Preference-Driven Refinement of Service Compositions*

Cheikh Ba[1,3], Umberto Costa[2], Mirian Halfeld-Ferrari[3], Rémy Ferre[3], Martin A. Musicante[2], Veronika Peralta[4] and Sophie Robert[3]

[1]*Université Gaston Berger, Saint Louis, Sénégal*

[2]*DIMAp, Universidade Federal do Rio Grande do Norte, Natal, Brazil*

[3]*Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, FR-45067 Orléans, France*

[4]*Université François Rabelais Tours, LI, Blois, France*

cheikh.ba.sn@gmail.com, {mirian,sophie.robert}@univ-orleans.fr, remy-ferre@laposte.net, {mam,umberto}@dimap.ufrn.br, veronika.peralta@univ-tours.fr

Keywords: Service Compositions, abstract services, refinement, user preferences

Abstract: The Service Oriented Computing Paradigm proposes the construction of applications by integrating pre-existent services. Since a large number of services may be available in the Cloud, the selection of services is a crucial task in the definition of a composition. The selected services should attend the requirements of the compound application, by considering both functional and non-functional requirements (including quality and preference constraints). As the number of available services increases, the automation of the selection task becomes desirable. We propose a method for the refinement of service compositions that takes the abstract specification of a composition, the definition of concrete services and user preferences. Our algorithm produces a list of refinements in preference order. Experiments show that our method can be used in practice.

1 INTRODUCTION

Cloud computing provides a cheaper, reliable infrastructure in which service providers may deploy applications or store data to be used by their customers. Access to the resources is granted to users by offering a client-server infrastructure that can be used to perform tasks at different levels of abstraction. Cloud infrastructure includes models at three levels of abstraction : Cloud Infrastructure as a Service (*IaaS*), Cloud Platform as a Service (*PaaS*) and Cloud Software as a Service (*SaaS*). The three models of Cloud computing propose the use of *services* to access resources. Services may be used to query data or to perform computations.

The long-term goal of our work is the construction of a platform (*PaaS*) to provide a high-level, abstract way of specifying service compositions. The compositions would be obtained according to the software development process composed by *Specification*, *Refinement based on user preferences* and *Coding*.

*This work was partly supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq (Brazil) 573964/2008-4 and 305619/2012-8; CAPES/UdelaR (Brazil) 021/10; CAPES/STIC-AmSud (Brazil) 052/14.

In this paper, we only deal with the refinement step: The specification of a compound service is defined in terms of *abstract* services (*i.e.*, specifications of basic service capabilities). Abstract services may be combined to specify the functional and non-functional behaviour of the composition. This abstract composition will be refined into a composition of concrete, available services. The refinement algorithm presented here builds new compositions by combining concrete services, in order to match the specification. Concrete services are available in the Cloud. We suppose that the capabilities of each concrete service are also specified in terms of abstract building blocks, similar to those used in the abstract composition. The choice among different possible compositions is guided by user preferences.

In (Costa et al., 2013), we have outlined a refinement technique that obtains all the possible combinations of concrete services to match the specification. This leads to an algorithm with NP time complexity. In this paper we use the notion of *user preferences* to set conditions to favour the use of certain services. The search space (combinations of actual services in the Cloud) may be ordered in accordance to these preferences. This order may be used to produce the solutions (concrete compositions) in an incremental

way, so that the refinement process can stop when the desired number of compositions is reached.
Paper Organisation: Section 2 presents related work. Section 3 discusses how to use preferences to select services. Section 4 presents our method. A prototype is shown in Section 5. Section 6 concludes the paper.

2 RELATED WORK

In (Costa et al., 2013), we presented a method (referred here as Refinement1) for the automatic refinement of web service compositions, based on MiniCon algorithm (Pottinger and Halevy, 2001). This method takes into account both functional and non-functional requirements, as well as semantic information. User preferences are not taken into account and no ranking is provided for the produced refinements.

Several works (Zhao et al., 2011; Mesmoudi et al., 2011; Paiva Tizzo et al., 2011) are close to Refinement1. In (Zhao et al., 2011) the authors use the original MiniCon algorithm to rewrite *information-providing* compositions in terms of available services. The main difference between their approach and ours is that we have substantially modified the original MiniCon algorithm to support ontology alignment, optional parameters, quality constraints and conversion functions. In (Mesmoudi et al., 2011) the authors propose a method of service discovery that builds buckets of relevant services for each sub-goal in a composition, by taking into account the semantics description of services. Like us, they distinguish between the phase of refinement (or discovery) and a latter step of orchestration. In (Paiva Tizzo et al., 2011) the authors propose a model to improve the dynamic selection of services in the context of WS-BPEL processes. Service compositions are described as SPARQL queries; those queries are evaluated at runtime by a SPARQL server to produce service endpoints; those endpoints are used to invoke the selected services. The proposed approach is semi-automatic and does not take user profiles or QoS into account.

