Human-centric Computing and Information Sciences
a SpringerOpen Journal

RESEARCH Open Access

Publishing and discovering context-dependent services

Naseem Ibrahim[1*], Mubarak Mohammad[2] and Vangalur Alagar[2]

*Correspondence:
naseem.ibrahim@asurams.edu
[1]Department of Mathematics and Computer Science, Albany State University, Georgia, USA
Full list of author information is available at the end of the article

Abstract

In service oriented computing, service providers and service requesters are main interacting entities. A service provider *publishes* the services it wishes to make public using service registries. A service requester initiates a *discovery* process to find the service that meets its requirements using the service registries. Current approaches for the publication and discovery do not realize the essential relationship between the service contract and the conditions in which the service can guarantee its contract. Moreover, they do not use any formal methods for specifying services, contracts, and compositions. Without a formal basis it is not possible to justify through a rigorous verification the correctness conditions for service compositions and the satisfaction of contractual obligations in service provisions. In our recent works, we have identified the role of contextual information, trustworthiness information and legal rules in service provision. This paper focuses on the publication and discovery of trustworthy context-dependent services as supported by the novel framework *FrSeC*. It introduces a novel ranking algorithm that ranks trustworthy context-dependent services according to the degree they match service requesters requirements. Finally, this paper introduces a prototype implementation for the matching and ranking of services as supported by *FrSeC*.

Keywords: Service publication, Service discovery, Service ranking, Context-awareness, Trustworthy service provision

Introduction

In traditional Service-oriented Architecture (SOA) interactions, the three main interacting elements are the *service provider*, the *service requester* and the *service registry*. The service provider defines a service and publishes it through the service registry. The service registry acts as a data center that holds the services published by the different service providers. The service requester accesses the registry to get information about available services. It will then use this information to select a specific service that meets its requirements and will interact with the service provider of the selected service.

Hence, the main activities in SOA are *service publication*, *service discovery* and *service provision*. *Service publication* refers to the process of defining service contracts by service providers and publishing them through available service registries. *Service discovery* is the process of finding services that have been previously published and that meet the requirements of a service requester [1]. Typically, service discovery includes *service*

query, service matching, and *service ranking*. Service requesters define their requirements as service queries. Service matching refers to the process of matching the service requester requirements, as defined in the service query, with the published services. Service ranking is the process of ordering the matched services according to the degree they meet the requester requirements. The ranking will enable the service requester to select a specific or a most relevant service from the list of candidate services. *Service provision* refers to the process of executing a selected service. The execution may include some form of an interaction between the service requester and the service provider.

In practice, before publishing services a service provider defines the contract that can be guaranteed by a service. This contract includes the functionalities and quality of services guarantees that the provider can make. But such guarantees are not absolute. A service cannot guarantee its contract in all situations. It can only guarantee its contract in a predefined set of conditions. These conditions are usually related to the *context* of the service provider and requester. Context information has been defined as any information used to characterize the situation of an entity, such as location, time and purpose [2]. Legal rules also play a crucial role in constraining the publication and discovery of services. For example, a wireless phone provider may include in the service contract a guarantee of excellent quality, but this guarantee is not absolute. It may have a constraining condition stating that in order to ensure excellent quality, the consumer should be located within 1000 meters from cell phone stations. This constraint is related to the contextual information of the service consumer. In addition, local legal rules may black-out wireless service in secure-critical locations. Such legal rules should be an essential part of every contract.

It is necessary to distinguish between legal rules and nonfunctional requirements. If a nonfunctional property is "a soft" requirement it may be ignored. However ignoring a legal rule is equivalent to a "legal violation", which might land in legal disputes and even lead to loss of entire business. In essence, not enforcing a legal rule prevents the execution of a contract. Almost all current approaches use only functional and nonfunctional properties to enable the publication, discovery and provision of services. In [3], no distinction is made between legal rules and nonfunctional properties. Failure to include contextual information and legal rules will only mislead the consumer to believe in the advertised excellent quality of wireless service, regardless of where the consumer is domiciled which is not true.

To remedy the drawbacks of available service provision frameworks we have introduced two main concepts [4-6]. The first is a formal service model, which is called *ConfiguredService*. In this model, service and contract are packaged together. The service part includes functional and nonfunctional aspects of service, and the data parameters and attributes that are essential to define the functional and nonfunctional aspects. The contract part includes business rules, legal aspects, and context information. The second concepts is the *Formal Framework for Providing Context-dependent Services (FrSeC)* in which *ConfiguredServices* and their compositions are formally embedded. Service publication, service discovery, service selection and ranking, and service delivery are rigorously defined. The significance here is the way context information is defined for each stage, and is used in the interactions between the different components of the architecture elements in order to sustain the trustworthiness properties at all stages.

Motivation and contribution

In *FrSeC*, we have introduced contextual information, trustworthiness properties and legal rules as first class elements in both service publication and discovery. The introduction of contextual information and legal rules in service contracts introduces many challenges in service adaptation during service rediscovery process. The main goal of this paper is to address these challenges, and offer novel approaches for service publication, discovery and ranking, taking into consideration contextual information and legal rules.

This paper is structured as follows. First, we provide an overview of our formal framework for providing context dependent services *FrSeC* [4-6]. *FrSeC* supports the provision of services with context dependent services. Second, we focus on the publication and discovery of context-dependent services. We provide a novel formal notation for service publication. The three issues that we emphasize are *context*, *legal rules*, and *adaptability*. In *FrSeC*, service providers publish only *ConfiguredServices* in the service registry. A *ConfiguredService* is a structure that bundles together service functionality, service contract, and provision context. Service requesters query the service registry to find available *ConfiguredServices*. Often there is a semantic gap between the service query and the services in the registry. To deal with this, we introduced in the third part of this paper three novel query types for service discovery, and a ranking algorithm. Fourth, we discuses adaptability in *FrSeC*. Fifth, we introduce a prototype implementation of the ranking and matching supported by *FrSeC*. Finally, we briefly compare our work with the related work and provide concluding remarks.

An overview of FrSeC

This section briefly introduce the formal framework for providing context-dependent services *FrSeC*. The introduction of the main elements of *FrSeC* is essential to understand the contribution of this paper, namely the publication, discovery and ranking of services supported by *FrSeC*.

FrSeC was motivated by the need for a framework that supports the publication, discovery and provision of services with context-dependent contracts. The main elements of *FrSeC* are shown in Figure 1. A complete formal definition of *FrSeC* is presented in the two recent papers [4,6]. Below is a brief summary of *FrSeC* elements.

Service Provider (SP)

It is the entity that provides an implementation of a service specification. The service specification is published by the SP using SRe.

Service Registry (SRe)

It is a central repository for services, in which Service Providers publish their services and PU discovers services. It includes semantic definitions for domain specific concepts.

Context Gathering Unit (CGU)

FrSeC contains at least three context gathering units. One unit collects contextual information to assist SR in formulating their service queries. Another unit collects contextual information relevant to SP. The third unit collects contextual information to assist EU and

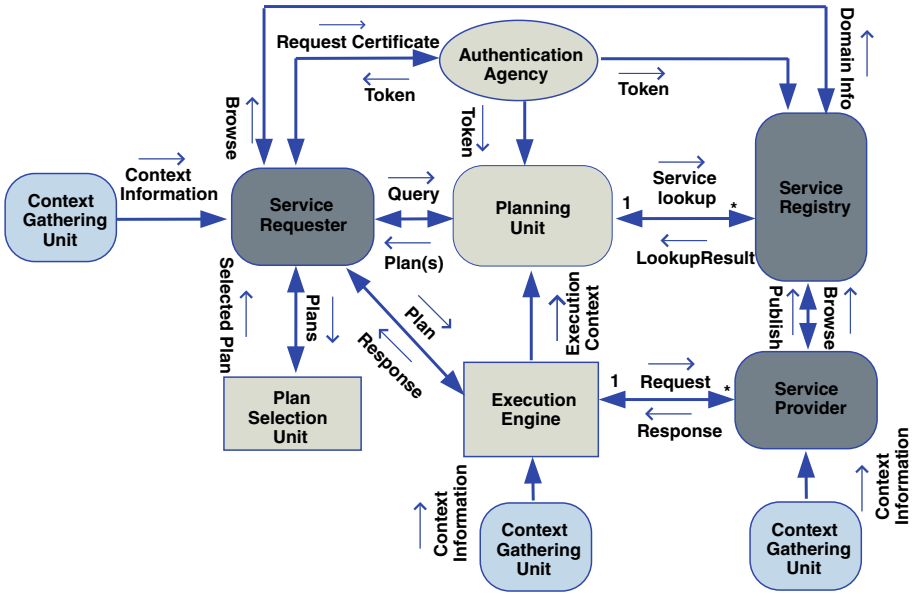

Figure 1 *FrSeC* Architecture. A presentation of the architectural components of the FrSeC architecture and the interactions between those components.

