Automatic Refinement of Service Compositions*

Umberto S. Costa[1,**], Mirian Halfeld Ferrari[2],
Martin A. Musicante[1], and Sophie Robert[2]

[1] Universidade Federal do Rio Grande do Norte, DIMAp, Natal, Brazil
{umberto,mam}@dimap.ufrn.br
[2] Université d'Orléans, LIFO, Orléans, France
{mirian,sophie.robert}@univ-orleans.fr

Abstract. We propose a method for the automatic refinement of web service compositions: given a composite web service specification over *abstract* modules, our method generates lower-level versions of this composition. The refinement process is based on query rewriting techniques extended to take into account not only functional and non-functional requirements but also semantic information. Experimental results illustrate the performance and scalability of the method.

Keywords: Web Services, Service Compositions, Automatic Refinement.

1 Introduction

The composition of web services is a central task in Service-Oriented Software Development [1]. This task consists in combining pre-existing services in order to achieve new functionalities. The selection of services is based on the requirements of the compound service as well as on the descriptions of individual services. Services from different providers may not agree on the representation of data or functionality. The successful combination of services depends on the correct matching between their interfaces. In this scenario, the composition designer is in charge of providing mechanisms to find suitable services and to adapt their interfaces as required by the composition. A number of initiatives were proposed to tackle the problem of automatically composing web services. Approaches include the adaptation of techniques from areas such as Databases [2] or AI Planning [3].

In this paper we propose a mechanism for the automatic refinement of web service specifications, using semantic information. The Semantic Web can help to broaden the choice of services. Ontologies [4] may be used to align the representation of concepts, as well as to describe the relationships between services. The developer can describe a compound application in terms of semantic descriptions (*abstract services*). Each abstract service may correspond to one or

* This work was partly supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq (Brazil) 573964/2008-4; CAPES/UdelaR (Brazil) 021/2010; CAPES/STIC-AmSud (Brazil) 020/2010; ANR project ExaviZ.

** Bolsista da CAPES - Brasília/Brasil.

F. Daniel, P. Dolog, and Q. Li (Eds.): ICWE 2013, LNCS 7977, pp. 400–407, 2013.

more concrete services, as published by individual providers. We assume that the construction of a composition of concrete services is based on a software development process formed by *Specification*, *Refinement*, *Evaluation* and *Coding* steps. This paper focuses on the *Refinement* step and presents an algorithm to automatically refine high-level specifications of service compositions into lower-level ones. Our method is based on the MiniCon algorithm [5] for query rewriting, known in the database domain. We begin with a higher-level composition specification expressed over abstract services and quality constraints. Our approach generates several translations of this specification into compositions over concrete services. The solutions produced will be ranked (*Evaluation*) and coded into concrete orchestrations. These two steps are beyond the scope of our work.

This paper is organised as follows: Section 2 describes our method; Section 3 presents some experiments; Section 4 concludes the paper.

2 Rewriting Compositions

Our algorithm for refining specifications of abstract compositions is structured in two main phases. In the first one, each concrete service definition is scanned in order to identify what parts of the specification it covers. The second phase of our algorithm combines concrete services, in order to cover the whole specification.

Both the abstract composition and concrete services are defined in the same way, by using the syntax: $C(\bar{t}) \equiv_{def} A_1(\bar{t_1}), \ldots, A_n(\bar{t_n}), Q_1(\bar{t'_1}), \ldots, Q_m(\bar{t'_m})$. The elements of the tuple $\bar{t}$ on the left-hand side of a definition are the formal parameters. These parameters represent input (marked with "?") or output (marked with "!") data. The right-hand side of the definition consists of abstract service calls and quality constraints. The same decorations are used for the parameters of these items. Additionally, optional parameters of abstract services inside the definition of concrete services are marked with "$*$". Quality constraints $Q_i(\bar{t})$ are of the form $(X\ op\ Y)$, $(X\ op\ a)$ or $(X \in C)$ where X and Y are variables, a is constant, $op \in \{<, >, \leq, \geq, =\}$ and C is a set of constants.