Recently, the QoS-based web service selection and composition in service oriented applications has gained the attention of many researchers (Alrifai et al., 2008; Alrifai et al., 2010; Sandionigi et al., 2013; Zeng et al., 2003). These papers focus on *how to select services* on the basis of QoS, *i.e.*, in solving a multi-criteria decision making problem which corresponds to the identification of the best candidate web services from a set of functionally equivalent services. For instance, (Alrifai et al., 2008) models the problem as the Multi-Choice Multidimensional Knapsack problem and proposes a scalable QoS computation approach based on a heuristic algorithm, which decomposes the optimization problem into small subproblems. Their heuristic is a hybrid approach that combines global optimization with local selection in order to find a close-to-optimal selection more efficiently. Improvements of this method are presented in (Alrifai et al., 2010). Some approaches are very specific to the targeted application. For example, in (Sandionigi et al., 2013) the authors are capable to express the service selection in situational computing applications as a Mixed Integer Linear Programming problem from decision variables expressing binding and *ad hoc* QoS constraints. Contrary to them, our goal is to show how *to use* the result of a QoS-based service selection *to decrease the search space* while looking for a concrete instantiation of an abstract composition. Moreover, several QoS aspects may be summarized in a given user preference.

Qualitative preference models, including conditional and temporal clauses, are used in (Lin et al., 2008; Sohrabi and McIlraith, 2010) in order to model expressive service goals that should be satisfied as much as possible. However, most works rely on quantitative preference models, expressing a utility function (generally a weighted sum that combines multi-criteria preferences). For example, given a set of concrete compositions, (Haddad et al., 2010) recommends the compositions that maximize a utility function. As another approach, (Lamparter et al., 2007) uses utility policies and linear programming for selecting compositions. These proposals use user preferences for choosing the best compositions among a set of pre-generated compositions. Our goal is to use user preferences for guiding and optimizing the generation process. The work of (Lemos et al., 2012) use PreferenceSQL constructors for expressing hard and soft preferences ; the former are used to prune the set of candidate services while the latter are used to order them (they do not address service composition). However, preferences are used after performing a structural matching between a requested service and each candidate service, while we aim at using preferences for optimizing the process.

3 BACKGROUND

Abstract compositions are specified by equations, in accordance to the following:

Definition 1 (Abstract Composition) *Let*

$$C(\bar{t}) \equiv_{def} A_1(\bar{t_1}),\ldots,A_n(\bar{t_n}),Q_1(\bar{t_1'}),\ldots,Q_m(\bar{t_m'}). \quad (1)$$

be the definition of an abstract composition. The left-hand side of the specification defines the interface of

the composition. The elements of the tuple $\bar{t}$ are formal parameters and represent input and output data. We use the ? and ! markings to indicate input and output parameters, respectively.

The right-hand side of the definition consists of abstract services $(A_1,\ldots,A_n)$, and constraints $(Q_1,\ldots,Q_m)$, expressing capabilities of the composition. Abstract services correspond to semantic descriptions of service functionalities and specifies the relationship between their required inputs and expected outputs. Ontologies may be used to align the representation of such concepts and to describe the relationships between services. □

Abstract services are the building blocks used to specify the composition to be refined. Constraints are relational expressions built on variables and constants. They can express static conditions (to be verified during the refinement process) and dynamic conditions (to be verified dynamically by the refined composition). The composition designer specifies an order of importance between abstract services.

Example 1 *Let us suppose the specification of a composite service for buying books over the Internet:*

$$\begin{aligned} &LookForBooks(Uid?,Pwd?,Isbn?,Loc?,Price!,Addr!,Ack!)\\ &\equiv Authentication(Uid?,Pwd?,Tkn!,Prot!,Form!),\\ &\quad BookStore(Tkn?,Isbn?,Loc?,Addr!,Price!,Invoice!),\\ &\quad Payment(Tkn?,Invoice?,Ack!),\\ &\quad Prot = \text{“}REST\text{”}, Ack = \text{“}OK\text{”}. \end{aligned}$$

□

The client supplies an identification, password, ISBN and location and expects to receive the book's price, a pickup address and the transaction acknowledgement. The composition begins with the authentication of the client. This step returns a token to identify the client in the store, as well as the exchange protocol and format. The BookStore service also uses the client's location to return the nearest pickup address for the book, as well as the price and the invoice, to process the payment. After looking up the book's price, the bill is paid by using the credit card information.There are two constraints in the composition. The first one is a static constraint that specifies the protocol to be used for authentication. The second condition establishes that the whole process was successfully finished (to be verified dynamically).