PU in dynamic planning activities. A central context manager may be added to monitor and trigger the adaptive context-aware behavior of the framework.

Service Requester (SR)

It is the entity requiring a certain functionality to be satisfied. It represents the client side of the interaction. It can be an application or another service. SR defines its requirements in a service query.

The Authentication Agency (AU)

It is the entity responsible for ensuring trustworthy access to SRe. It provides requesters with certificates (tokens) that allow them to access SRe. The certificate type depends on the legal and contextual information of the requester.

Planning Unit (PU)

It is responsible for managing the service discovery process by interacting with SR, SRe and AU. It also defines service composition. The composition includes defining the *plans* that can satisfy a query requirement. A plan defines the execution logic of a service or multiple services that collectively achieve the functional, nonfunctional and trustworthiness requirements of the requester. A complete formal composition theory is defined in [5]. This theory considers the functional, nonfunctional, legal and contextual parts of the service when defining the composition result.

Plan Selection Unit (PSU)

It is responsible for helping SR to select one or more plans from the set of plans received from PU. For each plan received, it requests additional information, such as data parameters, from SR and verifies that the information in the plan is complete with respect to the request. If it finds the information incomplete the chosen plan is ignored, otherwise the plan is selected for SR.

Execution Unit (EU)

It is responsible for managing the provision of services. It executes the selected plan. The execution process will include communicating with the service providers involved in the plan by sending service *requests* and obtaining service *responses*.

Service publication

Service providers publish service contracts through the Service Registry (introduced above) in order to make them available for discovery. In current approaches (discussed in the Related Work Section), the service contract includes only the functional and nonfunctional requirements together with any semantic information the service provider wishes to make public. But in *FrSeC* the service definition is much richer. It includes the service contract together with the related contextual conditions. Hence, we introduced the concept *ConfiguredService*, which is a structure in which service functionality, service contract, and service provision context are bundled together. SP publishes the two main elements, namely the *contract* and *context*, in the *ConfiguredService*.

The contract includes *function*, *nonfunctional properties* and *legal issues*. The context part of the *ConfiguredService* includes the main parts *context info* and *context rules*. The *context info* defines the contextual information of the *ConfiguredService*. The *context rules* define the contextual information related to SR that should be true for SP to guarantee its *ConfiguredService* contract.

Example 1. *Table 1 shows a ConfiguredService of a Car Repair Shop. The repair shop charges 60$ per hour and requires a deposit of 300$ with the condition that the car owner is a member of the* Canadian Automotive Association (CAA).

In [5], we introduced the novel service model *ConfiguredService*. *ConfiguredService* is formally defined using a model-based approach. The formal model is built from set theory and logic. Below is a formal presentation of *ConfiguredService*.

Constraints

A constraint is a logical expression, defined over data parameters and attributes in first order predicate logic. If $\mathbb{C}$ denotes the set of all such logical expressions, $X \in \mathbb{C}$ is a constraint. The following notation is used in our definition:

- $\mathbb{T}$ denotes the set of all data types, including abstract data types and $Dt \in \mathbb{T}$ means Dt is a datatype.
- $v : Dt$ denotes that v is either constant or variable of type Dt.
- X_v is a constraint on v. If v is a constant then X_v is true.
- V_q denotes the set of values of data type q.
- $x : \Delta$ denotes a logical expression $x \in \mathbb{C}$ defined over the set of parameters Δ.

Parameters

A parameter is a 3-tuple, defining a data type, a variable of that type, and a constraint on the values assumed by the variable. We denote the set of data parameters as $\Lambda = \{\lambda = (Dt, v, X_v) | Dt \in \mathbb{T}, v : Dt, X_v \in \mathbb{C}\}$.

Table 1 RepairShop *ConfiguredService* description

ConfiguredService	***Function***			***NonFunctional***	***Legal***	***ContextRule***	***ContextInfo***
RepairShop	**Name**:ReserveRS **Pre:**CarBroken==T **Post:**HasAppointment==T **Address:** XXX	**InputParameters**: CarBroken:bool deposit:double CarType:string failureType:string	**ResultName**:ResultRS **OutputParameters**: HasAppointment:bool numberOfHours:int	Price=60$/h	deposit=300$ **PriceCondition**: CarType=toyota	membership==CAA	(location,montreal)

A description of the RepairShop *ConfiguredService* containing the contract and context information.

Attributes

An attribute has a name and type, and is used to define some semantic information associated with the name. The set of attributes is $\alpha = \{(Dt, \nu_\alpha) | Dt \in \mathbb{T}, \nu_\alpha : Dt\}$.

Context

A context is formalized as a 2-tuple $\beta = \langle r, c \rangle$, where $r \in \mathbb{C}$, built over the contextual information c. Context information is formalized using the notation in [7]: Let $\tau : DIM \rightarrow I$, where $DIM = \{X_1, X_2, \ldots, X_n\}$ is a finite set of dimensions and $I = \{a_1, a_2, \ldots, a_n\}$ is a set of types. The function τ associates a dimension to a type. Let $\tau(X_i) = a_i$, $a_i \in I$. We write c as an aggregation of ordered pairs (X_j, ν_j), where $X_j \in DIM$, and $\nu_j \in \tau(X_j)$.

Example 2. *Using the context notation, the contextual information of ConfiguredService RepairShop(rs) presented in Table 1, is written as* $\beta_{rs} = \langle r_{rs}, c_{rs} \rangle$, *where* $r_{rs} = \{(membership == caa)\}$ *is the context rule and* $c_{rs} = \{(Location, Montreal)\}$ *is the contextual information of the service provider.*

Contract

A contract is a 3-tuple $\sigma = \langle f, \kappa, l \rangle$, where the service function f, the set of nonfunctional properties κ and the set l of legal issues that bind the service contract are defined below.

Example 3. *Using this formalism the RepairShop contract presented in Table 1, is written as* $\sigma_{rs} = \langle f_{rs}, \kappa_{rs}, l_{rs} \rangle$.

- ***Service function:*** A service function is a 4-tuple $f = \langle g, i, pr, po \rangle$, where g is the function signature, i is the function result, pr is the precondition, and po is the postcondition. A signature is a 3-tuple $g = \langle n, d, u \rangle$, where n is the function identification name, d is the set of function parameters and u is the function address, the physical address on a network that can be used to call a function. The result is defined as $i = \langle m, q \rangle$, where m is the result identification name and q is the set of parameters resulting from executing the *ConfiguredService*. The precondition pr and the postcondition po are data constraints.

