\emptyset^*$:

$$type\_int = \bigcap_{i=1,n} type(X, Si)$$

$$type(X, Q') = \begin{cases} type\_int, & (type\_int \neq \emptyset) \\ \emptyset^*, & (type\_int = \emptyset) \end{cases}$$

Consequently the forward chaining conditions in Section 2.2 above have to be adapted such that they consider the wildcard data-type. For complete matches the condition becomes:

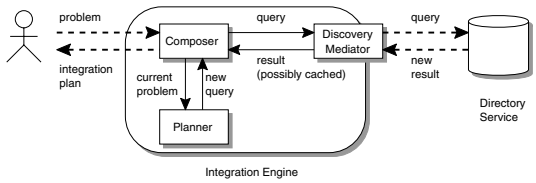


Figure 1. The architecture of our service integration engine.

$$(\forall P \in in(S)) \quad (P \in in(Q) \wedge (type(P, Q) = \emptyset^* \vee type(P, Q) \subseteq type(P, S))).$$

Similary for partial matches the condition becomes:

$$(\forall P \in in(S)) \quad (P \in in(Q) \wedge (type(P, Q) = \emptyset^* \vee type(P, Q) \cap type(P, S) \neq \emptyset)).$$

3. Computing type-compatible service compositions

In this section we will present algorithms for computing type-compatible service compositions. Their design is motivated by two aspects specific to large scale service directories operating in open environments:

- **large result sets** - for each query the directory could return a large number of service descriptions.
- **costly directory accesses** - being a shared resource accessing the directory (possibly remotely) will be expensive.

We address these issues by interleaving discovery and composition and by computing the "right" query at each step. For that, the integration engine (see Fig. 1) uses three separate components:

- **planner** - a component that computes what can be currently achieved from the current query using the current set of discovered services. From that the problem that remains to be solved is derived and a new query is returned.
- **composer** - a component that implements the interleaving between planning and discovery. It decides what kind of queries (partial/complete) should be sent to the directory and it deals with branching points and recoursive solving of sub-problems.
- **discovery mediator** - a component that mediates composer accesses to the directory by caching existing results and matching new queries to already discovered services.

IEEE COMPUTER SOCIETY

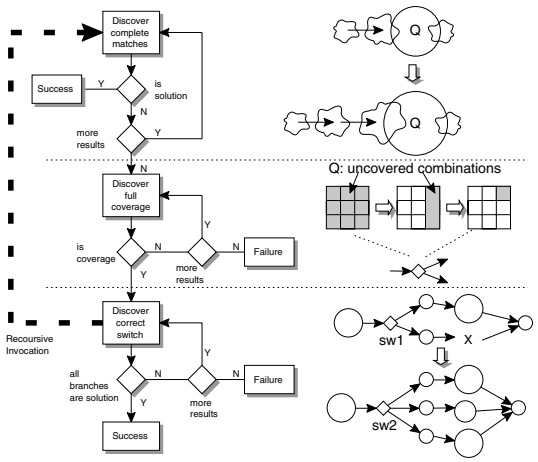


Figure 2. Flow of algorithm for composition with partial type matches.

3.1. Composition with complete type matches

Composing complete matching services using forward chaining is straightforward: once the condition for complete type matches in Section 2.3 is fullfilled (all inputs required by the service S are present in the query Q and the types in the query are more specific than the types accepted by the service) a new query Q' can be computed by adding to the set of available inputs of the current query Q all the outputs provided by the service S:

$$out(Q') = out(Q) \cup out(S)$$

The data-types of the parameters available in the new query Q' are computed accordinlgy to Section 2.3 as either the intersection of the current data-types or when the intersection is empty as the wildcard data-type $\emptyset^*$.

3.2. Composition with forward partial type matches

Conceptually the algorithm that we use for composing services with forward partial type matches has three steps:

- Discovery of complete matching services (see above Section 3.1).
- Discovery of services for full coverage of available inputs.
- Discovery of services for correct switch handling.

3.2.1. Discovering full input coverage The second step of the algorithm assumes that a solution using only complete matches was not found and that services with partial type matches have to be assambled in order to solve the problem. By definition any of the partially matching services is able to handle only a limited sub-space of the values available as inputs. In order to ensure that any combination of input values can be handled, the space of available inputs is first discretized in parameter value cells. One cell is a rectangular hyperspace containing all dimensions of the space of available inputs but only a single interval for each dimension. A cell corresponds to the guard condition of the switch. Cells are built in such a way that any of the required inputs for the retrieved partially matching services could be expressed as a collection of cells. Each of the retrieved partially matching services is assigned to the cells that it can accept as input. The coverage is considered complete when all cells have assigned one or more services. When all cells are covered the algorithm proceeds at the next step. If no more partially matching services can be found and a complete coverage was not achieved the algorithm returns failure.