Concrete services are also specified by using abstract services and constraints:

Definition 2 (Concrete Services) *Let*

$$S(\bar{t}) \equiv_{def} A_1(\bar{t_1}),\ldots,A_k(\bar{t_k}),Q_1(\bar{t'_1}),\ldots,Q_r(\bar{t'_r}). \quad (2)$$

be the definition of a concrete service. As in the case of Definition 1, the left-hand side of the specification defines the interface of the service. The elements of the tuple $\bar{t}$ are formal parameters and represent input and output data. We use the ? and ! markings to indicate input and output parameters, respectively.

The right-hand side of the definition consists of abstract services $(A_1,\ldots,A_k)$, and constraints $(Q_1,\ldots,Q_r)$, expressing capabilities of the service. □

In this case, the left-hand side of the definition gives the name and interface of the concrete service. The right-hand side uses abstract services and constraints to express the capabilities of the service. The semantic information embedded in the abstract services can help to broaden the number of available services for the refinement process. We suppose that the specification of each concrete service is given by the service supplier/publisher.

Our refinement method matches the specification of concrete services with that of the abstract composition. The algorithm considers the definition of each available concrete service to cover parts of the abstract composition. These matchings will be used later by the algorithm, to build a concrete solution (refinement) of the specification.

The refinement process uses preference information to guide the traversal of the search space of concrete services. This traversal should be done in an order determined by the user preferences. In this way, the refinement algorithm should be capable of proposing concrete compositions that comply with the composition specification and that are presented in decreasing order of preferences. We associate a score to each concrete service available to the composition developer. Services having a greater score are preferred to the ones having lower scores.

Definition 3 (User Preferences) *Given a set $\mathcal{S}$ of concrete services and a real-valued scoring function function $u : \mathcal{S} \mapsto [0,1]$, a preference $P = (\mathcal{S}, <_P)$ is derived from the scoring function, where for two concrete services $x,y \in \mathcal{S}$, $x <_P y$ iff $u(x) < u(y)$.* □

The above definition is interpreted as “*I like y better than x if it has a better score*”. Notice that scoring functions can represent very different types of preferences, ranging from simple scores (e.g.based on response time) to complex, multi-criteria expressions that combine various quality perspectives. These scores may be manually defined by users or communities, may be deduced or estimated from user activity, or may be automatically assessed. Our approach is independent of how the scoring function is defined.

The notion of preference classifies the available concrete services in accordance to the user's point of view. However, this notion is orthogonal to the semantics of each concrete service, since the definition of preferences does not need to take into account the functionality of concrete services.

We aim to use preference information to produce compositions that maximize the combined weight of its component services. In order to adapt this information to the context of our refinement procedure, we introduce the notion of *Coverage Domain*.

Definition 4 (Coverage Domain) *For each abstract service A_i of an abstract composition we define the Coverage Domain $\mathcal{A}_i$, such that: (1) $\forall S \in \mathcal{S}$ s.t. $S \in \mathcal{A}_i$, S contains the abstract service A_i in its definition. (2) For each $\mathcal{A}_i$, there is a threshold ξ_i, defined at the same time as the abstract composition. Only services S such that $u(S) \geq \xi_i$ will be included in $\mathcal{A}_i$.* □

Notice that, since each abstract service A_i is a semantic annotation that denotes that S provides a defined functionality, the coverage domain $\mathcal{A}_i$ represents the set of services that can perform such a provision. Also, according to Definition 4, an empty coverage domain $\mathcal{A}_i$ indicates that no concrete service in $\mathcal{S}$ will provide the functionality described by A_i. In this case, no refinement can be produced over $\mathcal{S}$.

4 REFINEMENT PROCESS

High-level compositions are specified in terms of abstract services, constraints and a total order over the coverage domains. Given this high-level composition and the specification of the available concrete services, our approach generates refined compositions, built upon concrete services. The refinement process incrementally generates concrete compositions, in accordance to the user's preferences.