  Example 4. *The RepairShop contract functionality presented in Table 1, is formally written as* $f_{rs} = \langle g_{rs}, i_{rs}, pr_{rs}, po_{rs} \rangle$, *where*

  - $g_{rs} = \langle n_{rs}, d_{rs}, u_{rs} \rangle$, *where* $n_{rs} = (ReserveRS)$, $d_{rs} = \{(CarBroken, bool), (deposit, double), (CarType, string), (failureType, string)\}$, *and* $u_{rs} = (XXX)$.
  - $i_{rs} = \langle m_{rs}, q_{rs} \rangle$, *where* $m_{rs} = (ResultRS)$ *and* $q_{rs} = \{(HasAppointment, bool), (numberOfHours, int)\}$.
  - $pr_{rs} = \{(CarBroken == true)\}$ *and* $po_{rs} = \{(HasAppointment == true)\}$.

- ***Nonfunctional property:*** Defined as a 6-tuple $\kappa = \langle \rho, \epsilon, \psi, \eta, p, tr \rangle$. The safety guarantee ρ includes time guarantee ρ_t and data guarantee ρ_d. The time guarantee is defined as the time the service takes to provide its function. The data guarantee refers to the accuracy of data. The security guarantee ϵ defines the set of security protocols that the Service Provider has followed to guarantee confidentiality and integrity constraints. The reliability guarantee ψ refers to the guaranteed maximum time

between failures. The availability guarantee η refers to the guaranteed maximum time for repairs. The price is defined as a 3-tuple $p = \langle a, cu, un \rangle$, where a is the price amount defined as a non negative double, cu is currency tied to a currency type $cType$, and un is the pricing unit. Provider Trust is defined as a 3-tuple $tr = \langle ce, pg, re \rangle$. Lowest price guarantee pg is represented by a Boolean flag that is true when a *ConfiguredService* can guarantee its price to be lower than the price of any other *ConfiguredService* providing the same functionality. Client recommendations ce and recommendations from independent organizations re can be defined as sets of ordered pairs representing the clients or organization and associated recommendation.

Example 5. *The nonfunctional property in the RepairShop contract presented in Table 1, is formalized as* $\kappa_{rs} = \langle p_{rs} \rangle$, $p_{rs} = \langle a_{rs}, cu_{rs}, un_{rs} \rangle$, *where* $a_{rs} = (60)$ *is the cost,* $cu_{rs} = (dollar)$ *is the currency, and* $un_{rs} = (hour)$ *is the pricing unit.*

- ***Legal issues:*** A legal issue is a rule, expressed as a logical expression in $\mathbb{C}$. A rule may imply another rule, however no two rules may conflict each other. We write $l = \{y | y \in \mathbb{C}\}$ to represent the set of legal rules. The legal aspect of the RepairShop contract presented in Table 1, is formally written as $l_{rs} = \{(deposit = 300), (CarType == toyota)\}$.

Putting the above definitions together we arrive at a formal definition for *ConfiguredService*.

Definition 1. *A ConfiguredService is a 4-tuple* $s = \langle \Lambda, \alpha, \beta, \sigma \rangle$, *where* Λ *is a set of parameters,* α *is a set of attributes,* β *is a context, and* σ *is a contract.*

Service discovery

To be able to select and invoke a service that meets its requirements, a service requester should initiate a discovery process. The discovery process includes *service query, service matching,* and *service ranking*. First, SR defines his requirements in the service query. Second, the query is matched with available *ConfiguredServices* by PU. Third, PU ranks available candidate *ConfiguredServices*. Fourth, SR with the help of PSU, selects a *ConfiguredService* from the set of ranked *ConfiguredServices*. The novelty of the discovery process supported by *FrSeC* is two-fold. First, the discovery process takes into consideration legal requirements and context conditions together with functional and nonfunctional requirements. Second, depending on the requirements of the service requester, *FrSeC* supports the two types of queries *traditional style* and *buffet style*. The rest of this section discusses service query and matching. Service ranking is discussed separately in the next section.

Traditional style

In traditional style discovery, the requester has a clear idea about the requirements. But, the semantic information necessary to define the query is missing. Hence, SR accesses SRe to get the domain knowledge which will help in defining the query. The query process can be defined in the following steps:

1. SR sends a request to the AU for a certificate to access SRe.
2. AU provides the certificate depending on the legal and contextual information of SR.

3. SR, with the help of the certificate, browses SRe to gain domain knowledge.
4. SRe provides SR with domain knowledge, such as available domains and their associated functionalities.
5. SR uses this domain knowledge to construct the query and sends it to PU.
6. PU defines and sends service lookups to SRe.
7. The service lookup result is then used by PU to perform matching between the query and available services.
8. PU defines the query result (plan) and sends it to SR with any feedback if necessary.

Traditional style discovery can be either *exact-match* discovery or *weighted-match* discovery as discussed below.

Exact-match

In exact-match discovery, the requester requires an exact match to the requirements. The candidate *ConfiguredServices* should be able to guarantee all the requirements. The exact-match query, as shown in Figure 2, consists of the five main parts *required function, required nonfunctional properties, required legal issues, consumer contextual information,* and *authentication certificate*. The query also contains the set of parameters that it can understand. This set is a subset of the parameters associated with the functionality it chose when accessing SRe. The required nonfunctional properties are a subset of the nonfunctional properties associated with the functionality defined in SRe. The three following definitions formalize an exact-match query.

Definition 2. *An exact-match query q_e is defined as $q_e = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda} \rangle$, where $\hat{f}$ is a query required function, $\hat{\kappa}$ is the nonfunctional requirement, $\hat{l}$ is the legal rules requirements, $\hat{c}$ is the contextual information of the service consumer, E is the authentication certificate and $\hat{\Lambda}$ is the set of parameters SR can provide or understand. The formal definitions of context information, legal rules and parameters are identical to the definitions in the previous section.*

Definition 3. *The required function is defined as $\hat{f} = \langle \hat{pr}, \hat{po}, \hat{D}, \hat{SF} \rangle$, where $\hat{pr}$ is the set of preconditions of the required function, $\hat{po}$ is the set of postconditions of the required*

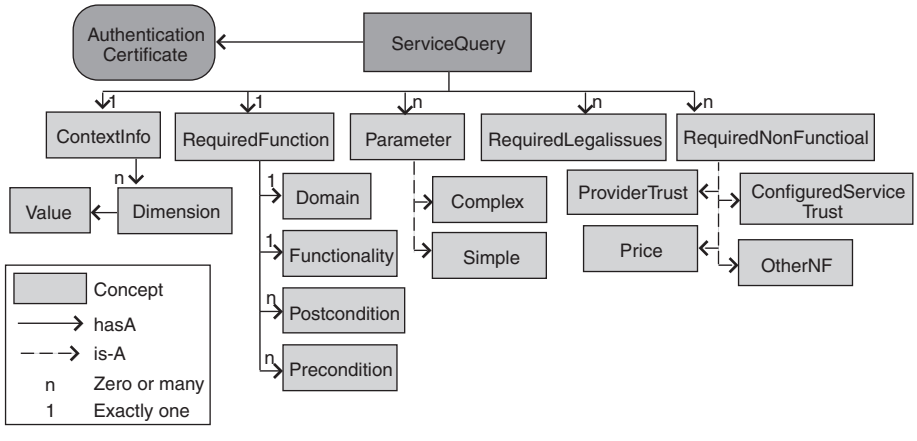

Figure 2 Exact-match query. The content of a traditional style query of type exact-match.

function, $\hat{D} = (x : x|string)$ *is the associated domain as defined in SRe and* $\hat{SF} = (x : x|string)$ *is the functionality as defined in SRe. The formal definition of precondition and postcondition is identical to the one in the previous section.*

Definition 4. *The required nonfunctional property is defined as* $\hat{\kappa} = \langle \hat{\rho}, \hat{\epsilon}, \hat{\psi}, \hat{\eta}, \hat{p}, \hat{tr} \rangle$*, where* $\hat{\rho}$ *is the required safety guarantee,* $\hat{\epsilon}$ *is the required security guarantee,* $\hat{\psi}$ *is the required availability guarantee,* $\hat{\eta}$ *is required the reliability guarantee,* $\hat{p}$ *is the maximum price required and* $\hat{tr}$ *is the required provider trust guarantee. The formal definition of each of those nonfunctional properties is identical to the definition in the previous section.*

