

## A DESCRIPTION-BASED HYBRID COMPOSITION METHOD OF MASHUP APPLICATIONS FOR MOBILE DEVICES

KORAWIT PRUTSACHAINIMIT, TAKEHIRO TOKUDA  
*Department of Computer Science, Tokyo Institute of Technology  
Meguro, Tokyo 152-8552, Japan  
{korawit, tokuda} @ tt.cs.titech.ac.jp*

Received September 1, 2015  
Revised January 22, 2016

Mashup application composition methods have been proposed for quick development of new mobile applications from existing resources. The existing methods have succeeded in developing data-flow mashup applications. However, they have limited capability to create event-driven mashup applications. A full treatment of data-flow and event-driven mashup composition is not yet achieved. This paper presents a new methodology for developing data-flow and event-driven mashup applications for mobile devices. Our hybrid composition method allows integration of mobile applications and REST Web services in a data-flow and event-driven manner. Description-based techniques and application generator tools are applied to reduce development cost. A mashup development system is implemented in Android mobile environment as the first experimental platform. The evaluation results show that our method is expressive and efficient in composing mobile mashup applications.

*Keywords:* Mobile device, description-based mashup, event-driven application  
*Communicated by:* M. Gaedke & O. Diaz

### 1. Introduction

Mobile devices, such as smartphones and tablets, have recently gained popularity and become common computing devices. As a result, millions of mobile applications are published through major delivery channels covering a variety of user's requirements. Even though a great number and huge diversity of mobile applications are available, they are still not covering the long tail of users' requirements [1]. This situation drives the need for quick development of new mobile applications. However, developing mobile applications is not practical for non-programmers or even for novice programmers, because extensive knowledge of mobile application development and intensive programming skills are required. Thus, recently, tools assisting mobile application development have emerged. Mobile mashups, which employ the concept of lightweight composition of existing resources, are one of the efficient tools supporting rapid mobile applications development.

Mobile mashups allow the creation of new applications by using lightweight composition of existing resources. They take advantage of a combination of mobile Internet, Web service APIs and device-specific components for enriching mobile services and enhancing user experiences [2]. In particular, mashup applications composed using mobile device's information are able

to create new results that cannot be achieved by using conventional mashup composition [3]. For example, a smartphone's camera can be used to create a mashup application that scans a barcode of a book, and then finds the review information from an online bookstore using Web services [4]. Therefore mobile mashup composition becomes a capable tool that reduces programming efforts and required skills for mobile application development.

Practically, mobile devices monitor their states and report changes in an event-driven manner. Changes of states, such as call status, current location or battery level, are notified to other processes as events. Mashup applications can take great advantage of the event notification by using it as a trigger to control the execution of mashup components. For example, we can create a mashup application that automatically shows a list of restaurants around the current location by using a "location-changed" event as an input to a restaurant search Web service. Event-driven mashup applications that automate tasks such as putting the device into airplane mode when arriving at a specific location are also possible. The event-driven composition model can expand the expressivity of mashup composition on mobile devices by allowing the integration of various mobile events with a huge number of available Web services. Thus, we believe that the event-driven mechanism is a compulsory attribute of mashup composition for mobile devices.

Although a number of mobile mashup approaches are proposed [5], they still have limited capability regarding event-driven mashup composition. Since most mobile mashup approaches inherited the composition techniques from well-engineered Web mashup frameworks, they employ data-flow composition style and aim for enabling data integration. Some mashup frameworks are designed to support event-driven composition. However, the event-driven capabilities are limited to specific domains such as location-aware applications, telephony functions or user interface integration. A hybrid composition method that supports functional integration in both data-flow and event-driven as well as dealing with general events of mobile devices is not yet achieved.

To enable hybrid mashup composition for mobile devices, we first explore a novel mashup composition model. We set up practical scenarios that represent usage of both data-flow and event-driven mobile mashup applications, and derive a suitable component integration model and an efficient mashup development process. To reduce development cost, we apply a description-based mashup development technique where description languages are used as an input to mashup generators in order to generate mobile mashup applications. The first prototype of our mashup development tool is designed and implemented on Android platform. We also evaluate the applicability of our approach by conducting a user experience study. In summary, the main contributions of this paper are as follows:

- We propose a new mashup composition method for developing data-flow and event-driven mashup applications that utilize the functional integration of mobile applications and REST Web services.
- We present description languages and generator tools that allow developing mashup applications with minimal programming effort.
- We show that our approach improves the efficiency of mobile mashup application composition, particularly the expressivity of user requirements, reusability of mashup components, and robustness of mashup execution.

