Large scale testbed for type compatible service composition

Ion Constantinescu and **Boi Faltings** and **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

In a service-oriented environment services can be composed such that new value added services are created. The problem of service composition raises a number of specific issues like the openness of the environment and the large number of possible services. As currently such massive service deployments do not exist, the current techniques proposed for service composition are difficult to test and compare. In this paper we propose a synthetic testbed that can be used for simulating large deployments of services and also for generating service composition problems. The testbed could prove to be an useful tool for the understanding, characterization and comparison of different service composition approaches.[1]

Introduction

Service oriented computing has opened several new research directions which include service discovery, composition, orchestration and execution monitoring. Service composition seems to be one of the most challenging area which has received a significant amount of interest in the last period. One of the major difficulties in solving composition problems lies in the large number of possible services. But since to date there are no massive deployments of services, the approaches currently proposed are difficult to test and compare.

Initial approaches to web service composition (Thakkar *et al.* 2002) described the Building Finder application, where a number of manually defined data-sources like the Microsoft Terraservice, U.S. Census Bureau information files, geocoding information and different real estate property tax sites where composed using a forward chaining technique.

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., ConGolog (McIlraith, Son, & Zeng 2001; McIlraith & Son 2002) and SWORD (Ponnekanti & Fox 2002)) or on hierarchical task planning (e.g., SHOP-2 (Wu *et al.* 2003)). In the scenario used in the ConGolog approach the composition engine would have to book flight tickets and arrange ground transportation and hotel reservations. For SWORD the example used was of a composed service giving driving directions to one's home. The composed service was formed from two services, one that mapped names to addresses and another that was giving driving directions to a given address. In the motivating example in the SHOP-2 approach, for handling a medical emergency, several data sources had to be composed and a schedule had to be computed.

The above approaches consider application domains with that have discrete numbers of services. This is more similar to classic planning approaches that assume small numbers of operators and where the difficulty is mainly due to the large space of possible states and to embedded hard resource-allocation problems. Still for our testbed we are interested to investigate issues that are specific and unique to the web services context like:

1. **large scale service directories** - we assume that our testbed will contain a large number of available services in the form of a yellow-page directory. But with what terms should those services be defined and what kind of relations should be between different terms ? Moreover what kind of transformations should services perform between different sets of terms ?
2. **partial type matches** - composition engines should be able to discover and also reason with services with types that match partially but not completely[2]. How can we represent in our testbed these kinds of services and how should we discover them ?

The main contribution of this paper is a synthetic testbed which addresses the above issues: large numbers of services with various relations can be generated and the formalism used for representing services supports complete and partial type matches. The testbed is highly parameterizable and allows instantiations to different configurations. The testbed has a graph structure where each graph node corresponds to

[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 (Paolucci *et al.* 2002) and the **intersection** or **overlap** match type identified by Li (Li & Horrocks 2003) and Constantinescu (Constantinescu & Faltings 2003).

an application domain and graph edges correspond to services that perform transformations between parameters in different domains.

This paper is structured as following: in the next section we present the formalism used for defining services as long as a more in-depth view regarding the problem of type-compatible service composition. Then in the section "Large scale testbed for type compatible service composition" we describe in more detail our testbed and the parameters trough which different configurations can be instantiated. The "Conclusion" section ends the paper.

Type compatible service composition

We consider service composition approaches that are 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 (Blum & Furst 1997) and more recently for service integration (Thakkar *et al.* 2002). We describe next the formalism that we use to model, match, and chain services.

Formalism and assumptions

We represent services and queries in the standard way (W3C b) as two sets of parameters (inputs and outputs). Preconditions and effects as in (DAML-S) can also be directly represented: inputs and outputs can be seen as information preconditions and effects. Negative effects cannot be directly represented but they are also not explicitly allowed by usual services decscription formalisms (e.g., (DAML-S)).

A parameter is defined through its name and a type that can be primitive (W3C d) (e.g., a decimal in the range [10,12] or [14,16]) or a class/ontological type (W3C a). 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 below the section "Representing types".

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 (W3C a). 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 (DAML-S) or strings in the case of WSDL (W3C b)) attach also some semantic information to the parameters[3]. Thus, in our composition algorithm we not only consider type compatibility between parameters but also semantic compatibility.

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 (Thakkar *et al.* 2002). In the case of backward chaining

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

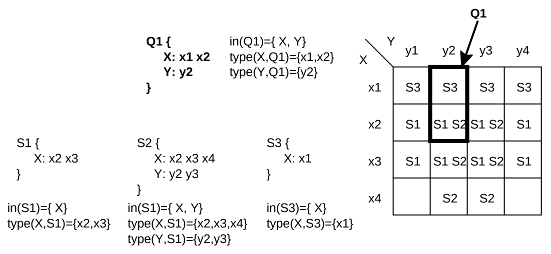


Figure 1: Composing services with partially matching types.

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 formalized 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:

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

This kind of matching between the inputs of query Q and of service S corresponds to the **plugIn** match identified by Paolluci (Paolucci *et al.* 2002).

