A Web-based Mashup Environment for On-the-fly Service Composition

Qi Zhao, Gang Huang, Jiyu Huang, Xuanzhe Liu, Hong Mei
Key Laboratory of High Confidence Software Technologies, Ministry of Education
School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China; {zhaoqi06, huanggang, huangjy07, liuxzh}@sei.pku.edu.cn; meih@pku.edu.cn

Abstract

The web-based service composition, e.g. mashup, is becoming a popular style to reuse web services. From the perspective of reuse, existing work has limitations on qualifying whether the service or the service composition satisfies user requirements and adapting the service or composition according to the qualification results. For addressing these limitations, this paper proposes an on-the-fly approach to web-based service composition. Firstly, we do not distinguish the design-time and run-time of services and their composition so that they can be qualified in a what you see is what you get manner when services are selected or assembled. Secondly, we propose a component model for separating the service business and user interface so that they can be changed dynamically and independently in the adaptation of service selection and composition. This approach is demonstrated by a browser-based mashup tool.

1. Introduction

Currently, there are many services published via Internet with open APIs, such as Amazon S3 and Google Map. More and more developers are able to access these services through web and assemble them to construct their own applications, e.g. [4][5], so-called mashups.

Many existing work [2][3][4][5] provided web-based service composition environments, in which the service business logic and User Interface (UI) are encapsulated into a single component, called web-based service (WBS) components. Developers are able to create applications by assembling these components in web-based environments, such as web browsers, in a rich user experience manner. During the composition, WBS components are loaded into the environment and then become available for design. Such design-time WBS components only have appearance but without any real functionality. Developers can configure these 'fake' components (e.g. setting its layout, changing the fonts and colors) and assemble them. Once composition is completed, developers should change these 'fake' components to runtime. Then the components will be really instantiated and assembled. However, the strict separation between design-time and run-time brings some serious limitations on software reuse, especially for qualifying WBS components and their composition.

Firstly, developers should qualify each WBS component before assembling them. Given a set of components, qualification means to check whether the proposed component satisfies a given set of requirements [1]. In current web-based composition environments, there are a large amount of WBS components. Each component has its own pros and cons. In particular, rich user experience is one of the most important differences between the web-based service composition and other component-based or service-based composition. Hence, it is hard to qualify one component whether it satisfies the requirement without really using it. Nevertheless, the separation between design-time and run-time prevents developers from qualifying WBS components effectively. Because when components are loaded into the environment, they cannot be qualified directly since they are just in design-time and do not have real functionality.

Secondly, even if each component has been qualified, there are still many problems which prevent components from being assembled in a correct and desired way. Accordingly the composition of these components should be qualified as well. In current web-based environments, composition qualification is always carried out through interfaces checking that can find out the number and type mismatch of parameters. However, even if interfaces are matched, there may still have some mismatches. For instance, two date strings may be in different formats. This mismatch cannot be found if the composition does not execute really. Therefore the separation between design-time and run-time makes the mismatch hard to be found.

Unlike traditional off-the-shelf components that need compilation and deployment, services are actively running entities [8]. Such a significant difference implies that the separation between design-time and run-time is not an inherent nature of web-based service composition. In other words, it is possible to assemble services in an on-the-fly manner, that is, 1) when a service is loaded into the composition environment, it becomes available with real appearance and functionality immediately, 2) when two services are assembled, they can interact with each other actually. Developers can qualify a service by invoking it directly and checking the real result immediately. They can also qualify a service composition

