A Flexible Service Selection for Executing Virtual Services

Nassim Laga, Emmanuel Bertin, Ivan Bedini, Noel Crespi, Benjamin Molina, and Zhenzhen Zhao

Abstract. With the adoption of a service-oriented paradigm on the Web, many software services are likely to fulfil similar functional needs of the end-users. We propose in this paper to aggregate functionally equivalent software services within one single virtual service, that is an association of a functionality, a graphical user interface (GUI), and a set of selection rules. When an end-user invokes such virtual service through its GUI to answer his/her functional need, the software service that better answer to the end-user selection policy is selected and executed, and the result is then rendered to the end-user through the GUI of the virtual service. A key innovation in this paper is the flexibility of the service selection policy we propose. First, each selection policy can refer to heterogeneous parameters (e.g. service price, end-user location, and QoS). Second, additional parameters can be added to an existing or new policy with little investment. Third, it is the end-user who defines which selection policy to apply during the selection process, thanks to the GUI element added as part of the virtual service design. This approach has been validated by designing, implementing, and testing an end-to-end architecture, including the implementation of several virtual services, and considering several software services available today in the Web.

Key words. Service Discovery; Service Selection, Virtual Service; Service Selector; Service Aggregation; Marketplace.

N. Laga・E. Bertin
Orange Labs France,
42 Rue des coutures, 14000, Caen, France

N. Laga
e-mail: nassim.laga@orange-ftgroup.com
E. Bertin
e-mail: emmanuel.bertin@orange-ftgroup.com

I. Bedini
Alcatel-Lucent, Bell Labs Ireland,
Blanchardstown Industrial Park,
Blanchardstown, Dublin 15,
e-mail: ivan.bedini@alcatel-lucent.com

N. Crespi・Z. Zhao
Institut Telecom, Telecom SudParis,
Rue Charles Fourier, 91011, Evry Cedex, France

N. Crespi
Email: noel.crespi@it-sudparis.eu
Z. Zhao
e-mail: zhenzhen.zhao@it-sudparis.eu

B. Molina
Universitat Politècnica de València,
Camino de Vera, s/n 46022 Valencia
Valencia, Spain

1 Introduction

The transition from Web 1.0 to Web 2.0 is characterized by an increasing number of services and by a more and more user-centered design (rich user interfaces, user self service, and user service creation) [1]. Following the lead of these characteristics, the concept of a "services marketplace" has emerged [2, 3, 4, 5, 6, and 7]. It provides end-users with a common place where services are published, discovered, and in some cases created and hosted. Current marketplaces embed a huge number of services, and many services can fulfill similar functional needs. Therefore, virtual service and service selector, already introduced in different development environments such as IBM WebSphere and Microsoft .Net, are pertinent features in this context. A virtual service is a mediation endpoint between the service requestor and the providers of actual services (real implementations of services). The service selector is then in charge of selecting the actual service to be executed for a given virtual service, according to the needed functionality and the selection criteria. In this paper, we study the application of such concepts from end-users perspectives, in the marketplace context. We study the requirements for such goal, and propose our own solution.

As several services may have exactly the same functional signature, selecting a service only by matching the end-user's goal and the functionality provided by the service is no longer sufficient. Currently, there is a significant body of research on service selection in software engineering. The first approaches focused on goal-based discovery and selection using semantic technologies [8, 9, 10, and 11], but researchers have noted the need for more criteria in the discovery process. Thus, new approaches that consider non-functional parameters such as the Quality of Service (QoS) and the end-user's context have emerged [12, 13, 14, 15, 16, 17, 18, 19, 20, 21 and 22]. The non-functional parameters may be static, such as price, or dynamic (known only at runtime), such as the location and the presence of the end-user.

Two important limitations can be noticed in existing service selection approaches. First, they are based on a limited set of parameters that can be used in the selection process. Adding a new parameter usually requires rethinking the selection process implementation. Second, they lack of user centricity design. Indeed, while some end-users may want to select the service that minimizes the price, others may want to select the service that is most suited to their context, while some others may want to select the service according to the language they speak (and some will want a mix of criteria to evaluate). It is important to allow the end-user to define the selection policy to apply during the selection process.

We propose in this paper such a mechanism; a service selection mechanism built on the top of a marketplace of services, where the end-user defines the criteria to apply for selecting the actual service to execute for a given virtual service. For this purpose, we design a virtual service as an association of a functionality, a Graphical User Interface (GUI), and a set of selection rules that could be applied in the selection process. This association is defined by the marketplace provider. When, a virtual service is invoked, the GUI is displayed. It enables the end-user to define the policy to use in the selection process. It also displays a form that enables the end-user to provide the inputs required for the execution of the functionality. Once the selection policy is defined and configured, and the form is filled, the marketplace selects the services that satisfy the policy

defined by the end-user. Finally, the selected service is executed and the results are displayed within the GUI.

There are two important contributions in this paper. First, it is the end-user who defines the policy to apply. Second, in addition of being able to refer to static parameters such as service price, the marketplace administrators can also refer to any dynamic parameter generated by a service in the marketplace (e.g. end-user location, end-user presence, network traffic…etc), and that in a seamless way. This makes the policy language very flexible. The marketplace administrators can easily add new rules that refer to new parameters.

This paper is organized as follows. In section 2, we first present the requirements for a virtual service execution by end-users; and second review the state of the art related to service selection field and its limitations. In section 3, we summarize our proposal. In section 4, we present the end-to-end architecture for a user-centric service selection mechanism we propose. We focus essentially on three aspects: the specification of the virtual service, the specification of actual services, and the specification of our selection rules language. We detail the implementation of our approach and validate it by prototyping the different virtual services and the list of software services considered in section 5. In section 6 we conclude with a summary and discuss the potential application of our proposal in other research domains.

2 Requirements and Related Work

This section begins by defining the requirements for using virtual services by end-users. The two categories of related work we have identified, service selection based on static parameters, and service selection based on dynamic parameters, are then reviewed.

2.1 Requirements

This paper proposes a novel approach for executing a virtual service in the marketplace context. For this purpose, we have identified a set of requirements. The first requirement is related to the heterogeneity of parameters that can be considered in a selection policy. Service price, QoS parameters (bandwidth, response time, availability, security...etc), service reputation, end-user location and presence are all examples of parameters that can be pertinent in the service selection. Therefore, as the service selection mechanism can not anticipate all these parameters, it is important to provide a seamless approach for adding new ones.

The second requirement is related to the heterogeneity of end-users' selection policies. When a virtual service is invoked, the marketplace must be able to select the best actual service to execute for the needed functionality. However, the criteria that define the best service are likely to be different from an end-user to another. Let us take the execution of a Send SMS virtual service. Some end-users may choose the cheaper service, while others may choose the best ranked service for the same functionality. It is therefore important to enable end-users to define their own selection policy for a given virtual service invocation. In other words, it is important to design a user-centric service selection mechanism, where end-users can define their own selection policy.

The third requirement is related to the intuitiveness of the GUI that enables the end-user to define the selection policy for a given functionality. Script languages must be transparent to the end-users.

2.2 Related Work

Various mechanisms may facilitate the selection of the best available service according to a user's goal. In the current Web environment, several services provide similar functionalities, and thus aim to fulfil similar goals. Therefore, it is important to investigate other criteria to enable the selection of the most appropriate service among functionally equivalent ones. In this section we classify such criteria into those that refer to static parameters and those that refer to dynamic parameters. Basically, we define static parameters as a parameter whose value does not depend on the runtime context. The service price and negotiated bandwidth are examples of such a parameter. Dynamic parameters are in contrast parameters whose value is known only at runtime. User location, user presence, and actual network traffic are typical examples of such a parameter.

2.2.1 Service Selection based on Static Parameters only

In order to select a single service among functionally equivalent ones, selection mechanisms based on non-functional parameters, mainly SLA- (Service Level Agreement) related parameters (e.g. price, availability, and response time), have emerged [17 and 18]. This selection includes several issues [23]:

- Identifying the most relevant non-functional parameters;
- Incorporating the non-functional parameters into the service description; and
- Aggregating these different non-functional parameters into a single quantitative value when selecting a service.

Non-functional parameters may be classified into quantitative and qualitative parameters [24]. Quantitative parameters include, for example, the price of a service, service availability, bandwidth, loss rate, and response time. Some qualitative parameters are the quality of the user interface, and the quality of a code; these are usually related to the end-user service perception [25].

Once we have identified the non-functional parameters to be considered in the service selection, these properties should be integrated into and be able to be retrieved from the service registries. To this end, several description languages and UDDI (Universal Description, Discovery and Integration) extensions have been proposed (HQML [26], QML [27], OWL-S [28], and [29]). We do not discuss the performances of these languages in this paper, but instead focus on the process of selecting services.

In [12] and [13], this process of selecting a service among functionally equivalent ones has been called "horizontal composition" -- in contrast to "vertical composition" which aims at finding the best combination of "tasks" to respond to a consumer's needs (these "tasks" are also called "abstract web services" [12]). These authors have focused on selecting a service that satisfies a set

of constraints (expressed as a condition referring to non-functional parameters) while optimizing a given linear objective function:

$$f(x1,...,xn) = c1.x1 + ... + cn.xn,$$

where $x1,...,xn$ are the considered and normalized non-functional parameters, and $c1,...,cn$ are the corresponding weights of each parameter, respectively. Thus, the importance of a parameter in the service selection is expressed by its assigned weight.

2.2.2 Service Selection based on Static and Dynamic Parameters

Another approach for choosing a service among others that offer the same functionalities is to take into account runtime parameters such as the context of the end-user (e.g. end-user location, or end-user presence status)[21], the actual quality of service parameters [19], or the reputation of services [21, 30, 31 and 32]. This kind of service selection has been investigated in many studies [14, 15, 16, 33, and 34]. In [15] for example, a framework called eFlow was proposed; a service composition framework that supports automatic adaptation according to the composite service parameters. Indeed, this framework enables the service consumer (a service composer) to express his/her needs through service nodes, associating a selection rule to each one that refers to the runtime values of the current parameters of the composite service.