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.

In order to be able to handle the situations when the types of services *partially match* we extend our framework by introducing a relaxation of the above condition for forward chaining. We do that by replacing type inclusion with a weaker overlap:

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

This kind of matching between the inputs of query Q and of service S corresponds to the **overlap** or **intersection** match identified by Li (Li & Horrocks 2003) and Constantinescu (Constantinescu & Faltings 2003).

We will also consider the condition needed for a backward chaining approach in the case of *complete type matches*. 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:

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

The above condition can be also relaxed for backward chaining services with *partial type matches*:

$$(\exists P \in out(S))\ (P \in out(Q) \wedge type(P,Q) \cap type(P,S) \neq \emptyset)$$

Type-compatible service composition versus planning

As the majority of service composition approaches today rely on planning we will analyze 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 corresponds 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 (Pednault 1989) or more recently PDDL (McDermott 1998) (concerning the disjunctive preconditions) and planning with non-deterministic actions (Kushmerick, Hanks, & Weld 1995) (regarding the disjunctive effects), but the combination as a whole (positive-only disjunctive preconditions and effects) stands as a novel formalism.

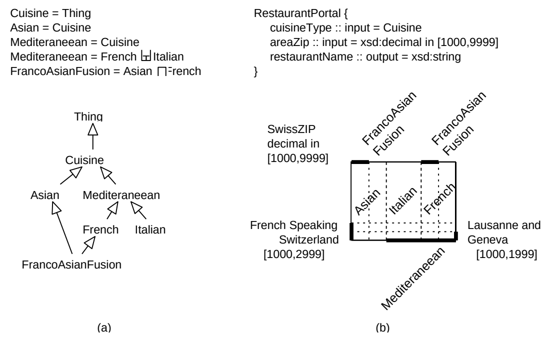
Representing types

In this section we will present the encoding used for numerically representing service descriptions in more detail. 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., (W3C b), (UDDI),

```
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.



```
Cuisine = Thing
Asian = Cuisine
Mediteraneean = Cuisine
Mediteraneean = French ⊔ Italian
FrancoAsianFusion = Asian ⊓ French
```

