A Service Composition Model for Dynamic Service Creation and Update in IMS/Web 2.0 Converged Environment

Cuiting HUANG, Noël CRESPI
Institut Telecom, Telecom SudParis
9, rue Charles Fourier
91000 Evry Cedex, France
{cuiting.huang, noel.crespi}@it-sudparis.eu

ABSTRACT
Relying on an IMS/Web 2.0 converged environment, this paper proposes an extensible service composition model which uses the concept of policy for encapsulating the service composition information. Then based on this model, the policy concept is enhanced by being divided into two types (abstract service policy and concrete service policy) for realizing the automatic service composition which mainly relates to the service update and service creation processes. Moreover, an active approach, a passive approach, and a hybrid approach are proposed for balancing the latency and real-time characteristic. With these enhancements, both user’s experience and involvement are greatly improved, the service lifecycle is optimized, and the costs for service creation and maintenance are decreased.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures;
H.3.4 [Information Storage and Retrieval]: Systems and Software

General Terms
Management, Design, Theory

Keywords
IMS, Web 2.0, Policy, Service Composition, Convergence, Automatic, User-centric

1. INTRODUCTION
Service composition, which enables the reuse of existing services and network resources for creating a novel service, has been widely recognized as an important trend for next generation service delivery. The relevant research work has been carried out for years catering to different requirements, nevertheless, the attributes like convergence, automaticity, user-centricity are still the open issues pursued by both industry and academia. Addressing to some of unresolved issues, this paper intends to propose a service composition model relying on an IP Multimedia Subsystem (IMS)/Web 2.0 converged environment for unifying the service composition process for different types of services (Telecom or Web services), simplifying the service creation process and optimizing the service lifecycle through an automatic service creation and update mechanism.

The paper is organized as follows: Section 2 introduces briefly the convergence, user-centricity and automaticity issues. Section 3 provides an overview for our service composition model. Then, section 4 extends the policy concept used in the model and proposes a mechanism for catering to the automatic service composition requirements. In section 5, a use case is analyzed with a discussion regarding the actual service environment. Section 6 concludes by discussing the advantages of the proposed solution and presenting some of our future work.

2. BACKGROUND
Regarding the evolutions in IT and Telecom worlds, convergence is widely acknowledged by both industry and academia as an inevitable trend. One of the most prominent examples is the alliance between Web 2.0 and IMS for benefiting from combining the openness and flexibility of Web and the reliability and trustworthiness of Telecom. Web 2.0 proclaims that it will bring a significant impact on the Internet service provisioning by encouraging the contribution from end-users for innovative services and/or content creation. At the same time, IMS is widely recognized by Telecom world as the reference architecture for Next Generation Network (NGN) by enforcing the service control over IP based infrastructures, enabling the access independent feature, guaranteeing a secure, ubiquitous service experience for user [1]. Apparently, such convergence predicatively will bring a new perspective for next generation service delivery.

Meanwhile, both in the Web 2.0 and IMS environments, service composition is considered as a basic service delivery mechanism that allows the creation and execution
of complex services by integrating the existing distributed service building blocks (called Capabilities or Enablers in Telecom world). However, most of such service composition mechanisms within the IT and Telecom worlds are evolving rather independently from each other based on different underlying infrastructures. E.g., in Telecom, the Serving-Call Session Control Functional (S-CSCF) and initial Filter Criteria (iFC) based service chaining mechanism were specified for IMS based service delivery, the Service Capability Interaction Management (SCIM) [2] was proposed by 3rd Generation Partnership Project (3GPP) and the OMA Service Environment (OSE) [3] was specified by Open Mobile Alliance (OMA) for providing service brokering and interoperability management for Telecom capabilities and services. However, within these entities, there is no such mechanism for cross application domain brokerage and service description.

In order to achieve the goal of composing a new service regardless the types of the element services, we devise a unified service composition model relying on an IMS/Web converged control platform specified in SERVERY project as shown in figure 1.