The first phase of our method consists in matching the specification of each concrete service with parts of the abstract composition. Each concrete service may be used to implement parts of the composition. Given the abstract composition $C(\ldots) \equiv_{def} A_1(\ldots), \ldots, A_n(\ldots), Q_1(\ldots), \ldots, Q_m(\ldots)$ and each concrete service $S_i(\ldots) \equiv_{def} A_i(\ldots), \ldots, A_j(\ldots), Q_k(\ldots), \ldots, Q_l(\ldots)$, our algorithm tries to match some abstract services on the right-hand side of the definition of S_i with the same services on the right-hand side of the definition of C. This matching consists in a (semantic) mapping to make their parameters compatible. For each possible matching, a tuple containing the mapping information is produced. Each of these tuples is called a PCD (*Partial Coverage Descriptor*).

A Partial Coverage Descriptor D for a concrete service S and a composition C is a tuple $\langle S, h, \varphi, G, Def, has\_opt \rangle$, where:

- S is the name of the concrete service involved in the matching.
- φ is a partial mapping from $Terms(C)$ to $h(Terms(S))$. This mapping defines the correspondence between the terms appearing on the abstract composition and terms that appear on the concrete service definition.

- h is a mapping from $Terms(S)$ to $Terms(S)$. For every term x that is not a parameter of S, $h(x) = x$. For terms x and y that are parameters of S, h may be such that $h(x) = h(y)$, where for every parameter x we have that $h(x) = h(h(x))$. This mapping is the head homomorphism in [5].
- G is the set of abstract service names and quality constraints covered by S.
- Def is a set of quality constraints of the abstract composition. Intuitively, this set will contain those conditions that cannot be guaranteed by S alone.
- has_opt is a boolean flag used to indicate that some abstract service in the definition of S has been used in G and has an optional parameter. □

Roughly speaking, a PCD D indicates *(i)* which part of the abstract composition is covered by a concrete service S and *(ii)* how to relate the data processed by the composition with the parameters of the concrete service.

Example 1. Let $C(x?, y!) \equiv_{def} A_1(x?, x?), A_2(x?, y!)$ be an abstract composition. Let $S(a?, b?) \equiv_{def} A_1(a?, b?), A_3(a?)$ be the specification of a concrete service. Let us consider the abstract service call $A_1(x?, x?)$ in C. We can use the definition of S to cover part of the composition. Indeed, it is possible to obtain the PCD $D = \langle S, \varphi, h, \{A_1\}, \emptyset, \textbf{false}\rangle$ where $h(a) = a$, $h(b) = a$ and $\varphi(x) = h(a)$. □

The algorithm below builds a set of PCDs, given an abstract composition C and a set of concrete service specifications $\mathcal{S}$.

Algorithm 1. (Build PCDs)

```
procedure build PCDs(C, S)                                                    1
  PCDs := ∅;                                                                  2
  for each abstract service A in the definition of C do                       3
    for each concrete service S∈ S do                                         4
      if there are mappings h and φ for A in the definitions of C and S then  5
        G := {A};                                                             6
        Def := ∅;                                                             7
        PCD := ⟨ S, h, φ, G, Def, has_opt ⟩;                                  8
        AS := {A' | A' is an abstract service or quality constraints in C sharing  9
              parameters with A or with other elements of AS}                 10
        PCD_OK := true;                                                       11
        while AS≠ ∅ and PCD_OK do                                             12
          A' := choose an abstract service from AS;                           13
          if h, φ can be extended to cover A' then                            14
            Update PCD w.r.t. h, φ, G, Def, has_opt                           15
            AS := AS − A';                                                    16
          else PCD_OK := false;                                               17
        if PCD_OK then PCDs := PCDs ∪ PCD;                                    18
```

In the mappings for the abstract service A (line 5), parameters appearing on the left-hand side of C should only be mapped to parameters appearing on the left-hand side of concrete service definitions or optional ones. Then, Algorithm 1 looks for other abstract services or quality constraints connected to A. The set AS contains all abstract services or quality constraints of C that *(i)* have a data dependency to A and *(ii)* are not mapped by φ to parameters of S (line 9).