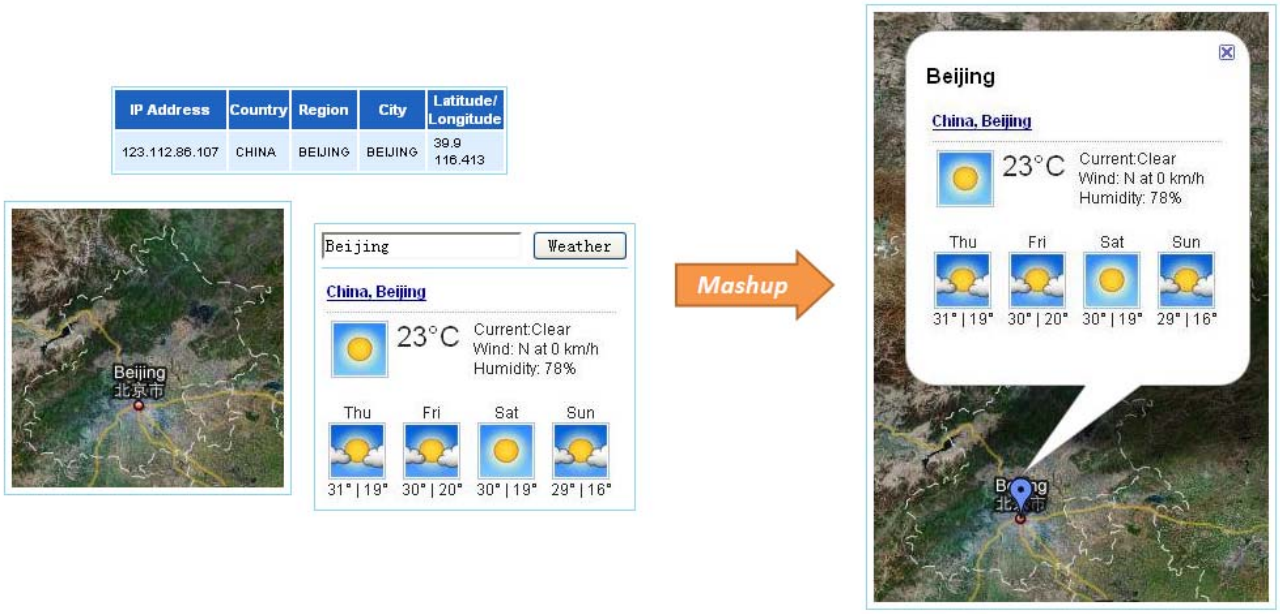

Figure 1 Motivation Scenario

by activating the service flows in the composition and checking the real changes of the services. Briefly, WBS components and composition can be qualified in a more effective and better user experience manner. However, existing web-based service composition environments do not take the on-the-fly way.

Qualification is always coupled with adaptation, which changes the service or composition more or less for passing the qualification. Just like on-the-fly qualification, on-the-fly adaptation is needed for web-based service composition. WBS components encapsulate UI, which is the most variable element considering the fast evolving composition context. If on-the-fly adaptation is not supported, developers may need to modify component source codes, redeploy components and reload them into the environment. Such offline adaptation makes on-the-fly composition impossible. Nevertheless, in most current WBS component models, such as current mashups, UI is described by fixed markup files and cannot be modified. Moreover, WBS components also include some interaction logic which manipulates the relationship between UI and services. But the interaction logic and UI in current WBS component models are tightly coupled. Hence, once UI is modified, the interaction logic will break down. These limitations prevent developers from adapting components and composition in an on-the-fly manner.

This paper proposes an on-the-fly approach to web-based service composition and implements a browser-based mashup tool[1]. First, we present a composition environment. Neither the environment nor components in it are distinguished design-time and runtime, Thus when developers select and assemble components, they can qualify them and their composition in a "what you see is what you get" way. Secondly a component model is proposed, which decouples service business logic and user interface. It ensures that components and their composition adaptation can be carried out in on-the-fly manner.

The rest of this paper is organized as follows. Section 2 illustrates a sample scenario to explain the problems in web-based service composition we mentioned above. Section 3 gives an overview of our approach. Section 4 provides the model of our component and explains how it supports on-the-fly adaptation. Section 5 presents the details about how our composition environment supports on-the-fly composition. Section 6 gives some discussions. Finally, we discuss related work in Section 7 and conclude this paper in Section 8.