The organisation of this paper is as follows. We briefly explain difference of data-flow and event-driven mashup composition, discuss the limitations of data-flow mashup composition and review related research in the next section. The motivating scenarios and research aims are presented in Section 3. An overview of the composition method, mechanism of mashup component integration and mashup development process is explained in Section 4. Section 5 describes the implementation of the mashup proxy, mashup orchestration messaging and design of the description languages. We show the result of usability evaluations, explain the boundary of expressivity, and discuss current problems and possible improvements in Section 6. Finally, we give our conclusion in Section 7.

## 2. Background and Related Work

The objective of this paper is to present a mobile mashup approach that deals with data-flow and event-driven mashup composition on mobile devices. In our view, most of the current composition methods and models target on the data-flow style of composition. To clearly define our research problems and state the motivations, background information about characteristics and limitation of data-flow mashup composition, the definition and scope of event-driven mashup composition, advantages of the event-driven composition style and related research on mobile mashups are discussed in the following subsections.

### 2.1. Characteristics and Limitations of Data-flow Mashup Composition

In general, data-flow mashup composition is based on the data-flow programming paradigm in which mashup components are connected as a directed graph and communicate via message passing. The mashup resources, such as Web content and mobile device’s information, are combined as a workflow; passing input/output parameters between connected components to produce mashup output [6].

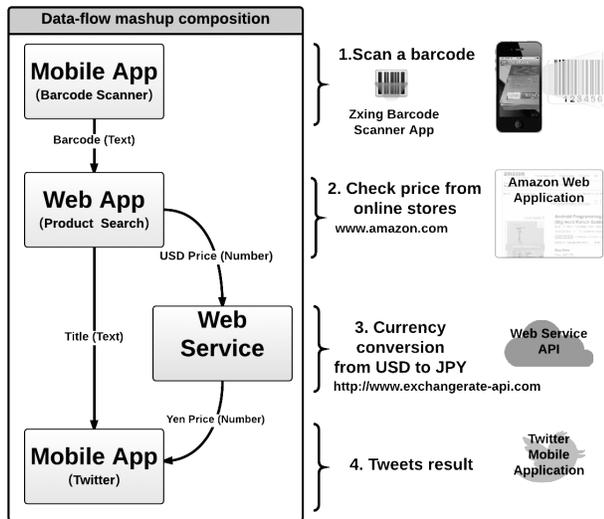


Fig. 1. A sample of data-flow mashup composition on mobile device.

Figure 1 shows a sample scenario for a data-flow mashup application running on a mobile device. This mashup application helps users to search for product information on an online store by using device's camera as a barcode scanner. The mashup then converts the price of product into a specific language and shares the information on Twitter. With data-flow mashup composition, the first mashup component, i.e. the barcode scanning application, takes a photo of a barcode and converts it to a barcode number and sends this as an input to the next component. The next component is a product search Web service that takes the barcode as an input to query title and price of the product. Then, the price is converted into a specific currency by a currency conversion Web service component. Finally, the product title and price will be used as input for Twitter applications to create a post on a social network.

The data-flow composition pattern can represent various kinds of mobile mashup applications and it is efficient. However, using only data-flow composition patterns limits the capability of mashup composition for mobile devices. For example, let us consider a popular location-aware mashup application that displays point of interest (POIs), e.g. restaurants, hotels or ATMs, around a user's current location. By using a data-flow composition, the mashup application gets the current location and passes it as an input to a location-based Web service API. The Web service then finds coordinates and additional information of POIs, and displays them as pins on a map. In a real-life situation, mobile devices may move to a new location after the mashup execution is completed. As a result, the displayed pins of POIs are no longer valid. Users have to manually execute the mashup application every time they want to get updated POI locations, or let the application set a time interval to update the result. The missing mechanism of this mashup application is the capability to listen for a specific event and automatically execute the mashup to update the result. We can consider this to be an event-driven mashup.

## **2.2. Event-Driven Mashup Composition for Mobile Devices**

Event-driven mashup composition is based on event-driven programming, an architectural style in which one or more components in a software system execute in response to receiving event notifications [7]. We can define event-driven mashup composition as a mashup composition style in which one or more mashup components in the application logic are executed in response to receiving an event notification. In other words, event-driven mobile mashup applications work as a set of autonomous components that monitor their internal state for changes, and give notice of events to other components as well as listening to events produced by other components. When an event notification occurs, the components that are listening to the event invoke functions or services of other components. Thus, we can compose mashup applications in an event-driven style by pairing an event listener component with an event handler component. Figure 2 shows a comparison of mashup composition models between data-flow and event-driven composition.