![Unified service composition model relying on IMS/Web converged environment](image)

Briefly, this service composition model relies on a unified Service Composition Engine (SCE) and an IMS/Web converged control plane. This converged control plane is named as Web Multimedia Subsystem (WMS) by SERVERY. It provides a common set of control capabilities, media capabilities and service delivery capabilities that are required to allow end user to access converged Web, media and IMS services seamlessly through different types of terminals. It mainly contains a WMS Control Plane which leverages existing network standards to provide the common capabilities on all service access paths, and a WMS Media Plane which provides the multimedia content in a controlled way. To be noted that this article does not intend to illustrate the IMS/Web converged control plane in details, but focuses on the service composition engine relying on it which enables users to reuse the existing IMS or Web services for creating their personal services. The more detailed specifications for WMS can be referred to SERVERY [4].

As one feature of this proposed service composition model is extensibility which enables us to enrich it with the user-centricity and automaticity attributes. Nowadays, users are involved in the value-added chain: they are not only regarded as the consumers for the services, but are also encouraged to be the contributors for the content resources or personalized services. Much work has been done and many solutions have been proposed based on Service Oriented Architecture (SoA) and using varying technologies like business process management, AI planning, semantic and ontology, context-aware, etc. However, most of the currently proposed frameworks or systems for composing service are static or semi-automatic; even some famous user-centric solutions like YahooPipe [5], MashMaker [6], and MARGMASH [7] still need human intervention for completing a service creation. Such human invention is always not friendly enough for the ordinary users by comparing the actuality that most of them have no technical knowledge neither on Telecom nor Internet. In order to reduce the manual operation, to hide the backend complexity, and to improve the efficiency of composite service creation and execution, in the following sessions, an automatic mechanism is presented for addressing the automatic composition issue relating to the service creation and the update processes, and eventually achieving the ultimate goal of user-centricity.

3. SERVICE COMPOSITION MODEL WITHIN IMS/WEB 2.0 CONVERGED ENVIRONMENT

3.1 Architecture Overview

From the functional view, the main role of SCE is to enable the reuse of the existing capabilities, which relates to the actions like service creation, service exposure, service discovery, and service orchestration. Figure 2 is a simplified model for the SCE linked to the converged IMS/Web 2.0 environment.

![Service composition engine in IMS/Web 2.0 converged environment](image)
This model adopts “Policy” concept for encapsulating the service composition and orchestration relevant information. Such policies can be considered as the formalisms used to express business, engineering or process criteria. More concretely, a policy can be represented by a combination of a condition, and an action to be performed if such condition is true. The “action” in policy can be an invocation of a function, script, code, or workflow. And the “condition”, it can be Boolean predicate that yields true or false, or it can be an extension to some other machine readable data [8].

3.2 Main Components
The main components in Service Composition Engine comprise: Service Policies Repository and Policies Enforcement which reside in the Composite Services Module; Service Executor, Context Repository and various adaptors which locate in the Service Orchestration Module; Capabilities Binding, Capabilities Policies Repository and Policies Comparator in Capabilities Module.

3.2.1 Composite Service Module
In order to manage the aggregation and the interoperability among different capabilities, SCE needs the service composition information for controlling the accesses to the service capabilities. Such information is stored in Service Policies Repository as a service policy. Each policy in Service Policies Repository is associated to a unique combined service, and it specifies a sequence of service capabilities to be invoked, as well as indicates which capability will be invoked in which condition.

And the policies for security, charging, Service Level Agreements (SLAs), logging, Quality of Service (QoS), privacy and so on can be enforced by Policy Enforcement. It also provides a generic mechanism to control access for third party applications to use these service capabilities, e.g. whenever a third party is authorized to access a resource in the operator’s network, a policy is assigned to it.