Example 6. *If a service requester is attempting an exact-match query for the repair shop functionality defined in Table 1, the query could be defined as* $q_e = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda} \rangle$ *where:*

- $\hat{f} = \langle \hat{pr}, \hat{po}, \hat{D}, \hat{SF} \rangle$*, where* $\hat{pr} = \{(CarBroken == true)\}$, $\hat{po} = \{(HasAppointment == true)\}$, $\hat{D} = (CarDomain)$, and $\hat{SF} = (RepairShopFunctionality)$.
- $\hat{\kappa} = \langle \hat{p} \rangle$*, where* $\hat{p} = \langle \hat{a}, \hat{cu}, \hat{un} \rangle$, $\hat{a} = (50)$, $\hat{cu} = (dollar)$ and $\hat{un} = (hour)$.
- $\hat{l} = \{(deposit = 500)\}$.
- $\hat{c} = \{(membership == caa)\}$.
- $\hat{\Lambda} = \{(CarBroken, bool), (deposit, double), (CarType, string), (failureType, string)\}$

After receiving the lookup result from SRe, PU will match available *ConfiguredServices* with the service query. In exact-match the matching process will result in a *ServiceType* which is a list of candidate *ConfiguredServices*. All *ConfiguredServices* in the *ServiceType* provide the exact match to all the requirements defined in the service query. For each *ConfiguredService* in the lookup result the matching will:

1. compare the query functionality with the *ConfiguredService* functionality,
2. compare the query nonfunctional requirements with the *ConfiguredService* nonfunctional properties,
3. compare the query legal issues with the *ConfiguredService* legal rules, and
4. use the query contextual information to make sure the *ConfiguredService* context rules are met.

Weighted-match

The formal definition of the weighted-match query is very close to the exact-match query. The only difference is the inclusion of the weights. In weighted-match discovery, the requester knows the requirements, however unsure about them. SR states the requirements in a query with the expectation that the best matched services, that might not be exact matches, will be given. That is, the *ConfiguredServices* received by the service requester do not have to match all the stated requirements. When stating the query the requester assigns a weight, representing the priority, with every property requirement. A higher weight indicates a higher priority. SR can also state *exact* property to indicate that an exact match is necessary for this particular property. Stating the weight is valid for the elements of the required function, nonfunctional requirements and the required legal rules. With respect to contextual information, SR can state more than one possible set of contextual information. As an example, the context information for service delivery

can be either the service be delivered at home or at office. Each contextual information will be assigned a weight to indicate the preference of the requester. In our further discussion we assume that the assigned weights belong to the set *{Low, BelowAverage, Average, AboveAverage, High, Exact}*, in which the values are listed in strictly decreasing order of priority.

Definition 5. *A weighted-match query is defined as* $q_w = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda}, \Xi \rangle$, *where* $\hat{f}$, $\hat{\kappa}$, $\hat{l}$, $\hat{c}$, E *and* $\hat{\Lambda}$ *are defined as in the traditional query, and* $\Xi : (x \in \{Low, BelowAverage, Average, AboveAverage, High, Exact\}) \rightarrow (y \in \{\hat{pr}, \hat{po}, \hat{\rho}, \hat{\epsilon}, \hat{\psi}, \hat{\eta}, \hat{p}, \hat{tr}, \hat{l}, \hat{c}\}$ *is a function that assign weights to the elements of the weighted-match query.*

Example 7. *Adding weights to the query defined in Example 6 the weighted-match style query will be defined as* $q_w = \langle \hat{f}, \hat{\kappa}, \hat{c}, \hat{l}, E, \hat{\Lambda}, \Xi \rangle$ *where* $\hat{f}$, $\hat{\kappa}$, $\hat{c}$, $\hat{l}$, E *and* $\hat{\Lambda}$ *are defined as in Example 6, and* $\Xi = \{((CarBroken == true), \mathbf{Exact}), ((HasAppointment == true), \mathbf{Exact}), (\hat{p}, \mathbf{High}), ((deposit = 500), \mathbf{Average})\}$.

The matching process in weighted-match discovery considers all possible *ConfiguredServices* even if some properties are not satisfied. All candidate *ConfiguredServices* will be included in the matching result *ServiceType*, with the exception of the *ConfiguredServices* that do not provide a match for a requirement with *Exact* weight.

Buffet style

The main difference between traditional style and buffet style is that in traditional style the requester is more or less clear about the requirements in order to be able to define the query in terms of these requirements, whereas in buffet style the requester is not at all clear about the requirements. Hence, the SRe is browsed for available *ConfiguredServices* and a query is defined only in terms of existing *ConfiguredServices*. The buffet style query process can be defined in the following steps:

1. SR sends a request to the AU for a certificate to access SRe.
2. AU provides the certificate depending on the legal and contextual information of SR.
3. SR, with the help of the certificate, browses SRe for available *ConfiguredServices*.
4. SRe provides SR with high level information about the set of available *ConfiguredServices*.
5. SR defines the query in terms of specific *ConfiguredServices* and sends it to PU.
6. PU will access the SRe to get the complete information about the required *ConfiguredServices*.
7. SRe will verify that SR has the required authentication to use the required *ConfiguredServices*.
8. PU defines the query result (plan) to include the complete *ConfiguredServices* information and sends it to SR with any feedback if necessary.

No matching process is necessary in buffet style, because the service requester is querying only *ConfiguredServices*. As a consequence, the definition of buffet style query, shown in Figure 3, consists of the three main parts *required ConfiguredService, consumer*

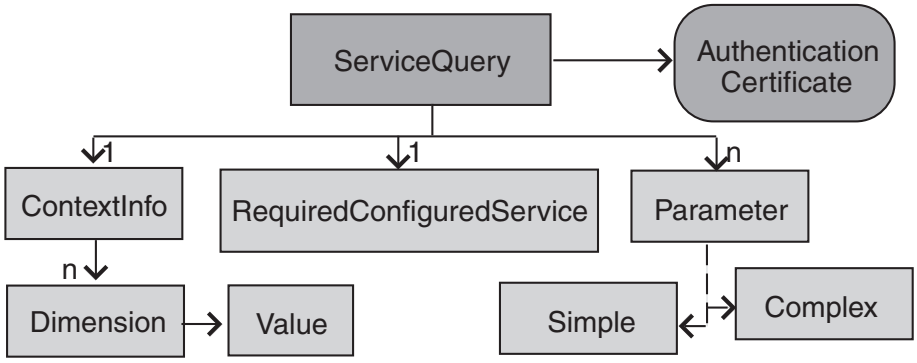

Figure 3 Buffet style query. The content of a buffet style query.

contextual information, and *authentication certificate*. The following two definitions formalize a buffet style query.

Definition 6. *A buffet style query is defined as* $q_b = \langle \hat{cs}, \hat{c}, E, \hat{\Lambda} \rangle$*, where* $\hat{cs}$ *is the required ConfiguredService defined as in Section 2,* $\hat{c}$ *is the contextual information of the service consumer defined as in Section 2, E is the authentication certificate and* $\hat{\Lambda}$ *is the set of parameters SR can provide or understand, all defined as in Section 2.*

Example 8. *If SR is attempting a buffet style query for the ConfiguredService RepairShop defined in Table 1, the query will be defined as* $q_b = \langle \hat{cs}, \hat{c}, E, \hat{\Lambda} \rangle$*, where* $\hat{cs} = s_{rs}$*.* $\hat{c} = \{(membership == caa)\}$*, and* $\hat{\Lambda} = \{(CarBroken, bool), (deposit, double), (CarType, string), (failureType, string)\}$*.*

Service ranking

In buffet-style the service requester queries for a concrete *ConfiguredService*, and hence no ranking is necessary. Service ranking is necessary for weighted-match service queries, for the following reasons. In exact matching, the only difference between the *ConfiguredServices* in the *ServiceType* is the *order* they were discovered. A *ConfiguredService A* that was discovered before *ConfiguredService B* will precede it in the *ServiceType* list. This is because all services provide an exact match to the requirements, which is *not* the case in weighted-match. For a weighted-match, the position of a service in a *ServiceType* should indicate the degree to which the requester requirements are met. That is, in weighted-match search a service that appears first in a *ServiceType* should have the best match with the stated requirements than a service that appears later in the *ServiceType* list. These considerations have motivated us to discover a ranking method.