Moreover, in [14] researchers introduced an approach for context-aware services composition, one result has been the MAIS (Multi-channel Adaptive Information System) project [35]. Their contribution consists in having improved the current SOA platform with context awareness capabilities. These researchers propose a new template to publish services; a template that includes a description of accessing channels, quality of service, and other non-functional parameters. When an entity wants to access a service, it provides a request that includes requestor context information such as the accessing channel and the device. Thus, the platform is put in charge of discovering the service that matches both the functional needs of the requestor and his/her context. A very similar approach is introduced in [16], where context information is published along with services, and requestor context may be included in the request to enable the platform to select the service that best matches that context.

Finally, extensive work has been done on the context aware systems, such as location-aware systems, context-managing frameworks and context-aware service composition architecture [14, 16 and 34].

2.3 Limitations

Table 1 presents the limitations of the related work regarding the requirements we have identified for using the concept of virtual service by end-users.

Table 1 Related work limitations.

Requirement	Related Work limitation
Heterogeneity of parameters that can be considered in a selection policy.	Existing solutions consider only a limited set of parameters; the solutions that consider the QoS parameters, for instance, do not consider the end-user context parameters such as location and presence. In addition, the issue of how to integrate a new selection parameter to an existing solution is not discussed.
Heterogeneity of end-users' selection policies.	Existing solutions are conceived from developer perspectives. More precisely, they are designed from service composers' perspectives, where a chaining of virtual services is defined, and for each virtual service, a set of selection criteria is associated. The solutions presented in [14] and [16], for instance, require the service requestor to first detect the context of the end-user and then invoke the service platform with that information. This requires service integrators to master the context enabler API and to manage cross format adaptation between what was generated by the enabler and the platform format. Consequently, end-users without computing skills cannot define their own selection criteria.
Intuitiveness of the GUI for defining a selection policy.	Existing solutions have not considered a GUI element for creating selection policy.

3 Overall Approach

The goal of this paper is to propose an end-to-end architecture and implementation for a virtual service execution by end-users. We present two innovations in this paper. First, we enable end-users to specify the service selection policy they want to apply during the execution of a virtual service. A selection policy is defined as a combination of selection rules. For each virtual service we associate a set of applicable selection rules. Then, using a GUI element added as part of the virtual service, the end-user chooses which rule(s) to apply, and by consequence defines the selection policy. Second, the language we propose for defining a selection rule can refer to any static parameter and to any dynamic parameter which can be generated by a service present in the marketplace. Therefore, to integrate a new parameter in our solution, the administrator needs only to publish the service that generates it to the marketplace. For example, to add end-user location as a parameter which can be referred to in a selection rule, the administrator should only publish the location service to the marketplace.

Figure 1 illustrates the overall architecture of the proposed solution. We distinguish three main concepts. The first one is "Actual Service". It is a computer software system that exposes a software interface (typically provided by third parties), with methods that can be defined by a generic goal (e.g. get route, send message, translate news…); Web Services [36] or RESTful services [37] are common examples. The Flickr REST photos search API or the Telefonica SOAP

send MMS API are actual services. In our proposal, actual services are published to the marketplace by providing a description. This description would include a functional view (goal, inputs and outputs) as well as a non-functional view (e.g. QoS, price, usage limitations) of the service.

The second important concept is “Virtual Service”. It is an association of a goal concept (e.g. send MMS, send email, search pictures), a set of selection rules (built following a rule language we propose), and a GUI. The GUI enables the end-user to enter the inputs expected by the functionality, define the policy to apply by choosing the selection rules, invoke the service selector to select the best actual service, and execute the selected actual service.

Finally, the third important concept is the service selector, implemented in our solution by the *Interpreter* component of Figure 1. It is in charge selecting the best actual service to execute for a given virtual service. It receives as inputs the functionality (goal concept) of the virtual service, the inputs provided by the end-user, and the selection policy defined by the end-user (a set of selection rules).

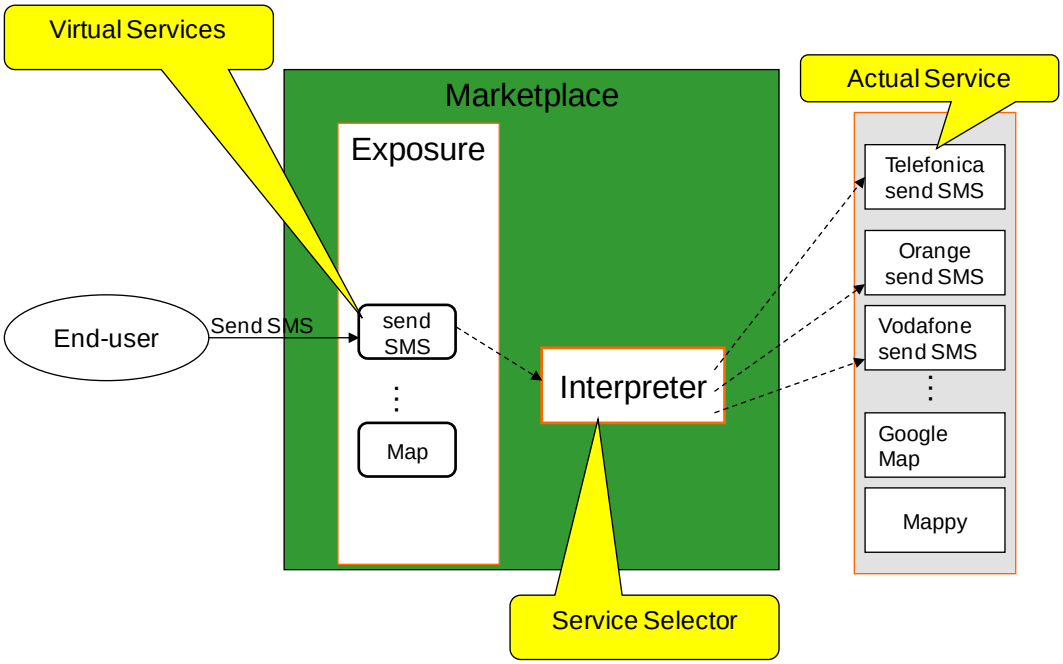

Figure 1 Virtual services exposure.

4 Framework Design

In this section we will detail our proposed design of an end-to-end architecture to enable a user-centric virtual service execution. We first show the different roles involved in our marketplace with a use case diagram, and then we detail how each use case is realized.

4.1 Framework Use Case Diagram

The components we define to enable a user-centric execution of virtual services are based on service marketplace. Figure 2 is a high-level view of the different roles and use cases involved in this mechanism. The administrator of the marketplace is in charge of specifying the virtual service;

this includes the design of the GUI, the definition of a set of selection rules that could be applied, and the association of these selection rules with the GUI and the goal concept (functionality).

The service providers publish their actual services into the marketplace by filling out a form designed to capture the functional and non-functional parameters of the actual service being published. The publication process may require an adaptation of the actual service on the part of the provider to comply with a functionality signature.

Finally, the end-user can use a virtual service, which comprises the definition of the selection policy, filling the GUI form (required inputs), the selection of the best actual service (by the marketplace), and the validation and invocation of the selected actual service.

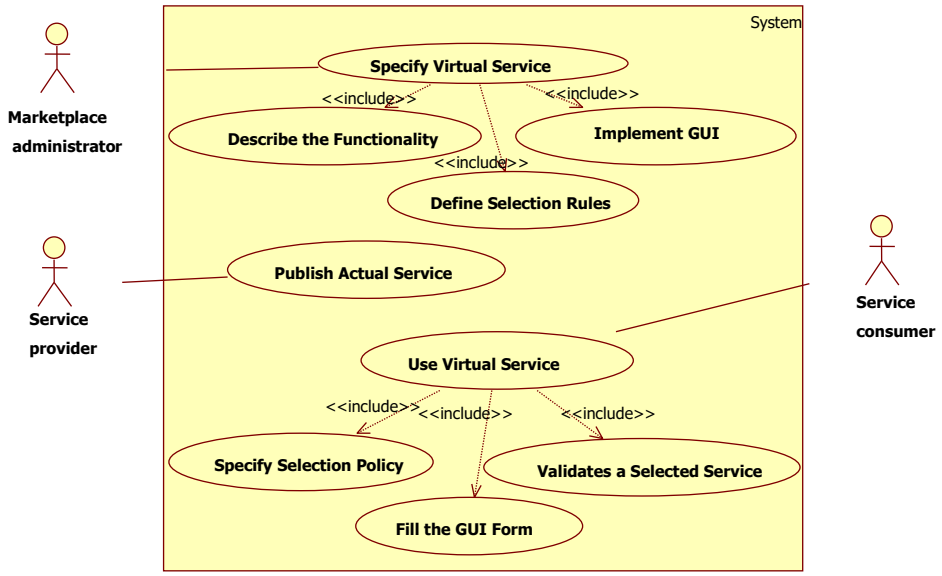

Figure 2 Framework use case diagram.

4.2 Specifying a Virtual Service

In this section we detail the concept of the virtual service and design the generic model for specifying the selection rules. These selection rules are intrinsic elements to the concept of virtual service. Figure 3 summarizes the whole model and highlights the virtual service section. It shows the different relationships among the various concepts.

A virtual service is basically an association of a functionality description, a GUI (virtual service front-end), and a set of selection rules that can be applied during the selection process on the functionality. The following subsections describe each part.

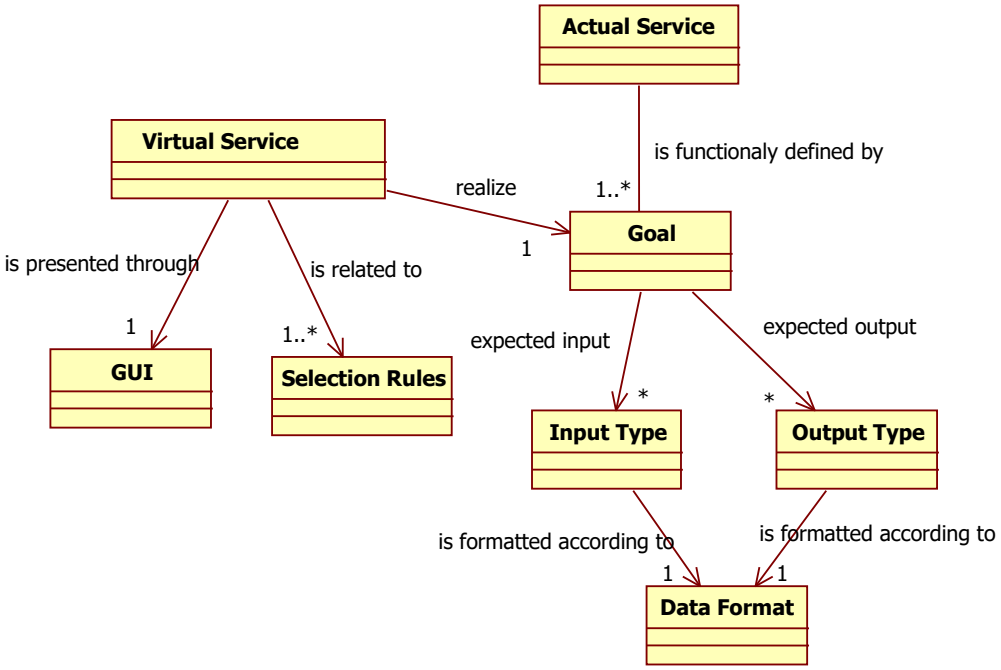

Figure 3 Service related concepts.