Algorithm 1 traverses a search space formed by combinations of concrete services, in order to build a list of concrete compositions that refine the abstract composition specification.

Algorithm 1 (Refinement Algorithm)

```
procedure refinement (C, A)                                   1
  I:= BuildServiceOrganiser(C, A);                            2
  BuildStorePCD(C, A, I);                                     3
  while number of desired solutions is not reached do         4
    for each P := GetNextSetOfPCD(I); do                      5
        R := ProduceRewriting(C, P);                          6
        Publish(R); //R may be empty.                         7
    end do                                                    8
  end while                                                   9
end procedure                                                 10
```

Given the abstract composition C, and the coverage domains $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$, built from user's preferences as defined in Section 3, we suppose that the order between coverage domains is defined by the indices ($1 \leq i \leq n$) in the right-hand side of the specification of C. The procedure *BuildServiceOrganiser* initialises an index-like data structure to store the concrete services in the lexicographical order (based on the score of concrete services and the coverage domain order). For each concrete service, the procedure *BuildStorePCD* produces mappings (called PCD - Partial Coverage Descriptors) to cover parts of the abstract specification. A PCD describes *(i)* which part of the abstract composition is covered by a concrete service and *(ii)* how to relate the data processed by the composition with the parameters of this concrete service.

The procedure *BuildStorePCD* associates each PCD to a concrete service and stores them for future use. Algorithm 1 implements an iterator which generates sets of PCD by traversing the index I, in decreasing lexicographical order, according to the user's preferences. The preference of a PCD is that of the concrete service to which it is associated. Notice that the search space is segmented to be processed stepwise. At each step, the iterator computes sets $\mathcal{P}$ of PCD such that: (i) they have the same rank on the lexicographical order and (ii) for each $\mathcal{P}$ there is only one PCD from each coverage domain. In line 5, function GetNextSetOfPCD returns one of these sets $\mathcal{P}$, which, in turn will be passed to the last phase of the refinement method. In line 6 the combination of the PCD in $\mathcal{P}$ is verified and possibly published.

The rationale for the search space segmentation is to reduce the number of produced combinations. Indeed, Refinement1 generates all possible rewritings by testing all possible PCD's combinations. In this new approach the algorithm may exit before exhausting all the possibilities, emphasizing our goal of avoiding the cost of a combinatorial explosion.

The following sub-sections detail our proposal.

4.1 Partial Coverage Descriptors

Given the abstract composition defined as
$C(\ldots) \equiv_{def} A_1(\ldots), \ldots, A_n(\ldots), Q_1(\ldots), \ldots, Q_m(\ldots)$.
and each concrete service S, defined as
$S(\ldots) \equiv_{def} A_i(\ldots), \ldots, A_j(\ldots), Q_k(\ldots), \ldots, Q_l(\ldots)$.
our algorithm tries to match some services on the right-hand side of the definition of each concrete service with the same abstract services appearing on the right-hand side of C. This matching consists in producing a (semantic) mapping to make their parameters compatible.

For each possible matching, a tuple containing the mapping information will be produced. Each of these tuples is a PCD. The definition of PCD is based on the MiniCon Descriptions in (Pottinger and Halevy, 2001). It was adapted in (Costa et al., 2013) to the context of web services. Intuitively, a PCD

$\langle S, h, \varphi, G, Def, has\_opt \rangle$ defines how a service can be part of a concrete composition that meets the abstract specification. S is the name of the concrete service involved in the matching. A mapping φ defines the correspondence between the terms appearing on the abstract composition and terms that appear on the concrete service definition. For example, given $C(\ldots) \equiv_{def} A_1(\ldots), \ldots, A_i(x,y), \ldots, A_n(\ldots), Q_1(\ldots), \ldots, Q_m(\ldots)$ and $S(\ldots) \equiv_{def} \ldots, A_i(u,v), \ldots$ we have that $\varphi(x) = u$ and $\varphi(y) = v$. The mapping h corresponds to the head homomorphism in (Pottinger and Halevy, 2001). For example, given $C(\ldots) \equiv_{def} A_1(\ldots), \ldots, A_i(x,x), \ldots, A_n(\ldots)$ and $S(u,v) \equiv_{def} \ldots, A_i(u,v), \ldots$ we need to equate u and v (*i.e.*, $h(u) = u$, $h(v) = u$), so that φ may be defined as a function (*i.e.*, $\varphi(x) = u$). G is a set of abstract service names and quality constraints covered by S. The set Def contains conditions that cannot be guaranteed by S alone. The flag has_opt indicates the use of an optional parameter.