In *FrSeC*, the ranking process is performed by PU. The process takes as inputs the weighted-match query and the *ServiceType*, and generates as output an ordered *ServiceType*. The ranking process can be defined in the following 3 steps.

Form weight vector

In formulating a weighted-match query the requester assigns a weight to each property that is relevant for him. PU extracts these weights and constructs the weight vector, as in Equation 1, where Q_w is the weighted-match query weight vector and w_i is the weight of

property *i* as defined by the service requester. Property *i* can be a precondition, a postcondition, a nonfunctional requirement or a legal requirement. The number of properties *n* depends on the weighted-match query defined by the service requester.

$$Q_w = [\, w_1, w_2, w_3, .., w_n] \tag{1}$$

We have mentioned earlier that the weight can be *{Low, BelowAverage, Average, AboveAverage, High, Exact}*. *ConfiguredServices* that do not satisfy *Exact* values are filtered when doing the weighted-match matching. So the possible weight values are *{Low, BelowAverage, Average, AboveAverage, High}*. We assume in further discussion that weight values are whole numbers in the range 1 . . . 5, where 1 denotes *Low* and 5 denotes *High*.

Construct weight matrix

By using the weight vectors constructed in Step 1, the weight matrix for the *ConfiguredServices* in the *ServiceType* is constructed. This is shown in Equation 2, where *n* is the number of properties defined in Equation 1 and *m* is the number of *ConfiguredServices* in the *ServiceType*. Each column represents the weights of the properties in a single *ConfiguredService*. Each row represents the weights of a single property in the different *ConfiguredServices*.

$$CS_w = \begin{bmatrix} w_{1,1} & w_{2,1} & .. & w_{m,1} \\ w_{1,2} & w_{2,2} & .. & w_{m,2} \\ .. & .. & .. & .. \\ w_{1,n} & w_{2,n} & .. & w_{m,n} \end{bmatrix} \tag{2}$$

The value of the *ConfiguredService* property weight depends on the property type. If a property *j* is a precondition, postcondition, legal rule (without values) or a security property, a weight $w_{i,j}$ is equal to 1, if *ConfiguredService i* satisfies property *j* and is equal to 0 otherwise. If property *j* is price, legal rule (with values), availability or time-safety, $w_{i,j}$ is calculated according to Equation 3, where *z* is the required property value as defined in the weighted-match query and *x* is the actual property value specified in *ConfiguredService*.

$$w_{i,j} = \begin{cases} 1 & \text{if } x \leq z \\ 1 - \left(\frac{x-z}{2z-z}\right) = 2 - \frac{x}{z} & \text{if } z < x < 2z \\ 0 & \text{if } x \geq 2z \end{cases} \tag{3}$$

Equation 3 assumes that actual value that is more than double the required value will be given a weight of 0. Anything that is less than the required value will be given 1. And an actual value between the required value and double the required value will be given a weight that depends on how close the actual value is to the required value. For example, if the required price as defined in the service query is 50, an actual *ConfiguredService* price of 55 should be given a better weight than a price of 80.

If property *j* is reliability, $w_{i,j}$ is calculated according to Equation 4, where *z* is the required reliability value as defined in the weighted-match query and *x* is the *ConfiguredService* actual reliability value.

$$w_{i,j} = \begin{cases} 1 & \text{if } x \geq z \\ 1 - \left(\frac{z-x}{z-\frac{z}{2}}\right) = \frac{2x}{z} - 1 & \text{if } \frac{z}{2} < x < z \\ 0 & \text{if } x \leq \frac{z}{2} \end{cases} \tag{4}$$

Equation 4 assumes that actual values that is less than half the required value will be given a weight of 0. A value that is more than the required value will be given a weight of 1. And an actual value between the required value and half the required value will be given a weight that depends on how close the actual value is to the required value. The main difference between reliability and other properties is that reliability values represent a minimum while other properties represent a maximum.

Calculate weights for ranking

A single weight value for each *ConfiguredService* is computed, and the services are ranked based on these weights. Equation 5 uses the results of steps one and two to calculate the raking weight vector.

$$W = Q_w \times CS_w \tag{5}$$

The ranking weight vector W contains the weights of the different *ConfiguredServices*. These weights are used to rank the *ConfiguredServices*. The *ConfiguredService* with the highest weight value is placed first in the *ServiceType*. The *ConfiguredService* with the second highest weight value is placed second in the *ServiceType* and so on for the rest of the *ConfiguredServices*.

Example 9. *Two ConfiguredServices RepairShopA and ReapirShopB provide the functionality required in Example 7. They don't provide an exact match to the nonfunctional and legal requirements, but rather a partial match. The list of properties will include: {RequiredPrecondition, RequiredPostcondition, RequiredPrice, RequiredDeposit}. The RequiredPrecondition and the RequiredPostcondition will be filtered out because they require an exact match. So we are left with RequiredPrice and RequiredDeposit. Hence, the weighted-match query weight vector is* $Q_w = \begin{bmatrix} High & Average \end{bmatrix}$. *In numbers,* $Q_w = \begin{bmatrix} 5 & 3 \end{bmatrix}$. *ConfiguredService RepairShopA* (rsA) *has a cost of* $rsA_c = 40\$/hour$ *and requires a deposit of* $rsA_d = 600\$$. *ConfiguredService RepairShopB* (rsB) *has a cost of* $rsB_c = 70\$/hour$ *and requires a deposit of* $rsB_d = 400\$$. *Hence, the ConfiguredServices weight matrix is defined, using Equations 2 and 3, as:*

$$CS_w = \begin{bmatrix} w_{rsA,c} & w_{rsB,c} \\ w_{rsA,d} & w_{rsB,d} \end{bmatrix}$$

where, $w_{rsA,c} = 1$, $w_{rsB,d} = 1$ *and:*

$$w_{rsA,d} = 2 - \frac{600}{500} = 0.8$$

$$w_{rsB,c} = 2 - \frac{70}{50} = 0.6$$

The ranking weight vector will then be defined using Equation 5 as:

$$W = \begin{bmatrix} 5 & 3 \end{bmatrix} \begin{bmatrix} 1 & 0.6 \\ 0.8 & 1 \end{bmatrix} = \begin{bmatrix} 7.4 & 6 \end{bmatrix}$$

Hence, ConfiguredService RepairShopA scores 7.4 and ranked first, while ConfiguredService RepairShopB scores 6 and ranked second.

FrSeC adaptability

One of the main features of *FrSeC* is its ability to adapt to situations that trigger a need for a rediscovery or re-ranking process. Below is a discussion of the most important triggers and how they are handled in *FrSeC*.

Context change

The discovery process uses the contextual information of the service requester at service discovery time. But during service execution, the contextual information of the service requester might have changed. As a consequence, the contextual rules of the discovered service(s) might be violated, other services may be more suitable. In order to deal with the dynamic change in context we introduce an adaptable discovery mechanism. In *FrSeC*, this mechanism includes the following steps:

1. CGU senses the new context information and informs SR.
2. SR generates a new query with the new context information.
3. The context change may result in a change to the security level. So SR contacts AU with the new context information.
4. AU sends a new token to SR.
5. SR sends the new query to the PU which will initiate a new discovery process.
6. PU will send a new plan with the set of new ordered *ConfiguredServices*.
7. EU will migrate from the old *ConfiguredService* to the new *ConfiguredService*.