2. Motivation Scenario

We begin with a typical scenario to explain the problems in WBS composition mentioned above. This scenario is a weather map, as shown in Figure 1, which is similar with a popular mashup, Weather Bonk. In this scenario, there are three WBS components: A city weather forecast component, which displays 24-hours weather for a given cities; a Google Map which displays locations of given addresses and a geo-position detector, which can find users' geographical position from their IP addresses. The weather map displays 24-hours weather of the user's geographical position on the map. It can be implemented by assembling the three components.

In this scenario, we can find some problems mentioned above. Firstly, there may be several available map components, such as Google Map and Yahoo Map. Google Map component is suited for our requirement,

[1] A prototype is open source at http://sourceforge.net/projects/imashup/

because in specific cities its information is richer than Yahoo Map. However, there is no effective way to discover this without really using it. Secondly, when developers assemble the geo-position detector and map, the geo-data of detector and map may be mismatched because of an offset. Therefore position markers may display at wrong places. This kind of mismatch cannot be found by interfaces checking. Accordingly, when developers use a traditional composition environment, this mismatch cannot be found until the application is deployed. Finally, the weather forecast information should be nested in the maps position markers and redundant UI elements (i.e. city search input) should be removed. Therefore, there should be an on-the-fly way to customize UI and assemble them together. Otherwise these components are hard to be used in an on-the-fly composition process.

3. Approach Overview

In this section, we give a brief overview of our on-the-fly composition approach, which is briefly illustrated in Figure 2.

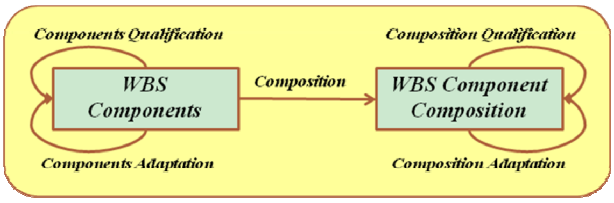

Figure 2 On-the-fly Approach Overview

Differently from other WBS component models, our component model is not distinguished as design-time and run-time. Once loaded, each component is at runtime with full and actual functionality. Therefore developers can qualify components by really using it. If components do not satisfy the requirements, developers could adapt them on-the-fly. Our component model decouples the UI and service business logic. Furthermore the model leaves the capability of configuring UI structure and presentation to developers. Briefly, developers can qualify and adapt components on-the-fly.

On the other hand, since components are always at runtime. When developers assemble them, components can be connected and services can be invoked just-in-time. This enables developers to qualify current composition immediately. Moreover, UI of composition has the same structure as UI of a component. Therefore composition adaptation can be carried out on-the-fly since component adaptation has this ability.

Considering our weather map scenario, developers should first use the searcher panel to search and retrieve the three components into environment. The components will be initialized once they are retrieved, and then displayed as the left part of Figure 1. Next, developer can interact with these components (i.e. searching some cities' weather forecast) for qualification. If the components do not satisfy the requirements completely, they can adapt with the configuration panel. In our scenario, developers will nest the weather information into the map and remove city search input. For on-the-fly adaptation, the UI will change immediately once developers provides new UI configuration. The adaptation result should be like the right part of Figure 1. And then developers can assemble the components. Since our component model and composition environment support on-the-fly capability, qualification can be carried out simultaneously with composition. For example, position markers will display immediately once the connection sets up. If any geo-data mismatch exists, the position markers will display at wrong places. Developers can discover it just in time and resolve it.

4. Web-based Service Component Model

As we mentioned above, two reasons make current WBS components hard to support on-the-fly adaptation. First, UI of components is described by fixed markup files and cannot be modified. Second, UI is tightly coupled with interaction logic and then the interaction logic may break down once UI modified. Our component model tries to resolve the problems.

Our WBS component model consists of two parts, interface and implementation. The interface of component model consists of User Interface and Programming Interface since WBS component model encapsulates UI. The component model is shown in Figure 3 [10].