In additions, the event-driven composition style is suitable for mobile mashup. Mobile devices, such as smartphones and tablets, keep track of context information by monitoring and reporting that information to users. Changes of state, such as a location, mail receipt, and battery level are informed to users. These notifications can be viewed as events, which can be used as triggers to develop event-driven applications. As a result, event-driven mobile applications are now popular. There are many commercial applications that take advantage of

mobile events. These applications can help users to automate pre-defined tasks when a specific event occurs. For example, a popular Android application called “Taskers” [8] can listen for changes in time, location, hardware/software states and system events, then executes one or more of 200 actions, such as launching other applications or making a phone call. While many mobile applications are utilizing the mechanism of event notification to expand coverage of user’s requirements, mobile mashups still cannot take full advantage of these features.

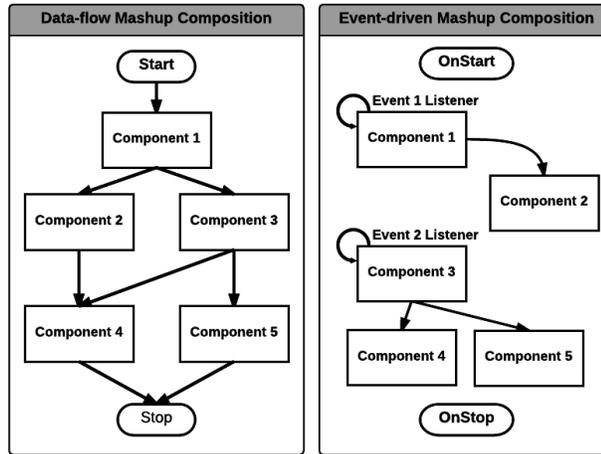


Fig. 2. The comparison of data-flow and event-driven mashup composition model.

The event-driven execution style also improves robustness of the mashup applications. For data-flow mashup composition, the component execution is done in synchronous manner where users start the mashup application and wait for the result. When problems occur during the execution, such as a loss of the network connection or Web resources becoming unavailable, the mashup application will be interrupted or terminated. In contrast, event-driven mashup can deal with these problems by performs the component execution in asynchronous manner. After starting the mashup application, users can switch to other applications while the mashup application is running in the background. However, the data-flow component execution is still required because not all mashup components support event-driven execution, and data-flow component execution gives better performance regarding functional integration.

### 2.3. Related Work

The techniques of mobile mashup frameworks inherited composition methods from Web mashup. Web mashup editors such as Yahoo Pipes [9], Intel MashMaker [10], and a mashup platform proposed by Kaltofen et al. [11] are capable of creating mobile mashup applications. They create mashup application as Web applications and use mobile Web browsers as a channel to deliver mashup result to mobile users. Since these mashup tools create Web-based mashup applications, they cannot fully access and make use of mobile device’s information.

Research on context-aware mobile mashup generally applies event-driven composition. They use a specific set of mobile events as a trigger to start mashup processes. The TELAR mashup platform [12] presents a solution to combine location data from GPS sensors with

existing Web service APIs to create a mobile mashup application that displays POIs around the current location. Lively for Qt [13], a mobile Web mashup implemented in JavaScript, also presented the possibility of building location-aware mashups that utilize GPS information. The IVO framework [14] presents the ECW (Event-Condition-Workflow) model that allows building context-aware applications by creating workflows that decide the application logic when a specified event is detected. These even-driven mashup approaches usually focus on mashup composition for a specific domain, such as location-aware applications or context-aware applications.

Telco (Telecom) mashup is another research topic that is targeted to mobile devices. Telco mashup frameworks [15] accommodate the integration of telephony functions with Web services. The EasyMobile composition tool [16] enables mashup composition by using telephony functions and mobile events, such as incoming call status or change of location, as a trigger to start mashup processes. Similar to context-aware mashup, Telco mashups allow integration of telephony functions with Web contents using a set of pre-defined mashup components.

Tasker [8] and Trigger [17] are samples of popular Android applications that employ the concept of mashup on mobile devices. These applications help users to automate common tasks such as setting ringtone volume, switching Wi-Fi on or off, or launching other applications. A task-automation is composed of a situation and set of actions. The situation can be a certain time, location, incoming call, phone orientation, and so on, while the actions can trigger system settings on the smartphone directly. Once again, the capability of these applications is limited to pre-defined sets of situations and actions. Importantly, they cannot integrate their components with Web resources such as Web services.

