# A Request Oriented Model for Web Services

**Gaurav Mitra[1], Xuan Zhou[2], Athman Bouguettaya[3], Xumin Liu[4]**
**[1]Research School of Computer Science, Australian National University, Australia**
**[2]School of Information, Renmin University, China**
**[3] School of Computer Science and Information Technology, RMIT, Australia**
**[4]Department of Computer Science, Rochester Institute of Technology, USA**

**Abstract**

The existing service modeling methodologies, such as WSDL and OWL-S, are *service-oriented*, which mainly focus on providing formalisms for the important features of a web service, including its functionality and QoS parameters. Conforming to these models, users need to first find what the available services are, go through the descriptions, and then shape the request specifications based on the functionality of these services. These modeling methodologies do not cope with the ever-increasing number and variety of web services, which introduces significant difficulties to users when discovering and selecting services in a large scale and heterogeneous environment. To address this issue, we propose a *request oriented model*, where the formalisms focus on user expectations and experiences on the usage of services, i.e., what a user wants as the result of accessing to services and what the user will experience during the service invocations. The model lays out a foundation for efficient and personalized service selection. It also provides formalisms for describing a service functionality, which supports *service reasoning tasks* to improve automation of service selection and usage. Based on the model, we propose a *Web Service Request Language* (WSRL), which allows users to specify their requests in a *declarative* way. We also present the reasoning procedure that mediates the interactions between users and web services.

## 1 Introduction

The Internet today has a widespread distribution of e-marketplaces offering diverse products to consumers. Websites usually represent consumer-vendor interfaces of such online markets. Consumers navigate to websites and specify their needs via intuitive features and options offered by these websites. In a typical scenario, a consumer would go through the following steps to purchase a product from the Internet : (i) Using a web directory or a web search engine, a consumer locates a suitable website that meets the needs; (ii) The consumer navigates to the website and attempts to understand its structure, available options and suitability; (iii) The consumer expresses his or her needs, determines whether they will be satisfied, explores payment and security options, and possibly completes a business transaction. Companies hosting the websites typically have a specialized domain. For instance, airlines offer flight reservation, hotels offer room booking, vehicle agencies offer car rentals. In case a consumer's needs cannot be fulfilled by any single website, e.g., booking a travel package including all above services, he or she will have to repeat the above steps for each corresponding website, which is tedious, time consuming, and error-prone.

Service Oriented Computing (SOC) offers a means to alleviate users from tedious work of manually selecting and interacting with web-based services by automating service discovery and invocation [20, 7]. Service composition techniques are also considered as a promising tool to generate and deliver value-added services, by integrating multiple services into workflows. The standardization efforts, the key enabler of SOC, significantly improve interoperability in a heterogeneous environment. Such standards include Simple Object Access Protocol (SOAP) [15], the Extensible Markup Language (XML), the Web Service Description Language (WSDL) [17], the Universal Description, Discovery and Integration (UDDI) repositories [16], Business Process Execution Language (BPEL) [11] , and so on.

As SOC keeps gaining significant interests and attentions from both industry and academia, many research efforts have been conducted aiming to reduce the human efforts in service usage, such as efficient service discovery, and automatic service composition. These works are supported by several formal *service oriented* models, where the formalisms focus on describing the features of a web service, such as its functionality and performance parameters [3, 19, 12, 21]. User requirements need to be specified in a way that conforms to these models to be processed. This leads to a *service oriented* interaction pattern, where users need to first find what the available services are and then shape their request specifications based on the description of these services. Driven by the popularity of SOC, the amount and variety of web services have been tremendously increased. Meanwhile, service composition technologies allow to dynamically generate a *value-added* service by composing multiple services together, which exponentially increases the number and the kinds of services available for users. This introduces great difficulties in discovering, selecting, composing, and accessing services based on the existing service oriented models.

To address the above issue, we propose a *request oriented* model, where the formalism focuses on user expectations and experiences on the usage of services, i.e., what a user wants as the result of accessing services and what the user will experience during the service invocations. The model lays out a foundation for efficient and personalized service selection. It also provides formalisms for describing a service functionality, which supports *service reasoning tasks* to improve automation of service selection and usage. Guided by user expectation and experience, matched services will be selected and invoked. Services will also be composed if needed in a static (e.g., predefined business processes) or dynamic way. On top of the model, we propose a *Web Service Request Language* (WSRL), which allows a *formal* and *declarative* specification of user requirements. More specifically, users do not need to specify which services will be involved and how to access the services when describing their requirements. This frees users from the tedious work of reading service documentations to learn the technical details for service selection and invocation. Moreover, if a user's requirement demands the composition of several services, i.e., a value-added service, the proposed framework will automatically select related services and orchestrate them together.

The rest of the paper is organized as follows. We describe a laptop purchase scenario as a running example to motivate and illustrate the proposed approach in Section 2. We concentrate on defining the request oriented model and the request language in Section 3. In Section 4 we provide an overview of the request framework and how the request system is used. In Section 5 we provide a brief description of the implementation of the WSRS. In Section 6, we discuss some representative related work and describe how this work is different from them. We finish with a conclusion in Section 7.

## 2 A Laptop Purchase Scenario

In this section, we describe a laptop purchase scenario to show how a *Web Service Request System* (WSRS), a supporting framework of WSRL, works. We also use it as a running example to illustrate the important concepts and approach proposed in this paper.

As depicted in Figure 1, suppose a user wants to buy a laptop equipped with a 15-in screen and an Intel processor, and with a high rating in customer review. At this point, the user does not know what services are available, what services are related, and what formats are needed to follow to invoke the services. He follows WSRL format for his request specification, which is not linked to any concrete service. Once he submits the request, the system will analyze it and process it.

By performing some service reasoning tasks, the system will locate several services, including a laptop review service for finding a desirable laptop, a customer purchase service for performing the purchase process, and a delivery service for arranging the laptop delivery. After invoking the laptop review service, the system returns a list of matched laptops to the user. Once the user selects a laptop and submits the decision, the system orders the laptop by invoking a computer purchasing service with the input obtained from the user. A receipt will be generated as the result of the invocation and be forwarded to the user by the WSRS. Once the receipt is ready, the WSRS system will automatically invoke a delivery service for the shipment.