4.2.1 Functionality Description

The functionality is described through a goal concept that is used to discover the corresponding actual services. Goals are formally defined via semantic tags. Each goal is modeled through a unique tag, the type of mandatory inputs it expects, and the type of outputs it generates. The input types, as well as the output types, are associated to a data format. For example, an SMS sending functionality would be modeled with a unique tag "*send_SMS*", which expects a mobile phone number ("*mobile_phone_number*") and a text message ("*text_message*") as its input parameters, and expects the acknowledgement of receipt ("*ack*") as its output parameters.

4.2.2 Selection Rules Definition

To select the best available actual service at runtime, we also need mechanisms to assess actual services and decide which one to select. We propose here a user-centric approach in the selection process; an approach in which the end-users themselves specify the selection policy to apply in the selection process. This requires a generic model, in which both static and dynamic parameters can be considered on the one hand, and where new selection criteria can be added easily on the other hand. In this section we design this model, which is also summarized in Figure 4 (the model is shown focused on the selection rule portion).

A selection policy is defined as a set of selection rules. There are two types of rules:

- Constraint rules, designed to remove services that do not fulfil a list of constraints. The result of a rule evaluation is either a true or false value. Constraint rules enable the marketplace provider to specify conditions that the selected service must satisfy. For instance, if we consider an SMS sending functionality, a constraint rule could be

formulated as follows: select actual services whose home network location is the same as the location of the recipient.

- Objective rules, structured so as to rank an actual service from the perspective of a given objective. The result of a rule here is a quantitative value that enables the classification of different actual services. Objective rules could include, for example, the price optimization of a selected service. This can also be a linear objective function that refers to several parameters such as price, bandwidth, and/or reputation.

Both constraint rules and objective rules indirectly refer to static parameters and/or dynamic parameters. Static parameters are those whose value is known before runtime; service price and end-user preferences are typical examples. They are usually provided when the service is published to the marketplace. When such a parameter is referenced in a rule, the engine invokes the knowledge base component, which is a component that accesses the marketplace database to retrieve static parameters. Dynamic parameters are those whose value is known only at runtime. These parameters are usually the results of the execution of other services, such as presence and location. One of important innovations in this paper is the ability of the rule engine we propose to get the value of dynamic parameters. Indeed, when a reference to such a parameter is found in a rule, the engine invokes the corresponding service to get its current value. Consequently, to add a new parameter to the selection mechanism, we only need to create the service that generates it.

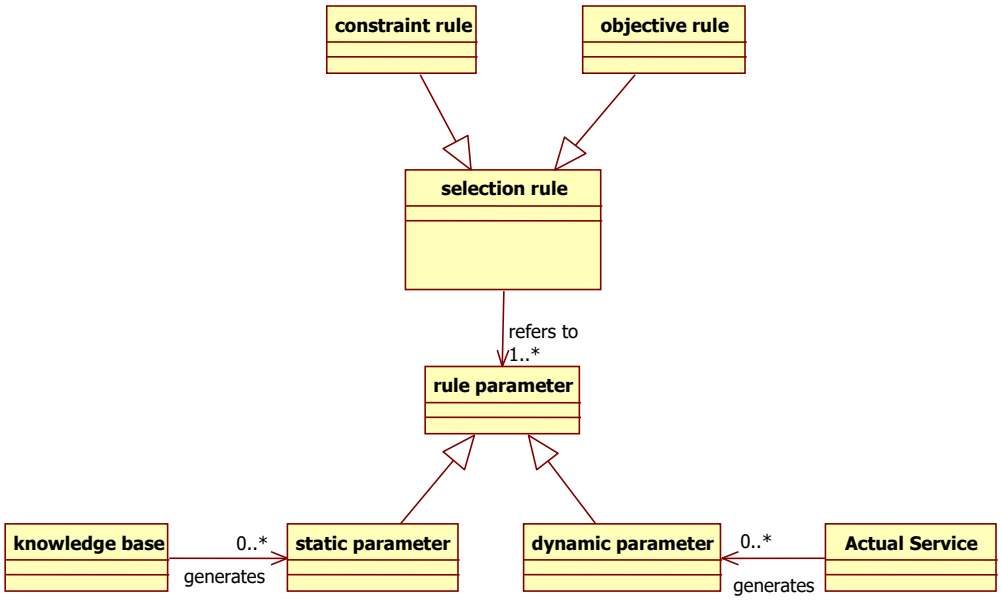

Figure 4 Selection rules model.

4.2.3 GUI Implementation

The virtual service also includes a GUI that enables the end-user to use and take advantage of the functionality and the associated selection rules and selection mechanism we define. It represents the virtual service front-end. It displays the GUI elements that enable the end-user to enter the inputs required by the functionality; it enables the end-user to define the selection policy by

choosing the rules to apply during the selection process; it invokes the selection mechanism; and then it executes the selected actual service and displays the result to the end-user.

The GUI must fulfill four conditions. First, it must implement the form that enables the end-user to provide the inputs needed by the functionality. This form is different from a virtual service to another as it depends on the required inputs of the corresponding functionality. Second, the GUI must enable the end-user to define the selection policy to apply by choosing/configuring the selection rules. Third, after validation, the GUI must invoke the selection mechanism implemented by a component called *Interpreter*. The *Interpreter* must receive the functionality tag of the virtual service, the selection rules (constraint rules and objective rules) chosen by the end-user, and the input values provided by the end-user as its input parameters. The Interpreter responds with a set of selected services. Fourth, the GUI must prompt the list of selected services to the end-user, enables him/her to select one of them, execute it, and display the execution results.

4.3 Actual Service Publication

Figure 5 depicts our model, emphasizing the actual service section. It shows the different information that must be provided by service providers when registering a new actual service in the marketplace.

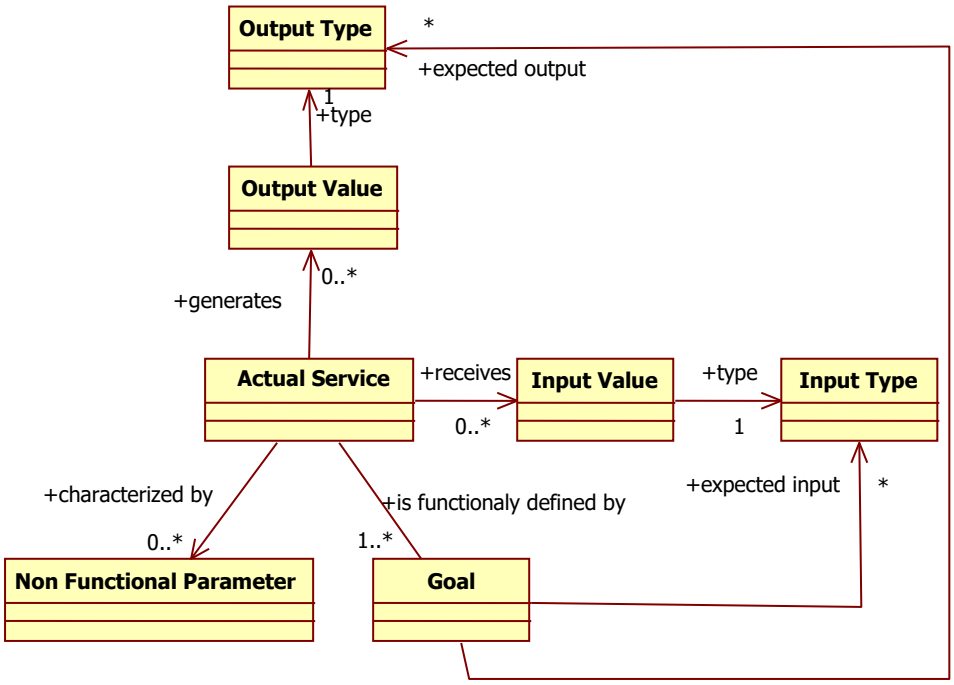

Figure 5 Actual service specification.

In summary, service providers must specify the following:

- The goal(s) of the actual service;
- The inputs and outputs of the actual service, referring to the inputs and outputs linked to the service goal(s);
- The non-functional parameters associated with the actual service; and

- Optionally, a specific interface, implemented to support interaction with the virtual service. This interface is required when the format of inputs and outputs of a specific service are different from the format of the inputs and outputs of the corresponding goal. In other words, this interface ensures the grounding compatibility of the actual service with the platform.

To avoid unusable free text values, these elements should be expressed through a predefined list of semantic tags. The marketplace is in charge of providing such a list. This task is performed manually by the administrator, who maps and adds new concepts as new services and functionalities appear.

4.4 Interpreter Component Design

The *Interpreter* component is one central component of our mechanism. It is the service selector of our platform. It assesses existing actual services regarding selection rules (policy) chosen by the end-user. Thus, it receives as input parameters the needed functionality, the selection rules to apply, and the input values provided by the end-user, and it generates a list of selected services. As we illustrate in Figure 6, the first action carried out by the *Interpreter* component is the discovery of all available actual services that perform the received functionality. Thereafter, the discovered services are filtered according to a set of constraint rules. Each constraint rule may refer to static parameters, dynamic parameters, and the inputs provided by the end-user. The static parameters are referenced through the knowledge base component (e.g. service prices, QoS parameters); the dynamic parameters are referenced through the corresponding actual service (e.g. invocation of localization actual service to get an end-user location parameter); and the inputs are referenced through the corresponding tag. Once all constraint rules have been applied and a set of actual services selected, the *Interpreter* evaluates the objective rule(s) if present. This objective rule may refer to static parameters, dynamic parameters, and/or the input values provided by the end-user. At the end, a set of actual services is selected; services that satisfy the constraint rules and optimize the objective rule. This list of selected actual services is sent back to the requestor (e.g. the GUI).