Example 2. Let $C(y!) \equiv_{def} A_1(x?, y!), A_2(x!), x \geq 10, y \in \{5, 4, 3\}$ be an abstract composition. Let us suppose $S(b!) \equiv_{def} A_1(a?, b!), A_2(a!), a = 10$. We will obtain a PCD covering not only the abstract service call $A_1(x?, y!)$, but, due to the mapping of x (on the composition) to a on S (*i.e.*, $\varphi(x) = a$), the PCD must also cover the abstract service call $A_2(x!)$ and the condition $x \geq 10$. This matching is possible because service S may cover A_2 and specifies that $a = 10$. □

One important difference between our algorithm and MiniCon [5], is that our method supports the notion of optional parameters in the specification of a concrete services, *i.e.*, parameters that can be ignored. The information about optional parameters is supposed to be provided by the vendor of the service as part of its specification. This situation is described in the next example.

Example 3. Let $C(u?, x!) \equiv_{def} A_1(u?, v!, w!), A_2(v?, w?, x!)$ be an abstract composition and $S(a?, c!) \equiv_{def} A_1(a?, *b!, c!)$ a concrete service specification where b is an optional parameter. Algorithm 1 builds the PCD $D = \langle S, h, \varphi, \{A_1\}, \emptyset, \mathbf{true} \rangle$ where h is the identity function; $\varphi(u) = a$, $\varphi(v) = b$, $\varphi(w) = c$. There are two data dependencies between A_1 and A_2, given by the parameters v and w on both service calls. None of these data dependencies is taken into account when the set AS is built at line 9 of Algorithm 1: *(i)* the variable v is mapped to the optional parameter b on the specification of S and *(ii)* the variable w is mapped by φ to c, which is a parameter of S. Notice that this PCD is marked as having optional parameters (last component of the tuple is **true**). This information will be used in the algorithm of the second phase to restrict combinations of PCDs. □

Our second phase algorithm combines PCDs to produce compositions over concrete services. To this end, it takes the set of PCDs produced by Algorithm 1 and looks for combinations of these PCDs to cover the right-hand side of the abstract service composition C. This procedure is described by Algorithm 2.

Algorithm 2. (Combine PCDs)

procedure Combine PCDs(C, PCDs) 1
  Given $C = C(\bar{t}) \equiv_{def} A_1(\dots), \dots, A_n(\dots), Q_1(\dots), \dots, Q_m(\dots)$ and 2
    PCDs $= \{\dots, \langle S_i, h_i, \varphi_i, G_i, \text{Def}_i, \text{has\_opt}_i \rangle, \dots\}$; 3
  for each combination $\{\text{PCD}_1, \dots, \text{PCD}_k\} \subseteq$ PCDs such that 4
   (a) $\{A_1(\dots), \dots, A_n(\dots)\} \subseteq G_1 \cup \dots \cup G_k$; 5
   (b) $\forall\, i, j \;.\; G_i \cap G_j \subseteq \text{Def}_i \cap \text{Def}_j$; 6
   (c) All deferred constraints in $\text{Def}_1 \dots \text{Def}_k$ hold; 7
   (d) Input and output optional parameters should match. 8
  do $Pre := \emptyset$; $Pos := \emptyset$; 9
    for each variable $x \in Q_i$ such that $Q_i \notin G_1 \cup \dots \cup G_k$ **do** 10
      if x is an input parameter of C **then** $Pre := Pre \cup Q_i$ **end if**; 11
      if x is an output parameter of C **then** $Pos := Pos \cup Q_i$ **end if**; 12
    publish $\langle\ Pre\ \rangle\ C'(EC(\bar{t})) \equiv_{def} S_1(\bar{t_1}), \dots, S_k(\bar{t_k})\ \langle\ Pos\ \rangle$; 13