3.2.2 Service Orchestration Module
In a converged service environment, the incoming service invocation requests may be SIP based (e.g. sent by Telecom terminals) or HTTP based (e.g. sent by Internet terminals, or web pages, or some widgets). For ensuring a unified user experience, all of those SIP or HTTP based requests should be adapted by SCE. Thus, some additional adaptors are considered as necessary. One of such adaptors is a Protocol Adaptor. Consequently, different protocols based requests are sent to Protocols Adaptor firstly. Then according to the predefined rules, Protocol Adaptor extracts the necessary service information from those newly received service invocation request, likes the unified Service ID, and formalizes them into a uniform HTTP based request for the subsequent steps. Afterwards, Protocol Adaptor transfers the extracted service information to Service Policies Repository. Depending on the received information, Service Policies Repository searches the corresponding policy from its directory for the composite service which user demands to execute, and then transfers it to Service Executor.

After receiving a policy from Service Policies Repository, Service Executor performs the services chaining by transferring the service invocation request to the corresponding capabilities. Thanks to the Exposure Layer which can rely on Parlay X Gateway [9], or other service exposure frameworks [10][11], even the Telecom service enablers can be invoked as the web applications (e.g. Widgets, Portable Service Elements) or accessed through the APIs, and that enables the service chaining regardless of the services’ types (IMS or Web). Meanwhile, Service Executor updates the service execution information which has been stored in Context Repository for memorizing the demanded composite service. Then Service Executor begins to execute the capabilities chaining logic indicated in the retrieved policy by sequentially routing the service invocation request among multiple service capabilities.

During the processing of the request, each capability performs its own specific role. After the execution of the service capability, a response is sent back to SCE with the execution result. According to the received response and by comparing with the policy, Service Executor decides to forward the request to next service capability for continuing the capabilities chaining, or terminate the capabilities chaining and reply to user with the execution result.

Moreover, in a complete service chaining procedure, it needs to ensure the seamless transition from one capability to the next. The intermediate information such as user information and other contextual information should be maintained during the service chaining process, and such information is also stored in Context Repository.

3.2.3 Capabilities Module
The Capability Module relates to the service exposure and service creation processes. Referring to WS-Policy [12] and WS-PolicyAttachment, policies can be used to represent the capabilities and requirements of an existing reusable service. Thus, in our service composition model, once a new capability is introduced into the network or an existing capability is updated, it advertises its capabilities and requirements to SCE as a policy. This means, the newly introduced or updated capability initiates a connection between itself and SCE – we assume each capability server has an additional module which contains the description of its functionality and usage requirements, and is responsible for initiating the connection to SCE, and then it embeds a policy which contains its information into the exchanged messages that may rely on Extensible Markup Language (XML). When SCE receive these messages, it extracts the Capability Policy from them and stores it at a policy repository.
The Capabilities Binding module provides the specific formats to access these actual capabilities according to the information presented by service capability policy.

3.2.4 Service Creation Environment

Service Creation Environment is an environment which empowers users to create a new service by facile method. Such service creation environment can be Natural Language Enabled Service Creation Environment (e.g. SPICE Natural Language Composer [13]), Friendly Graphical Service Creation Environment (e.g. YahooPipe), widget based service creation environment (e.g. EzWeb), or Integrated Development Environments (e.g. Eclipes IDE).

After user validates a new created composite service in the friendly service creation environment, the service creation environment abstracts the requirements for the new service according to end user’s input and sends it to SCE (more concretely, relying on Policies Comparator component) for selecting the appropriate capabilities for this composite service. Then, it generates a service policy for this composite service and stores it in the Service Policies Repository. Finally, Service Policies Repository sends the newly introduced service policy to an adaptor (e.g. widget adaptor) for automatically generating a widget or the client code for different types of terminals.

4. ENHANCED POLICIES FOR AUTOMATIC SERVICE COMPOSITION

Above introduced service composition model is easy to be extended for satisfying different requirements without underlying modification. Automatic service composition is an example which is enabled by extending the policy used in our previously proposed model.

