Large scale, type-compatible service composition

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

1. Introduction

Service composition is an exciting area which has received a significant amount of interest in the last period. Initial approaches to web service composition [15] 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 [10, 11] and SWORD [14]) or on hierarchical task planning (e.g., SHOP-2 [21]). 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 [7] has addressed the problem of interleaving discovery and integration in more detail, but he has 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 should 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 completley[2], 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 contributes with solutions to the two aforementioned issues: As a first contribution we present an algorithm that interleaves the discovery and composition process by using a partial order planning approach that focuses the directory searches. Our second contribution is a technique for discovering and composing services with partial type compatibility. In our approach we discretize the

1 The work presented in this paper was partly carried out in the framework of the EPFL Center for Global Computing and was supported by the Swiss National Science Foundation as part of the project MAGIC (FNRS-68155), as well as by the Swiss National Funding Agency OFES as part of the European projects KnowledgeWeb (FP6-507482) and DIP (FP6-507483).

2 We consider as partial matches the **subsume** match type identified by Paolluci [12] and the **intersection** or **overlap** match type identified by Li [8] and Constantinescu [3].

IEEE COMPUTER SOCIETY

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.

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 following: next in Section 2 we present the formalism and assumptions used in the rest of the paper as long as a more in-depth view regarding the problem of type-compatible service composition. In Section 3 we describe the machinery and algorithms that we actually use for computing service compositions. Section 4 analyses some experimental results on randomly generated problems and 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, or in a backward way, starting from the problem requirements. Forward or backward chaining techniques are used by different types of reasoning systems, in particular for planning [2] and more recently for service integration [15]. 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 [18] as two sets of parameters (inputs and outputs). A parameter is defined through its name and a type that can be primitive [20] (e.g., a decimal in the range [10,12] or [14,16]) or a class/ontological type [17]. 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 Section 2.5.

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 [17]. 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 sub-class (possibly null) 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 [18]) attach also some se-

mantic information to the parameters[3]. Thus, in our composition algorithm we not only consider type compatibility between parameters but also semantic compatibility.

2.2. Composing services

We are considering two kinds of composition approaches: forward chaining and backward chaining. 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 [15]. In the case of backward chaining we start from the set of parameters required by the query Q and at each step of the process we choose a service S that will provide at least one of the required parameters. Applying S might result in new parameters being required which can be formalised as a new query Q'. Again the process is iterated until a solution is found.

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 [12].

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.

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

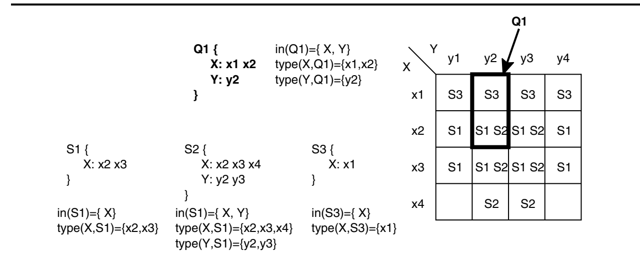

Figure 1. Composing services with partially matching types.

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 [8] and Constantinescu [3].

We will also consider the condition needed for a backward chaining approach. The service S has to provide at least one output which is required by the query Q. This corresponds to the **plugIn** match for query and service outputs. Using the formal notation above this can be specified as:

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

The above condition can be also relaxed such that services with *partial type matches* can be backward chained. In this paper we do not address this but we consider this issue for future work.

2.3. Type-compatible service composition versus planning

As the majority of service composition approaches today rely on planning we will analyse the correspondence between our formalism for service descriptions with types and an hypothetic planning formalism using symbol-free first order logic formulas for preconditions and effects.

As an example let's consider the service description S which has two input parameters A and B and two output parameters C and D. Their types are represented as sets of accepted and provided values and are a1, a2 for A, respectively b1, b2 for B, c1, c2 for C, and d1, d2 for D. This cor-

IEEE COMPUTER SOCIETY

responds to an operator S that has disjunctive preconditions and disjunctive effects. Negation is not required.

Written in this way our formalism has some correspondence with existing planning languages like ADL [13] or more recently PDDL [9] (concerning the disjunctive preconditions) and planning with non-deterministic actions [6] (regarding the disjunctive effects), but the combination as a whole (positive-only disjunctive preconditions and effects) stands as a novel formalism.

2.4. The structure of type-compatible service composition problems

As described previously in Section 2.1 a service integration query is specified in terms of a set of available input parameters and a set of required output parameters. An integration solution consists of a given ordering of services that can be invoked such that finally all parameters required by the query are known.