During the entire process, there is no direct interaction between the user and the three services. This significantly alleviates the efforts and knowledge required from users. In general case, such a WSRS system can potentially be deployed to any specific business domains, as a single entry point to multiple related web services. More importantly, with the request oriented model, WSRS allows a flexible and customized service delivery on a large scale.

## 3 A Request Oriented Model and Web Service Request Language

In this section,we propose a novel model for web services as a formal grounding of user-service interactions. We use the running example, the laptop purchase scenario, to illustrate the concepts in the model.

In the literature, the proposed web service models, are mainly *service-oriented* and do not describe a web service from a user's perspective. To provide a complete view, our model defines two types of information, which are described below.

- *User state*: A user's state is defined and traced once he/she enters a WSRS system. The state is altered every time when the user invokes a web service through the system. The initial state of a user shows his/her expectation on using the system. The transitions of the state shows his/her experience, i.e., how the expectation gets fulfilled. It is worth knowing that there are dependent constraints between user states. For example, a user cannot have such a state, where `purchasedlaptop` = false && `initiatedDelivery`=true. That is, `initiatedDelivery` depends on `purchasedlaptop`. In this section, we formally define a user state and the dependency between states.
- *Web service*: We define a web service from a user's perspective. We differentiate two types of services, *abstract services* and *concrete services*, where abstract services focus on the service functionality and concrete services focus on service implementation. A service functionality is defined in two aspects, transforming input to outptut (i.e., data related), and altering user states (i.e., state related).

### 3.1 The User State Model

To design a request oriented model, it is important to understand a service, single or composed, from a user's point of view. In a typical scenario, a user considers two types of features of a service, *functional* and *non-functional.* For the functional features, a user's perspectives include: the state transition by invoking a service, the information retrieved from invoking a service. For the non-functional features, a user concerns the quality delivered by a service, such as *duration*, *fee*, *reliability*, and *security*. Therefore, it is reasonable to say that a users expectation on his/her interactions with services is centred around these features. Along with this line, we define a user state variable as follows.

**Definition 1 (User State Variable):** *A user state variable represents an attribute of a user state. It is defined as a triple, var =$<$ Name, Val, T $>$ where:*

- **Name** *is the name of the variable.*
- **Val** *is value of the variable.*
- **T** *is the type of the variable. Our model considers two types of variables:*
    1. **Information Variable** $<$ *info* $>$ *: representing data used by a service, i.e. a services input and output messages.*
    2. **Functionality Variable** $<$ *fn* $>$ *: representing whether a functional goal is achieved in the business process. It is usually Boolean.*

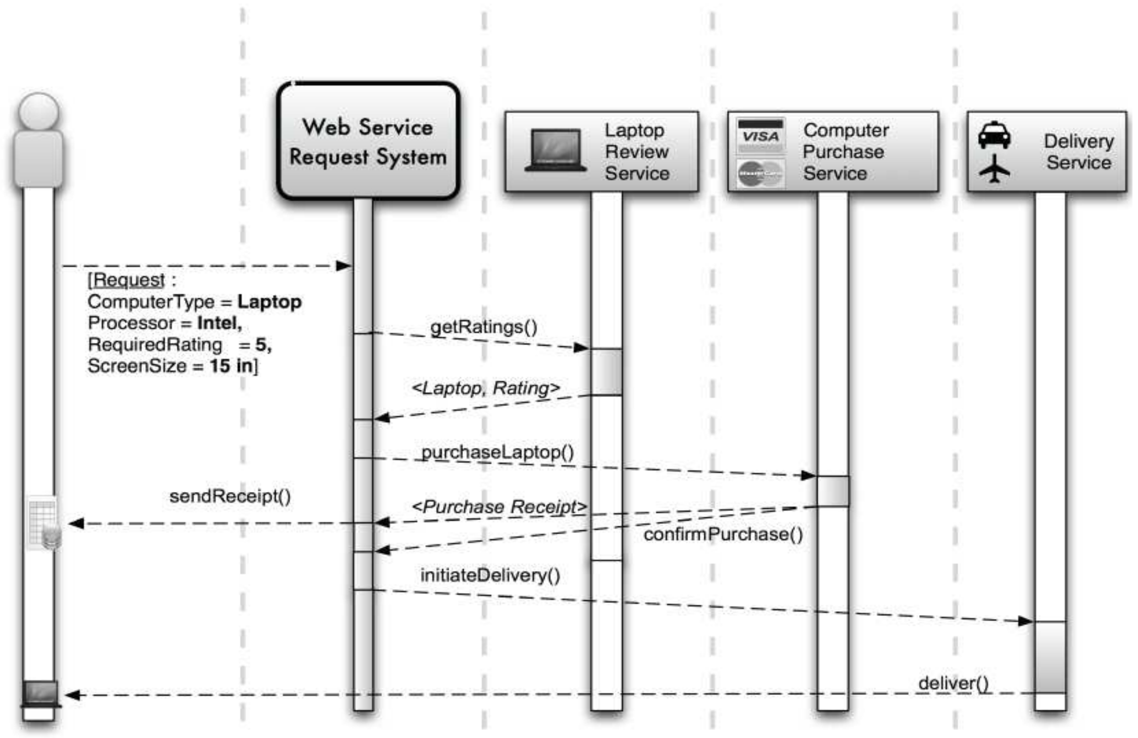


Figure 1: Purchasing A Laptop Via The Request System

In our laptop purchase example, the information variables would include *ComputerType, RequiredRating, Processor, ScreenSize, DeliveryAddress, ComputerModelNo, Computer-Rating, ComputerPrice, PurchaseReceipt* and *TrackingNumber*. They are used as inputs and outputs of the laptop review service, computer purchase service and the delivery service. The functionality variables would include *gotRatings, purchasedLaptop, initiatedDelivery.* Each of the functionality variables represent the status of the operations of these services. We formally define a user state as follows:

**Definition 2 (User State):** *A user state can be defined as a tuple,* $S =< ID, Var_{fn}, Var_{Info} >$ *where:*

- *ID is a unique global identifier for the state. ID = 0 indicates the root state.*
- $Var_{fn}$ *is a set of instantiated functionality variables.*
- $Var_{info}$ *is a set of instantiated information variables.*

A user can change the state by invoking services. When an atomic service is executed, its effects and outputs would change the values of one or many user state variables, so that the user is transferred to a different state. This is demonstrated in Figure 2(a), where invocation of the *getRatings* operation results in the change of user state variables by the laptop review service. The functionality variable corresponding to the invoked service is given value true as a result of the invocation, and the information returned by the invoked service (i.e. output messages of the invoked operation) is incorporated into the information variables. We model the transitions between states as state dependencies, which are defined as follows:

**Definition 3 (User State Dependency):** *A user state dependency can be defined as a triple,* $S_{Dep} =< S_{pred}, S_{succ}, Comp_{fn} >$ *where:*

- $S_{pred}$ *is the predecessor (or previous) state .*
- $S_{succ}$ *is the successor (or next) state .*
- $Comp_{fn}$ *is an atomic service. Its invocation can transfer a user from state* $S_{pred}$ *to state* $S_{succ}$*.*

User state dependencies connect user states into a user state graph, which is defined as follows:

**Definition 4 (User State Graph):** *A user state graph is a directed acyclic graph (DAG) with a single root, denoted by a triple* $USG =< r, V, E >$ *where:*

- *r is the root user state, representing the state when a user first enters the system.*
- *V is a set of user states that are the nodes of the DAG.*
- *E is a set of user state dependencies that are the edges of the DAG.*

The purpose of the user state graph is to show all the possible execution sequences of a dynamic business process. The detailed user state graph cannot be generated at design time as the values of information variables are unknown until the actual invocation of services. However, the values of functionality variables are usually predictable based on functionality descriptions of a service. It is possible to generate an *abstract user state graph* at design time, where the functionality variables are completely bound and the information variables remain unbound. Our service request system generates this abstract graph as a pre-computing step. Figure 2 (b) shows an example of an abstract user state graph for the laptop review service, computer purchase service and the delivery service. To generate an abstract user state graph, we need the descriptions of functionalities of web services. A service operation functionality is usually modeled as pre and post-conditions. We define these aspects in the functionality component of the *web service description model* in section 3.2.