4.1 Extended Policy: Abstract Service Policy and Concrete Service Policy

Firstly, in order to realize the automatic service composition functionality, we split the service composition process into two processes: Abstract Composition and Concrete Composition. Abstract composition consists of defining an appropriate combination of the functionality tasks and their data dependency. Concrete composition consists of determining the most appropriate services from a set of functionally equivalent ones for each functionality task. Both processes result in the generation of a policy: an Abstract Service Policy or a Concrete Service Policy. In this solution, when a composite service is created at design time, a loose coupling service invocation logic is generated and represented by an Abstract Service Policy. Abstract Service Policy defines the service functionalities combination through the high-level functional tasks description for each execution step as shown in Figure 3. In Figure 3, each node is related to an abstract service which can be mapped to one or several existing concrete capabilities which are functionally equivalent. In order to facilitate the concrete service discovery, each functional node description, it relates to a unified abstract function ID.

According to the abstract service policy and the capabilities policies, Policy Comparator automatically selects the appropriate concrete capabilities for this composite service, and creates a concrete service instance which is represented as a policy based on some predefined rules and constraints. Thus, an abstract service policy may be instantiated several times in respect to the individual user’s requirements, and several instances may be running concurrently.

Based on above improvements for the model, the automatic service composition functionality can be further enhanced. Generally, the service composition process can be divided into three phases: service creation, service update and service execution. And the automaticity feature is primarily relevant to the service creation and service update phases as introduced hereinafter.

4.2 Automatic Service Creation Process

With above mentioned enriched policy, our proposed service composition model is empowered with automatic service creation feature. When a service is created by a professional or non professional user, the user only needs to define the functions to be engaged in this created service. He does not need to know or define precisely which existing capabilities are actually invoked and how such concrete capabilities can be accessed and to be configured. That will greatly facilitate the service creation process and encourage non professional user to create more personalized services since the user is not required to have application development skills and does not need to spend
much time to get to know about the concrete capabilities and network environment which are always very difficult or even impossible for non professional user to comprehend.

A reference process for utilizing extended policy in automatic service creation process is as follows: when a non professional end-user wants to create a personal service through some friendly service creation environment, such as a Natural Language Enabled Service Creation Environment or a Friendly Graphical Service Creation Environment, he can input his requirements by natural language or wiring some instantiated building blocks, and the backend processor (e.g. a Lexical Syntax Engine) will parse the information (e.g. user defined functionalities, user’s preference, necessary contextual information) retrieved from service creation environment, and formalize it into a readable user service creation request which encapsulates an abstract service policy as well as the auxiliary information which is mainly germane to the non-functional properties, and then send it to SCE. Depending on the information extracted from the received service creation request, such as the abstract functional IDs included in the abstract service policy, Policy Comparator retrieves the corresponding capabilities policies. After consulting the predefined capabilities selection rules, and analyzing the received service creation information (e.g. abstract functionalities, data dependency), concrete service capabilities requirements (e.g. goal, output and input requirements, constraints), as well as some other contextual information (e.g. user’s preferences, user’s location, user’s terminal capabilities), Policies Comparator selects an appropriate capability for each abstract function ID. Eventually, after an additional verification for the behavioral consistency among these selected capabilities, Policies Comparator creates a new service policy which represents the capabilities invocation logic and stores it in the Service Policies Repository. If the result of the behavioral consistency verification is negative, the service capabilities selection process will be re-performed by Policies Comparator for finding out a replaceable service capability with the equivalent functionality.

Since during the abstract service policy definition process, which may be handled by friendly service creation environment, the service node can be a basic abstract service, but can also be a composite abstract service. If the node is a composite abstract service, SCE should be able to decompose it into several basic abstract services. On the other hand, in order to balance the time consuming, SCE should also be empowered to map the abstract composite service to a concrete composite service directly without the decomposition action.