The implementation of our WBS component model adopts the Model-View-Controller pattern. 1) The model implements business logic by invoking a remote service. 2) The view constructs and manages UI elements. An element of UI can be an atomic HTML element or a group of HTML atomic elements. 3) The controller manages the interaction logic between model and view. The controller part consists of several element controllers. Each element controller encapsulates the interaction logic of one specific UI element.

The programming interface exposes the business logic of WBS component. It consists of properties, methods and events. Properties describe the state of a component and can be queried and modified. Methods can query and modify the component state. Events notify changes of the component state. WBS components interact with others by their programming interface.

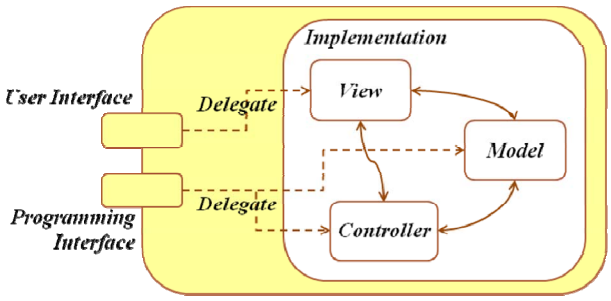

Figure 3 WBS component Model

The UI responds to users' actions and invokes the corresponding functions in the implementation. An interface of a reusable component consists of two parts [1]: the fixed part, which consists of the part of the component that must be used, and the variable part, which depends on the particular use. Our WBS component UI makes elements in UI as the fixed part, while presentation and structure of the elements as the variable part. The structure defines the elements' place and their relationship, while the presentation defines the elements' presentation information such as size and color. When developers reuse the WBS component, they can adjust configurations to adapt UI with particular scenario.

```
/* City Weather Forecast Component Uesr Interface*/
{
  elements: [
    {name : 'keyword'},
    {name : 'searchbutton'},
    {name : 'weatherdisplayer'}
  ]
}

<!-- Weather Forecast Structure Configuration-->
<div class='comp_container'>
  <element component-id='weather_forecast1'
    name='keyword' />
  <element component-id='weather_forecast1'
    name='searchbutton' />
</div>
<span class='clear_element' />
<div class='comp_container'>
  <element component-id='weather_forecast1'
    name='weatherdisplayer' />
</div>
```

Figure 4 User Interface and Configurations

In our example, the weather forecast component's programming interface includes "keyword" property, "search" methods and "onSearchCompoleted" events. The component also has three UI elements, a key word, a button and a weather displayer, and it has two element controllers: one is a button controller which responds to button clicked event and calls weather search functionality, the other is a weather displayer controller which gets and displays the result. Figure 4 gives the weather forecast component UI elements, default structure configurations.

Another problem which prevents on-the-fly adaptation is how to adapt the interaction logic while keeping the component correct when some elements are changed or removed. According to the controller of our WBS component model is divided into element-combined controllers, UI elements can invoke proper business logic and respond to return results through a specific controller no matter where it is placed and how it is presented. Moreover, if a UI element has been removed, the specific controller will be removed automatically while not affect the other UI elements and controllers as well.

5. On-the-fly Service Composition

To support on-the-fly composition, components should have no strict separation from design-time and run-time. However, some of current WBS components and environments, such as [2][3], are implemented by compiled languages, which separate design-time and run-time strictly. Hence, when developers use these environments, they have to compile and deploy composition results if they adapt or assemble components. That makes on-the-fly composition impossible.

Some other WBS components and environments [4][5] are implemented by dynamic languages and then on-the-fly composition is possible in these environments. Nevertheless, the components in these environments are still separated as design-time and run-time. Accordingly, developers in the environments still need to change components from design-time to run-time to carry out qualification, and change back to design-time if there is any error in composition. That makes components and composition hard to be qualified.

In our approach, dynamic languages and the no separation between design-time and run-time are the most important in design rationales, which lead to a web-based service composition environment with on-the-fly capabilities.