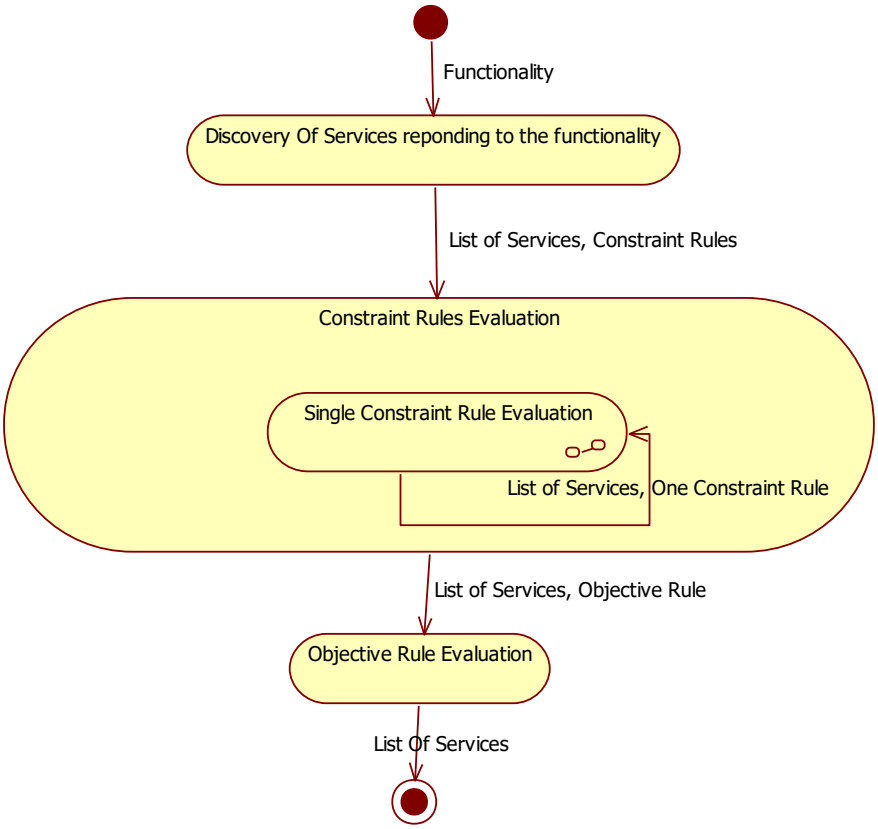

Figure 6 Service selection algorithm.

5 Implementation and Validation

In order to validate the different concepts and components we have introduced, we first detail the implementation of our contributions. We then show their pertinence by detailing the list of virtual services we implement and the list of actual services we consider. To demonstrate the efficiency of using virtual services, we show the scalability of the *Interpreter* component in terms of response time.

5.1 Implementation

The main contributions we introduce in this paper are a generic model for specifying service selection rules in the context of a marketplace of services, and the concept of virtual service. After detailing the implementation of the rule language with the *Interpreter*, we show in the following subsections how a virtual service is implemented by the marketplace administrator.

5.1.1 Rule Language and the Interpreter Component

In order to implement the generic model for specifying selection rules we define a language whose finite state machine diagram is depicted in Figure 7. The grey states are legacy final states, which means, for example, that a rule is considered complete when we reach state S2 or S3 (Figure 7.1). Each rule could be an objective or a constraint rule as depicted in Figure 7.1. Objective rules

necessarily contain an optimization operator (e.g. Max or Min) and a function to optimize. Constraint rules are simple conditions that are evaluated to true or false (Figure 7.1). Each condition includes one comparison statement between two functions (Figures 7.2 and 7.3). A function, referred to from both constraint rules and objectives rules, contains mainly numbers, variables, and operations. There are three types of variables:

a. Dynamic Parameters

A dynamic parameter must be resolved at the runtime. Indeed, its value differs from an end-user to another and instant t and t'. It is computed by invocating the corresponding actual service at the runtime. The service to invoke is directly specified in the rule. For example, the selection rule hereafter selects an actual service according to the end-user location. End-user location is a dynamic parameter known only at runtime. It is generated by the context service referred to within the rule.

$$\$Context(identifier: knowledge.phoneNumber).location.country = \\ = selectedService.country;$$

The keyword *$Context* enables the Interpreter to identify the actual service to invoke to get the end-user location. $knowledge.phoneNumber$ key word refers to the input values provided by the end-user. $selectedService.country$ key word refers to the country covered by an actual service. To be selected. $selectedService.country$ must be equal to the end-user location.

b. Static Parameters

Static parameters are also resolved at the runtime by the *Interpreter* component. Static parameters are related to a service. Therefore, they are referred to using "$selectedService.parameterTag$" key word. For example, "$selectedService.price$ refers to the price of the service being assessed regarding the rule. When such kind of parameter is present in the rule, the *Interpreter* component retrieves the actual value within the database.

c. Input Values

Input values provided by an end-user to execute a virtual service can also be used in as criteria in the service selection process. They are referred to through $knowledge.inputTag$ key word. For example, $knowledge.phoneNumber$ refers to the phone number parameter needed as an input for the virtual service being executed. These parameters are natively known by the *Interpreter* component as they are received as input parameter for the *Interpreter*

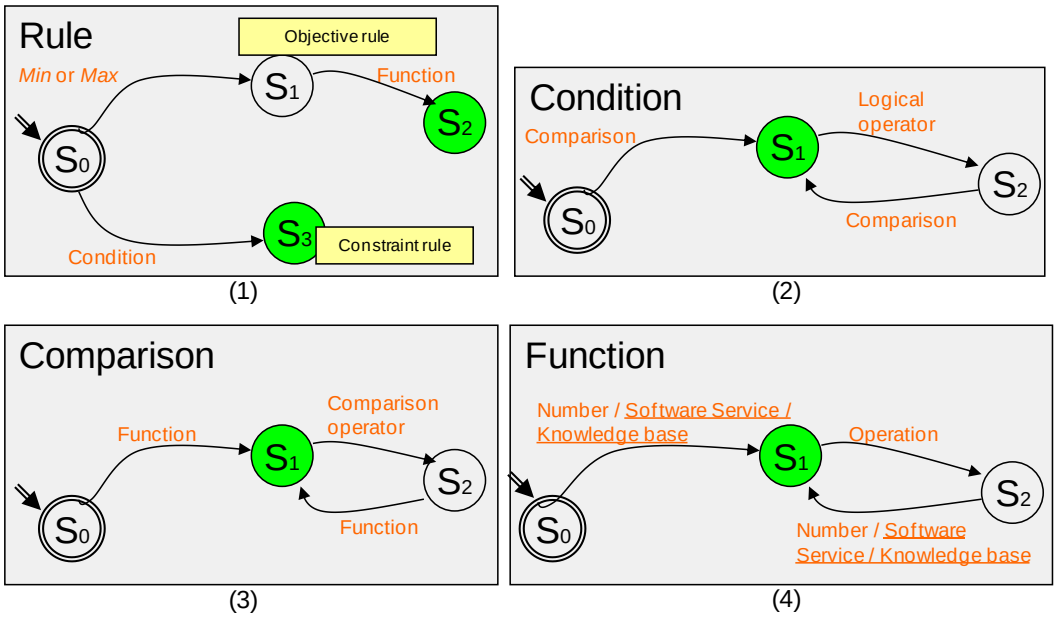

Figure 7 Finite state machine diagram of the selection rules language.

To interpret such a language, we used LEX[1]/YACC[2] tools to generate a compiler and an evaluator for the grammar, as illustrated in Figure 7. More precisely, we used JLex[3] and CUP[4] Java libraries. Basically, these tools receive as input the finite state machine diagram and automatically generate the corresponding Java code for a compiler and for an evaluator. The generated code is then integrated as part of the *Interpreter* component. The *Interpreter* component is implemented as a Java Servlet. It is exposed through an Http API; it is accessible through a URL using the Get or Post method. Table 2 shows the different parameters that must be transmitted.

Table 2 *Interpreter* invocation details.

Parameter name	Value example	Description
user_id	*alice@host.com*	The identifier of the virtual service consumer (end-user).
format	*Json (or xml)*	Specifies the format of the output (list of selected services) of the *Interpreter*. Current supported formats are JSON and XML.
functionality	*Send_SMS*	The functionality of the virtual service.
constraint_rules	*[“$Context(identifier:knowledge.phoneNumber).location.country*	The list of constraint rules to apply during the selection process.

[1] Lex tutorial: http://dinosaur.compilertools.net/lex/index.html, accessed on August 12th, 2010.
[2] Yacc tutorial: http://dinosaur.compilertools.net/yacc/index.html, accessed on August 12th, 2010.
[3] JLex: www.cs.princeton.edu/~appel/modern/java/JLex/, accessed on August 12th, 2010.
[4] CUP: www.cs.princeton.edu/~appel/modern/java/CUP/, accessed on August 12th, 2010.

	== selectedService.country;", ...]	
objective_rule	*MIN(selectedService.price);*	The objective rule to apply during the selection.
Input values list	*[callerPhoneNumber:0123456789]*	The input values provided by the end-user for the execution of the needed functionality.

5.1.2. Virtual Service Implementation

To create a virtual service three steps must be followed by the marketplace administrator. First, the target functionality tag of the virtual service must be identified (e.g. Send_SMS). Second, the administrator must create and associate a set of selection rules that could be applied for that virtual service. Third, he/she must create the GUI that enables the end-user to execute the virtual service. Each step is discussed hereafter. Figure 8 and 9 summarize the three steps.