4.3 Automatic Service Update Process
It is common that most capabilities are updated regularly, or there are some variations in the highly dynamic service environment such as new services become available, existing services become obsolete, or business rules change. Once a modification occurs, it is necessary to perform some corresponding adjustments in the service invocation logic and the corresponding service binding in order to cope with these exceptional situations with minimal user intervention. This derivates the concept of automatic service update which essentially means: when the capabilities or service execution environment have been modified, the relevant composite service logic and the corresponding configuration is automatically adjusted, either by modifying the corresponding configuration parameters, or by re-selecting another appropriate capability for this composite service. From the SCE’s point of view, this automatic update process can be “passive” or “active”.

4.3.1 Passive Update Process
The “passive” update means SCE is always on listening state. All the updates for the composite service logic are triggered either by the incoming capability policy published by capability, or by the runtime service execution result.

As show in Figure 4, each composite service is linked to a couple of policies: abstract policy and concrete policy. The abstract policy describes the functionalities of the service and the invocation logic. The concrete policy indicates the selected concrete capabilities and their triggering conditions. Both the abstract policy and the concrete policy are created at design time and are bound together.

When a new capability is introduced into the network or an existing capability is updated, it advertises its capabilities and requirements to the SCE. Since such capability policy contains at least an abstract function ID and a unified capability ID, once SCE receives such capability update request, it searches in Capability Policies Repository for finding out if a previously advertised policy from the same capability has been stored according to the unified capability ID indicated in the received policy and in the previously stored policy, and then makes a decision for this incoming policy: Create or Update.

If the decision is Create, a new item is added into Capability Policy Repository for storing the incoming capability policy, and a relevant capability binding item is generated and stored in Capabilities Binding component according to the necessary information extracted from this
incoming policy. Meanwhile Capability Policies Repository also forwards this capability policy to the Policies Comparator. According to the Abstract Function ID in the capability policy, Policies Comparator retrieves all the corresponding service policy couples which contain the same Abstract function ID. By analyzing the information stored in the incoming capability policy with the relevant concrete service policies, Policies Comparator decides whether the concrete service policy should be updated or not. If the outcome of the decision is positive, the previously selected capability will be replaced by the newly introduced capability in the concrete service policy.

If the decision is Update, the Capability Policy Repository replaces the previously stored item by the new one, and the corresponding item in Capabilities Binding component will also be updated if necessary. Then, Capability Policies Repository transfers the policy to Policies Comparator. According to the Concrete Capability ID, extracted from the newly updated capability policy, Policies Comparator retrieves the relevant service policy couples from Service Policies Repository. By analyzing these service policies and the update capability policy, Policies Comparator makes a decision whether the existing service policy should be adjusted or not according to the requirements indicated in the updated capability policy. If the decision is positive, the corresponding service policy will be updated and then restored into the Service Policies Repository.

In above case, we assume that once the capabilities update, they send their capability policies to SCE successfully. However, there also exists the exceptions that such capability policy update process fails and SCE always keeps the obsolete information for the capability, or even all the capabilities information stored in SCE are the latest ones, but the capabilities accesses or executions are failed due to some external reasons like there are some conflicts among the involved capabilities after their updates. In such situations, to ensure user experience, the automatic update process should be triggered by the runtime service execution result. Thus, the abstract service policy is transferred to Policies Comparator, which reactivates the service discovery process and re-creates a concrete service policy that is rebound to previous abstract service policy by replacing the old concrete service policy.

### 4.3.2 Active Automatic Update Process

The “Active” solution means that SCE will check the Capability Policies Repository regularly to make sure that all the selected capabilities for the composite service are the optimal ones at any given time.

