Scalable Automated Service Composition Using a Compact Directory Digest

Walter Binder, Ion Constantinescu, and Boi Faltings

Ecole Polytechnique Fédérale de Lausanne (EPFL)
Artificial Intelligence Laboratory
CH–1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract. The composition of services that are indexed in a large-scale service directory often involves many complex queries issued by the service composition algorithm to the directory. These queries may cause considerable processing effort within the directory, thus limiting scalability. In this paper we present a novel approach to increase scalability: The directory offers a compact digest that service composition clients download to solve the hard part of a composition problem locally. In this paper we compare two different digest representations, a sparse matrix and a Zero-Suppressed Reduced Ordered Binary Decision Diagram (ZDD). Our evaluation confirms that both representations are compact and shows that the ZDD enables more efficient service composition.[1]

Keywords: Web services, service composition, service directories, ZDDs.

1 Introduction

Service-oriented computing enables the construction of distributed applications by integrating services that are available over the web [9]. The building blocks of such applications are web services[2] that are accessed using standard protocols.

Service discovery is the process of locating providers advertising services that can satisfy a given service request. Automated service composition addresses the problem of assembling individual services based on their functional specifications in order to create a value-added, composite service that fulfills a service request. Most approaches to automated service composition are based on AI planning techniques [11,4,10,13]. They assume that all relevant service advertisements are initially loaded into a reasoning engine.

However, due to the large number of service advertisements and to the loose coupling between service providers and consumers, services are indexed in service directories. As loading a large directory of service advertisements into a reasoning engine is not practical, planning algorithms have been modified in order to dynamically retrieve relevant service advertisements from a service directory during composition [2,1].

[1] This work was supported by the Swiss National Funding Agency OFES as part of the European project KnowledgeWeb (FP6-507482).

[2] In this paper, *service* stands for *web service*.

S. Bressan, J. Küng, and R. Wagner (Eds.): DEXA 2006, LNCS 4080, pp. 317–326, 2006.

In such an approach, a single service composition may involve several complex directory queries, and each query may have to process a significant part of the directory. E.g., in the case of service composition algorithms using forward chaining, a single directory query may require the processing of up to 20% of the directory data, even though the directory uses an optimized index structure [1].[3] As the service directory is a shared resource, it is likely to become a performance bottleneck. Massive replication of the service directory is needed for scalability, which is expensive due to the large number of needed directory servers.

In this paper we present a novel approach to service composition, which avoids complex directory queries. In our approach, the directory offers a compact digest that summarizes the input/output behaviour of advertised services. The service composition clients download the digest and use it to solve the hard part of the composition problem locally. Only simple directory queries are issued to obtain the final result.

We compare two different representations of the directory digest, a sparse matrix and a Zero-Suppressed Reduced Ordered Binary Decision Diagram (ZDD). Both of them are compact, but the ZDD-based digest enables more efficient service composition algorithms.

The original contributions of this paper are: (1) a new approach of integrating service composition with large-scale service directories, (2) the introduction of a directory digest, (3) an evaluation of the digest size using different data structures, and (4) an evaluation of the performance of service composition algorithms leveraging the digest.

This paper is structured as follows: Section 2 introduces a simplified service description formalism and our definition of service composition. Section 3 gives an overview of service composition exploiting a directory digest. Section 4 evaluates the size of different digest representations. Section 5 evaluates the performance and scalability of service composition algorithms operating on the digest. Finally, Section 6 concludes this paper.

2 Definitions and Basic Service Composition Algorithm

Service descriptions are a key element for service discovery and service composition, as they enable automated interactions between applications. We distinguish between an invokable *service instance* (including service grounding) and its *service signature*. A service signature specifies the input/output behaviour of one or more service instances.

We describe a service signature S by two parameter sets – the *required input parameters* $in(S)$ and the *generated output parameters* $out(S)$. Each parameter is identified by a unique name in its set. We assume that the parameter name defines the semantics of the parameter. Despite its simplicity, our formalism is consistent with existing service description formalisms, such as WSDL [12] and OWL-S [7]. Part of the input/output specification of services described in WSDL or OWL-S can be mapped to our simplified formalism.