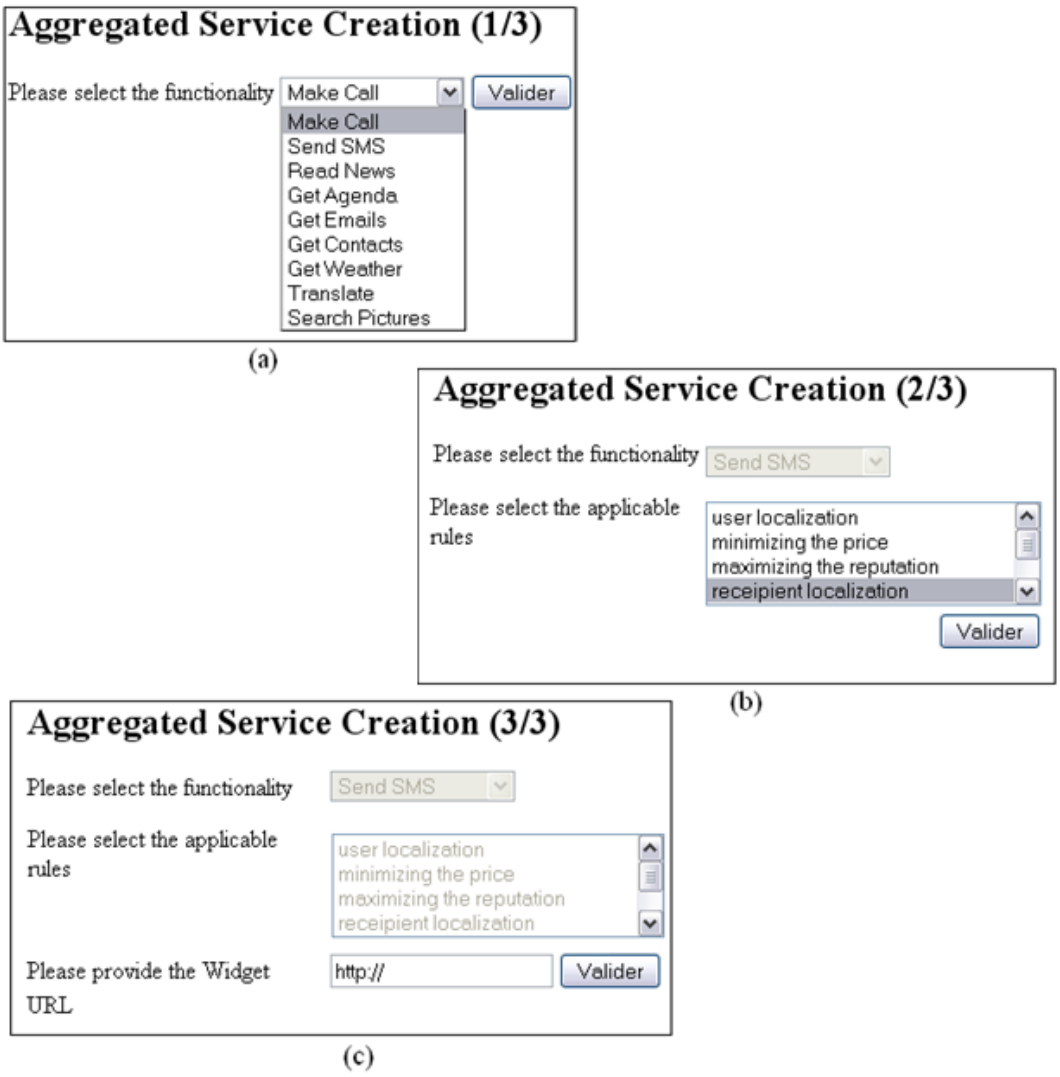

Figure 8 Virtual service creation.

a. Identifying Functionality

In its current implementation, the functionalities, the inputs they expect, and the outputs they generate are defined as list of RDF (Resource Description Framework) triplets. A virtual service can be created through a Web page we implement (Figure 8.a). This Web page displays the list of available functionalities extracted from the RDF graph. To start the creation of a virtual service, the marketplace administrator must select one of those functionalities (e.g. *Send_SMS* functionalities). In the current implementation we used MySQL database to store the list of virtual services.

b. Associating Selection Rules

The second step to be performed by the administrator of the marketplace is associating selection rules to the virtual service to be created. In our implementation, this is performed through a Web page displayed just after the previous step (see Figure 8.b). This Web page displays the list of available rules, extracted from the database. The administrator has just to choose those to associate to the virtual service and validate. A new rule can be added to the database as illustrated in Figure 8. In the current implementation we used MySQL database to store the rules and the associations with different virtual services.

c. Creating the GUI

The GUI provides the end-user with the capability of visualizing the selection rules, applying some of them (defining his/her own selection policy), entering the required inputs, viewing the service selection results, and finally invoking one of the selected services. It is implemented using Web standards (XHTML, JavaScript, and CSS). The URL of the GUI is also provided in the virtual service definition process as illustrated in Figure 8.c.

Figure 9 is a screenshot of a *send_SMS* virtual service GUI. Our implementation is based on Widgets to encourage lightweight composition; a new composition approach trend as detailed in [38, 39, and 40]. Three modes must be implemented in the Widget: the configuration mode, the view mode, and the selection mode.

The configuration mode contains the selection rules. This enables saving the chosen selection rules for further use by the same Widget (to avoid having the end-user reconfigure the selection rules each time he/she accesses the same virtual service). The constraint rules are displayed as check-boxes. The end-user can choose several of constraint rules to apply during the selection process. Some constraint rules can be configurable (e.g. Price). The objective rules however are displayed as radio-boxes, because only one objective rule can be applied. Nevertheless, each objective rule can refer to several parameters (e.g. in linear objective function). The combination of a set of constraint rules and an objective rule defines the selection policy of the end-user.

The view mode contains the form that enables the end-user to provide the required inputs. As the end-user enters the required inputs, AJAX requests are sent to the *Interpreter* component. It responds with a JSON array containing the selected services. This array is displayed in the selection mode area for validation by the end-user. This area is updated each time the Widget receives a new selected services array. The AJAX requests contain the selection policy (set of selection rules) and current inputs provided by the end-user.

Once a service is selected and validated by the user, it is invoked using AJAX requests. In the current implementation, each actual service when invoked, respond with a GUI defined using Web

technologies (XHTML, JavaScript, and CSS) and displayed directly within the view mode of the virtual service Widget.

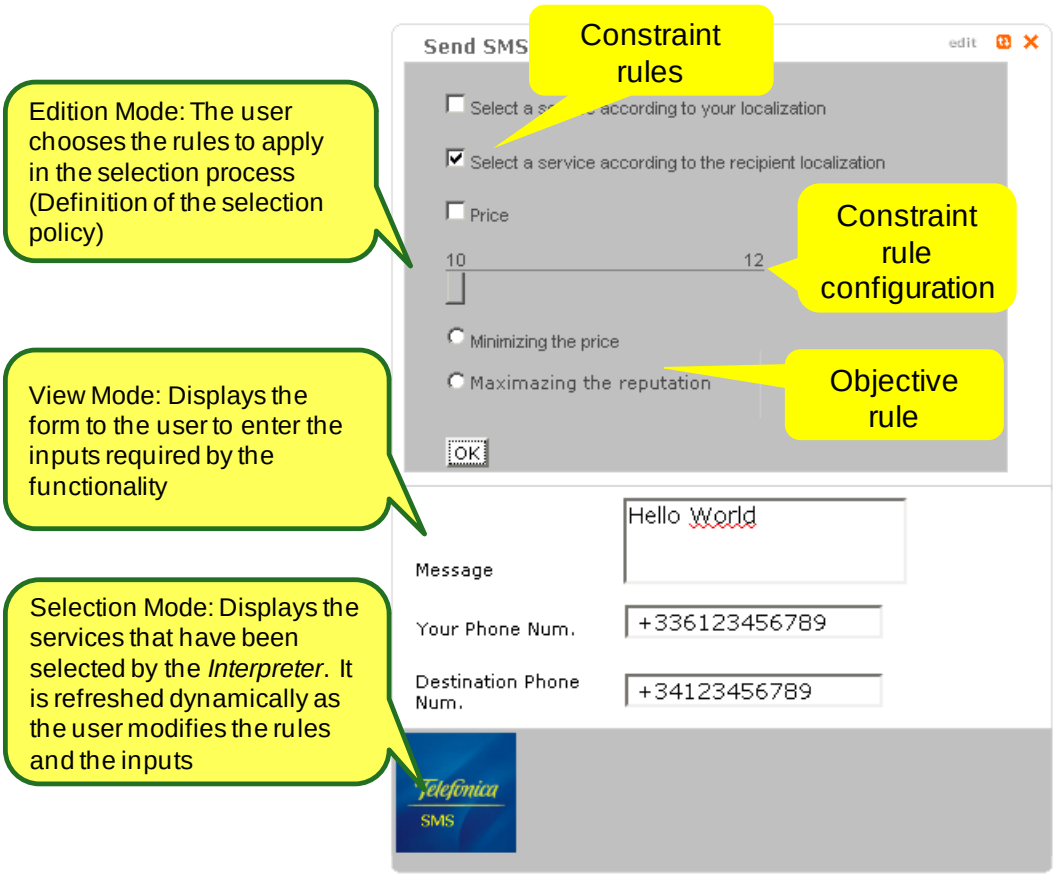

Figure 9 Illustration of a Send SMS virtual service.

5.2. Scenario and Validation

In this section, we validate our implementation of the virtual service concept. We first summarize the scenario of use of our implementation. Then, we detail usage feedback gathered from the different demonstration we made internally and externally to France Telecom – Orange. Finally, we validate our implementation against the requirements we have defined in this paper.

5.2.1 Scenario of Use

To demonstrate the usefulness of the mechanisms we introduce in this paper, we detail in Table 3 the different virtual services we have implemented. We also enumerate the different actual services we have considered; some of which are available on the Web, while others are private services used within France Telecom – Orange. Figure 9 shows an end-user environment that makes use of various virtual services.

Table 3 Virtual services and actual services list.