To take full advantage of mobile device's information, novel mashup approaches generate mashup applications as native mobile applications. Cappiello et al. introduced MobiMash [18], a model-driven approach for developing mobile mashups as native mobile applications. However, this approach focuses on data integration and service orchestration rather than taking advantage of mobile device's information and functional integration. Our previous research [19] proposed a mashup construction system for the integration of Web applications, Web services and mobile applications. This approach realizes the integration of Web resources and device-specific components and employs data-flow as the main composition method. Event-driven composition is also possible, but it requires manual programming efforts.

As discussed above, mobile mashup approaches have been proposed to facilitate mashup composition for mobile devices. There are a number of mashup platforms that use data-flow and event-driven composition patterns. However, the proposed methods still have limitations when it comes to mashup composition for mobile devices. The existing approaches cannot take full advantage of mobile device's information. Most mashup platforms are designed to work as a client-server system, which mashup applications run on a server and client mobile devices access the mashup applications via mobile Web browsers. In this way, the capability to access device-specific components such as locations, user information or sensor data, is limited to the capability of mobile Web browsers. Even though JavaScript APIs of modern browsers can access device-specific features [20, 21], it is still limited when compared to the capability of the native APIs of a mobile platform. Thus, we target our mashup approach to develop mashups as native mobile applications in order to maximize usage of mobile device's information. In addition, the existing approaches overlook the advantages of functional integration, especially

using mobile applications as mashup components. Apart from system APIs and sensors data of mobile devices, mobile applications can be used as valuable mashup components since they provide both functions and events that can be integrated with other types of mashup components. Thus, in this work, we aim to allow functional integration of mobile applications and REST Web services in data-flow and event-driven manners, as well as utilizing general events and device-specific components of mobile devices.

Since we aim to reduce the mashup development cost by using description-based techniques, description-based mashup approaches, i.e. mashup description languages [22, 23, 24], are reviewed. We found that these novel works only focus on modelling Web resources. For using mobile applications as mashup components, we have presented a description language for modelling the functionality of mobile applications to allow interoperability between them [25]. The proposed description language can be applied to enhance component integration of mashup approaches. However, additional extensions are needed to support event-driven mashup composition.

### 3. Scenarios and Analysis

In order to explore issues around hybrid mashup composition for mobile devices, we setup three scenarios that illustrate characteristics of the hybrid composition pattern for mobile mashup applications. These scenarios were selected from different aspects in order to reflect important challenges for enabling both data-flow and event-driven paradigms within mashup composition. Given these scenarios, we derive requirements needed to achieve hybrid mashup composition and discover challenges that need to be solved.

#### 3.1. *Motivating Scenarios*

The first scenario is an “event-trigger mashup”, which is used in well-known commercial mobile applications such as Tasker and Trigger. It allows a user to create event-based automation by listening for an event and performing additional data-flow logics when the event is fired. The second scenario is a “data streaming mashup”. This mashup handles the stream of data that is continuously produced from mobile devices as events, and automatically executes additional logics to update mashup results. For the final scenario, we aim to study how to use event-driven composition to represent mashup compositions that are usually done in a data-flow manner.

- (i) **Email Translator.** This scenario focuses on a common event-trigger application where an event notification of a mobile device is used as a trigger to execute additional mashup logics. This scenario simulates a requirement for translating the content of emails. The situation is an international student studying in Japan. He/she uses an Android smartphone for receiving emails from colleagues and university newsletters. However, sometimes the content of the emails is written in Japanese. In general, email client applications do not provide a translation function. When a new Japanese-language email is received, he/she has to copy and paste the content into a translation application and save the result in a note application. To solve this problem, a mobile mashup application that helps translating email from Japanese to English is needed. In our view, this application can be created by using event-driven mechanism with integration of existing

mobile applications and Web services. We can listen for new incoming emails using an email client application, translate the content to a specific language using a translation Web service and save the translated content to a note application. In addition, the application should allow him/her to specify which email will be translated by creating filters for email from a specific person or email containing a specific keyword.