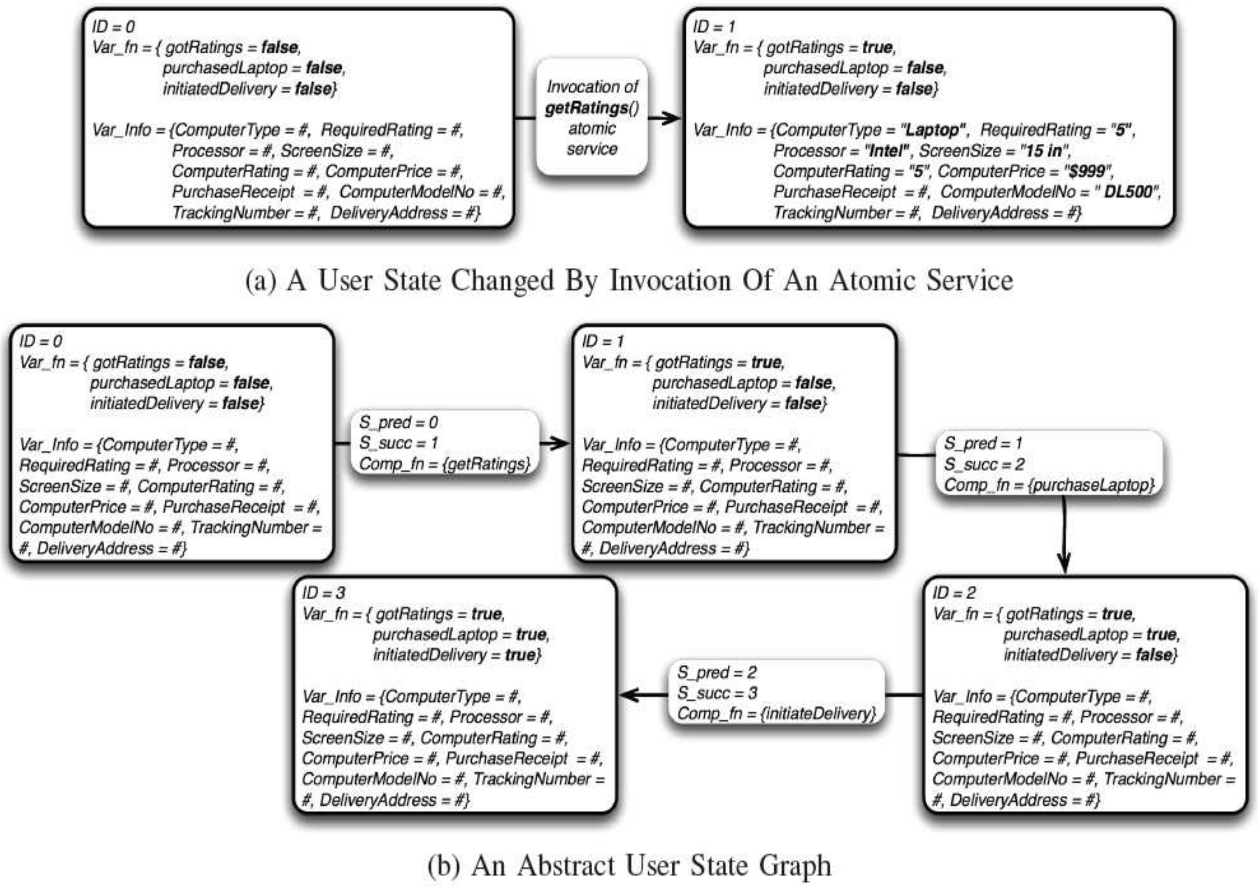

ID = 0
Var_fn = { gotRatings = false, purchasedLaptop = false, initiatedDelivery = false}
Var_Info = {ComputerType = #, RequiredRating = #, Processor = #, ScreenSize = #, ComputerRating = #, ComputerPrice = #, PurchaseReceipt = #, ComputerModelNo = #, TrackingNumber = #, DeliveryAddress = #}
Invocation of getRatings() atomic service
ID = 1
Var_fn = { gotRatings = true, purchasedLaptop = false, initiatedDelivery = false}
Var_Info = {ComputerType = "Laptop", RequiredRating = "5", Processor = "Intel", ScreenSize = "15 in", ComputerRating = "5", ComputerPrice = "$999", PurchaseReceipt = #, ComputerModelNo = " DL500", TrackingNumber = #, DeliveryAddress = #}
(a) A User State Changed By Invocation Of An Atomic Service
ID = 0
Var_fn = { gotRatings = false, purchasedLaptop = false, initiatedDelivery = false}
Var_Info = {ComputerType = #, RequiredRating = #, Processor = #, ScreenSize = #, ComputerRating = #, ComputerPrice = #, PurchaseReceipt = #, ComputerModelNo = #, TrackingNumber = #, DeliveryAddress = #}
S_pred = 0
S_succ = 1
Comp_fn = {getRatings}
ID = 1
Var_fn = { gotRatings = true, purchasedLaptop = false, initiatedDelivery = false}
Var_Info = {ComputerType = #, RequiredRating = #, Processor = #, ScreenSize = #, ComputerRating = #, ComputerPrice = #, PurchaseReceipt = #, ComputerModelNo = #, TrackingNumber = #, DeliveryAddress = #}
S_pred = 1
S_succ = 2
Comp_fn = {purchaseLaptop}
ID = 2
Var_fn = { gotRatings = true, purchasedLaptop = true, initiatedDelivery = false}
Var_Info = {ComputerType = #, RequiredRating = #, Processor = #, ScreenSize = #, ComputerRating = #, ComputerPrice = #, PurchaseReceipt = #, ComputerModelNo = #, TrackingNumber = #, DeliveryAddress = #}
S_pred = 2
S_succ = 3
Comp_fn = {initiateDelivery}
ID = 3
Var_fn = { gotRatings = true, purchasedLaptop = true, initiatedDelivery = true}
Var_Info = {ComputerType = #, RequiredRating = #, Processor = #, ScreenSize = #, ComputerRating = #, ComputerPrice = #, PurchaseReceipt = #, ComputerModelNo = #, TrackingNumber = #, DeliveryAddress = #}
(b) An Abstract User State Graph

Figure 2: Examples of User State, User State Dependency and User State Graph

## 3.2 The Web Service Description Model

The purpose of the web service description model is to describe the operations, messages, data types, port bindings and policy parameters of atomic web services, in order to create a composite service automatically based on user requirements. The model is subdivided into two parts, i.e. the model of an *abstract service* and that of a *concrete service*. An abstract service represents a class of services of the same functionality and interface [8]. A concrete service is an actual instantiation of an abstract service. Each abstract service is usually created by a service designer and stored in a local repository. Each concrete service is implemented by a specific service provider with or without being aware of the corresponding abstract service description. In the latter case, the mapping between abstract services and concrete services should be generated in a manual or dynamic way [9]. In principle, each abstract service can have multiple service providers, and can therefore correspond to multiple concrete services. The abstract service model we propose in this paper consists of two components, namely the *information* and *functionality* components, which correspond to the two types of user state variables defined in section 3.1, i.e. the information and functionality variables.

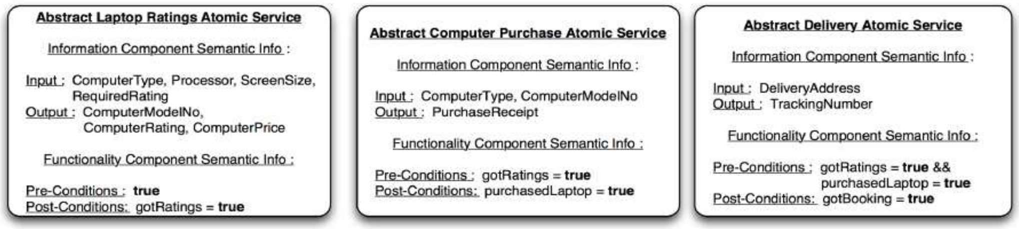
1. *Information Component*: This component is used to describe the input and output messages for the atomic service that this component represents, in the form of information variables in the user state model. It specifies exactly which information variables are required by this atomic service as input and which ones are required as output.
2. *Functionality Component*: This component represents an atomic service operation. This component provides the pre and post conditions of the operation in the form of *first order logic* propositions. Each logic proposition is composed of a set of functionality variables of the user state model. This is precisely the semantic information required to generate the user state graph.

Some examples of abstract service descriptions are given in Figure 3 (a). To define concrete services, we first introduce *concrete service variables.*

**Definition 5 (Concrete Service Variable):** *A concrete service variable represents an attribute of an executable service offered by a provider. It is defined as a tuple, var =$< Name, Val, T >$ where:*

- **Name** *is the name of the variable.*
- **Val** *is the value of the variable.*
- **T** *is the type of the concrete service variable. There are two types of variables :*
  1. **Policy Variable** $< pol >$: *Represents a WS-Policy attribute of a web service. This includes quality of service, security and versioning information.*
  2. **Implementation Variable** $< impl >$ *: Represents the service provider, abstract service and service implementation elements such as network address and port bindings.*

The mapping between abstract services and concrete services is made by an implementation variable e.g. the *Abstract Service* variable in Figure 3 (b). The



**Abstract Laptop Ratings Atomic Service**

Information Component Semantic Info :

Input : ComputerType, Processor, ScreenSize, RequiredRating
Output : ComputerModelNo, ComputerRating, ComputerPrice