A *service directory* stores *service advertisements*, describing features of service instances available over the web. Each service advertisement includes the service signature of the advertised service instance.

[3] A naive directory implementation may have to process the whole directory contents in order to discover all relevant service advertisements.

A *service request* R is a query for a particular service functionality. R consists of a set of *provided input parameters* $in(R)$ and a set of *required output parameters* $out(R)$.

Service discovery involves the submission of a service request R to a service directory and the retrieval of service advertisements with a service signature S that matches R. In the literature, different matching relations between R and S have been studied [14,8,3]. One particularly useful matching relation is the *plugin match*, which requires that $in(S) \subseteq in(R)$ and $out(S) \supseteq out(R)$. I.e., given the provided input parameters $in(R)$, a service instance with service signature S can be invoked, which generates all required output parameters $out(R)$.

Even if there is no single service advertisement to fulfill a given service request R, it may be still possible to *compose* multiple services in such a way that the composite service (which can be represented as a workflow) meets R. E.g., assume there are service advertisements with the following signatures S_1 and S_2: $in(S_1) = \{A, B\}$, $out(S_1) = \{C\}$, $in(S_2) = \{B, C\}$, $out(S_2) = \{D\}$ Neither S_1 nor S_2 matches the service request R, where $in(R) = \{A, B\}$ and $out(R) = \{C, D\}$. However, a service instance with signature S_1 can be invoked with the provided input parameters $\{A, B\}$, which generates the output parameter C. Now the parameters $\{A, B, C\}$ are available, enabling an invocation of a service instance with signature S_2, which generates the required output parameter D.

Algorithm 1 shows a simple service composition algorithm using forward chaining. It takes a set of service signatures Dir and a service request R as inputs and returns $success$ resp. $failure$, depending on whether R can be fulfilled. For the sake of simplicity, we show only a decision algorithm; an extension to keep track of the applied services is trivial.

Algorithm 1. Simple decision algorithm based on forward-chaining to determine whether a service request R can be fulfilled by a set of service signatures Dir.

$Compose(Dir, R)$:
  $services \leftarrow Dir$;
  $availableInputs \leftarrow in(R)$;
  $requiredOutputs \leftarrow out(R)$;
  while $requiredOutputs \not\subseteq availableInputs$ **do**
    $applicableServices \leftarrow \{s \in services \mid in(s) \subseteq availableInputs\}$;
    if $applicableServices = \emptyset$ **then**
      return $failure$;
    $newAvailableInputs \leftarrow \bigcup_{s \in applicableServices} out(s)$;
    $availableInputs \leftarrow availableInputs \cup newAvailableInputs$;
    $services \leftarrow services \setminus applicableServices$;
  return $success$;

The algorithm iteratively extends the set $availableInputs$. In each iteration of the loop, it selects those service signatures for which all required input parameters are available and adds the output parameters of the selected service signatures to the set $availableInputs$. The algorithm terminates if all required output parameters are available or no further service signatures can be selected. In order to avoid the repeated

selection of the same service signature, the set $services$ is updated to include only those service signatures that have not been selected yet.

3 Service Composition with Directory Digest

In previous work [1], we used complex directory queries in order to dynamically retrieve relevant service advertisements from a large-scale service directory during service composition. This approach caused very high workload within the service directory and consequently also slowed down the service composition algorithm because of expensive remote interactions with the service directory.

The approach presented here increases scalability by avoiding complex directory queries during composition. The service directory offers a compact digest that summarizes the service signatures of all service advertisements stored in the directory. Service composition clients download the digest, which contains sufficient information to perform service composition locally on the client side. The service composition client interacts with the service directory as follows:

1. Download Digest. The client periodically downloads the most recent version of the digest. As service advertisements usually remain valid for a longer period of time, the clients do not have to reload their copy of the digest for every service request. Typical refresh rates would be once per day, once per week, etc.
2. Transform Service Request R. As the digest is a compressed representation of the service signatures in the directory, full parameter names are not available in the digest. Hence, the composition client sends R to the directory and receives R^T. The directory maps each parameter of $in(R)$ (resp. $out(R)$) to the corresponding digest parameter of $in(R^T)$ (resp. $out(R^T)$). This translation is a simple mapping and can be processed in linear time with the number of parameters in R.
3. Compute Composition. Using the digest and R^T, the client locally computes a service composition, without any queries to the service directory. If the composition fails, the client may update its local copy of the digest and retry (the new version of the digest may include additional service signatures of recently added service advertisements).
4. Transform Composition Result. If the service request has been successfully solved in the step before, the client knows the signatures of the selected services that are part of the composition workflow. It asks the directory to provide service advertisements that match the selected service signatures. The resulting directory query looks only for *exact matches*, which can be processed very efficiently by the directory. E.g., the directory may simply use the service signature to compute a hash key and look up matching service advertisements in a hash table. If the desired service signature is not found in the directory (i.e., services have been removed recently), the client has to download an up-to-date digest and re-run the composition algorithm.

4 Directory Digest Representation

The digest representation should meet the following requirements:

- Incremental Update. The addition or removal of a service signature shall not require to rebuild the digest from scratch.
- Incremental Addition of Service Parameters. The addition of a new service parameter shall not require restructuring the digest.
- Compact Network Transfer Format. The (serialized) digest shall be as small as possible in order to reduce network bandwidth.
- Compact in-memory Representation. Also the in-memory representation of the digest shall be compact in order to allow clients with limited computing resources to keep a copy of the digest in memory.
- Enabling Efficient Service Composition. The digest representation shall enable efficient service composition algorithms.

In the following we consider different ways to represent the directory digest. In Section 4.1 we discuss simple matrix representations, whereas in Section 4.2 we argue for an efficient representation as a combination set.

4.1 Matrix Representation

The directory digest can be regarded as a bit matrix, where each column corresponds to a certain parameter and each row describes a service signature. Assume there are n different parameters used in service signatures in the directory, which are identified by their index i ($0 \leq i < n$). The column position $2i$ corresponds to the i^{th} parameter used as input, while the position $2i+1$ corresponds to the i^{th} parameter used as output. This representation is extensible, i.e., a new parameter can be included by adding 2 columns.

As an example, assume we have the parameters A (index 0), B (index 1), C (index 2), D (index 3) and the two example service signatures S_1 and S_2 of Section 2: $in(S_1) = \{A, B\}$, $out(S_1) = \{C\}$, $in(S_2) = \{B, C\}$, $out(S_2) = \{D\}$ This can be represented by the following bit matrix:

Column index	0	1	2	3	4	5	6	7	Corresponding
Column meaning	A_{in}	A_{out}	B_{in}	B_{out}	C_{in}	C_{out}	D_{in}	D_{out}	service
	1	0	1	0	0	1	0	0	S_1
	0	0	1	0	1	0	0	1	S_2

For a large number of parameters and a large number of different service signatures, the matrix may become quite large. For instance, assume we have 1000 different parameters and 10^6 different service signatures. The resulting matrix has $2 * 10^9$ bits, i.e., about 238MB.

However, in practice, the matrix is sparse, i.e., each row has only a few bits set, because the number of parameters per service signature is limited. Hence, a more compact representation can be obtained by listing the column indices of only those parameters that are present. If there are 1000 different parameters, the column indices range from 0 to 1999, which can be represented by 11 bits. As row delimiter, either an otherwise unused index (e.g., $2^{11} - 1$) may be defined or we add one extra bit to mark the end of a row. If we assume that on average each service has 3 input and 3 output parameters and we use an extra bit to mark the end of rows, the matrix can be represented

by $(3+3)*12*10^6$ bits, which is less than 9MB. Further reductions of the network transfer format could be obtained e.g. by applying standard compression algorithms. Concerning the in-memory representation, typically 16 bits would be used for a column index (byte alignment). I.e., a client keeping the matrix in memory would consume $(3+3)*16*10^6$ bits, less than 12MB.

While the sparse matrix representation is compact and can be updated incrementally, it is not the best data structure for efficient service composition. In order to discover applicable service signatures, the whole matrix has to be processed repeatedly. Each iteration of the service composition algorithm causes processing effort that is proportional to the size of the matrix.