- (ii) **Around Me.** The second scenario is a data streaming mashup. It is a mashup application that continuously receives notification of events as a stream of data. We reuse the common scenario of a location-aware mashup that continuously monitors the “location-changed” event and integrates the location with Web services to display POIs on a map [26]. This scenario simulates a situation where a tourist is travelling in a foreign country. While he/she is traveling in a city, he/she wants to find a restaurant around his/her current location by using a smartphone. Using a map application, e.g. Google Maps, he/she can use the “Local Search” feature to search for restaurants nearby. However, when he/she walks around or moves to another place, the search result becomes invalid. Thus, he/she has to do the search again to update the restaurant location on the map. In this situation, he/she needs a mobile application that keeps track of the current location and continuously pinpoints nearby restaurants. In the other words, the mashup application should automatically update the result when the “location-changed” event is fired.
- (iii) **Barcode Book Review.** The third scenario focuses on a problem that is usually solved using data-flow mashup composition. To compare event-driven composition with data-flow composition, we reuse a scenario presented in [19]. The scenario simulates a mobile mashup application that helps users finding information about selected books from online stores by using a smartphone’s camera as a barcode reader. The situation depicts a developer who wants to buy a book from a bookstore. Before deciding whether to buy or not, he/she wants to compare the price of the local bookstore with that of online stores. He/she also wants to read reviews and comments of people who have read that book. With his/her smartphone, he/she needs an application that uses the camera as a barcode reader to scan a book’s barcode. The scanned code is then used as an input for a bookstore and a book review Website to get the title, price, description and first review entry of the selected book.

### 3.2. *Requirements*

Based on the scenarios in Section 3.1, we derive objectives that our approach should fulfil and address challenges that we should overcome.

**Maximize usage of system event and mobile application events.** In our sample scenarios, we found that a key feature is integration of device-specific information with Web resources. Information produced from sensors and the mobile operating system, such as a “location-changed” (in Scenario ii.) and a “photo-taken” (in Scenario iii.), are used as triggers to start the mashup process. In addition, mobile applications can act as event producers by sending changes of their state to notify other processes. For example, mobile applications, such as email clients (in Scenario i.), social network applications and etc., always send notifications about their states to users and other applications. In our view, these notifications are valuable

components in mashup application composition. Thus, our approach aims to deal with events produced from the mobile operating system and events produced from mobile applications.

**Encourage reusability of mashup components.** From the scenarios, we found that mashup components can be reused for more than one scenario. However, reuse of components requires different component configurations. For instance, scenarios i. and iii. use the same translation Web service for translating Japanese to English and English to Japanese. In practice, major configurations of the translation Web service in both mashup applications are identical. However, some execution parameters, such as source and target language, may be different in each mashup application. Thus, in order to enhance reusability, the mashup components should be flexible enough to be reused in different mashup compositions. For example, the mashup components should provide a user interface for managing execution parameters at run-time in order to maximize reusability.

**Support an event-driven execution model.** In a conventional data-flow mashup, users start the mashup application and wait for the result. We found that the data-flow execution style is not suitable for our sample scenarios. For instance, in Scenario ii., users may start the mashup application and move to other locations. The mashup application requires event-driven execution style that deals with this situation by automatically updates the mashup result when location is changed. Another drawback of the data-flow mashup execution is the mashup applications have to run in the active context of mobile device. Users cannot switch to others applications during the mashup execution. Thus, event-driven mashup execution model is required to overcome these problems. For example, in Scenario i., users must be able to use other mobile applications while the mashup application is running in background. The mashup application must be activated when new email is received. To enable event-driven execution model, mashup applications have to listen for events in background, hold up the mashup execution, and stop the listening process when a condition is met or a user action is received. Given this condition, resource consumption, such as the number of background processes, should be well managed.

**Simplify the mashup development process.** Our approach aims to provide tools that simplify the component configuration and mashup composition process. We target all sample scenarios to users who do not have skills in programming. The technical knowledge on developing mashup components, as well as composing mashup applications should be reduced. However, when we adjust the design toward simplicity, the expressive capability will be reduced. Therefore, another challenge is to make a balance between simplicity of configuration and capabilities of the approach.

#### 4. Our Approach

The key concept of our approach is the orchestration of mashup components that represents functionality of actual mashup resources and provides a uniform integration interface. Instead of performing integration between mashup components directly, our approach uses a separated process to control the orchestration and enhance loose coupling among components. Communication among mashup components was simplified by an event bus system adapted

from the state of the art in event-driven architecture [7]. A mashup composer can use an XML description language to serialize configurations of mashup components and plan the composition logic. The description language is used as an input to the code generator tool to generate the mashup components and the output mashup application. The final output is in form of a native mobile application that is deployed to the target device and executed as an ordinary mobile application. In the following subsections, we describe the architectural overview of our approach. We also highlight the key ideas and techniques that we have applied to deal with challenges in enabling hybrid mashup composition on mobile devices.