Failure in service availability

During service execution, the executing service might fail or become unavailable. For example, the wireless router might fail. *FrSeC* is designed to adapt to service failures. In our design, PU uses *ServiceType*, and not specific *ConfiguredServices* when defining query result. The list *ServiceType* contains ordered *ConfiguredServices* that can meet the requirements of a specific query. These *ServiceTypes* will be part of the plan sent to EU. During run time, if a *ConfiguredService* fails or becomes unavailable, the EU will select the next *ConfiguredService* in the *ServiceType*. The worst case is that an equivalence class has only one *ConfiguredService* and it fails. The feedback loop in *FrSeC* will restart the discovery process in this case.

New alternative services

Service executions may be performed over days, or even months. But service selection and binding are usually performed only the first time the requester uses the service. This might not be practical because new services might be available during this long execution time. The new alternative services might be new services or old services with new modified contracts. A new contract might include a lower price or a better quality. For example, a wireless provider with cheaper price and same quality guarantees might become available. In order to adapt to new alternative services during run time, SR registers with PU. This registration will guarantee that PU will inform SR in case of a new *ConfiguredService* that provides the requester the required functionality becomes available. Thus, SR can initiate a new discovery process.

New contract rules

Contracts bound to *ConfiguredServices* may be either *strict* or *flexible*. In a strict contract, the life-time of contract is made explicit. Providers and requesters are bound by this

timeline. In a flexible contract, there is no life-time specification, which allows providers to change the contract terms at any point of time. For example, the service provider might increase the price of his wireless Internet connection. Providers might not be aware of the identity of their clients. This design decision was made to enshrine privacy issues. In *FrSeC*, providers inform EU of changes to service contract. At the time of service delivery, EU informs SR of changes to the contract and delivers the service only upon receiving the acceptance of new contract terms from SR. In order not to deny service, requesters are allowed to initiate a rediscovery process in accordance with the new contract terms.

New requester requirements

Some service executions might be too long and during this service time the requirements of the requester might change. To deal with new requirements, the requester has the choice of a rediscovery process or a re-ranking process. In the rediscovery process the requester will define a new query and go through all steps of service discovery. In a re-ranking process, the requester will ask PU to re-rank the *ConfiguredServices* in the *ServiceType* taking into consideration the new assigned weights to the elements in the modified query.

Evaluation and experiments

To evaluate the contributions presented in this paper, a Java based application has been implemented to represent the Planning Unit. This application takes as input the set of *ConfiguredService* that provide a specific functionality as returned from the Service Registry, and the service query. The application will then match between the service query and the candidate *ConfiguredServices* taking into consideration the functional, nonfunctional, legal, and contextual information. Two types of matching has been implemented 1) exact match and 2) weighted-match. The ranking algorithm has also been implemented. Figure 4 shows a snap shot of the Planning Unit application.

The application was tested on a standard PC using an Intel Centrino processor with 4GB of memory and running Windows 7 Professional. The average matching and ranking time was in in milliseconds for each *ConfiguredService* which eliminate the concerns of scalability issues.

The tool was tested on multiple case studies including the Automotive Emergency Case Study. Table 2 shows the set of *ConfiguredServices* that are matched and ranked according to the service requester query presented in Table 3. The matching and ranking result of these *ConfiguredServices* are:

- *ConfiguredService* RepairShop5 is matched by 100.0%
- *ConfiguredService* RepairShop4 is matched by 88.19%
- *ConfiguredService* RepairShop1 is matched by 83.33%
- *ConfiguredService* RepairShop2 is matched by 80.56%
- *ConfiguredService* RepairShop3 is matched by 78.12%

Related work

Related work can be divided into related publication approaches and related service discovery approaches.

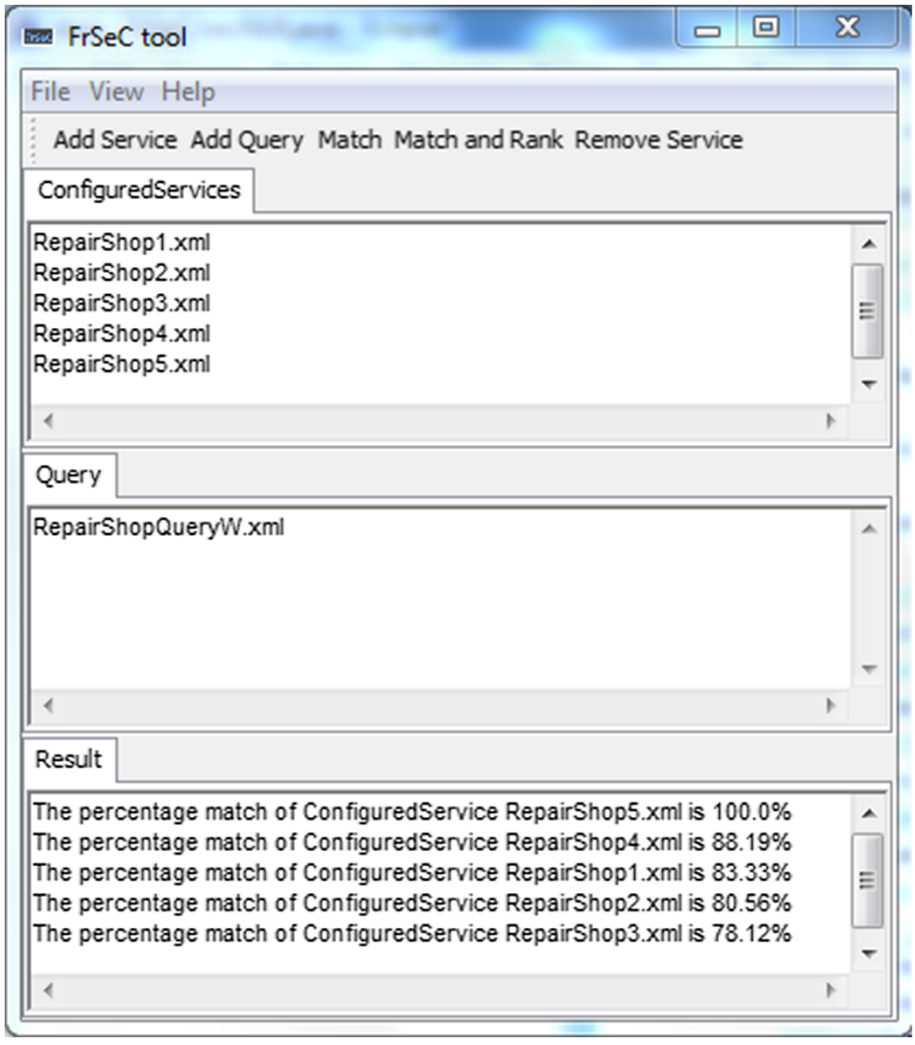

Figure 4 Planning unit tool. A snapshot of the planning unit tool.

The most notable publication approaches are UML-based such as [8,9], WSDL [10], OWL-S [11], WSMO [12], SOADL [13], SRML [14], and (SOFM) [15]. These approaches are compared and the result of comparison is presented in Table 4. It is clear that all approaches support the modeling of the functional behavior. Nonfunctional and trustworthiness properties are only supported in a simple manner by few approaches. Contextual information is not represented by any approach, hence the relationship between contract and context is totally ignored. A couple of approaches, which have ignored the modeling of nonfunctional and trustworthiness properties, have used formal methods and conducted formal verification. Except for UML and Web services languages most of the approaches provide a minimum amount of tools to support the modeling using their service models.

The most notable service discovery approaches are SeGSeC [16], eFlow [17], SELF-SERV [18], SHOP2 [19], SWORD [20], Argos [21], FUSION [22], Proteus [23], SPACE [24], StarWSCoP [25], METEOR-S [26], SeCSE [27], DynamiCoS [28] and TSCN [29]. These approaches have been compared with respect to the following criteria.