Algorithm 2 tries to cover the definition of the abstract composition C by searching all subsets of PCDs such that: *(a)* they cover all the abstract services

$A_1, \ldots, A_n$ of C (line 5); *(b)* there is no overlapping of the abstract services covered by these PCDs, except for deferred quality constraints (line 6); *(c)* the deferred quality constraints of the PCDs must hold when their variables are instantiated using the mappings of the PCDs (line 7); *(d)* each term in C mapped to an optional output parameter (in the definition of S_i) can only be mapped to optional input parameters (in the definition of any concrete service) (line 8).

For each combination of PCDs satisfying the conditions above, one concrete composition is produced. The refined composition is published in line 13, with its pre- and post-conditions. These conditions are properties of the abstract composition that cannot be statically verified. Each concrete composition $C'(EC(\bar{t})) \equiv_{def} S_1(\bar{t_1}), \ldots, S_k(\bar{t_k})$ has a parameter tuple obtained by applying the function $EC(\bar{t})$ to the parameters of the abstract composition. This function expresses an equivalence class of parameters. The function $EC(\bar{t})$ permits to equate parameters that are different on the abstract composition but that are mapped to the same term on a concrete service as shown in Example 4.

Example 4. Let $C(x?, y?, z!) \equiv_{def} A_1(x?, y?, w!), A_2(w?, z!)$ be an abstract composition, $S_1(a?, r!) \equiv_{def} A_1(a?, a?, r!)$ and $S_2(c?, d!) \equiv_{def} A_2(c?, d!)$ be the specifications of concrete services. Algorithm 1 builds the following PCDs: $D_1 = \langle S_1, h_1, \varphi_1, \{A_1\}, \emptyset, \textbf{false}\rangle$, where h_1 is the identity, $\varphi_1(x) = a$, $\varphi_1(y) = a$ and $\varphi_1(w) = r$; $D_2 = \langle S_2, h_2, \varphi_2, \{A_2\}, \emptyset, \textbf{false}\rangle$ where h_2 is the identity, $\varphi_2(w) = c$ and $\varphi_2(z) = d$. In D_1, both x and y are mapped by φ_1 to a and thus define the equivalence class $\{x, y\}$. So, x and y correspond to the same parameter. Each occurrence of a must be replaced with the representative term of the equivalence class $\{x, y\}$. Thus, we can generate the concrete composition C' by using the terms in $EC(\langle x, y, z\rangle)$, as follows: $C'(x?, x?, z!) \equiv_{def} S_1(x?, w!), S_2(w?, z!)$. □

The parameters of $S_1(\bar{t_1}), \ldots, S_k(\bar{t_k})$ in the concrete composition (Algorithm 2, line 13) are represented by the tuples $\bar{t_i}$. The terms in these tuples are obtained as $\bar{t_i} = f_i^{-1} \circ EC \circ \psi_i \circ h_i(\bar{t'_i})$, such that: *(i)* $\bar{t'_i}$ are the parameters of S_i; *(ii)* the mappings ψ_i rename the variables of the service S_i into the corresponding variables of the abstract composition; and *(iii)* the conversion functions f_i are provided by a set of ontologies. For each $t'_j \in h_i(\bar{t'})$, $\psi_i(t'_j) = t_j$, if $\varphi_i(t_j) = f_i \circ h_i(t'_j)$, and t'_j otherwise. As usual, conversion functions are bijective. In the case of the same representation of data, conversion functions are the identity.