Functionality Component Semantic Info :

Pre-Conditions : **true**
Post-Conditions: gotRatings = **true**

**Abstract Computer Purchase Atomic Service**

Information Component Semantic Info :

Input : ComputerType, ComputerModelNo
Output : PurchaseReceipt

Functionality Component Semantic Info :

Pre-Conditions : gotRatings = **true**
Post-Conditions: purchasedLaptop = **true**

**Abstract Delivery Atomic Service**

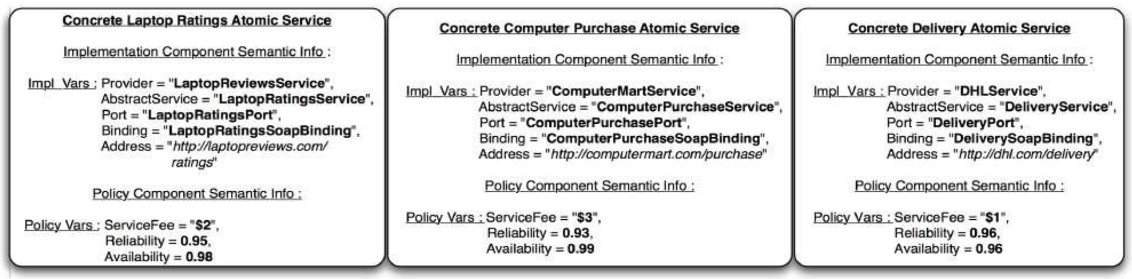
Information Component Semantic Info :

Input : DeliveryAddress
Output : TrackingNumber

Functionality Component Semantic Info :

Pre-Conditions : gotRatings = **true** && purchasedLaptop = **true**
Post-Conditions: gotBooking = **true**

(a) Descriptions of Abstract Services Using Information and Functionality Components



**Concrete Laptop Ratings Atomic Service**

Implementation Component Semantic Info :

Impl Vars : Provider = "**LaptopReviewsService**",
AbstractService = "**LaptopRatingsService**",
Port = "**LaptopRatingsPort**",
Binding = "**LaptopRatingsSoapBinding**",
Address = "*http://laptopreviews.com/ratings*"

Policy Component Semantic Info :

Policy Vars : ServiceFee = "**$2**",
Reliability = **0.95**,
Availability = **0.98**

**Concrete Computer Purchase Atomic Service**

Implementation Component Semantic Info :

Impl Vars : Provider = "**ComputerMartService**",
AbstractService = "**ComputerPurchaseService**",
Port = "**ComputerPurchasePort**",
Binding = "**ComputerPurchaseSoapBinding**",
Address = "*http://computermart.com/purchase*"

Policy Component Semantic Info :

Policy Vars : ServiceFee = "**$3**",
Reliability = **0.93**,
Availability = **0.99**

**Concrete Delivery Atomic Service**

Implementation Component Semantic Info :

Impl Vars : Provider = "**DHLService**",
AbstractService = "**DeliveryService**",
Port = "**DeliveryPort**",
Binding = "**DeliverySoapBinding**",
Address = "*http://dhl.com/delivery*"

Policy Component Semantic Info :

Policy Vars : ServiceFee = "**$1**",
Reliability = **0.96**,
Availability = **0.96**

(b) Descriptions of Concrete Services Using Implementation and Policy Components

Figure 3: Examples of Web Service Descriptions

concrete description model also has two components, namely the *policy* and *implementation* components:

- *Implementation Component*: This component contains the implementation variables, such as the name of the service provider, and port binding information for the service implementation. They can be mapped to WSDL description elements, such as <binding>,<port> and <service>.

- *Policy Component*: This component describes the Quality of Service, security protocols, and service versioning information in a list of policy variables. Attributes in WS-Policy can be incorporated into this component [2].

Figure 3 (b) illustrates some concrete services involved in our laptop purchase example.

### 3.3 The Web Service Request Language

WSRL allows a user to express a service request in terms of service variables and specify the target state the user wants to arrive at (i.e. goal state). When expressing a service request, we assume that the user is presented with all user state variables and concrete service variables in the system and the user has adequate knowledge about the variables and their respective purposes. This is a reasonable assumption as far as a normal user is concerned, as the variables are mostly represented by terms the user would be well versed in. Users express constraints on service variables to reflect their service requirements, through our service request language.Formally defining a variable constraint, we have:

**Definition 6 (Variable Constraint):** *A variable constraint is defined as a logical expression described in propositional calculus. To form an atomic variable constraint, a variable and its possible value are combined by an equational logic connective (i.e.,* $<, >, =, <=, >=$*) . Two atomic variable constraints can be combined using a propositional logic connective (i.e.,*$\vee, \wedge, \neg$*)to form a compound variable constraint.*

A web service request is formed by variable constraints.

**Definition 7 (Web Service Request):** *A web service request is defined as a tuple* $R =< Var_{info}, Var_{fn}, Var_{impl}, Var_{pol}, Ret >$*, where:*

- $Var_{info}$ *is a set of variable constraints on information variables.*

- $Var_{fn}$ *is a set of variable constraints on functionality variables.*

- $Var_{impl}$*is a set of variable constraints on implementation variables.*

- $Var_{pol}$ *is a set of variable constraints on policy variables.*

- *Retis a set of information variables that the request defines as return values.*

*Except the variables in Ret, each constraint is associated with one of the unary operators require and optional, indicating whether it is a mandatory constraint or an optional constraint.*

An example service request for a laptop computer, with a 15 screen, an intel processor, a five star rating, and a maximum $10 fee for the services invoked, would be:

```
request{
        require<info> Laptop : : ComputerType = “ Laptop ”
                and Laptop : : Processor = “Intel”
                and Laptop : : RequiredRating =“5”
                and Laptop : : ScreenSize = “15 in ”
        require<fn> Laptop : : initiatedDelivery
        require<pol> Laptop : : ServiceFee < “$10”
        return Laptop : : PurchaseReceipt
}
```

The semantics of WSRL operators are clearly defined in Table 1.