Given an abstract composition C, the algorithm below builds PCD and stores each one in the index I.

Algorithm 2 (Building and storing PCD)

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

In line 5, parameters appearing on the left-hand side of a composition should only be mapped to parameters appearing on the left-hand side of concrete service specifications or optional ones. Then, Algorithm 2 looks for other abstract services or quality constraints connected to A. Thus, in line 9, 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 the parameters of S.

Example 2 *Let us consider the composition LookForBooks specified in Example 1. Assume the following concrete service specifications:*

PickAndPayBooks(T?, I?, M?, A!, P!, K!) $\equiv_{def}$
BookStore (T?, I?, M?, A!, P!, V!) ,
Payment (T?, V?, M?, K!).
Amazon(T?, I?, M?, A!, P!, V!) $\equiv_{def}$
BookStore(T?, I?, M?, A!, P!, V!)
Visa(T?, V?, M?, K!) $\equiv_{def}$ *Payment(T?, V?, M?, K!).*

The PCD $D_1 = \langle PickAndPayBooks, h_1, \varphi_1, \{BookStore, Payment\}, \emptyset, \textbf{false} \rangle$, *where* h_1 *is the identity function and* φ_1 *maps terms Tkn?, Isbn?, Loc?, Addr!, Price!, Invoice!, Ack! to terms T?, I?, M?, A!, P!, V!, K!, respectively, is built by Algorithm 2 since: (i) the parameters appearing on the left-hand side of the composition are mapped by* φ_1*, and (ii) Invoice is a term that appears only on the right-hand side of the definition of LookForBooks and is mapped by* φ_1 *to V in PickAndPayBooks.*
The inclusion of Payment in D_1 *is explained by the fact that* $\varphi_1(Invoice)$ *is a term which does not appear on the left-hand side of PickAndPayBooks and acts as output parameter of BookStore and input parameter of Payment. Thus,* D_1 *can be used in the rewriting of LookForBooks to cover BookStore and Payment.*

Two other PCD can be built: $D_2 = \langle Amazon, h_2, \varphi_2, \{BookStore\}, \emptyset, \textbf{false} \rangle$ *and* $D_3 = \langle Visa, h_3, \varphi_3, \{Payment\}, \emptyset, \textbf{false} \rangle$ *where* φ_2 *and* φ_3 *are the intuitively expected mappings. Since these two PCD are feasible, then our approach can propose a rewriting of LookForBooks with service Amazon for BookStore and Visa for Payment.* □

Algorithm 2 detects two situations where the construction of PCD is not possible (*i.e.*, when nothing is stored in I): (i) In line 5, if no match between services S and A exists and (ii) in line 15, if no match between services S and A' exists. In the second situation, as $A' \in AS$, S must cover the abstract service A'.

When the construction of a PCD is possible, it is stored in I. For each coverage domain $\mathcal{A}$ of an abstract service $A \in G$, the computed PCD is inserted in I, associated to the concrete service $S \in \mathcal{A}$. After the execution of *Algorithm 2*, the index I contains all possible PCD, computed on the basis of a composition C and organised according to an user's preferences.

4.2 An Iterator

Our method proposes an organiser, an index-like structure, which is built for each abstract composition, according to the user's preferences and the coverage domain order. An iterator traverses this organiser in the decreasing lexicographical order and analyses, progressively, each concrete service combina-

tion susceptible to be a composition rewriting. The iterator avoids building useless compositions by respecting the following rules: (1) A concrete service can take part in a composition iff it does not cover a domain already covered by previous services. (2) A coverage domain is not visited if it has already been covered by a service in the composition.

The iterator generates each set $\mathcal{P}$ of PCD involved in a generated composition. At each iteration, we obtain all the sets $\mathcal{P}$ corresponding to the same preference. The function GetNextSetOfPCD in Algorithm 1 returns one of these sets to be possibly completed by the last step of our method.

Figure 1 illustrates the three-level structure used to implement our index. The first level sets the order of coverage domains. The second level is organised according to the concrete service scores. For instance, in Figure 1, concrete services in $\mathcal{A}_0$ having the highest weight are found in the slot labelled $rank_{0.0}$ while those having the lowest weight are found in the slot labelled $rank_{0.m_0}$. In this figure, we suppose that for domain $\mathcal{A}_0$ we have m_0 ranks, while for $\mathcal{A}_1$ we have m_1 and so on. In the third level we use a hash table to store concrete services. More precisely, a hash function associates a slot to a concrete service and its PCDs are stored in the corresponding linked list.