Table 2 Multiple RepairShop *ConfiguredServices* descriptions

ConfiguredService	***Function***			***NonFunctional***	***Legal***	***ContextRule***	***ContextInfo***
RepairShop1	**Name**:ReserveRS **Pre**:CarBroken==T **Post**:HasAppointment==T **Address**: XXX	**InputParameters**: CarBroken:bool deposit:double CarType:string failureType:string	**ResultName**:ResultRS **OutputParameters**: **HasAppointment**:bool **numberOfHours**:int	Price = 60$/h Client Rec.=5	deposit=300$ Warranty= 3 **PriceCondition**: CarType=toyota	membership==CAA	(location,montreal)
RepairShop2	**Name**:ReserveRS **Pre**:CarBroken==T **Post**:HasAppointment==T **Address**: XXX	**InputParameters**: CarBroken:bool deposit:double CarType:string failureType:string	**ResultName**:ResultRS **OutputParameters**: HasAppointment:bool numberOfHours:int	Price=50$/h Client Rec.=4 Recommended by CAA	deposit=400$ Warranty= 2 **PriceCondition**: CarType=toyota	membership==CAA	(location,montreal)
RepairShop3	**Name**:ReserveRS **Pre**:CarBroken==T **Post**:HasAppointment==T **Address**: XXX	**InputParameters**: CarBroken:bool deposit:double CarType:string failureType:string	**ResultName**:ResultRS **OutputParameters**: HasAppointment:bool numberOfHours:int	Price=40$/h Client Rec.=3 Recommended by CAA	deposit=500$ Warranty= 1 **PriceCondition**: CarType=toyota	membership==CAA	(location,montreal)
RepairShop4	**Name**:ReserveRS **Pre**:CarBroken==T **Post**:HasAppointment==T **Address**: XXX	**InputParameters**: CarBroken:bool deposit:double CarType:string failureType:string	**ResultName**:ResultRS **OutputParameters**: HasAppointment:bool numberOfHours:int	Price=70$/h Client Rec.=5 Recommended by CAA	deposit=300$ Warranty= 4 **PriceCondition**: CarType=toyota	membership==CAA	(location,montreal)
RepairShop5	**Name**:ReserveRS **Pre**:CarBroken==T **Post**:HasAppointment==T **Address**: XXX	**InputParameters**: CarBroken:bool deposit:double CarType:string failureType:string	**ResultName**:ResultRS **OutputParameters**: HasAppointment:bool numberOfHours:int	Price=40$/h Client Rec.=5 Recommended by CAA	deposit=250$ Warranty= 4 **PriceCondition**: CarType=toyota	membership==CAA	(location,montreal)

A descriptions of 5 ConfiguredServices providing the same RepairShop functionality.

Table 3 Service query description

Service query	***Function***		***NonFunctional***	***Legal***	***ContextInfo***
Repair Shop Query	**RequiredPre**: CarBroken==T Weight=6 **RequiredPost**: HasAppointment==T Weight =6	**Parameters**: CarBroken:bool deposit:double CarType:string failure Type:string	Price = 45$/h Weight = 3 Client Rec. = 4 Weight = 3 Recommended by CAA Weight = 3	deposit = 280$ Weight = 4 Warranty= 3 Weight = 5 **PriceConditione**: CarType=toyota Weight = 6	membership==CAA (location,montreal)

A description of the service query content.

- *Dynamic selection:* The service provision framework should be designed to allow service requesters specify the requirements with the full knowledge that some service bindings may occur only at run time.
- *Dynamic composition:* With the increased number of services and the increased composition complexity, it is difficult to have all service compositions predefined in a static manner.
- *Context support:* Contextual information is essential at service publication, service query, service selection and planting, and service execution.
- *Semantic support:* Semantic information is essential at service specification, service query, and service composition.
- *Formal:* Formalism is necessary to 1) verify the interaction between services by making sure there are no incompatible behaviors between services in a composition, 2) achieve correct automatic composition by verifying that the composition satisfies the requirements of the requester, and 3) check the conformance of requester requirements and the contracts of the services being provided.
- *Negotiation support:* Each service requester has his own set of requirements. In many cases, none of the available services may fully match these requirements. The service provision framework should provide a mechanism to support the negotiation between service requesters and providers.
- *Nonfunctional and trust:* The consideration of nonfunctional and trustworthiness properties in service publication, discovery and ranking is essential.
- *Replanning support:* At run time, the contextual information of the service consumer and requester might change. The service provision framework should support a replanning process to generate a new plan that best satisfies the requirements in the new context.
- *Fault-tolerance:* If a service fails or becomes unavailable at run time, the service provision framework should recover from this failure by selecting alternative services.

Table 4 Related service publication

	Functional	**Nonfunctional and trust**	**Legal rules**	**Context**	**Formal**	**Verification support**	**Tool support**
UML-based	YES	SOME	SOME	NO	NO	NO	YES
SRML	YES	NO	SOME	NO	YES	YES	YES
SOADL	YES	SOME	NO	NO	YES	YES	YES
SOFM	YES	SOME	NO	NO	YES	YES	NO
WSDL	YES	NO	NO	NO	NO	NO	YES
OWL-S WSMO	YES	SOME	NO	NO	NO	NO	YES

A list of related service publication approaches.

Table 5 Related service discovery

	Dynamic selection	Dynamic composition	Context support	Semantic support	Formal	Negotiation support	Nonfunctional & trust	Replanning support	Fault-tolerance
SeGSeC	YES	YES	YES	YES	NO	NO	NO	YES	YES
eflow	YES	No	NO	SOME	NO	NO	NO	NO	YES
Self-serv	YES	NO	NO	NO	NO	NO	NO	NO	NO
SHOP2	YES	YES	SOME	YES	NO	NO	NO	NO	NO
SWORD	NO	YES	NO	SOME	NO	NO	NO	NO	NO
Argos	NO	YES	YES	YES	NO	NO	NO	NO	-
Composer	YES	NO	YES	YES	NO	NO	SOME	NO	-
FUSION	YES	YES	NO	NO	NO	NO	NO	YES	-
Protus	YES	YES	NO	NO	NO	NO	SOME	YES	YES
SPACE	YES	NO	NO	NO	YES	NO	NO	YES	YES
StarWSCop	YES	YES	NO	YES	NO	NO	YES	NO	YES
Meteor-s	YES	NO	NO	YES	NO	NO	YES	YES	-
SeCSE	YES	NO	NO	NO	NO	NO	YES	YES	YES
DynamiCos	YES	YES	NO	YES	YES	NO	NO	NO	NO
TSCN	YES	NO	NO	NO	YES	NO	NO	NO	NO

A list of related service discovery approaches.

The result of this comparison is presented in Table 5 and it shows the following:

1. With the exception of SWORD and Argos, all approaches support dynamic selection.
2. Dynamic composition is considered by almost half of the approaches. In most of these approaches AI planning techniques are used.
3. Contextual information is used by very few approaches. In these approaches, context is used to filter the services, but not to constrain the service contract. Hence, the relationship between the contract and context is not considered.
4. Semantic information using ontology is supported by almost half of the approaches. The use of ontology restrains the semantic support due to the complexity and difficulty of composing ontologies.
5. With the exception of three approaches, all remaining approaches are not formally based. This will limit their verification support.
6. None of the investigated approaches supports negotiation.
7. The support of nonfunctional and trustworthiness properties is very simple and limited.
8. Replanning is supported by almost half of the approaches. With the exception of protus, approaches that support replanning do not support dynamic composition and hence the replanning is manually performed.
9. Fault-tolerance is supported by only few approaches. A number of approaches such as Argos, Composer, FUSION and Meteor-s do not mention fault-tolerance. Hence, by default we consider that they do not support fault tolerance.