5.1 Retrieving Components

The first step of service composition in our web-based composition environment is retrieving components. In order to support the on-the-fly composition, our WBS component model does not distinguish between design-time and run-time. Also, our composition environment does not follow the design-run-debug cycle. The composition environment is hosted in a web browser and implemented by JavaScript, which is a dynamic script language. Hence, the environment can load WBS component definitions without restarting. Also, because a WBS component is implemented by JavaScript, a component can be instantiated without compilation and deployment once its definition is retrieved. After being instantiated, a component is at runtime. It connects with a running service and can respond to user actions with full functionalities. Therefore developers can qualify the

component in on-the-fly manner. After qualification, components are ready for being assembled.

5.2 Programming Interface Composition

We define three types of connectors to enable components to interact with others' functionalities through programming interfaces, as shown in Figure 5.

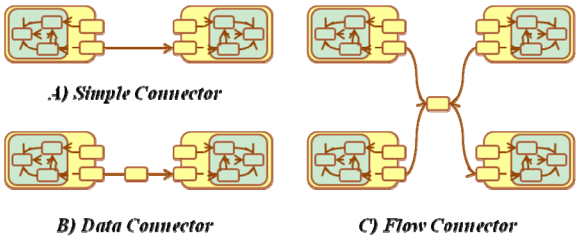

Figure 5 Component connectors

The first connector type is simple connector, which enables two components to interact directly. This connector supports a one-to-many publisher/subscriber relationship among components. That is, one component publishes an event, and other components subscribe to it. The publisher/subscriber relationship is specified via event listeners. Each listener specifies an event publisher, event type, event subscriber, and a method of the subscribing component.

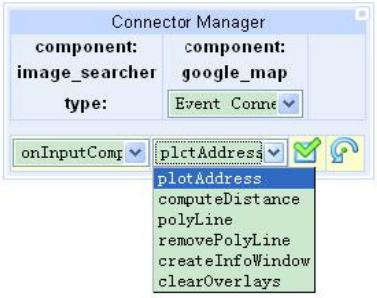

Figure 6 Connector Manager

Since programming interfaces of component models describe the methods and events of components, our composition environment can get this information at runtime by using reflection. Thus developers can select required methods and events, assemble them together in an on-the-fly manner. Figure 6 is a snapshot of connector manager of composition environment.

However, in many cases, data of multiple components may be in different formats, which prevents components from being connected directly. The second connector type is data connector handling data format mismatch. Data connectors work with the built-in data wrappers that handle specific kinds of data transformation, e.g. content-filter. Developers are allowed to program a JavaScrpt function to enhance the functionalities of connectors, in case the logic of data transformation is too complex to be handled by built-in data transformers. Since JavaScript is an interpretively-executed language, there is no need to restart or redeploy the composition environment, and the whole composition process is on-the-fly.

With the data connector, the data format mismatch can be resolved. But when trying to build a complex flow that involves more than two components, developers may feel frustrated if they can only connect two components at one time. So our composition environment provides the third type of connector, flow connector. Each flow connector corresponds to an integration-pattern [9], like "Aggregator" or "Process Manager", to support a special kind of flow. The flow connectors are implemented as a special type of WBS components, which have blank user interfaces and connect with other components through programming interfaces. So a flow connector can interact with other components through a simple connector, a data connector or even another flow connector. It enables users to build complex flow easily. And it also ensures the composition process is on-the-fly.

Considering every component is always at runtime, components can be connected just-in-time. Service invocations may be caused by this connection and these invocations may modify component internal states and UI. Thus developers can qualify the assemble results immediately.

5.3 User Interface Composition

Besides of programming interface composition, our composition environment also allows users to assemble user interface of WBS components. When developers assemble WBS components, their UI elements will be merged into a new UI. Then developers need to adjust the structure and presentation configuration of this new UI. The new UI's configuration is the same as the atomic component UI's.

Furthermore, to support the on-the-fly UI composition, our environment keeps detection on modifications of UI's configuration. Once a new configuration is set, it will be immediately applied on the environment and the new UI will be displayed. So developers can retrieve the feedback of assemble results immediately.