#### 4.1. Overview of Architecture

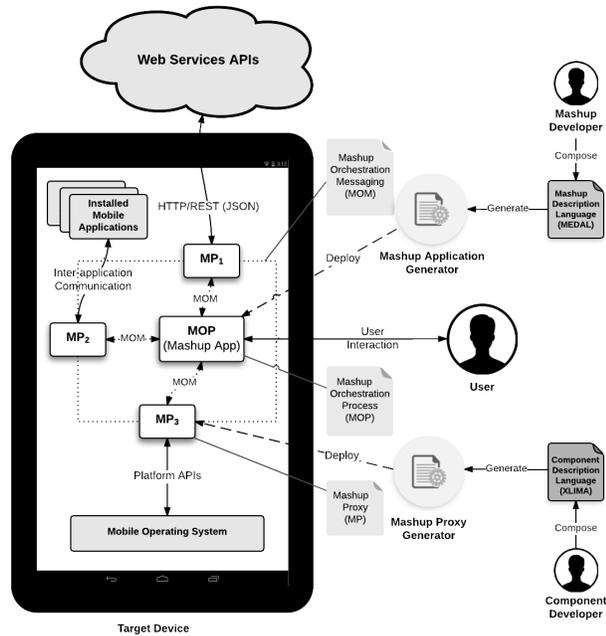


Fig. 3. Overview of architecture.

An overview of the proposed architecture is shown in Figure 3. The mashup process relies on cooperation between a Mashup Orchestration Process (MOP) and Mashup Proxies (MP). The MOP is responsible for controlling the execution of mashup logic and managing the orchestration of mashup components. It provides user interfaces that allow a mashup user to start/stop mashup process and view the output of mashup application. The MP is used to facilitate component integration. It acts as an intermediary between the MOP and the actual mashup components in order to transform a technology-specific programming interface into a common integration protocol, called Mashup Orchestration Messaging (MOM). The MP works with three types of mashup resources (REST Web services, mobile applications and system functions of a mobile operating system). They also augment the mandatory event-driven mechanism to the ordinary mashup components. More specifically, each MP listens to

actual events of a particular mashup component by using a proper programming interface, and sends notifications of events to the MOP in the form of MOM. The MOP then uses this notification and its parameters as a trigger to start particular mashup logic. To call a function of a mashup component, MOP sends a message and parameters to a corresponding MP via MOM. The target MP then translates the message into the proper programming interface and invokes the target function for a result. Finally, the result from the invocation is transformed to the common format (MOM) and sent back to the MOP.

#### 4.2. Mashup Orchestration Messaging

This section presents the details of the orchestration process between MOP and MPs, which is the key mechanism of our mashup composition approach. We use an orchestration process and proxy components to manage the integration of mashup components, as well as providing loose coupling in the integration. A messaging system and a delivery channel are used to facilitate the communication between MOP and MPs. A conceptual model that describes the integration process is illustrated in Figure 4. The model depicts a simple event-driven mashup process that performs the integration of two mashup components. This model emphasizes two key mechanisms, event listening and function calls, which are important for enabling event-driven component integration. The model consists of the following components.

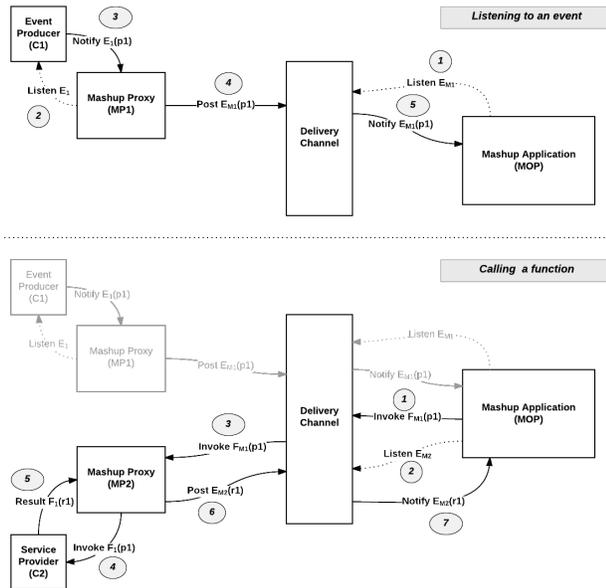


Fig. 4. Conceptual model of mashup component integration.

- *Mashup Orchestration Process (MOP)*. This is a process that controls the execution of mashup logic and interacts with a mashup user.
- *Event Source (C1)*. This component represents a mashup component that produces notifications to other components, i.e. system events or notification messages from installed mobile applications.