## 4 The Web Service Request Framework

The web service request framework is responsible for generating the user state graph as a pre-computing step and then translating a service request given by a consumer into a web service execution plan. The execution plan can then be directly executed to deliver the results desired by the consumer. The translation is conducted in four steps, which are described as follows.

First, a *request encoder* translates a request written in the request language to a web service request conforming to Definition 7. This encoded request is a set of variable constraints

Table 1: THE OPERATORS OF WSRL

| **WSRL Operator** | **Purpose** |
|---|---|
| $n :: var$ | Express the variable var as part of the service namespace $n$ |
| request $\{p_1, ..., p_n\}$ | Express ordering of variable constraints $p_1$ to $p_n$ in a service request |
| require $< var > p$ | Express a mandatory variable constraint p on a variable of type var. Here $var \in \{info, fn, pol, impl\}$ |
| if $p_1$ then $p_2$ [else $p_3$] | Express a conditional binary (if $p_1$ then $p_2$) or ternary (if $p_1$ then $p_2$ else $p_3$) variable constraint? |
| optional $p$ | Express a variable constraint as optional, i.e. not mandatory for the satisfaction of the request? |
| prefer $p_1$ to $p_2$ | Express a variable constraint as a preference to another i.e. if $p_1$ is satisfied then $p_2$ does not hold, else $p_2$ needs to be satisfied |
| return $p$ | Express a variable constraint p on information variables, as a return value for the satisfaction of a service request. |

given in Definition 6. These constraints are passed on to a request mapper algorithm, which maps them to an abstract user state, called goal state. The functionality variables of the goal state are instantiated with boolean values that agree with the functionality variable constraints in the request. The information variables of the goal state remain unbound.

Second, the goal state is passed on to a path finder, which finds a path in the abstract user state graph that takes the root user state to this goal state. This path is called the goal state path. A goal state path is defined as follows:

**Definition 8 (Goal State Path):** *A goal state path for a user state g is defined as a set* $Pg = \{D_1, D_2, ...., D_n\}$*, where* $D_i$ *is a user state dependency,* $1 < i < n$ *and* $D_1.S_{pred} = r$ *(root state) and* $D_n.S_{succ} = goal$ *(goal state).*

Third, the goal state path is passed on to an execution plan generator, which uses the concrete service descriptions in the system to create a service execution plan. This plan associates a concrete web service to each edge (or user state dependency) of the goal state path, and binds its inputs and outputs to the corresponding information variables. The concrete web services are chosen based on the implementation and policy variable constraints given by the users request. For example, in the laptop purchase request given in section 3, the ServiceFee policy variable has a constraint of being less than \$10. Therefore, the execution plan generator must guarantee that the aggregated service fee of the selected concrete services be less than \$10. A service execution plan is defined as follows:

**Definition 9 (Web Service Execution Plan):** *A web service execution plan is defined as a set* $ExP = \{ExI_1, ExI_2, ...., ExI_n\}$ *where* $ExI_i$ *is a concrete web service,* $1 < i < n$*. Figure 4 shows the set of variable constraints, the corresponding goal user state, goal state path and service execution plan for the laptop purchase example.*

Fourth, a *service executor* invokes the services indicated in the service execution plan. At this stage, when each invoked service returns results, the information variables in the user state graph start getting values assigned to them. The constraints on information variables are enforced by the service executor at two stages. Suppose a service is about to change a set of information variables.

- **Stage One**: Before executing the service, we first check whether the service can take the corresponding information constraints as inputs. If it does, we feed the constraints as input to the service, so that the execution results would satisfy the constraints directly. For instance, as the Computer Rating Service takes a constraint on ScreenSize as input, the service executor feeds the constraint ScreenSize = "15 in" to the service.
- **Stage Two**: After executing a service, we check again whether the resulting values agree with the information constraints in the service request. If a list of results are returned by the service, a single result that satisfies all the information constraints is selected. The values in the result are used to instantiate the corresponding information variables. For instance, a number of laptop models may be returned by the Computer Rating Service. Based on the information constraint RequiredRating = 5, one of the laptop models rated as 5 would be selected by our system. In case there is no result satisfying all the information constraints, the system reports an exception to the user who may ignore it, or abort the transaction.

## 5 Implementation

A prototype implementation was created in Java using a model-view-controller architecture pattern. The model consists of the implementations of the various elements of the request oriented model such as the abstract and concrete web service descriptions, functionality and information variables, components, user state graph, dependency etc. The controller consists of processes or steps defined in the request framework such as the user state graph generator and path finder. These processes accept user inputs and use objects defined in the model to produce outputs. The view consists of a user interface that accepts a web service request in WSRL and generates a service execution plan based on the request. A series of empirical and compound tests based on different scenarios are being conducted to determine the average and worst case computational costs of the steps in the framework.