Virtual services	Actual services

Functionality	Rules	
SendSMS	- Selection according to recipient location. - Selection according to sender (end-user) location. - Selection according to the price per SMS. - Minimizing the price (objective rule). - Maximizing the reputation (objective rule).	- Orange Partner Send SMS API (http://www.orangepartner.com). - Telefonica Send SMS API (http://open.movilforum.com)
GetCalendar	- Selection according to the activity context of the end-user (work/home).	- Google Calendar. - Microsoft Exchange Calendar (accessible only within Orange private network)
GetEmails	- Selection according to the activity context of the end-user (work/home).	- Gmail. - Microsoft Exchange emails (accessible only within Orange private network).
GetContacts	- Selection according to the activity context of the end-user (work/home).	- Gmail contacts. - Microsoft Exchange contacts (accessible only within Orange private network).
GetWeather	- Selection according to end-user preferences. - Selection according to end-user location. - Selection according to service price.	- Meteo France - WeatherForcast (http://www.webservicex.net)
SearchPictures	- Selection according to end-user preferences. - Selection according to end-user language. - Selection according to service price.	- Microsoft bing search engine. - Flickr search. - Picasa search.
Translate	- Selection according to end-user preferences.	- Google translate. - Microsoft bing translator. - Yahoo babelfish translator.

For this demonstration, we have implemented a Widget aggregator [38 and 39]; a customizable Web application that enables an end-user to display the different (preferred) Widgets on the same Web page. Figure 10 is a screenshot of the Widget aggregator. Our interest is in the Widgets we have implemented; those referring to virtual services. The screenshot shows the seven Widgets that correspond to the seven virtual services listed in Table 3. For example, it shows the "*GetContacts*" virtual service, which can be configured to select actual services according to the end-user activity context (work or home). Thus, if the end-user is at home, the selection mechanism will choose the personal contact list, and if the end-user is at work, the selection

mechanism will choose the professional contacts. This selection rule requires the invocation of an actual service, referenced in the rule, to get the context information of the end-user.

The Widget aspect enables end-users to configure the rules they want to apply only once. Indeed, in further use of the same virtual service, the mechanism will apply the same selection rules; unless they are changed by the end-user.

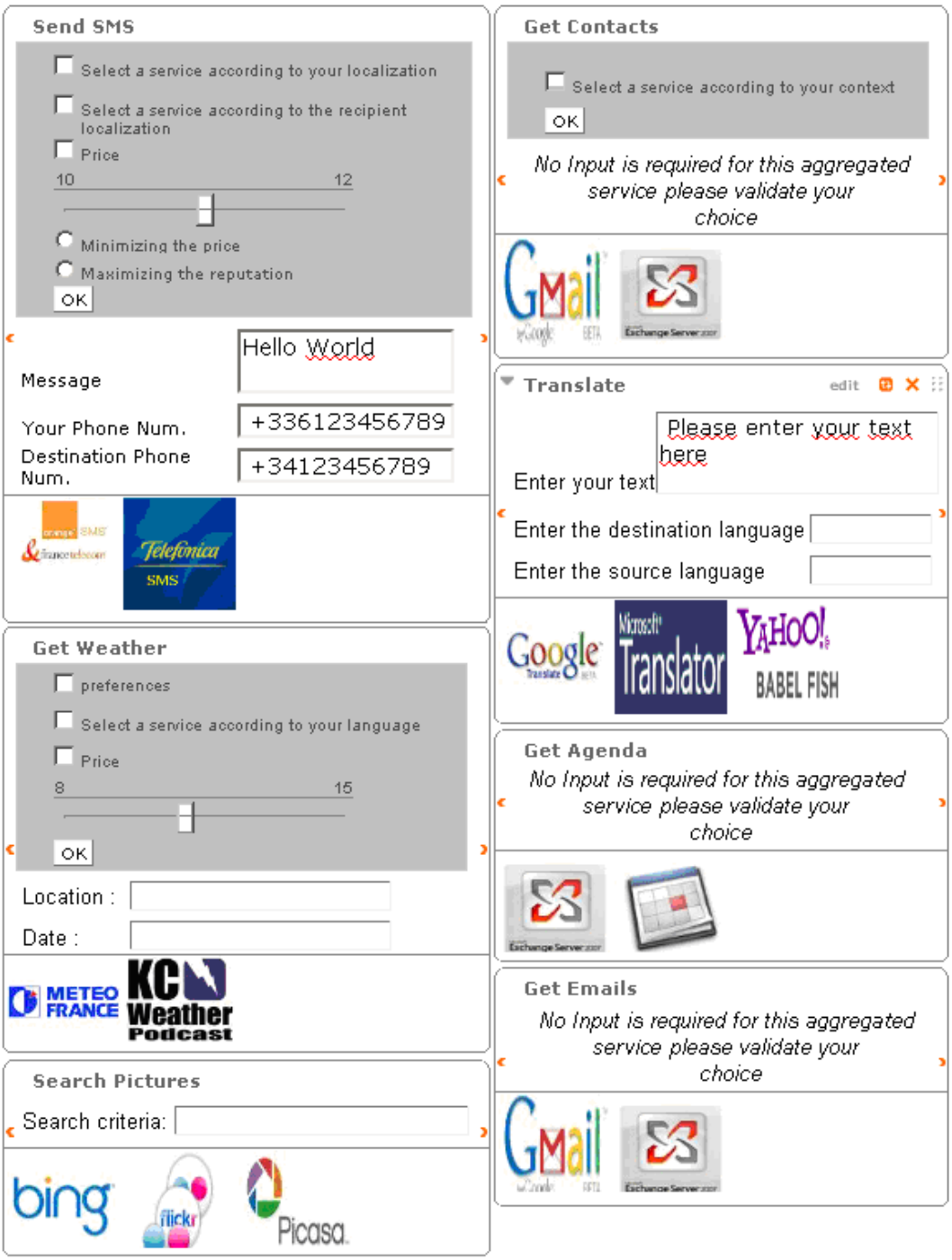

Figure 10 Virtual services.

5.2.2 Usage Validation

This demonstration has been shown within the European project SERVERY[5] (Service Platform for Innovative Communication Environment). The project aims to build a marketplace of converged services (Telecom and Web services), where service creation, service management, and their execution on heterogeneous platforms is supported.

The feedback collected from different project contributors (composed of non-technical (managers) and technical (researchers and developers) people) is very positive as:

- First the concept was included as an important part of the demonstrated mechanisms in the end review of the project.
- Second, the concept was integrated to different service composition components defined by the project such as the natural language composer. The concept enabled from the one hand to decouple composite services from the basic services they use, and from another hand to perform runtime adaptation of composite services according to rules that can refer to heterogeneous parameters such as user context, preferences, QoS…etc.
- Third, the concept was proposed and accepted for demonstration at the Orange Labs research exhibition (June 2010).
- Finally, the virtual service concept was adopted to implement an internal mediation entity to publish and search services within France Telecom – Orange.

This demonstration also raised two issues to be investigated. First, it is important to link the virtual service concept to the business engine of the SERVERY project. The business engine is essentially in charge of defining flexible business models for services (including prepaid, postpaid, and promotions) within the SERVERY marketplace. Thus, the virtual service concept should interact with the business engine in order to retrieve the services that could be used by the user (i.e. free services, or those the user is already subscribed to). Second, the price of a service is not usually a fixed value; instead it varies from a user and another (subscription option), and from a period to another (promotions). Consequently, in practice, it is not conceivable to define selection rules that refer to the service price as a fixed parameter; instead, it should be considered as a dynamic parameter which should be computed by the business engine at the runtime.

5.2.3 Validation against our requirements

We have defined three requirements to a successful implementation of virtual service concept:

- Considering the heterogeneity of parameters that can be considered in a selection policy ;
- Considering the heterogeneity of end-users' selection policies ;
- Considering the intuitiveness of the GUI that enables the end-user to define the selection policy as an important criterion.

Table 4 summarizes how our implementation tackles these three requirements.

Table 4. Implementation against requirements.

Requirement	Implementation response
heterogeneity of parameters that can be considered in a selection policy	We have defined and implemented a seamless approach for adding new parameters that could be considered in a selection rule. Indeed, to add a new parameter to be considered in a selection rule, we only need to expose it through a Web service, thanks to the capability to refer to Web services in a selection rule.
heterogeneity of end-users' selection policies	A selection policy is defined as a combination of selection rules. Each user can configure this combination and consequently define the policy he wants to apply. This is why we argue that our approach is user centric. Nevertheless, in the definition of a virtual service, it is important to associate several selection rules, in order to cover many selection criteria and provide to the end-users a wide combination capability.
the intuitiveness of the GUI	In our implementation we hided the complexity of the scripting language that enables the definition of selection rules. Thus, a selection policy is defined easily through checking and un-checking selection rules.

5.3. Scalability Validation

The main technical contribution of this paper is the design and implementation of the *Interpreter* component. It includes a rule engine, which enables the evaluation of selection rules that comply with the generic model we have designed and implemented. Therefore, it is important to evaluate the cost in terms of the response time that is added by the rule compilation and evaluation process. In other words, this section evaluates the scalability of the rule engine. This component was implemented as a Servlet running on Tomcat Servlet container 5.5.26 installed on a Debian GNU/Linux 3.1 server. The server hardware characteristics are: (Vendor: Intel, Model: Pentium III, Cache size: 256Kb, CPU:~1GHz).

The response time of the rule engine essentially depends on the number of operators (arithmetic, logical, comparison, and brackets) of the rules, the number of actual services referenced within the rules, and the response time of each. Two experimentations are achieved to test the impact of such parameters on the response time. In the first experimentation we created 20 rules with a number of operations from 25 to 500 (25, 50, 75…etc). In the second experimentation, we created 10 rules with a number of calls to an actual service from 1 to 10. It is the same actual service which is called each time.

[5] SERVERY is an ongoing Celtic project, which has received funding from each funding public authority of the involved countries (France, Spain, Hungary, Turkey), after having received Celtic label in 2008, http://projects.celtic-initiative.org/servery/, accessed on August 12th, 2010

The two following graphs (Figures 11 and 12) represent the evolution of the rule engine response time according to the number of operations and the number of calls to actual services, respectively. The objective is to figure out the response time as a function of these parameters in order to estimate the response time before executing a virtual service. This is important especially for interactive and real time (e.g. emergency) applications.