Conclusion and future work

We have presented *FrSeC* that supports the publication, discovery, and provision of services with context-dependent contracts. *FrSeC* is formally based and considers legal rules and contextual conditions during service provision. It also supports an adaptive rediscovery and reranking operations. We are currently working on a detailed service architecture extended with flexible contracts and trustworthiness guarantees. We are developing a set of tools and a process model in order to provide a platform in which service-oriented applications within the confines of *FrSeC* can be developed.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
The main contributions of NI are defining the *ConfiguredService* structure for service publication, defining the query structures for service discovery, and defining the service ranking algorithm. The major contribution of MM includes defining a service oriented architecture which influenced the formal framework *FrSeC*. VA has contributed to the formalization of the framework and issues related to adaptability. All authors participated in drafting and approving the paper manuscript.

Author details
[1]Department of Mathematics and Computer Science, Albany State University, Georgia, USA. [2]Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.

Received: 6 January 2012 Accepted: 28 December 2012
Published: 17 January 2013

References

1. Papazoglou MP (2008) Web services: principles and technology, First edition. Prentice Hall, England, UK
2. Dey AK (2001) Understanding and using context. Perso Ubiquitous Comput 5: 4–7
3. OSullivan J (2007) Towards a precise understanding of service properties. Phd thesis, Queensland University of Technology, Brisbane, Australia

4. Ibrahim N, Alagar VS, Mohammad M (2011) Managing and Delivering Trustworthy Context-Dependent Services. In: proceedings of the 2011 IEEE 8th International Conference on e-Business Engineering (ICEBE), Beijing China, pp 358–363
5. Ibrahim N, Alagar V, Mohammad M (2011) Specification and verification of context-dependent services. In: Kovács L, Pugliese R, Tiezzi F (eds) Proceedings of the 7th International Workshop on Automated Specification and Verification of Web Systems, Reykjavik Iceland. Volume 61 of EPTCS, pp 17–33
6. Ibrahim N, Mohammad M, Alagar V (2011) An architecture for managing and delivering trustworthy context-dependent services. In: Proceeding of the 8th IEEE International Conference on Services Computing, Washington, DC, USA, pp 737–738
7. Wan K (2006) Lucx: Lucid enriched with context. Phd thesis, Concordia University, Montreal, Canada
8. Mayer P, Schroeder A, Koch N (2008) MDD4SOA: Model-Driven Service Orchestration. In: EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference. IEEE Computer Society 2008, Washington, DC, USA, pp 203–212
9. Service oriented architecture Modeling Language (SoAML) (2008) Specification for the UML Profile and Metamodel for Services (UPMS). OMG Submission document: ad/2008-11-01. Available at http://www.omgwiki.org/SoaML/doku.php?id=specification
10. WSDL (2001) Web Services Description Language 1.1. W3C Note. March, http://www.w3.org/TR/wsdl
11. Martin D, Paolucci M, McIlraith S, Mark McDermott D, McGuinness D, Parsia B, Payne T, Sabou M, Solanki M, Srinivasan N, Sycara K (2004) Bringing Semantics to Web Services: The OWL-S Approach. In: First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), San Diego, California, USA, pp 243–277
12. Zaremba M, kerrigan M, Mocan A, Moran M (2006) Web services modeling ontology. In: Cardoso J, Sheth AP(eds) Semantic Web Services, Processes and Applications. Springer, pp 63–87
13. Jia X, Ying S, Zhang T, Cao H, Xie D (2007) A new architecture description language for service-oriented architecture. In: Sixth International Conference on Grid and Cooperative Computing (GCC 2007), Urumchi, Xinjiang, China, pp 96–103
14. Marino J, Rowley M (2009) Understanding SCA (Service Component Architecture). Person Education, Inc., Boston, MA, USA
15. Cao XX, Miao HK, Xu QG (2008) Modeling and refining the service-oriented requirement. In: TASE '08: Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering. IEEE Computer Society, Washington, DC, USA, pp 159–165
16. Fujii K, Suda T (2009) Semantics-based context-aware dynamic service composition. ACM Trans Autonomous Adaptive Syst 4(2): 1–31
17. Casati F, Ilnicki S, Jin Lj, Krishnamoorthy V, Shan MC (2000) Adaptive and dynamic service composition in eFlow. In: Proceedings of the 12th Int'l Conference on Advanced Info. Systems Engineering. Springer-Verlag, pp 13–31
18. Sheng QZ, Benatallah B, Dumas M, Mak EOY (2002) SELF-SERV: a platform for rapid composition of web services in a peer-to-peer environment. In: Proceedings of the 28th international conference on Very Large Data Bases, VLDB Endowment, Hong Kong, China, pp 1051–1054
19. Wu D, Parsia B, Sirin E, Hendler J, Nau D, Nau D (2003) Automating DAML-S web services composition using SHOP2. In: Proceedings of 2nd International Semantic Web Conference, Sanibel Island, Florida, USA, pp 195–210
20. Ponnekanti SR, Fox A (2002) SWORD: A developer toolkit for web service composition. In: Proceedings of the 11th International WWW Conference, Honolulu, Hawaii, USA. http://www2002.org/CDROM/alternate/786/index.html
21. Ambite JL, Weathers M (2005) Automatic composition of aggregation workflows for transportation modeling. In: Proceedings of the 2005 national conference on Digital government research, Digital Government Society of North America, Atlanta, GA, USA, pp 41–49
22. VanderMeer D, Datta A, Dutta K, Thomas H, Ramamritham K, Navathe SB (2003) FUSION: A system allowing dynamic web service composition and automatic execution. In: Proceedings of the IEEE Int. Conference on E-Commerce Technology, IEEE Computer Society, p 399
23. Ghandeharizadeh S, Knoblock C, Papadopoulos C, Shahabi C, Alwagait E, Ambite JL, Cai M, Chen CC, Pol P, Schmidt R, Song S, Thakkar S, Zhou R (2003) Proteus: A system for dynamically composing and intelligently executing web services. In: Proceedings of the 1st International Conference on Web Services, Las Vegas, NV, USA, pp 17–21
24. Jin C, Wu M, Ying J (2009) A Structure-based approach for dynamic services composition. J Software 4(8): 891–898
25. Sun H, Wang X, Zhou B, Zou1 P (2003) Research and implementation of dynamic web services composition. In: Zhou X, Jahnichen S, Xu M, Cao J (eds) Advanced Parallel Processing Technologies, 5th International Workshop, APPT 2003, Volume 2834 of Lecture Notes in Computer Science. Springer-Verlag, pp 457–466
26. Verma K, Gomadam K, Sheth AP, Miller JA, Wu Z (2005) The METEOR-S Approach for configuring and executing dynamic web processes. Technical report, LSDIS Lab, University of Georgia, Athens, Georgia
27. Penta MD, Bastida L, Sillitti A, Baresi L, Maiden N, Melideo M, Tilly M, Spanoudakis G, Cruz JG, Hutchinson J, Ripa G (2009) SeCSE–Service centric system engineering: An overview. In: Nitto ED, Sassen AM, Traverso P, Zwegers A (eds) At Your Service: Service-Oriented Computing from an EU Perspective. The MIT Press, Cambridge, Massachusetts, USA, pp 241–272
28. Silva E, Pires LF, van Sinderen M (2009) Supporting dynamic service composition at runtime based on end-user requirements. In: Proceedings of the 1st Workshop on User-generated Services (UGS2009) at the 7th International Joint Conference on Service Oriented Computing, (ICSOC 2009), Stockholm, Sweden, pp 20–30
29. Fan G, Yu H, Chen L, Liu D (2009) An approach to analyzing dynamic trustworthy service composition. In: Gómez-Pérez A, Yu Y, Ding Y (eds) The Semantic Web, Fourth Asian Conference, ASWC 2009, Shanghai, China, December 6-9, 2009. Proceedings, Volume 5926 of Lecture Notes in Computer Science. Springer, pp 261–275

doi:10.1186/2192-1962-3-1
Cite this article as: Ibrahim *et al.*: **Publishing and discovering context-dependent services.** *Human-centric Computing and Information Sciences* 2013 **3**:1.