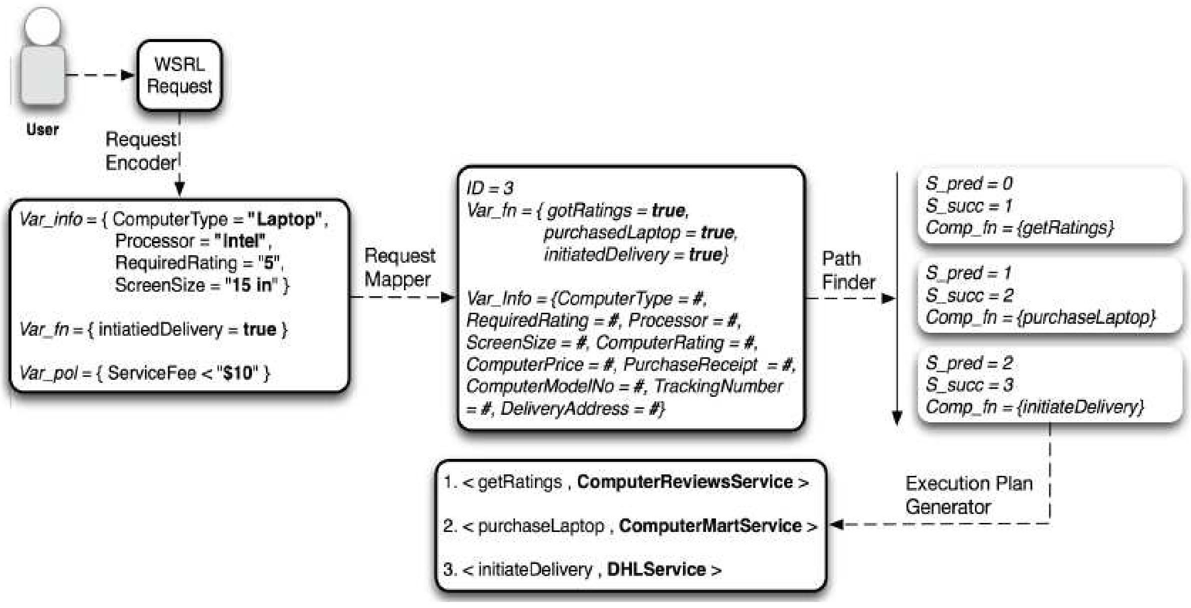
## 6 Related Work

Our work is related to three topics: service modeling, automatic service composition, and customized service delivery, which are important and interrelated research topics in service oriented computing. In this section, we discuss some representative works and differentiate our approach from them.

### 6.1 Service Modeling

Several semantic web service languages have been proposed to realize *semantic web services* to facilitate automatic service discovery and service composition [10]. Representative works include WSDL-S [18], OWL-S [3], and WSMO [19]. WSDL-S extends WSDL, the standard web service description language, with semantic annotations to its input and output parameters with OWL concepts. OWL-S is built on OWL. It provides a set of markup language constructs for describing properties and functionalities of web services from four aspects, including *service provider*, *service capabilities (i.e., input, output, precondition, and effect)*, *service execution process*, *and service accesses* (i.e., service grounding). WSMO is built on WSMF. It describes a web service from four aspects, including *non-functional properties*, *used mediators* (addressing data and process sequence mismatches), *capability* (i.e., precondition and postcondition), and *interface* (describes data flow and control flow).

Existing semantic web service languages mainly focus on a *service oriented* models, which aim at adding machine understandable annotations to service descriptions. As a result, users need to first understand a service description and conform their requests to it. Our approach, on the other side, provides a modeling methodology from user perspectives. It allows the specification of user expectation and experience of using a service which can be atomic or composite, which will facilitate customized service selection and composition. More specifically, instead of modeling and reasoning *states of services*, we define *states of users*, which capture users experiences when invoking services, to better achieve a customized and flexible service composition. We also enrich the logical expression of a



User
WSRL Request
Request Encoder
Var_info = { ComputerType = "Laptop", Processor = "Intel", RequiredRating = "5", ScreenSize = "15 in" }
Var_fn = { intiatiedDelivery = true }
Var_pol = { ServiceFee < "$10" }
Request Mapper
ID = 3
Var_fn = { gotRatings = true, purchasedLaptop = true, initiatedDelivery = true}
Var_Info = {ComputerType = #, RequiredRating = #, Processor = #, ScreenSize = #, ComputerRating = #, ComputerPrice = #, PurchaseReceipt = #, ComputerModelNo = #, TrackingNumber = #, DeliveryAddress = #}
Path Finder
S_pred = 0 S_succ = 1 Comp_fn = {getRatings}
S_pred = 1 S_succ = 2 Comp_fn = {purchaseLaptop}
S_pred = 2 S_succ = 3 Comp_fn = {initiateDelivery}
Execution Plan Generator
1. < getRatings , ComputerReviewsService >
2. < purchaseLaptop , ComputerMartService >
3. < initiateDelivery , DHLService >

Figure 4: Translating a Service Request into a Service Execution Plan

goal by adding a user's data-related expectation [1]. Therefore, a user's expectation is captured in two aspects, *data retrieval* and *user state alternation*, which are used to guide the process of service composition. We incorporate the idea proposed in [6, 22], where services are composed by matchmaking their input and output on a semantic level.

In [5], a Context-Based Web Service Description Language (CWSDL) was proposed to incorporate context information in a service modeling. Context can be internal, which is related to service features, or external, which is related to user features. To help better service, CWSDL allows the specification of a set of context functions, which take context information, such as a user's location, as input to evaluate and rank services. It can also be used to consider user preferences, defined as context functions, when selecting services. Different from [5], our approach, having a user-centric modeling theme, provides more expressive power to describe user preferences on service selection, which can be data centric and state centric. Data centric user preference specifies what the information flow a user prefers to have during the interaction with a service. State centric user preference specified how a user's state varies during the interaction. We use a User State Graph to specify the dependency between states, which can be used to explore all possible execution sequences of a dynamic business processes and then determine a concrete service selection and execution plan.

In [12], Papazoglou et al. proposed an AI-planning based language, XML Service Request Language (XSRL) for requesting services and performing planning under uncertainty. The language models the service domain as a nondeterministic state machine. An XSRL request is encoded into a constraint satisfaction problem and then passed to an AI planner, which automatically generates a service execution plan. As its basic model is nondeterministic, the outcome of an execution plan is usually unknown before the actual execution. To solve this problem, the authors proposed to continuously monitor and modify the plan to deliver the best-effort results. The work in this paper shares the philosophy of XSRL, but aims to provide a less computationally expensive and user friendly approach. We model the service domain as a deterministic state machine, which is only based on services functionality. The unexpected results are handled by the service executor at run time. In addition, our model treats information and functionality differently. In the request system, only constraints on functionality are considered in the planning process, while constraints on information are considered at run time. This allows us to substantially reduce the complexity of the model and the computational cost of planning.