4.2 Combination Set Representation

If we consider a service signature a combination of parameters, the directory digest can be seen as a combination set. E.g., the example service signatures S_1 and S_2 shown before can be represented by the following combination set: $\{\langle A_{in}, B_{in}, C_{out}\rangle, \langle B_{in}, C_{in}, D_{out}\rangle\}$

Zero-suppressed Reduced Ordered Binary Decision Diagrams (ZDD) are known as an efficient representation of sparse combination sets [5]. ZDDs efficiently support set operations, such as union, intersection, and difference. Moreover, many specialized ZDD operations have been developed for particular use cases, such as e.g. the restriction and exclusion operations in the context of constraint satisfaction [6].

We adopted ZDDs, because they allow a compact digest representation and enable the implementation of efficient service composition algorithms exploiting ZDD operations. Below we discuss some experimental results concerning the size of the ZDD representation. In Section 5 we present performance measurements for a service composition algorithm that leverages ZDD operations.

In order to measure the size of ZDDs for different settings, we created service directories with an increasing number of randomly generated, distinct service signatures (0–10^6). Each service signature S has 3 input and 3 output parameters, randomly chosen from a set of 1000 different parameters with the constraint $in(S) \cap out(S) = \emptyset$. In practice, services often have parameters only from a single, domain-specific ontology (e.g., travel domain, financial domain, sports domain, etc.). Hence, we partitioned the 1000 parameters into an increasing number of domains (1–100) and required the input parameters (resp. the output parameters) of each service signature to come from the same domain (the input parameters' domain can be different from the output parameters' domain).

Fig. 1 shows the size of a ZDD representing a directory digest depending on the number of different service signatures and the number of domains. As ZDDs are graphs, we use the number of nodes in the graph as metric. Not surprisingly, the ZDD size increases with the number of distinct service signatures. A small number of domains results in larger ZDDs than a high number of domains. In the worst case (10^6 service signatures, 1 domain), the ZDD has about $2.9 * 10^6$ nodes. For 100 domains (each consisting of 10 parameters), 10^6 service signatures require only about 10^6 nodes. The reason for the smaller ZDD size is that the number of possible parameter combinations is reduced.

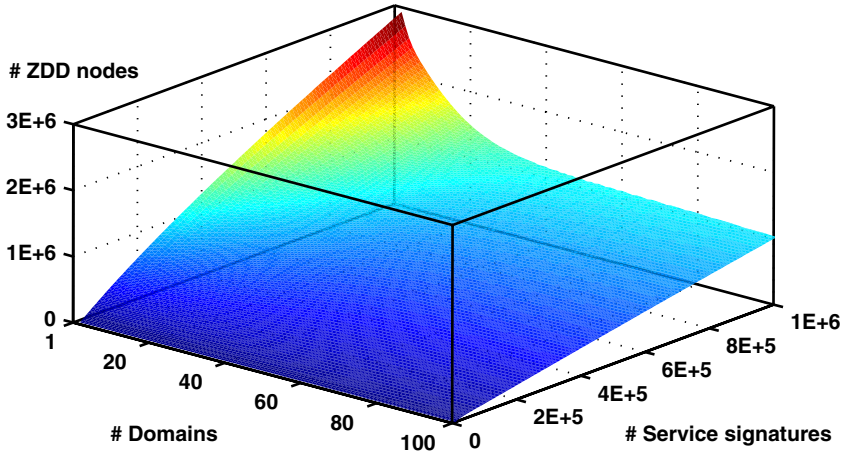

Fig. 1. Number of nodes in a ZDD representing a directory digest as a function of the number of service signatures in the digest and the number of domains

Our directory digest implementation is based on the JDD library[4], which is programmed in pure Java. Concerning the in-memory representation, JDD stores the ZDD nodes in an array of 32 bit integers. Each node uses 3 entries in the array: One entry stores a variable[5] and the other two entries store indices to other nodes. In total, a node consumes 12 bytes in memory. For the network transfer of the directory digest, the ZDD can be serialized more compactly: 11 bits are enough to store a parameter index and 24 bits suffice to index other nodes, i.e., 59 bits per node are sufficient. Further compression of the bitstream using standard compression techniques would be possible as well.