It can be proved that the concrete service compositions produced by the combination of Algorithms 1 and 2 meet the requirements of the abstract composition, in functional terms. This is described by the following property:

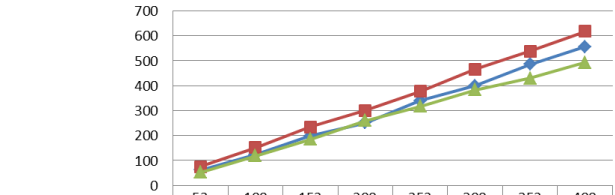
Property 1 (Correctness). Given an abstract composition C, for each concrete composition C' obtained by our algorithm, the following property holds: $\forall\ \bar{t}, \bar{t'}\ .\ C'(\bar{t}?, \bar{t'}!) \Rightarrow C(\bar{t}?, \bar{t'}!)$. □

Property 1 ensures that the solutions obtained by our method are functionally correct. Notice that the functionality implemented by the refined compositions may not cover all the cases considered by the abstract composition. The compositions obtained by our method depend on the available concrete services. The available services may not match all the cases of the specification.

3 Experiments

We have implemented a prototype of our method in Java on the basis of the MiniCon program. In the second phase of our method, all combinations of PCDs are considered, which implies an exponential time complexity (in the number of PCDs generated by the first phase of the method)[1]. This is due to the combinatorial nature of the problem, which is also faced by the MiniCon Algorithm. Figures 1 and 2 show the average time from 10 executions on a Dual Core 2.83GHz processor, 4GB RAM machine running Debian 6.

In Figure 1 we show the runtime for a composition with 10 abstract services and a varying number of concrete services (with two left-hand side parameters) defined by 10 abstract services. In these experiments each concrete service responds to the composition requirements with: (A) no quality constraint; (B) five quality constraints added to each definition; (C) MiniCon without quality constraints but with an optimization procedure. We have used an optimization to avoid the combinatorial explosion of the MiniCon approach, since each service can respond alone to composition requirements. The linear growth shown in Figure 1 is due to this optimization. The overhead introduced by the quality constraints in case (B) varies from 11% to 23% when compared to case (A).



	52	100	152	200	252	300	352	400
(A) 10 SGoals, 0 QLY	61	123	199	251	339	400	485	556
(B) 10 SGoals, 5 QLY	75	151	235	300	377	466	538	618
(C) 10 SGoals, 0 QLY	53	118	185	260	317	383	430	493

Fig. 1. # Services × Time (ms)

In Figure 2, we show the runtime for an abstract composition formed by six abstract services and one quality constraint. The number of concrete services taken into account varies from 96 to 228. This is shown on the X-axis. For each number of concrete services, we varied the proportion of them that satisfies the quality constraints of the abstract composition. Percentages range from 0% to 100%. This is shown on the Z-axis of the picture. The Y-axis of the picture corresponds to the average execution time of the program.

We observe that for a reduced percentage of services that complies with the quality constraint, the first phase of the algorithm will produce a reduced number

[1] The first phase of our method is $O(m.n^2)$, where m is the number of concrete services and n is the number of abstract services invoked by the abstract composition and concrete service definitions.

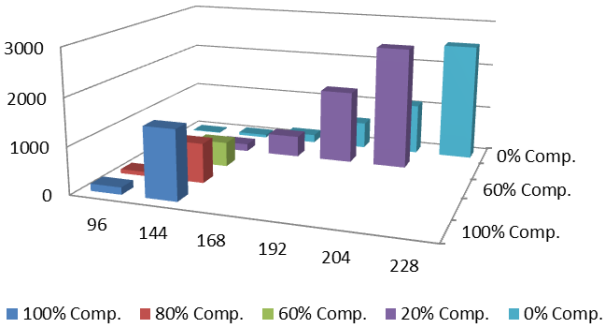


Fig. 2. # Services × Qly Compliance × Time (ms)

of PCDs, allowing the second phase to work with a fewer combinations. As the number of services that meet the quality restriction increases, the second phase of the algorithm shows its combinatorial nature, making it difficult to deal with more than about 150 concrete services. We should notice that on usual situations, the number of available concrete services is not expected to be that many. According to these preliminary experiments, our approach is feasible for problems with up to almost two hundred concrete services (depending on the proportion of quality constraints met by the concrete services).