The structure configuration of weather map is shown in Figure 7. In this scenario, the input and button of the city weather forecast component, along with the UI of the geo-position detector, are removed from the configuration. Moreover, since the weather information is displayed in the position marker on the map, the xml element corresponding to the city weather forecast component is nested in the xml element of the map in the configuration.

```
<!-- Weather Map Structure Configuration-->
<div class='comp_container'>
  <element component-id='map1' name='map'>
    <element component-id='weather_forecast1'
      name='weatherdisplayer' />
  </element>
</div>
```

Figure 7 UI Composition Structure Configuration

6. Discussion

As we have observed that the on-the-fly service composition ability does benefit qualification. There are still several open issues to further address.

First, our WBS components encapsulate service and UI. It is not an easy task to create WBS components. Hence, a WBS component builder or SDK, which helps developers to create WBS components in a "what you see is what you get" way, is required.

Moreover, to adapt WBS components, developers should write XML configuration by hand now. It makes component adaptation not easy-to-use. Therefore, A XML configuration panel which supports developers to adapt components in a visual way is required.

7. Related Work

The web-based service composition has involved a lot related work. In [2][3], some fundamental work about web-based service component was discussed, including the basic WBS component model and event-based composition model. Nevertheless, the component model cannot be used in on-the-fly composition since it separates the design-time and run-time strictly. Furthermore, this component model of the work is not suitable for on-the-fly adaptation, since its UI is fixed and hard to be customized.

Thousands of mashup applications [4][5] already exist, which allow users to create web applications by using widgets. However, current mashup tools always separate design-time and run-time. Therefore, in these tools, components and composition qualification is hard to be carried out.

Java Portal [7] lets users customize composite pages with full-fledged, pluggable components called portlets. However portlets, the name of WBS components in Portal, should be compiled, packaged and deployed into portals before developers can qualify it. Briefly, the qualification in Java Portal is hard.

8. Conclusion and Future Work

The web-based service composition is becoming a popular composition style in Service Oriented Computing. And on-the-fly ability is a significant difference between traditional components and web-based services. This paper makes the following contributions.

1) We present an on-the-fly approach for web-based service composition. With this approach, developers can qualify components and their composition in a more effective and better user experience manner.
2) We propose a component model which decouple service business logic and user interface. This component model can be adapted in an on-the-fly manner and suited for on-the-fly composition.
3) We provide a browser-based composition environment prototype to demonstrate our approach.

As we mentioned above, there are some open issues for our composition environment. In the future, our composition environment will have a more powerful configuration panel, which allows developers to adapt components in a visual way. And a WBS components builder is under development and will be open source soon.

References

[1] H Mili, A Mili, S Yacoub, E Addy. Reuse-based software engineering: techniques, organization, and controls. Wiley-Interscience New York, NY, USA, 2001.

[2] J. Yu et al. A Framework for Rapid Integration of Presentation Components. In the Proceedings of WWW'07, Banff, Canada, May 2007.

[3] F Daniel et al. Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities. IEEE Internet Computing, 2007.

[4] DataMashups: http://www.datamashup.com

[5] Yahoo! Pipes: http://pipes.yahoo.com

[6] Oscar Diaz, Salvador Trujillo, Sandy Perez. Turning Portlets into Services: The Consumer Profile. Presentation Components. In the Proceedings of WWW'07, Banff, Canada, May 2007.

[7] Java Portlet Specification. <http://jcp.org/aboutJava/communityprocess/final/jsr168>.

[8] Girish Chafle et al. An Integrated Development Environment for Web Service Composition. In the Proceedings of IEEE International Conference on Web Services 2007.

[9] Gregor Hohpe, Bobby Woolf, Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions, 2003.

[10] Qi Zhao, Gang Huang, Xuanzhe Liu, Jiyu Huang. Towards a Component Model for Web-based Service Composition. Journal of Frontiers of Computer Science and Technology, Vol.2 No.4, 2008, p378-389.