Fig. 2 compares the size of different directory digest representations. The sparse matrix is more compact, although in a setting with 20 domains, the ZDD representation is not much larger. As we will see in the next Section, the ZDD representation enables more efficient service composition algorithms.

5 Service Composition Performance

In this Section we present experimental results concerning the performance of service composition algorithms operating on a sparse matrix resp. on a ZDD representation of the directory digest. The algorithms are specialized implementations of the generic structure of Algorithm 1 introduced in Section 2.

As in-memory representation of the sparse matrix we chose an array of service signatures (matrix rows), where each service signature is an array of parameter indices (matrix columns). Moreover, we use a boolean array to indicate for each service signature whether it has been applied. The algorithm iterates through the array of service

[4] `http://javaddlib.sourceforge.net/`

[5] We map parameters to ZDD variables. We encode each parameter P as two different ZDD variables P_{in} and P_{out}. E.g., in our experimental setting we have 2000 ZDD variables.

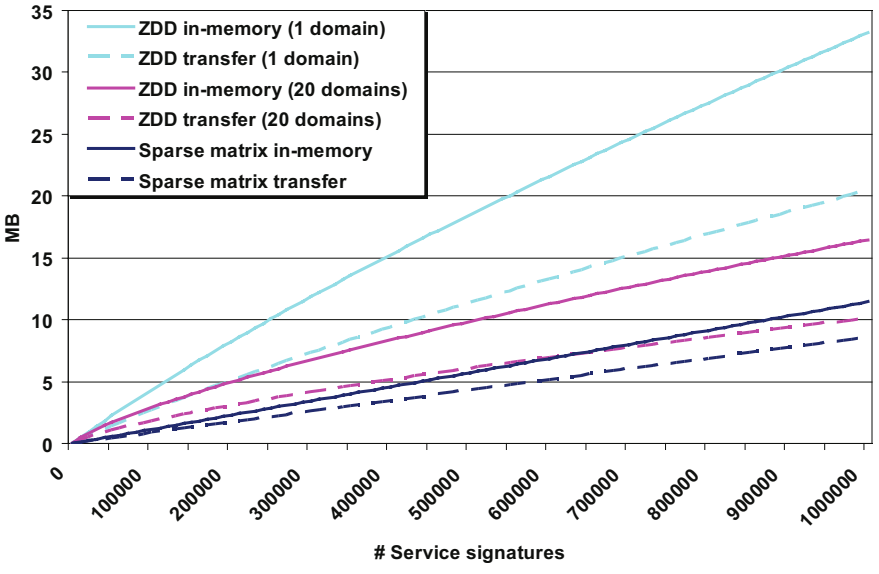

Fig. 2. Directory digest size (in megabytes) of in-memory and network transfer representations based on a sparse matrix resp. on a ZDD

signatures in a cyclic way, selecting and applying those signatures for which all required input parameters are available (and which have not been applied before).

Our ZDD-based service composition algorithm relies on the repeated application of the $subset0()$ primitive [5] in order to filter out those service signatures that require a parameter which is not available. The remaining service signatures are selected. Then the algorithm leverages the $subset1()$ primitive to test for each missing parameter, whether a selected service generates it as output. If at least one missing parameter becomes available, the whole process is iterated.

Fig. 3 compares the performance of the two service composition algorithms for different digest size (0–10^6) and distinct number of domains (1 and 20). As before, we assume 1000 possible parameters and service signatures with 3 input and 3 output parameters. Each measurement represents the total execution time for processing a set of 2000 random service requests. We chose service requests that are particularly hard to process, providing only 3 input parameters and requiring all 1000 output parameters. As we kept the set of 2000 service requests unchanged throughout all experiments, the percentage of solvable service requests is increasing with the number of service signatures in the digest. Both algorithms operate on the same digest contents and on the same set of service requests. For a fair comparison, we disabled all caches in the JDD library. As execution platform we chose a typical client system: Pentium 4, 2.4GHz clockrate, 512MB RAM, Windows XP, Sun JDK 1.5.0 Hotspot Server VM. In order to obtain reproducible measurements, we disabled background processes as much as possible. Each measurement represents the median of 15 executions on identical data.