## 6.2 Service Composition

There are many approaches proposed in the literature for automatic web service composition. [14] provides a survey of some AI planning based service composition methods. In these works, a service is modeled as an *action*, which can be performed to change *states*. Each action is specified by its *precondition* and *effect*, i.e., the states before and after performing the action, respectively. Once there is a service composition task, an *initial* state and final state, i.e., the goal, are described. A reasoning process is performed to generate the execution order of actions, so that the system will start at the initial state and reach the final state. These works are supported by existing semantic service models, such as OW-S [3] and WSMO [19], where preconditions and effects are treated as the properties of a Web service and presented as logical formulas. The limitation of these AI planing based approaches is that a goal is usually represented as a state. This does not capture a user's expectation on retrieving information from accessing services, which is a very common web service usage.

Aiming to address this issue, some extended AI approaches to model and automate service composition have been proposed [4, 13]. In [13], automatic service composition is achieved by automatically mapping process-level composite service descriptions to knowledge-level service interactions. A knowledge-level, i.e., data centric, goal is specified. Services are selected and composed, determined by their inputs and the transitions from their inputs to outputs, until the goal is achieved. In [4], the approach allows extensions to a goal specification, such as numeric variables, temporal constructs, and maintainability properties. It also allows specifying goals from two differential aspects: information gathering and state transitioning. We leverage these service modeling approaches and follow their reasoning principle, i.e., planing service invocations to reach a goal.

## 6.3 Service Delivery

Yu and Bouguettaya present a different perspective to the same problem in [21]. They proposed a query algebra for users to issue service requests declaratively. The query algebra aims to provide optimized access to web services based on their functionality and quality. It uses a relational model that abstracts web services based on their functionality. A predefined dependency graph is used to generate a service execution plan.

Different from their approach, our model provides an intuitive interface for users to specify only their objectives rather than exactly what abstract services to use. WSRS allows users to specify their goals in terms of required information and functionality, thereby frees them from specification of services to be used. This allows the request system with enough data to select the most appropriate services at runtime, that may fulfill the users request.

## 7 Conclusion and Future Work

This paper proposes a novel modeling approach for web services, *request oriented model*, which provides users a convenient and declarative way to express their requirement on the usage of web services. We formally define user states and their dependencies, web service descriptions, and web service requests. Based on the model, we propose a formal language, *Web Service Request Language* (WSRL) to express a service request, which can be automatically handled by a supporting framework, *Web Service Request Framework* (WSRF). We have shown through an example that the proposed WSRF can act as a new entry point for multiple web services, creating dynamic service compositions and delivering customized results directly to users. We illustrated that, by differentiating between information and functionality, computational cost involved in service composition could be reduced. As the future work, we plan to perform a comprehensive experimental study, where real-world services are used, to evaluate our approach. We also plan to add dynamic service selection component to our system, to improve its efficiency and robustness.

## References

[1] Serge Abiteboul, Victor Vianu, Brad Fordham, and Yelena Yesha. Relational transducers for electronic commerce. In *PODS '98*, pages 179–187. ACM Press, 1998.

[2] L. Baresi, S. Guinea, and P. Plebani. Ws-policy for service monitoring. In *6th VLDB Intl. Workshop on Technologies for E-Services, volume 3811 of Lect. Notes in Computer Science*, pages 72–83. Springer, 2006.

[3] The OWL Services Coalition. Owl-s: Semantic markup for web services. In *http://www.daml.org/services/owl-s/1.1B/owl-s/owl-s.html*, July 2004.

[4] Eirini Kaldeli, Alexander Lazovik, and Marco Aiello. Extended goals for composing services. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *ICAPS*. AAAI, 2009.

[5] S. M. Kouadr and M. Younas. Context-oriented and transaction-based service provisioning. *International Journal of Web and Grid Services*, (2):194–218, 2007.

[6] F. Lecue, E. Silva, L. Ferreira Pires, and F. Sevigne. A framework for dynamic web services composition. 2008.

[7] X. Liu, A. Bouguettaya, Q. Yu, and Z. Malik. Efficient change management in long term composed services. *Service Oriented Computing and Application (SOCA)*, 5(2), 2011.

[8] X. Liu and H. Liu. Bootstrapping operation-level web service ontology: A bottom-up approach. In *The 7th International Conference on Collaborative Computing (CollaborateCom 2011)*, Orlando, FL, October 2011.

[9] X. Liu and H. Liu. Automatic abstract service generation fromweb service communities. In *ICWS 2012*, Hololulu, HI, June 2012.

[10] S. McIlraith, T.C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems. Special Issue on the Semantic Web*, 16(2):46–53, March/April 2001.

[11] OASIS standard. Business process execution language (bpel). In *https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel*, 2003.

[12] Mike Papazoglou, Marco Aiello, Marco Pistore, and Jian Yang. Xsrl: A request language for web services. Technical report, Internet Computing, IEEE, 2002.

[13] M. Pistore. Automated composition of web services by planning at the knowledge level. In *In 19th Intl. Joint Conferences on Artificial Intelligence*, pages 1252–1259, 2005.

[14] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004*, pages 43–54, 2004.

[15] W3C. Simple Object Access Protocol (SOAP). *http://www.w3.org/TR/SOAP/*, 2003.

[16] W3C. Universal Description, Discovery, and Integration (UDDI). *http://www.uddi.org*, 2003.

[17] W3C. Web Services Description Language (WSDL). *http://www.w3.org/TR/wsdl*, 2003.

[18] W3C. Web service semantics - wsdl-s. In *http://www.w3.org/Submission/WSDL-S/*, November 2005.

[19] WSMO Working Group. Web Service Modeling Ontology (WSMO). *http://www.wsmo.org/*, 2004.

[20] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed. Deploying and managing web services: issues, solutions, and directions. *The VLDB Journal*, 17:537–572, May 2008.

[21] Qi Yu and Athman Bouguettaya. Framework for web service query algebra and optimization. *ACM Trans. Web*, 2(1):1–35, 2008.

[22] R. Zhang, I. Budak Arpinar, and B. Aleman-meza. B.: Automatic composition of semantic web services. In *Intl. Conf. on Web Services, Las Vegas NV*, pages 38–41, 2003.