From the perspective of the match type between services and queries (Fig. 2) we consider the following three cases: forward complete matches, backward complete matches, and forward partial matches.

By using forward-completely matching services the initial set of available parameters can be incrementally extended. As there is a single point from which a service can be applied - once all its required inputs are available - forward chaining services does not introduce any choice points.

Applying backward-completely matching services creates a directed graph of sets of required parameters as the order in which different parameters can be applied affects the set of parameters that still need to be provided.

Several forward-partially matching services can be aggregated together into a composite service as a software switch that maps each possible combination of parameter values from the space of available inputs to one or more

```
in(S) = [A, B]              :action S
type(A,S) = [a1, a2]        :precondition
type(B,S) = [b1, b2]           (and
                                (or a1 a2)
                                (or b1 b2))

out(S) = [C, D]             :effect
type(C,S) = [c1, c2]          (and
type(D,S) = [d1, d2]            (or c2 c2)
                                (or d2 d2)
```

Table 1. Service with types and corresponding planning operator.

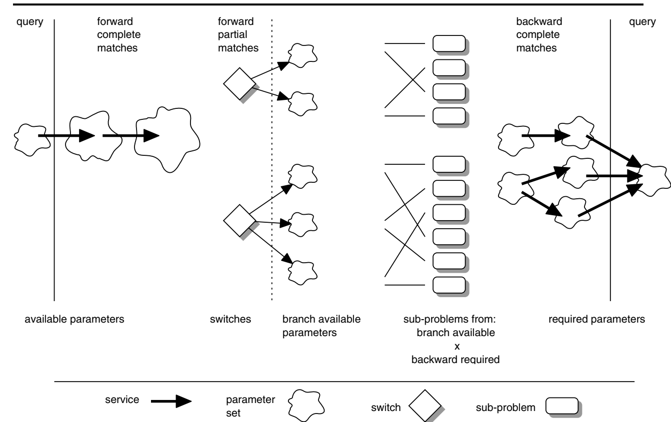

Figure 2. The structure of type-compatible service composition problems.

partially matching services as in Fig. 1. In order to be able to fulfill the same functionality as the *completely matching* service we have to have for each possible range combination of input parameters one or more services that can accept those values.

Our software switch corresponds to a non-deterministic planning operator in that the choice point that it introduces will allow for a number of possible service invocation paths to be followed without commitment at integration time to a particular one. The choice will be made only at runtime based on the values of the switch input parameters.

Each of the branches in a switch will provide a (possibly different) set of available parameters. It has to be noted that in order for the switch to be part of a service integration solution **all** of the distinct sets of available outputs of the switch will have to be part of an integration solution. Still for determining which branches can lead to a solution we might have to construct for each pair of branch available outputs and backward complete required inputs a subproblem that we then solve recursively.

2.5. Representing types

Service descriptions are a key element for service discovery and service composition and should enable automated interactions between applications. Currently, different overlapping formalisms are proposed (e.g., [18], [16], [4], [5]) and any single choice could be quite controversial due to the trade-off between expressiveness and tractability specific to any of the aforementioned formalisms.

In this paper, we will partially build on existing developments, such as [18], [1], and [4], by considering a simple table-based formalism where each service is described through a set of tuples mapping service parameters (unique

IEEE COMPUTER SOCIETY

names of inputs or outputs) to parameter types (the spaces of possible values for a given parameter). Parameter types can be expressed either as sets of intervals of basic data types (e.g., date/time, integers, floating-points) or as classes of individuals. Class parameter types can be defined through a descriptive language like XML Schema [19] or the Ontology Web Language [17]. From the descriptions we can then derive either directly or by using a description logic classifier a directed graph (DG) of simple is-a relations.

For efficiency reasons, we represent the DG numerically. We assume that each class will be represented as a set of intervals. Then we encode each parent-child relation by subdividing each of the intervals of the parent; in the case of multiple parents the child class will then be represented by the union of the sub-intervals resulting from the encoding of each of the parent-child relations. Since for a given domain we can have several parameters represented by intervals, the space of all possible parameter values can be represented as a rectangular hyperspace, with a dimension for each parameter. Details concerning the numerical encoding of services can be found in [3].

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:

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

Our algorithms address these issues by interleaving discovery and composition and by computing the "right" query at each step.

3.1. Composition with complete type matches

As described previously in Section 2.4 depending on the initial query and on how the services retrieved up to a given point can be chained (forward or backward) a set of available input parameters and several sets of possible required output parameters can be obtained.