3.2.2. Discovering solution switch The last step of the algorithm assumes that a coverage was found and a first switch can be created. The goal of this step is to ensure that the switch will function correctly for each of its branches. For each cell and its set of assigned services the algorithm will compute the set of output parameters that those services will provide. Then a new query is computed, having as available inputs the output parameters of the cell and as required outputs the set of required outputs of the complete matching phase. The whole composition procedure is then invoked recursively. In the case that all cells return a successful result the switch is considered to be correct and the algorithm returns success. Otherwise a new service is retrieved and the process continues. When no more services can be retrieved the algorithm returns failure.

4. Evaluation and Assessment

This section first presents two domains used by our service integration testbed: a basic domain with a layered structure and a more complex domain with a graph structure. A discussion of the results concludes this section.

4.1. The layered domain

We have first considered a domain with a layered structure where each layer defines a set of parameter names. Services are defined as transformations between parameters in adjacent layers and problems are defined between parameters of the first and last layer.

IEEE COMPUTER SOCIETY

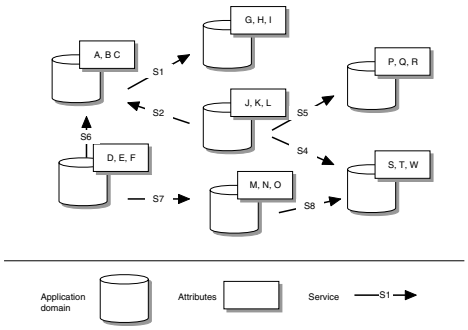


Figure 3. The service graph structure.

4.2. The graph domain

We have designed a second domain that generalizes the previous layered structure to a graph structure (for more details see [3]). The domains is designed on the assumption that the majority of future web services will be created by exposing in a machine readable form applications and systems that are currently accessible via human-level interfaces.

The graph domain builds on the previously identified concepts of data distribution and application domain distribution. In our framework an application domain represents a collection of terms and their associated data types. Then services and queries are defined as transformations between sets of terms in two application domains. Formally this is captured as a directed graph structure, where each node represents an application domain and each edge represents one or more web services. Each web service performs a transformation between two given set of terms from the two application domains associated with the two ends of the graph edge.

4.3. Evaluation of experimental results

For both domains, we have randomly generated services and queries. We then solved the queries using first an algorithm that handles only *complete* type matches and then an algorithm that handles *partial* type matches (and obviously includes *complete* matches). We have measured the number of directory accesses and the failure ratio of the integration algorithms.

Fig. 4 (a) and Fig. 5 (a) show the average number of directory accesses for the algorithm using *complete* type matching versus the average number of directory accesses for the algorithm also using *partial* type matching. Both algorithms scale well, as there is at most a slow increase in the number of directory accesses as the number of services in the directory grows. As it results from the experimental data for both domains the overhead induced by the usage of partial matches is not very significant and decreases as the directory gets saturated with services. This is due to the fact that having more choices makes the coverage problem intrinsic to the partial algorithm easier. More than that in the layered domain from some point the partial algorithm even performs better that the complete one (Fig. 4 (a) after 3000 services). This is due to the fact that the algorithm using partial matches fails less on sub-problems and as such makes better usage of already retrieved services.

The most important result concerns the number of extra problems that can be solved by using partial matches and can be seen in Fig. 4 (b) and Fig. 5 (b). The graph show that the failure rate in the case of using only *complete matches is much bigger than the failure rate when* partial matches are used: up to **7 times** in the case of the Layered domain and **2 times** in the case of the Graph domain. This shows that using partial matches opens the door for solving many problems that were unsolvable by the complete type matching algorithm.

5. Conclusions

With the increasing move towards web services, tools for service indexing, matchmaking, and composition are becoming increasingly important. Our contribution is twofold: first we have shown how service directories and composition methods can be easily extended to deal with *partial matches* of data types and ranges by incorporating software *switches*. Second, our composition algorithms *incrementally access* remote directories.