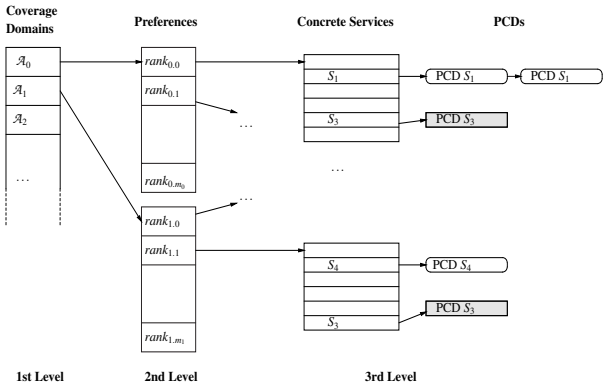

Figure 1: Organiser: to store PCD for concrete services according to the user's preferences.

4.3 Producing Refinements

Algorithm 3 is the last step of our approach. Its role is to verify whether a given set $\mathcal{P}$ of PCD can be correctly combined in order to produce a composition refinement. Algorithm 3 completes the composition according to imposed quality constraints and optional parameters, when they apply.

The input of Algorithm 3 is a set $\mathcal{P}$ containing k partial coverages for n abstract services. The number of PCD in $\mathcal{P}$ does not necessarily equal the number of abstract services, since a concrete service may cover more that one abstract service. The iterator guarantees that $\mathcal{P}$ respects the following properties: (i) PCD in $\mathcal{P}$ cover all the abstract services $A_1,\ldots,A_n$ of C and (ii) there are no overlapping between the abstract services covered by different PCD. The only allowable intersection between them is given by deferred quality constraints.

Algorithm 3 (PCD Combination and Rewriting)

function *ProduceRewriting($C,\mathcal{P}$)* 1
  if $\mathcal{P}=\{PCD_1,\ldots,PCD_k\}$ *is such that* 2
    (a) All deferred constraints in $Def_1\ \ldots\ Def_k$ *hold;* 3
    (b) Input and output optional parameters should match. 4
  then 5
    $Pre := \emptyset$; $Pos := \emptyset$; 6
    for each *var.* $x \in Q_i$ *such that* $Q_i \notin G_1 \cup \cdots \cup G_k$ ***do*** 7
      if *x is an input parameter of C* ***then*** $Pre := Pre \cup Q_i$ 8
      end if; 9
      if *x is an output parameter of C* ***then*** $Pos := Pos \cup Q_i$ 10
      end if; 11
    end for 12
    return $\langle Pre\rangle\ C'(EC(\bar{t})) \equiv_{def} S_1(\bar{t_1}),\ldots,S_k(\bar{t_k})\ \langle\ Pos\ \rangle$; 13
  end if 14
end function. 15

Algorithm 3 tries to cover the definition of the abstract composition C, by verifying whether, for partial coverages in $\mathcal{P}$, the following conditions hold: (a) The deferred quality constraints appearing in the PCDs must hold when their variables are instantiated using the mappings of the PCDs (line 3). (b) Each term in C mapped to an optional output parameter (inside the definition of S_i) can only be mapped to optional input parameters (inside the definition of any concrete service) (line 4).

If the combination of PCD in $\mathcal{P}$ satisfies the conditions above, one concrete composition is produced. The refined composition is returned (line 13) with its pre- and post-conditions. These conditions cannot be statically verified and need to be checked at runtime by the generated concrete composition. They represent quality constraints of the abstract composition that are not ensured by the concrete services (since they have a dynamic nature). An example is the condition $Ack =$ "*OK*" of Example 1.

The following example covers all the aspects of our refinement algorithm.

Example 3 *Given the specification of the composition of Example 1, suppose that we have defined the following coverage domains obtained from the available concrete services and user preferences:*

Bookstore*: Amazon (0.9), BarnesAndNoble (0.85).*

Payment*: Visa (0.4) and MasterCard (0.35).*

Authentication*: OrangeAuth (0.8), YahooAuth (0.7), Twitter (0.65), Facebook (0.6).*