- *Service Provider (C2)*. This component represents a mashup component that provides functionality, i.e. REST Web services or mobile applications that contain reusable functions.
- *Mashup Proxy (MP1 and MP2)*. These components work as intermediaries between actual mashup resources and the mashup orchestration process. Each actual mashup component has a designated MP to transform a technology-specific programming interface into a common programming interface.
- *Delivery Channel*. This component is a communication channel that delivers messages to facilitate the orchestration between MOP and MPs.

The model describes the minimum building blocks of mashup orchestration messaging, which is the invocation of a mashup component when an event is fired. The goal is to use an event E1 of event source C1 as a trigger to invoke function F1 of service provider C2. In order to get a result from F1, the mashup application has to extract parameters from event E1 and use them as input parameters for invoking F1. The detailed steps of integration are grouped into two major tasks, which are listening to an event and calling a function.

#### **Listening to an event.**

- 1 MOP listens for EM1 event messages via the delivery channel.
- 2 MP1 is used as an intermediary component between C1 and MOP. It listens to an actual event E1
- 3 Once C1 produces event E1 with parameter p1, MP1 transforms the technology-specific event E1 into a common event message EM1 including parameter p1.
- 4 MP1 posts the transformed event onto the delivery channel to notify the MOP.
- 5 Once the MOP receives the event notification, it extracts parameter p1 and uses it to invoke function F1.

#### **Calling a function.**

- 1 MOP sends an invocation message FM1 with parameter p1 through the delivery channel.
- 2 MOP then listens for a callback event EM2 to process the result.
- 3 Once MP2 receives the invocation message, it transforms the message into the proper programming interface.
- 4 MP2 forwards the call F1 and parameters p1 to actual component C2.
- 5 After C2 has finished the execution, it informs MP2 as a callback F1 with a result r1.
- 6 MP2 then transforms F1 with return parameter r1 into an event message EM2 with return parameter r1.
- 7 MP2 posts the transformed event onto the delivery channel to notify the MOP.

Finally, the MOP receives the notification (EM2 with return parameter r1) as an event, and then extracts r1 for using in the rest of the mashup process.

For the orchestration process, our model allows integration in an asynchronous manner, particularly when calling functions. In general, mashup components invoke a function of other components using synchronous messaging, i.e. sending an invocation message and waiting for a result. However, the synchronous method may not work well on mobile devices because they are often moved to different locations. Changing the location of devices may cause

network unavailability problems that interrupt the execution of resource-consuming mashup components, e.g. invoking a Web service function. Thus, performing asynchronous calls and waiting for notification of a result as an event can increase robustness of mashup applications.

### 4.3. Mashup Development Process

In general, mashup development consists of two major processes: component development and mashup composition. The component development is the process for describing functionality and properties of mashup components, while the mashup composition is related to defining how components interact with each other. Developing a high quality mashup component requires high-level technical knowledge that end-users cannot accomplish by themselves [27]. For example, developing a Web service component requires a good understanding of the REST protocol and the JSON data format. Therefore, in order to optimize the efficiency of the overall mashup development process, our approach divides the development process into three stages: 1) Component Development, 2) Mashup Composition and 3) Mashup Execution. Each state is designed for a different group of users, which require different levels of technical skill. An overview of the mashup development process is shown in Figure 5.

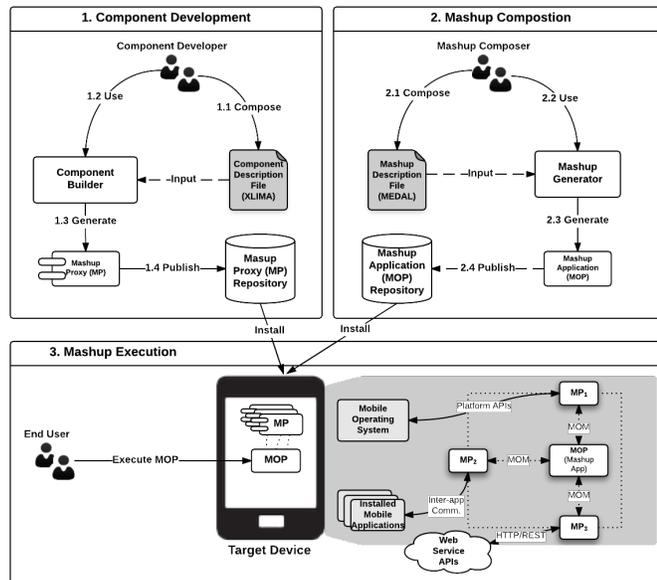


Fig. 5. Overview of mashup development process.