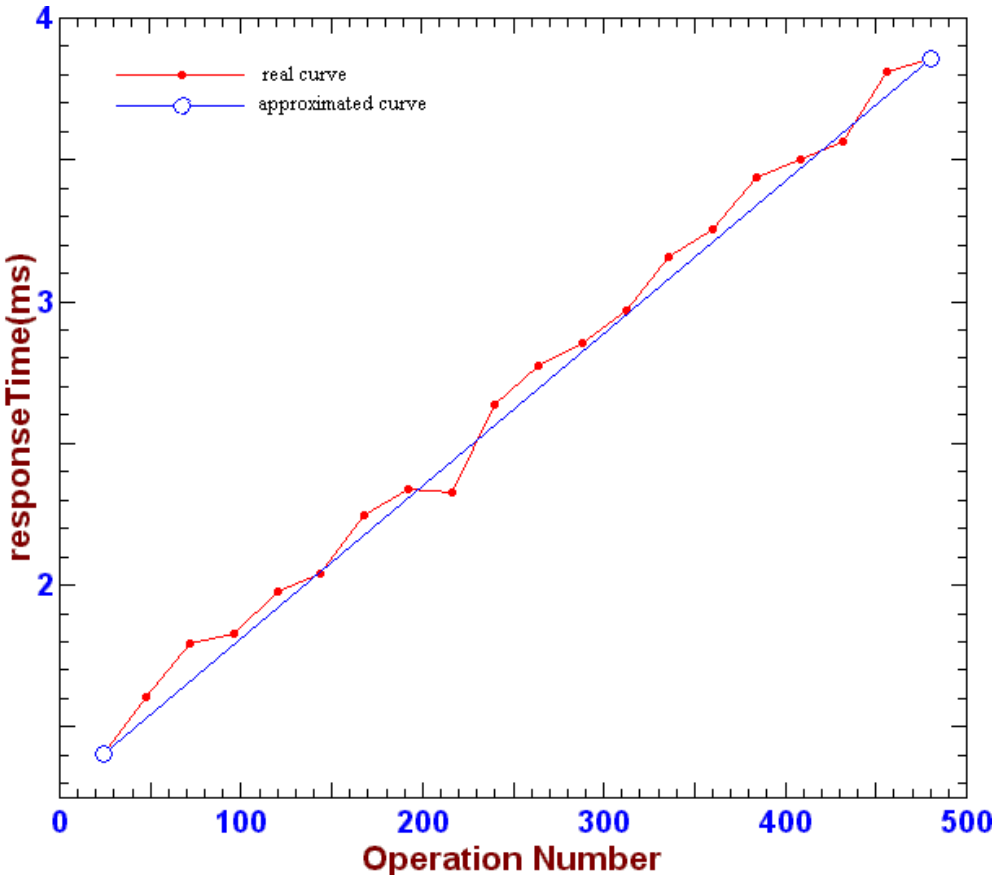

Figure 11 Response time variations according to operation number.

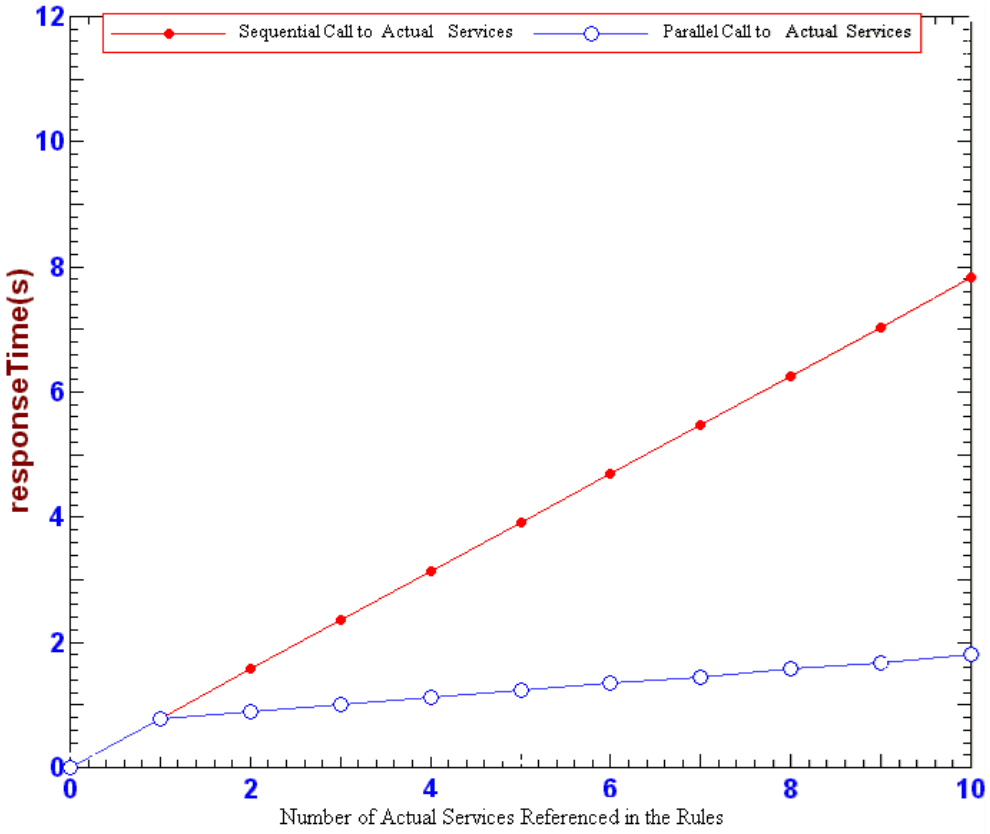

Figure 12 Response time variations according to the number of calls to actual services.

Concerning Figure 12, we have set the response time of an actual service to 670ms, which corresponds to the average response time of a Flickr REST API. This amount includes the computing time of the service as well as the network delay.

These two figures illustrate the scalability of our contribution. Figure 11 shows that the variation of the response time according to the number of operations exactly follows a linear function in the form of:

$$responseTime(opNumber) = a_1 * opNumber + b_1$$

where "*opNumber*" is the variable representing the number of operations within the rules.

In our experimentation we computed a_1 and b_1 using the computed values of "*responseTime(opNumber)*". Following are the corresponding formulae.

$$a_1 = Avr(\frac{responseTime_i - responseTime_{i+1}}{opNumber_i - opNumber_{i+1}}) = 0.004$$
$$b_1 = Avr(responseTime_i - a_1 * opNumber_i) = 1.53$$
$$responseTime(opNumber) = 0.004 * opNumber + 1.53(ms)$$

This function demonstrates that the weight of the operation number on the response time is insignificant. Indeed, according to the formula, even if we raise the number of operations to 400000, we will have a response time equal to 1.5s.

However, Figure 12 clearly indicates that the rule engine response time is influenced when the evaluated rules contain references to actual services. It shows two curves; each corresponds to an algorithm strategy. The "*sequential call to actual services*" curve depicts the variation of the response time when the rule engine calls involved actual services sequentially; and the "*parallel*

call to actual services" curve illustrates the variation of the response time when the rule engine calls involved actual services at the same time (in parallel). Though both strategies have a linear variation (*responseTime(SSCalls)=a_2* SSCalls+b_2*), the second one (parallel calls) shows a much better performance. "*SSCalls*" is the variable representing the number of references to actual services within the rules.

In our experimentation we computed a_2 and b_2 using the computed values of "*responseTime(SSCalls)*" and following the parallel calls to actual services strategy. The corresponding formulae are:

$$a_2 = Avr(\frac{responseTime_i - responseTime_{i+1}}{SSCalls_i - SSCalls_{i+1}}) = 0.114$$
$$b_2 = Avr(responseTime_i - a_2 * SSCalls_i) = 0.664$$
$$responseTime(SSCalls) = 0.114 * SSCalls + 0.664(s)$$

Thus, for the same response time (1.6 s) as above, the rule engine can evaluate a rule that contains eight calls to actual services.

The rule engine response time according to both parameters (Number of calls to actual services and number of operations) is:

$$responseTime = a_1 * opNumber + a_2 * SSCalls + b_1 + b_2;$$
$$responseTime = 0.004 * opNumber + 114 * SSCalls + 1.53 + 664(ms)$$
$$responseTime = 0.004 * opNumber + 114 * SSCalls + 665.53(ms)$$

These formulae enable us to associate an average response time to a virtual service execution. Consequently, the marketplace can provide an idea (average, max and min) of the response time of each virtual service to end-users. This information is especially useful for interactive and emergency applications.

The parallel call strategy for actual services is not always possible. Indeed, in some cases, the result of one actual service is used as an input parameter in another. For example, the following rule requires invocation of the end-user profile service before the invocation of the context service.

$Context(identifier:$Profile(userId).mobilePhoneNumber).location.country == selectedService.country;

Since calls to actual services are time consuming and negatively impact rule engine performance, it is recommended to reduce sequential calls as much as possible.

6. Conclusion and Future Work

The paper has presented a novel technique to perform service selection among functionally equivalent ones. We introduce the concept of virtual service, a GUI associated to a functionality and to a set of selection rules. The GUI enables an end-user to define his/her own selection policy by choosing the rules to apply during the process of selecting actual services at runtime, and then to execute the selected actual service. The selection rules are defined according to a language that provides the capacity of referring seamlessly to static parameter (e.g. service price) and to dynamic parameter (e.g. end-user location and presence status). These parameters are either accessible through the invocation of the corresponding actual service present in the marketplace (e.g. location

actual service to retrieve a end-user's location), or through the invocation of the knowledge base component (e.g. to retrieve an actual service price). To add a new parameter to be considered in the selection mechanism, it is only needed to publish an actual service that computes it to the marketplace. The defined and implemented mechanisms are part of the European SERVERY project. The marketplace provides a common place where services are bought and sold, including service creation tools addressed to end-users and developers.

In this paper, we tested and experimented with the concepts in relation to two different aspects: the usage pertinence of the framework in the current service marketplace concept, and its scalability in terms of response time. The usage study provided essential information regarding the possible application of these concepts to current services available today in the Web. The scalability study validated the scalability of the framework and estimated the cost added by the generic model we introduced to specify the selection rules. We have thus successfully considered various services available today in the Web, proving the pertinence of applying the concepts we defined in the emerging services marketplaces; and we have also demonstrated that the response time of the rule engine is not sensitive to the number of operation of a rule, but that it varies linearly according to the number of actual services referenced within a rule.