According to Algorithm 1, the first call to the function in line 5, will return a set of PCD built on the concrete services Amazon, Visa and YahooAuth. Notice that, although OrangeAuth is preferred over YahooAuth, the service cannot be used in the composition since it uses the SAML protocol, instead of REST, which is required by the specification. In this way, no PCD is generated on this service. Thus, our procedure generates the following refinement:

$$\begin{aligned} &LookForBooks'(Uid?, Pwd?, Isbn?, Loc?, Price!, Addr!, Ack!) \\ &\quad \equiv_{def} \quad YahooAuth(Uid?, Pwd?, Tkn!, Prot!, Form!), \\ &\qquad\qquad Amazon(Tkn?, Isbn?, Loc?, Addr!, Price!, Invoice!), \\ &\qquad\qquad Visa(Tkn?, Invoice?, Ack!), Ack = \text{“}OK\text{”}. \end{aligned}$$

In the case in which further solutions are sought, our algorithm will continue to produce refinements with less preferred services. □

Notice that Refinement1 modifies MiniCon by imposing more restrictions (and thus avoiding some rewritings). We can see that the solutions obtained by our algorithm are functionally correct. This claim is supported by the results in (Pottinger and Halevy, 2001) and the following facts: (1) Algorithm 2 builds PCD for each concrete service in $S \in \mathcal{A}_i$. Each PCD is designed to cover part of the abstract composition. It matches the parameters of the abstract composition with those of the concrete service and defines substitution functions to maintain that matching during the (future) execution of the service. (2) Algorithm 3 selects those combinations of PCDs that covers the entire abstract composition. The parameter substitutions are combined to keep track of the data exchanged by the concrete services.

5 OUR METHOD IN PRACTICE

Our iterator proposes concrete service combinations, which implies an exponential time complexity (in the number of PCDs generated by BuildStorePCD in Algorithm 1). This is due to the combinatorial nature of the problem, also faced by the original MiniCon and our refinement method. As, in the worst case, one can have $(n.M.m)$ PCDs, the complexity is $O(n.M.m)^n$ where n is the number of abstract services in the composition, m is the maximal number of abstract services in the specification of a concrete service and M is the number of concrete services. The worst case of our algorithm occurs when all services have the same preference. In this case, the complexity of our method if the same as that of MiniCon. Our solution prunes the search space by introducing the notion of coverage domains and by ordering solutions

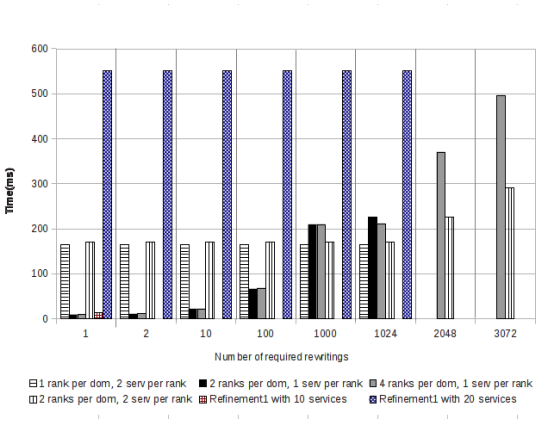

Figure 2: Execution time of our method *w.r.t.* nb of required (desired) rewritings. Varying nb of ranks and services: 10 or 20 (1 or 2 ranks) or 40 (2 or 4 ranks) services. Execution time of Refinement1 method computing all possible solutions for 10 and 20 services.

in accordance to the user preferences. As coverage domains offer a semantic classification of services, in general, a PCD does not belong to all the coverage domains. Our index structure organises PCDs according to coverage domains, so that there is no need to build all PCD combinations. By building solutions (concrete compositions) in a decreasing order of user preferences the refinement process can stop when the desired number of compositions is reached.

We have implemented a prototype of our method in Java. Figure 2 shows the average time from 60 executions on a Intel Quad-Core 2.70GHz, 2GB over synthesized services. Its goal is to show the result of synthesized compositions, done in order to evaluate the behaviour of our method in some bad situations.

Consider a composition with 10 abstract services and a varying number of concrete services defined by just one abstract service. Each concrete service covers at most one abstract service of our composition. In this case, a great number of rewritings is possible and our new proposal offers a considerable time saving when compared to our previous work.

We have run tests with 20 and 40 concrete services equally distributed over the 10 coverage domains (Figure 2). These tests have been chosen in order to illustrate the worst cases in terms of complexity. With only one preference rank all compositions have the same priority. For instance, with 40 services, 4 per rank, our algorithm behaves like Refinement1 and we have 4^{10} compositions with the same priority. Improvements are reported in situations where different priorities are assigned to services. For instance, Figure 2 shows that for 20 services (2 ranks per domain) we need 8.53*ms* and 21.5*ms* to offer 1 and 10 required rewritings, respectively. A similar execution

time is needed for 40 services, 4 ranks per domain, since in both cases we have 1 service per rank. Notice that before executing ProduceRewriting (Algorithm 1), the iterator generates all the combinations having the same priority. In this case, even if one solution is required, all of them are published. In Figure 2, for 40 services, 2 ranks per domain, we have 2^{10} rewritings to propose.