As expected, the algorithm based on the sparse matrix digest representation performs linear with the number of service signatures in the digest. The number of domains does

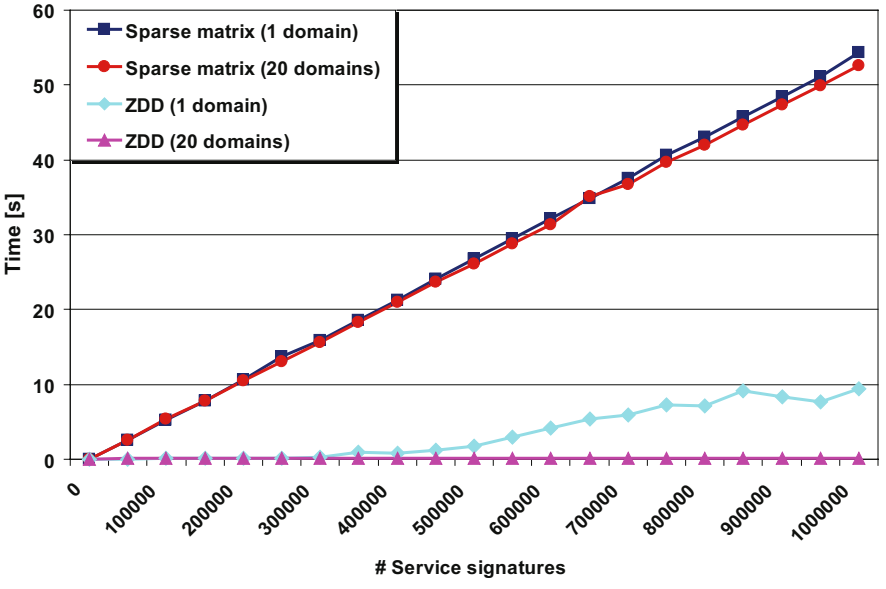

Fig. 3. Elapsed processing time [s] for 2000 random service requests

not have much impact on the performance (the matrix size is not affected by the number of domains). The algorithm based on the ZDD representation performs significantly better. In the case of a single domain (i.e., larger digest), the ZDD algorithm is 5–100 times faster than the matrix algorithm; in the case of 20 domains (i.e., smaller digest), the ZDD algorithm is 40–500 times faster. In the latter setting, the performance of the ZDD algorithm does not degrade with an increasing number of service signatures.

6 Conclusion

Service composition algorithms that dynamically retrieve relevant service advertisements from a large-scale service directory tend to issue a large number of complex directory queries, causing high workload within the directory. As such an approach does not scale well, we introduce a compact directory digest that is downloaded by service composition clients and allows to solve the hard part of a composition problem locally without expensive remote interactions with the directory. Directory queries are only needed to translate the initial composition problem and to obtain the final result. These queries are very simple and require only a lookup in a table, significantly reducing the workload in the directory and boosting scalability.

We compared two different representations of the directory digest – sparse matrix versus ZDD. Both representations result in a compact digest. However, a service composition algorithm based on the ZDD representation performs and scales significantly better than an algorithm operating on the matrix representation.

References

1. I. Constantinescu, W. Binder, and B. Faltings. Flexible and efficient matchmaking and ranking in service directories. In *2005 IEEE International Conference on Web Services (ICWS-2005)*, pages 5–12, Florida, July 2005.
2. I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, pages 506–513, San Diego, CA, USA, July 2004.
3. 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*, pages 331–339, 2003.
4. 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. 2002. Morgan Kaufmann Publishers.
5. S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 272–277, Dallas, TX, June 1993. ACM Press.
6. H. G. Okuno, S. ichi Minato, and H. Isozaki. On the properties of combination set operations. *Information Processing Letters*, 66(4):195–199, May 1998.
7. OWL-S. DAML Services, `http://www.daml.org/services/owl-s/`.
8. 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*, pages 333–347, 2002.
9. M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. *Communications of the ACM*, 46(10):24–28, Oct. 2003.
10. S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. 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. Web services description language (WSDL) version 1.2, `http://www.w3.org/TR/wsdl12`
13. 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 (ISWC-2003)*, pages 195–210, 2003.
14. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.