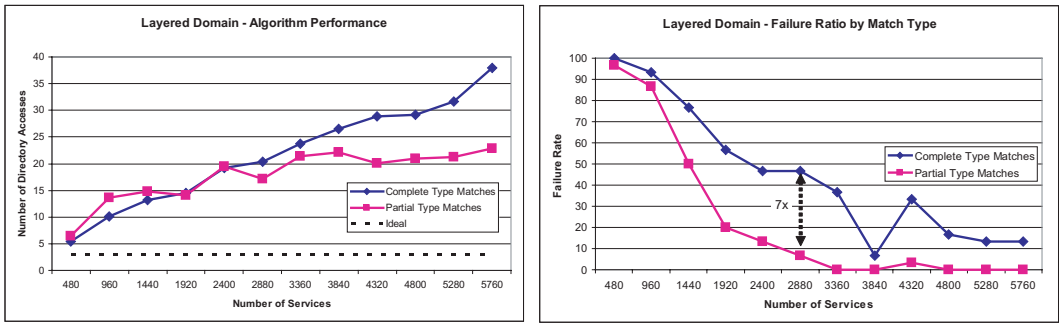
Experiments with randomly generated problems in realistic scenarios show that such partial matches bring significant gains in the range of problems that can be solved by automated composition with a given set of services. Furthermore, it appears that this comes at no increase in the complexity as measured by the number of accesses to service directories. Thus, we consider partial matches to be an essential element of any future service composition algorithm.

Note that we have carried out our experiments with a straightforward chaining approach. Other approaches to composition, such as composition as model checking, are being considered for service composition and would allow more complex constructions such as loops. We think that partial matches may become even more important in such a context as complex plans will create more uncertainty about data ranges that would make complete type matches less and less likely.

References

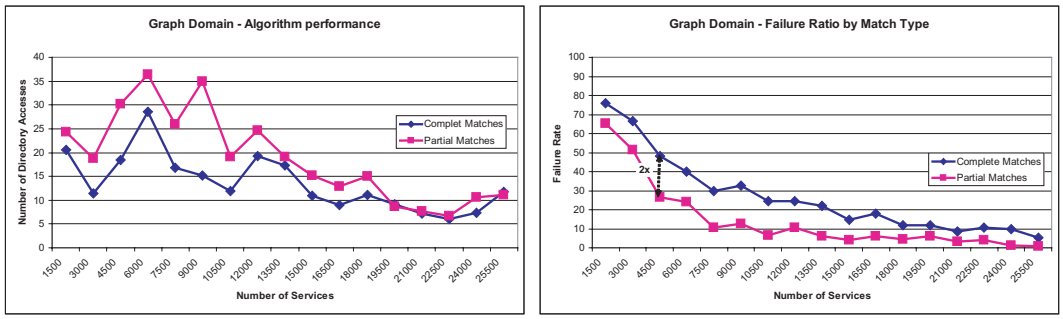
[1] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300, 1997.

IEEE COMPUTER SOCIETY



(a) (b)

Figure 4. The layered domain.



(a) *(b)*

Figure 5. The graph domain.

[2] I. Constantinescu and B. Faltings. Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*, 2003.

[3] I. Constantinescu, B. Faltings, and W. Binder. Large scale testbed for type compatible service composition. In *ICAPS 04 workshop on planning and scheduling for web and grid services*, 2004.

[4] DAML-S. DAML Services, http://www.daml.org/services.

[5] O. Lassila and S. Dixit. Interleaving discovery and composition for simpleworkflows. In *Semantic Web Services, 2004 AAAI Spring Symposium Series*, 2004.

[6] L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, 2003.

[7] S. McIlraith, T. Son, and H. Zeng. Mobilizing the semantic web with daml-enabled web services. In *Proc. Second International Workshop on the Semantic Web (SemWeb-2001)*, Hongkong, China, May 2001.

[8] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, San Francisco, CA, Apr. 22–25 2002. Morgan Kaufmann Publishers.

[9] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, 2002.

[10] S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *In 11th World Wide Web Conference (Web Engineering Track)*, 2002.

[11] S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically composing web services from on-line sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, pages 1–7, Edmonton, Alberta, Canada, July 2002.

[12] W3C. OWL web ontology language 1.0 reference, http://www.w3.org/tr/owl-ref/.

[13] W3C. Web services description language (wsdl) version 1.2, http://www.w3.org/tr/wsdl12.

[14] W3C. XML Schema Part 2: Datatypes, http://www.w3.org/tr/xmlschema-2/.

[15] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S web services composition using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, Sanibel Island, Florida, October 2003.

IEEE COMPUTER SOCIETY