```
RestaurantPortal {
    cuisineType :: input = Cuisine
    areaZip :: input = xsd:decimal in [1000,9999]
    restaurantName :: output = xsd:string
}
```

Figure 2: An example domain: a restaurant recommendation portal.

(DAML-S), (FIPA)) 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 (W3C b), (Ankolekar *et al.* 2002), and (DAML-S), by considering a simple table-based formalism where each service is described through a set of tuples mapping service parameters (unique 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 (W3C c) or the Ontology Web Language (W3C a). 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 (e.g., the is-a directed acyclic graph (DAG) for types of cuisine in Fig. 2 (a) derived from the ontology above).

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 sub-dividing each of the intervals of the parent (e.g., in Fig. 2 (b) Mediteranean is sub-divided for encoding Italian and French cuisine); 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 (e.g., the FrancoAsianFusion in Fig. 2 is represented through sub-intervals of the Asian and French concepts).

Since for a given domain we can have a number of parameters represented by intervals, the space of all possible parameter values can be represented as a rectangular hyper-space.

Let's consider as an example (see Fig. 2 (b)) a restaurant recommendation portal that takes the user preference for a cuisine type and the Swiss zip-code (four digit numbers between 1000 and 9999) of the area of interest and will return a string containing a recommended restaurant located in a given area.

In this example the service will accept for the cuisineType parameter any of the keywords $Mediteranean$, $Asian$, $French$, $Italian$, or $FrancoAsionFusion$ and for the $areaZip$ any decimal between 1000 and 9999 representing the location of the area of interest.

Large scale testbed for type compatible service composition

We present next the main contribution of this paper, our testbed for large scale service composition. The testbed is build 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.

From the technology perspective there are several kinds of options for building web sites ranging from static web pages to more dynamic one developed using languages specific to server side scripting (e.g., ASP, PHP or JSP). For generating the content to be presented, the dynamic pages can access directly a data layer (e.g., a relational database) or can use and intermediate objectual layer that implements the business logic and encapsulates data (e.g., an application server).

At a higher level, different applications are usually organized accordingly to their domain (e.g., traveling, entertainment, real-estate or medical). Understanding, developing and using the terminology required for a specific domain requires usually a significant knowledge engineering effort.

Our testbed builds on the previously identified concepts of data distribution and domain distribution by using application domains as the core idea. In our framework an application domain represents a collection of terms and their associated data types. Then services 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.

We will present next in more detail the application domain nodes and the service graph structure.

Application domain nodes

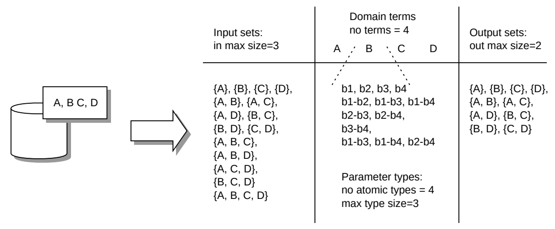


Figure 3: The application domain node.

In our framework terms are a first order concept used for the specification of an application domain. Each term maps in the frame of a service description specifiying it either to an input or to an output parameter. For each term an application domain defines a set of possible data types.

For speeding up the generation process we exhaustively generate all possible input or output parameter sets by generating the power set of the domain terms. Since we consider that the number of terms in a domain will be of an order of magnitude larger that the number of parameters that a service will usually use as input or output, we will establish a *maximum size* of the parameter sets and we will we filter the initial power-set accordingly. For example in the case of a domain with terms A, B, C, D with the maximum number of parameters for a service of 2, we will have as possible parameters the sets A, B, C, D, A, B, A, C, A, D, B, C, B, D and C, D. We will filter out the sets A, B, C, A, B, D, A, C, D, B, C, D and A, B, C, D since their cardinality passes 2.

As a given service could use a given set of terms in a domain either as input parameters or as outputs parameters, we make the same differentiation regarding the sets of possible terms such that by having different *maximum sizes* for possible inputs and possible outputs, we will have a different fan-in and fan-out regarding the number of services that could make transformation from or to an application domain. For example the domain previously mentioned could have 2 as the *output maximum size* but could have 3 as the *input maximum size*, which would result in a fan-in/fan-out rapport of 3/2.

For each term in a domain we define a number of possible data types. First, for each term a number of "atomic" types is specified. We consider that the number of occurrences for each of the atomic types obeys a power-law distribution of the form $1/i^a$ where i the index of the type and a is close to the unit. Using this we compute occurrence frequencies for each of the atomic types. We then create an "zipf atomic set" of a given size in which the atomic types appear once or more, accordingly to their frequencies. Then from the "zipf atomic set" we create as above the power set of the atomic types while keeping a higher bound for the cardinality of the obtained sets. These sets will represent "composite types" obtained from the concatenation of one or more "atomic types". The "composite types" are normalized such that double occurrences of the same atomic type are discarded (e.g., $\{b1, b1, b2\} \Rightarrow \{b1, b2\}$) and that consecutive types are merged (e.g., $\{b1, b2, b3\} \Rightarrow \{b1 - b3\}$).

The service graph structure

We generate services in our testbed as transformation between sets of terms in two application domains. For doing that for each service we first randomly pick two application domains. Then we randomly pick a parameter set from the set of input parameters of the first application domain and a parameter set from the set of output parameters of the second application domain. Then for each of the parameters in the two sets we randomly pick a parameter data-type from the respective application domains.

The main constraint that we enforce while making the choices above is to pick different application domains, as we are interested mainly in the cross-domain chaining of services. Another constraint that we enforce is the filtering of duplicates, services that have exactly the same inputs and outputs.

For generating test problems we use a similar algorithm as for creating services, by randomly picking domains, parameter sets and parameter data-types. The major difference stands in the different interpretation of input and output parameters in the case of a query (see the "Type compatible service composition" section). The parameters selected from the first domain input set are "available inputs" and the parameters selected from the second domain output set are "required outputs".

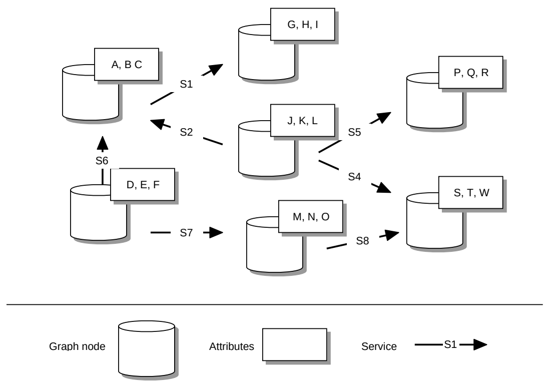


Figure 4: The service graph structure.

Conclusions

In this paper we have considered the problem of service composition and we have presented a synthetic testbed which addresses issues that are specific to this problem: the generation of large numbers of services with various relations, services described using formalism that supports complete and partial type matches. The presented testbed is highly parameterizable and allows instantiations to different configurations. We think our contribution could be highly useful for the developers of service composition algorithms and the users of composition engines as it will allow a better understanding and characterization of different techniques and a more precise comparison of different systems.

References

Ankolekar, D.-S. C. A.; Burstein, M.; Hobbs, J. R.; Lassila, O.; Martin, D.; McDermott, D.; McIlraith, S. A.; Narayanan, S.; Paolucci, M.; Payne, T.; and Sycara, K. 2002. DAML-S: Web service description for the Semantic Web. *Lecture Notes in Computer Science* 2342.

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

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

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

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

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

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

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

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

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

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

Pednault, E. P. D. 1989. 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)*, 324–332.

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

Thakkar, S.; Knoblock, C. A.; Ambite, J. L.; and Shahabi, C. 2002. Dynamically composing web services from online sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, 1–7.

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

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

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

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

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

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