In this case, as shown in Figure 5, Service Policy Repository only needs to store the abstract policies which are enacted by Service Executor whose responsibility, in this case, is to process messages by notifying completion status related to an abstract composite service and by subsequently scheduling next abstract function that needs to be activated. Hence, once the user logsins and executes the abstract service policy, Policy Comparator is invoked simultaneously. Before the execution of each abstract function, according to user’s information and preference, Policy Comparator contacts Capability Policy Repository at runtime for discovering the most appropriate concrete capability that fulfills the requirements specified in the abstract function definition. And eventually it routes the service invocation request to this newly selected capability in order to execute it. After the execution of the selected capability, a service completion notification is sent back to Service Executor with the execution result. Following necessary processing for this response and according to the predefined invocation logic, Service Executor continues to contact Policy Comparator for mapping a successor abstract function with the concrete capability as described above.

![Figure 5. Service policy for active update](image)

In the service execution process, Service Executor should ensure the service execution integrity since the runtime service selection can not guarantee that all the abstract functions can find out the appropriate concrete services and execute them successfully until the termination of the last execution step. Consequently the Service Executor has to provide a compensation mechanism for certain irreversible actions. If Policies Comparator can not discover an appropriate capability for current abstract function, then all the running services are aborted and the completed ones are compensated by executing a service-specific compensating action for ensuring that either all the services are executed or none is. An alarm is sent to the user for demanding the modification of current abstract composite logic or user’s preference. If the selected capability can not be executed successfully, e.g. the access to this capability is not authorized for this user, Service Executor re-contacts with Policies Comparator for selecting another capability with the same functionality but with lower priority.

### 4.3.3 Comparison and Discussion for Passive and Active Solutions

Comparing with the passive update solution, the runtime performance of active update will be improved, since all the selected capabilities are the optimal ones at any time, even for the case when the capabilities are being modified during
the composite service execution process. Nevertheless, the possible execution latency will be a problem and the QoS can not be guaranteed, since Policies Comparator needs to interact with Capability Policies Repository and select the capability before each capability execution which is both time and resource consuming, especially when no capability can be discovered during the service execution process or the discovered capability can not be executed successfully.

For the passive update solution, there will be no time latency brought by the service discovery process, since a complete set of concrete composite services have been discovered and selected at the design time, and the service policy update process and the service execution process can be performed in parallel by executing the already defined executable one. But it will incur the limitation for real-time characteristic, especially considering the possibilities when the capability policy advertisement process is failed, or there is no such advertisement mechanism for some capabilities, which may lead to the service execution failure that the selected capability can be accessed at design time, but is not accessible at runtime due to the real-time update.

Considering above comparison, a hybrid solution may be able to balance the time-consuming and the real time characteristic as shown in Figure 6.

The Policy Repository only stores the abstract policy as the active solution does, and service discovery is performed neither at design time, as passive solution, nor at runtime, as active solution, but just after the user’s login and before the execution of the service. Thus, once the user logs in, Policies Comparator retrieve an abstract policy and contacts Capability Policies Repository for defining a concrete policy for this composite service according to user’s login information, the runtime capabilities information and requirements, which is similar to the concrete service policy creation at design time for passive solution.

5. USE CASE AND DISCUSSION

In order to illustrate this proposed model more vividly, a use case “MSN robot like service composition” is analyzed based on above introduced model.

As shown in Figure 7, the backend service composition platform can be instantiated as a MSN robot like contact in the Contact List, which enables to reuse MSN authentication mechanism, the existing contact information, and ease the service creation process for end user without any additional installation or configuration. Thus, user just needs to add the email of such “contact” into his Contact List, and then he can talk with this “contact” by entering his request, e.g. “Send my next week available time slots to John”, for creating a composite service which involves the network agenda service, the presence service, the message service, and executing it at runtime.