In order to manage the current state of the search we use a data-structure that maintains the set of forward available parameters and a graph of sets of backward required parameters. For a given set of services the structure identifies which are the services that can be applied in a forward way and which services can be used for fulfilling required parameters in a backward way. The sets of backward required parameters are also computed based on the current forward available parameters.

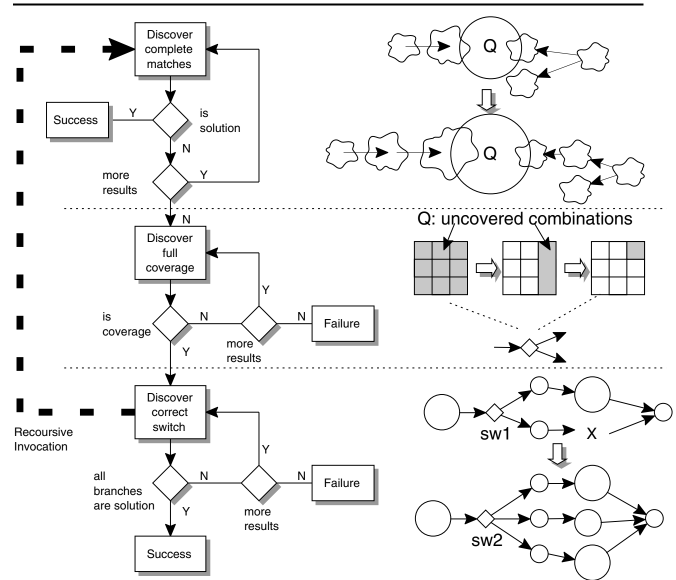

Figure 3. Flow of algorithm for composition with partial type matches.

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.
- Discovery of services for full coverage of available inputs.
- Discovery of services for correct switch handling.

3.2.1. Discovering full input coverage The discovery of complete matching services is done using one of the previous algorithms for complete matches. If a solution is found then the algorithm returns success. This step ends when no more complete matches can be found. this step ends.w

3.2.2. 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

IEEE COMPUTER SOCIETY

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 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 passes at the next step. If no more partially matching services can be found and a complete coverage was not achieved the algorithm returns failure.

3.2.3. 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 it's branches. For that the algorithm will compute for each cell and its set of assigned services 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 recoursively. 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: one more concrete and another that provides more symmetry. A discussion of the results concludes this section.

4.1. The Media domain

For experimental purposes we have considered the following scenario (see Fig. 4): a personal agent (PA) is delegated the task to find music provider for an album of good pop music. Since its user doesn't like to listen only to spare melodies we assume all-or-nothing semantics: only if all the melodies of a given album can be found then this is a solution. Otherwise another album has to be selected.

First PA has to determine what is currently considered as good pop and uses a recommendation site (e.g., billboard.com) to find Album X as a first option. Then the PA has to use a library site (e.g., cdcovers.cc) to determine which are the melodies in Album X - let's presume that they are Melody-1, Melody-2 and Melody-3. Then finally the PA searches different p2p networks for providers that have different melodies from the album.

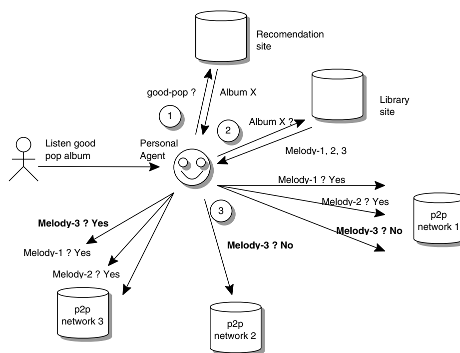

Figure 4. Putting together a good pop music album.

In this context partial type matches are used for selecting a peers that can provide (cover) a sub-set of the melodies of an album. Then by computing the software switch we select those peers that together can provide all the melodies from the album.

Since for solving the integration we need at least a recommendation, an album description and a provider for the melodies the minimum number of required directory accesses for any integration in this domain is 3.

4.2. The layered domain

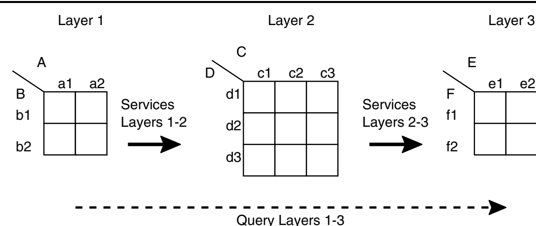

Figure 5. The layered domain - simplified example.