4 Final Remarks

Selecting and composing services is not a new problem [6,7,8,9]: Some authors [10,11] consider an automatic selection of services, from the semantic point of view. To others, web service compositions are obtained as refinements of more abstract specifications [8,12]. Query Rewriting techniques [13,2,5] have been considered for generating compositions from abstract specifications. Recently, researchers have started to apply this technique in the context of web semantics and web service composition [7,8,9]. Our work is inserted in this context, where we use non-functional properties for fine-tuning the selection of services.

Our work adapts and extends the query rewriting method MiniCon to service composition. In this new context, it is important to remark that the definition of a composition or a service is not seen as a database query and, thus, is not imposed to the same restrictions. Besides this adaptation (that, for instance, makes useless the notion of safe rules required in [5]), the original method has been expanded to deal with optional parameters and quality constraints.

The advantages of our approach are significant: it eases the user's work, deferring technical details to further steps; takes into account both functional and non-functional requirements; offers different solutions that can be used latter when dealing with service evolution in runtime; allows the use of domain ontology information to perform data transformations (*i.e.*, in practice, our algorithm

is capable of automatically performing data conversions in order to use services whose parameters do not match exactly).

Experiments using our prototype implementation show that our approach is feasible on real-life applications, where distinct concrete services rarely respond to the same non-functional requirements (restricting the possible choices during the rewriting). As a future direction, we are aware of the need of establishing theoretical properties of our approach. We are currently working on the classification of solutions according to an user profile.

Acknowledgements. Special thanks to Prof. R. Pottinger, who kindly made available the code of MiniCon, and to S. Munier for implementing a prototype of our method.

References

1. Marks, E., Bell, M.: Service-Oriented Architecture: A Planning and Implementation Guide for Business and Technology. Wiley (2006)
2. Levy, A.Y.: Logic-Based Techniques in Data Integration. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 575–595. Kluwer, USA (2000)
3. Rao, J., Su, X.: A survey of automated web service composition methods. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 43–54. Springer, Heidelberg (2005)
4. Dobson, G., Sanchez-Macian, A.: Towards Unified QoS/SLA Ontologies. In: Proceedings of the IEEE Services Computing Workshops, SCW 2006, pp. 169–174. IEEE Computer Society, Washington, DC (2006)
5. Pottinger, R., Halevy, A.Y.: Minicon: A scalable algorithm for answering queries using views. VLDB J. 10(2-3), 182–198 (2001)
6. Alrifai, M., Risse, T.: Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In: Proc. of the 18th International Conference on World Wide Web, WWW 2009, pp. 881–890. ACM, New York (2009)
7. Barhamgi, M., Benslimane, D., Medjahed, B.: A query rewriting approach for web service composition. IEEE Trans. Serv. Comput. 3(3), 206–222 (2010)
8. Thakkar, S., Ambite, J.L., Knoblock, C.A.: A data integration approach to automatically composing and optimizing web services. In: Proc. of the ICAPS Workshop on Planning and Scheduling for Web and Grid Services (2004)
9. Zhao, W., Liu, C., Chen, J.: Automatic composition of information-providing web services based on query rewriting. Science China Information Sciences, 1–17 (2011)
10. Berardi, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., Mecella, M.: Automatic composition of e-services. Technical Report 22-2003, Dipartimento di Informatica e Sistemistica, Universita di Roma La Sapienza, Roma, Italy (2003)
11. Izquierdo, D., Vidal, M.-E., Bonet, B.: An expressive and efficient solution to the service selection problem. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 386–401. Springer, Heidelberg (2010)
12. Mesmoudi, A., Mrissa, M., Hacid, M.S.: Combining configuration and query rewriting for Web service composition. Technical Report RR-LIRIS-2010-015, LIRIS UMR 5205 CNRS/INSA Lyon/U. Lyon 1/U. Lyon 2/EC Lyon (July 2010)
13. Duschka, O.M.: Query Planning and Optimization in Information Integration. PhD thesis, Department of Computer Science, Stanford University (December 1997)