A natural language interpreter linked to this “contact” extracts the information from user’s request and structures them into a specified format which encapsulates an abstract service policy, and forwards them to SCE. According to the received information, Policies Comparator selects the relevant service capabilities and defines their invocation logic which will be represented as composite service policy. Then this service policy is forwarded to Service Executor. After the execution of the explicitly corresponding services (get the agenda information from Network Agenda Capability) and the auxiliary services (get the John’s status from MSN Contact List, if John is offline, invoke the SMS capability for sending a SMS with user’s available time slots’ information got from Agenda capability), a final invocation result is sent back to the interpreter, and eventually this interpreter sends back a response to user such as “The message has been sent to John”.

Moreover, user can store his created service by entering “Save” for replying the question asked by this MSN robot. Then, the created abstract service policy and the concrete service policy are stored in Service Policies Repository, and are materialized as a contact displayed in user’s Contact List. Thus once user clicks this contact, the communication window displays below content:

![Figure 7. MSN robot likes service creation interface](image)

After user enters the time and destination’s contact name, this service is re-executed as illustrated in upper part. This user created service is also empowered with the automatic update feature due to the automatic update mechanism introduced in Section 4. For example, if the selected concrete SMS service has been updated (e.g. augment the
price for sending SMS), this service update information invokes Policies Comparator for re-selecting a replaceable service with the same SMS functionality but with lower price for catering to user’s preference requirements (e.g. the SMS price does not exceed certain limitation).

To create such composite service, end-user is not required any development skill and does not need to know which exact services (e.g. SMS service provided by Orange, or by Telefonica) selected form the capabilities repository. And the service update process is transparent for end user and does not require any user intervention.

As reflected in this specific use case, the main interest of the proposed approach lies upon the enhancement of user-centric feature. Considering the user-centricity, the Quality of User Experience (QoE) is a crucial factor which has to be taken into account. Many service creation platforms have been proposed by different communities nowadays, however, most of them are still far from being popularized for the ordinary users due to their complexity. It is evident that service composition is a really complex task, and it is difficult for the ordinary user’s to deal with the whole process manually, even he has some development skill. Such complexity is due to the facts that the dramatically increasing number of services makes it arduous to find out an appropriate service from a huge service repository. Moreover, the highly dynamic service environment requires that services can be created and updated on the fly, thus the service composition system needs to detect the creation/update demand and make the corresponding decision at runtime which is apparently inefficient for all being handled manually by user. These are the main reasons that spur us on to propose a service composition model which hides the backend complexity from user and simplifies the service creation interface, as well as satisfies the service creation variety requirements.

Evidently, to realize above service composition model, the intelligent service discovering and selecting mechanisms are indispensable for considering that the quality of the service composition result much depends on the selected concrete services. Some current service discovery solutions mainly rely on the technologies like Ontology [14], lexical database (e.g. WordNet), and data mining [15]. In order to complete our proposed model, a further investigation for the service discovery will be performed in our future research by referring to these technologies.

6. CONCLUSION AND FUTURE WORK
Considering the user-centric and convergence trends, this paper explored a novel service composition model relying on the IMS/Web 2.0 converged environment for facilitating innovative service creation by non professional user. The advantages of this enriched service composition model are explicit: it facilitates the service creation process for user, especially for non professional user, by enabling the automatic service selection, composition and monitoring. Thus, both the user experience and user’s involvement level will be greatly improved. It reduces the service maintenance cost for service providers due to the automatic update mechanism, since each modification, either from the capabilities side or from service environment, will invoke the adjustment on the corresponding composite service automatically. Moreover, it enhances the cooperation among different parties (e.g. end-users, service providers, operators). Finally, it optimizes the service lifecycle and enables delivering innovative services at a more rapid pace.

As it is an on-going work, we will continue our work in the service composition by refining current service composition model. Not only an intelligent service discovery approach, but also the mechanism for balancing the great amount of information received from end-users and service providers for reducing the possible end-to-end latency will be taken into account in our future research endeavors.

7. REFERENCES
[1] 3GPP TS 23.228, “IP Multimedia Subsystem (IMS)–Stage2”
[3] OMA, “OMA Service Environment”, OMA-AD-Service-Environment-V1_0_4-20070201-A