In addition to the usefulness and the scalability of the work presented in this paper, the two concepts we introduce also highlight a new opportunity to investigate, namely service composition based on virtual services. This general concept embraces different approaches in service composition: automatic composition based on natural end-user language, end-user mashup creation tools, and composition performed by developers. In our recent activities within the SERVERY project we have integrated the concept of virtual service to the different composition tools defined within the project. By performing service composition based on the concept of virtual service, we provided several additional features and resolve several issues faced by current service composition tools.

- First, virtual service-based composition enhances the loose coupling between service integrators and the providers of the basic actual services used. Indeed, as service integrators will define the compositions of virtual services, which are in turn defined by the marketplace provider and linked dynamically at runtime to a given actual service, the changes that occur at the actual service provider level would not affect the composition.
- Second, virtual service-based composition provides an automatic adaptation of a composite service according to the evolution of the marketplace services, as well as based on the changes that occur to different parameters, such as end-user context.

Acknowledgement

The authors wish to thank Mathieu Boussard, Benoit Christophe, Mariano Belaunde, Abderrahmane Maaradji, and Sivasothy Shanmugalingam for their comments. This work was

supported in part by SERVERY (Service Platform for Innovative Communication Environment), a CELTIC project that aims to create a Service Marketplace that bridges the Internet and Telco worlds by merging the flexibility and openness of the former with the trustworthiness and reliability of the latter, enabling effective and profitable cooperation among actors.

References

1. O'Reilly, T. What Is Web 2.0, Design Patterns and Business Models for the Next Generation of Software.
2. Boussard, M., Fodor, S., Crespi, N., Iribarren, V., Le Rouzic, J.P., Bedini, I., Marton, G., Moro Fernandez, D., Lorenzo Duenas, O., Molina, B. SERVERY: the Web-Telco marketplace. ICT-Mobile Summit 2009, Santander, Spain, June 2009.
3. Crespi, N., Boussard, M. Fodor, S. Converging Web 2.0 with telecommunications. eStrategies *Projects*, Vol. 10, June 2009. British Publishers, ISSN 1758-2369.
4. Wholesale Applications Community. WAC Informational Whitepaper. Available at http://www.wholesaleappcommunity.com/About-Wac/BACKGROUND%20TO%20WAC/whitepaper.pdf, accessed on August 12th, 2010.
5. Apple Inc. Apple app store. Available at: www.apple.com/iphone/appstore/, accessed on August 12th, 2010.
6. Google. Android market. Available at: www.android.com/market/, accessed on August 12th, 2010.
7. Windows Marketplace, Available at http://marketplace.windowsphone.com/default.aspx, accessed on August 12th, 2010.
8. Blum, N., Dutkowski, S., Magedanz, T. InSeRt - An Intent-based Service Request API for Service Exposure in Next Generation Networks. In Proceedings of 32nd Annual IEEE Software Engineering Workshop, Porto Sani Resort, Kassandra, Greece, 15-16 October 2008.
9. Rolland, C., Kaabi, R.S., Kraiem, N. On ISOA: Intentional Services Oriented Architecture. In Advanced Information Systems Engineering, volume 4495/2007, p 158-172, June 2007.
10. Piessens, F., Jacobs, B., Truyen, E., Joosen, W. Support for Metadata-driven Selection of Run-time Services in .NET is Promising but Immature. Journal of Object Technology, Special issue: .NET: The Programmers Per-spective: ECOOP Workshop (2003).
11. Google. Intents and Intent Filters. Available at http://developer.android.com/guide/topics/intents/intents-filters.html, accessed on August 12th, 2010.
12. Ben Hassine, A., Matsubara, S., Ishida, T. A constraint-based approach to horizontal web service composition. In ISWC, pp. 130-143, (2006).
13. Santhanam, G. R., Basu, S., and Honavar, V. On Utilizing Qualitative Preferences in Web Service Composition: A CP-net Based Approach. In Proceedings of the 2008 IEEE Congress on Services, Services - Part I, 2008, vol., no., pp.538-544, 6-11 July 2008.

14. Baresi, L., Bianchini, D., Antonellis, V.D., Fugini, M.G., Pernici, B., Plebani, P. Context-aware Composition of e-Service. In Technologies for E-Services: Third International Workshop, TES 2003, Berlin, German, September 7-8, 2003.
15. Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., Shan, M. Adaptive and Dynamic Service Composition in eFlow. In Proceedings of the 12th International Conference on Advanced Information Systems Engineering (June 05 - 09, 2000). B. Wangler and L. Bergman, Eds. Lecture Notes in Computer Science, vol. 1789. Springer-Verlag, London, 13-31.
16. Spanoudakis, G., Mahbub, K., Zisman, A. A Platform for Context Aware Runtime Web Service Discovery. In IEEE International Conference on Web Services, 2007. ICWS 2007, vol., no., pp.233-240, 9-13 July 2007.
17. Cibrán, M. A., Verheecke, B., Vanderperren, W., Suvée, D., and Jonckers, V. 2007. Aspect-oriented Programming for Dynamic Web Service Selection, Integration and Management. World Wide Web 10, 3 (Sep. 2007), 211-242.
18. Yu, Q., Bouguettaya,A., 2012. Multi-attribute optimization in service selection. World Wide Web 15, 1 (Jan. 2012), 1-31.
19. Rasch, K;, Li, F., Sehic, S., Ayani R., and Dustdar, S. 2011. Context-driven personalized service discovery in pervasive environments. World Wide Web 14, 4 (Jul. 2011), 295-319.
20. Liu, Y., Ngu, A. H., and Zeng, L. Z. 2004. QoS computation and policing in dynamic web service selection. In *Proceedings of the 13th international World Wide Web Conference on Alternate Track Papers &Amp; Posters* (New York, NY, USA, May 19 - 21, 2004). WWW Alt. '04. ACM, New York, NY, 66-73.
21. Ding, Q., Li, X., and Zhou, X. 2008. Reputation Based Service Selection in Grid Environment. In *Proceedings of the 2008 international Conference on Computer Science and Software Engineering - Volume 03* (December 12 - 14, 2008). CSSE. IEEE Computer Society, Washington, DC, 58-61.
22. Tsesmetzis, D., Roussaki, I., and Sykas, E. 2007. Modeling and Simulation of QoS-aware Web Service Selection for Provider Profit Maximization. *Simulation* 83, 1 (Jan. 2007), 93-106.
23. Yu, T., Zhang, Y., Lin, K. Efficient algorithms for Web services selection with end-to-end QoS constraints. ACM Transaction Web 1, 1 (May. 2007), 6.
24. Aggarwal, R., Verma, K., Miller, J., and Milnor, W. Constraint Driven Web Service Composition in METEOR-S. In Proceedings of the 2004 IEEE international Conference on Services Computing (September 15 - 18, 2004). IEEE Computer Society, Washington, DC, 23-30.
25. Reichl, P. From ‘Quality-of-Service’ and ‘Quality-of-Design’ to ‘Quality-of-Experience’: A holistic view on future interactive telecommunication ser-vices. 15th International Conference on Software, Telecommunications and Computer Networks, 2007. Soft-COM 2007. vol., no., pp.1-6, 27-29 Sept. 2007.

26. Gu, X., Nahrstedt, K., Yuan, W., Wichadakul, D., Xu, D. An Xml-Based Quality of Service Enabling Language for the Web. Technical Report. UMI Order Number: UIUCDCS-R-2001-2212., University of Illinois at Urbana-Champaign.
27. Frolund, S., Koisten, J. QML: A Language for Quality of Service Specification. HP Labs technical reports. Available at http://www.hpl.hp.com/techreports/98/HPL-98-10.html, accessed on August 12th, 2010.
28. Martin, D. et al. OWL-S: Semantic Markup for Web Services. W3C member submission, available at http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/, accessed on August 12th, 2010.
29. Xu, Z., Martin, P., Powley, W., Zulkernine, F. Reputation-Enhanced QoS-based Web Services Discovery. Web Services, 2007. In proceedings of IEEE International Conference on Web Services, ICWS 2007., vol., no., pp.249-256, 9-13 July 2007.
30. Wang, P., Chao, K., Lo, C., Farmer, R., and Kuo, P. 2009. A Reputation-Based Service Selection Scheme. In *Proceedings of the 2009 IEEE international Conference on E-Business Engineering* (October 21 - 23, 2009). ICEBE. IEEE Computer Society, Washington, DC, 501-506.
31. Wang, H., Yang, D., Zhao, Y., and Gao, Y. 2006. Multiagent System for Reputation--based Web Services Selection. In *Proceedings of the Sixth international Conference on Quality Software* (October 27 - 28, 2006). QSIC. IEEE Computer Society, Washington, DC, 429-434.
32. Malik, Z. and Bouguettaya, A. 2009. Rater Credibility Assessment in Web Services Interactions. *World Wide Web* 12, 1 (Mar. 2009), 3-25.
33. Sanchez, A., Carro, B., Wesner, S. Telco services for end customers: European Perspective. In Communications Magazine, IEEE, vol.46, no.2, pp.14-18, February 2008.
34. Dey, A. K., Salber, D., Abowd, G. D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, Human-Computer Interaction, 16, pp, 1-67, 2001.
35. MAIS (Multichannel adaptative information systems) Project. Available at http://www.mais-project.it/index.php, accessed on August 12th, 2010.
36. Newcomer, E. Understanding Web Services: XML, Wsdl, Soap, and UDDI. Addison, Wesley, Boston, Mass., May 2002.
37. Fielding, R.T. Architectural Styles and the Design of Network-based Software Architectures. Thesis dissertation, 2000.
38. Laga, N., Bertin, E., and Crespi, N. 2009. Building a User Friendly Service Dashboard: Automatic and Non-intrusive Chaining between Widgets. In *Proceedings of the 2009 Congress on Services - I* (July 06 - 10, 2009). SERVICES. IEEE Computer Society, Washington, DC, 484-491.
39. Laga, N., Bertin, E., and Crespi, N. 2010. Business Process Personalization Through Web Widgets. In *Proceedings of the 2010 IEEE international Conference on Web Services* (July 05 - 10, 2010). ICWS. IEEE Computer Society, Washington, DC, 551-558.

40. Nestler, T., Namoun, A., Schill, A. 2011. End-user development of service-based interactive web applications at the presentation layer. EICS 2011: 197-206.