Figure 2 also shows the running time for Refinement1 when dealing with 10 and 20 services. For 10 concrete services only one solution exists: both methods have the same performance ($\approx 9ms$). For 20 services, since there are 2 services per domain, Refinement1 computes all the 2^{10} rewritings in $550ms$.

With our method we can compute the 2^{10} rewritings in about $170ms$ when we have one rank per domain and 2 services per rank. In this case all the solutions are computed in one shot as they have the same priority. In the case with 2 ranks per domain and one service per rank, our method needs about $5ms$, $70ms$ and $210ms$ to compute one solution, 100 solutions and all the solutions, respectively. Notice that the time generation of the 1024 solutions in the first case is slightly smaller compared to the latter case as the iterator computes at each step only one rewriting (no two rewritings have the same priority). This difference illustrates the small price introduced by our priority-oriented method.

6 CONCLUSIONS

Given the abstract specification of a composition, our method produces combinations of services available in the Cloud, in order to refine the specification. The algorithm presented here extends improves our previous work in (Costa et al., 2013) by classifying solutions according to an user's profile. This new proposal generates solutions in a preference order and avoids the production of a combinatorial number of rewritings. Our experiments show that efficiency has been greatly improved *w.r.t.* (Costa et al., 2013). An important perspective is to consider web intelligence techniques to better explore user's preferences.

REFERENCES

Alrifai, M., Risse, T., Dolog, P., and Nejdl, W. (2008). A scalable approach for qos-based web service selection. In *ICSOC Workshops*, pages 190–199.

Alrifai, M., Skoutas, D., and Risse, T. (2010). Selecting skyline services for qos-based web service composition. In *WWW*, pages 11–20.

Costa, U., Halfeld Ferrari, M., Musicante, M., and Robert, S. (2013). Automatic refinement of service compositions. In *Proceedings of the International Conference on Web Engineering (ICWE)*, pages 400–407.

Haddad, S., Mokdad, L., and Youcef, S. (2010). Selection of the best composite web service based on quality of service. In *ISSS/BPSC*, volume 177 of *LNI*, pages 255–266.

Lamparter, S., Ankolekar, A., Studer, R., and Grimm, S. (2007). Preference-based selection of highly configurable web services. In *Proceedings of the 16th international conference on World Wide Web*.

Lemos, F., Grigori, D., and Bouzeghoub, M. (2012). Adding non-functional preferences to service discovery. In Brambilla, M., Tokuda, T., and Tolksdorf, R., editors, *ICWE*, volume 7387 of *Lecture Notes in Computer Science*, pages 299–306. Springer.

Lin, N., Kuter, U., and Sirin, E. (2008). Web service composition with user preferences. In *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 629–643. Springer.

Mesmoudi, A., Mrissa, M., and Hacid, M.-S. (2011). Combining configuration and query rewriting for web service composition. In *ICWS*, pages 113–120.

Paiva Tizzo, N., Coello, J., and Cardozo, E. (2011). Improving dynamic Web service selection in WS-BPEL using SPARQL. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, pages 864–871.

Pottinger, R. and Halevy, A. Y. (2001). Minicon: A scalable algorithm for answering queries using views. *VLDB J.*, 10(2-3):182–198.

Sandionigi, C., Ardagna, D., Cugola, G., and Ghezzi, C. (2013). Optimizing service selection and allocation in situational computing applications. *IEEE T. Services Computing*, 6(3):414–428.

Sohrabi, S. and McIlraith, S. A. (2010). Preference-based web service composition: A middle ground between execution and search. In *International Semantic Web Conference (1) - ISWC 2010*, volume 6496 of *Lecture Notes in Computer Science*, pages 713–729. Springer.

Zeng, L., Benatallah, B., Lei, H., Ngu, A. H. H., Flaxer, D., and Chang, H. (2003). Flexible composition of enterprise web services. *Electronic Markets*, 13(2).

Zhao, W., Liu, C., and Chen, J. (2011). Automatic composition of information-providing web services based on query rewriting. *Science China Information Sciences*, pages 1–17.