We have defined also a more abstract domain (see Fig. 5) where we consider a number of layers that define sets 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. For example, a possible service between layers1-2 with the parameters A, B could have as input the types A=a1, B=b1,b2 and for the output parameters C and D could have as types C=c2, c3 and D=d1, d2. A query could for the input parameters A, B the types A=a1,a2, B=b2 and for the output parameters E,F the type E=e1, F=f2.

IEEE COMPUTER SOCIETY

For the purpose of our experiments, we have created environments with 3 layers for which the minimum number of required directory accesses for any integration is 2. Each of the layers defines two parameters with 11 possible subtypes for layers 1 and 3 and 63 possible overlapping subtypes for layer 2; between each two layers (1-2 or 2-3) there are 480249 possible services.

It has to be noted that in contrast with the media domain there is a symmetry regarding inputs and outputs in the configuration used for the layered domain.

4.3. Evaluation of experimental results

For both domains, we have randomly generated services and problems. For each specific type of services (e.g., album recommendation, album description, melody provider or service layer1-2, service layer2-3) or for queries (e.g., find good pop album in mp3 format or find a service that transforms an input in layer1 to an output in layer3) we had a pre-determined set of parameters. For actually generating the descriptions (service or query) we randomly picked for any of the pre-determined parameters a random sub-type from a set of possible types.

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. 6 (a) and Fig. 7 (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.

Regarding performance, the first conclusion is that 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. For the media domain the number of accesses actually decreases.

As expected, the algorithm using partial matches performs comparable with the one using complete matches. 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 probably 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. 7 (a) after 3000 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. 6 (b) and Fig. 7 (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: **4 times** in the case of the Media domain and up to **7 times** in the case of the Layered 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 planning, such as planning as model checking, are being considered for web 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] D.-S. C. A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web service description for the Semantic Web. *Lecture Notes in Computer Science*, 2342, 2002.

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

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

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

[5] FIPA. Foundation for Intelligent Physical Agents Web Site, http://www.fipa.org/.

[6] N. Kushmerick, S. Hanks, and D. S. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1–2):239–286, 1995.

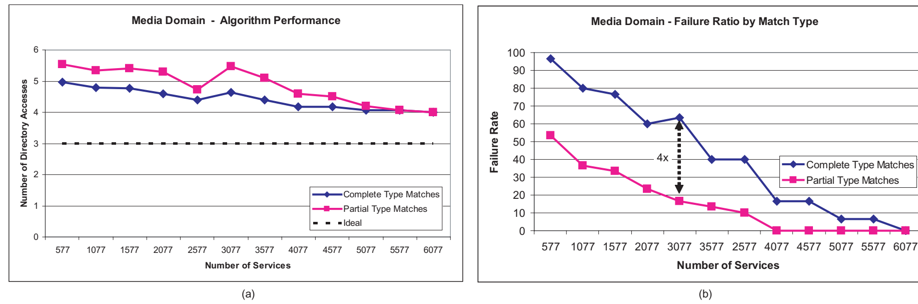

Figure 6. The media domain.

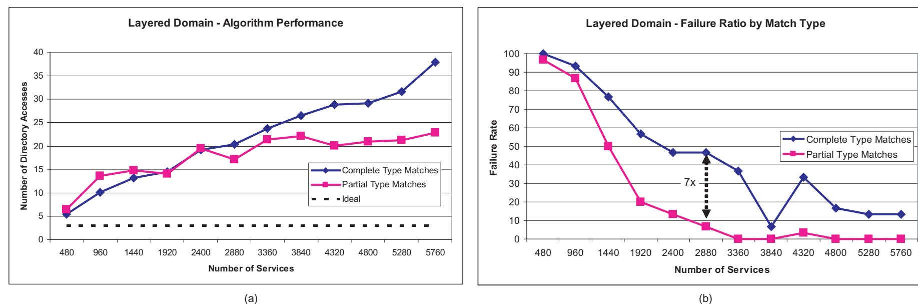

Figure 7. The layered domain.

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

[8] 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.

[9] D. McDermott. The planning domain definition language manual. Technical Report 1165, Yale Computer Science, 1998.

[10] 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.

[11] 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.

[12] 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.

[13] E. P. D. Pednault. Adl: Exploringthe middle ground between strips and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, pages 324–332, Morgan Kaufmann Publishers, 1989.

[14] 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.

[15] 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.

[16] UDDI. Universal Description, Discovery and Integration Web Site, http://www.uddi.org/.

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

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

[19] W3C. XML Schema, http://www.w3.org/xml/schema.

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

[21] 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