**1 Component Development.** The goal of this process is transforming actual mashup resources into reusable mashup components. To develop a mashup component, the component developer composes a component description file that contains configurations of functionalities and events. This file is then used as an input for the component builder tool to generate the mashup proxy. At the end, the generated mashup proxy is published to the repository. In order to efficiently describe the functionalities and events of each

component, the component developers require the knowledge of related technologies, i.e. Web service APIs, inter-application integration protocols, XML and JSON. Thus, we designed this process for an IT specialist or a programmer who understands the stated technologies.

- 2 **Mashup Composition.** This is the process of building a new mashup application by using existing mashup components. The process starts with a mashup composer describing composition logic of a mashup application and saves it as a mashup composition file. During the composition, the mashup composer can query the component repository to see which mashup components are available. After finishing the design of the composition logic, the mashup composition file is used as an input for the mashup generator tool in order to generate output mashup application. Then, the output mashup application is stored in mashup application repository. This process requires less technical knowledge than the component development, since the description in this stage is abstracted to be technologically independent. Thus, the mashup composer can be a programmer or an advanced user who understands idea of mashup (i.e. concept of operations, events and parameter mapping) and experienced in composing XML document.
- 3 **Mashup Execution.** In the mashup execution state, the generated mashup application is installed to a target device. The execution starts out by validating the required components of the installed mashup application. In case of a required mashup component is not installed, the mashup application will inform the user about the required mashup components. Once all required components are installed, an end-user can run the mashup application as an ordinary mobile application.

In this approach, we propose two XML description languages, one for component development process and the other one for mashup composition process. For the component development process, we used a description language called XLIMA (eXtended Language for Interoperability of Mobile Applications). XLIMA inherits the concepts and design from a description language called LIMA, which is our previous language for interoperability of mobile application [25]. A mashup component developer can use XLIMA for describing abstract functionalities and technical configurations of mashup components. XLIMA description files are then used as input for the component builder tool to generate mashup proxies as native mobile applications. For mashup composition, a description language called MEDAL (Mashup Event-Driven Annotation Language) is used for representing hybrid mashup composition logic. Instead of including mashup component configurations together with mashup composition descriptions as ordinary mashup description languages do, MEDAL contains only the description of mashup component integration, but provides a mechanism to link to external mashup component configuration files (XLIMA files). Similarly, MEDAL description files are used as input for the mashup generator to generate mashup applications as native mobile applications.

## 5. Implementation

The first prototype of the hybrid mashup development system is implemented on the Android platform. We selected Android as the first experimental platform because it is widely used and has the highest potential when compared to other mobile platforms. As discussed previously, an important technique of our approach is the Mashup Orchestration Messaging (MOM).

Therefore, challenges and solutions of implementing MOM and its components for Android smartphones and tablets are discussed in this section.

### 5.1. Mashup Proxy

The mashup proxy (MP) plays an important role in our component integration. It acts as an intermediary between actual mashup components and the orchestration process. An MP is responsible for forwarding function calls from a mashup component to an actual mashup resource, as well as passing the event notifications from an actual mashup resource to the other mashup components. Each mashup resource has a dedicated mashup proxy to transform a heterogeneous programming interface into a unified integration protocol. The following subsections discuss challenges and solutions for implementing a mashup proxy on Android platform.

#### 5.1.1. Invoking Functions of Mashup Components

Mashup proxies can deal with two types of mashup components: mobile applications and REST Web services. In the case of mobile applications, our approach applies Android's Intent [28] as the invocation protocol. Intent is an inter-application communication technique that abstracts description of an operation to be performed. It can be used for launching applications, sending messages and parameters to applications, or communicating with background services. In order to invoke a mobile application for a result, a mashup proxy sends an Intent with identifier information (i.e. the name and parameters of the target mobile application) to the Android system. The Android system then invokes the target application using the specified information. When the target application finishes the requested task, the result is sent back to the Android system as an Intent. The result Intent is then passed to the mashup proxy to extract the result parameters. In the case of Web service invocation, the HTTP request messages and callback functions are used. Therefore, we design each mashup proxy to contain an "Asynchronous Invocation" module that takes care of invoking mobile applications and REST Web service APIs, and a "Callback Handler" module that takes care of extracting the result returned from the target mashup component. Figure 6 shows the functional invocation mechanism of the mashup proxy.

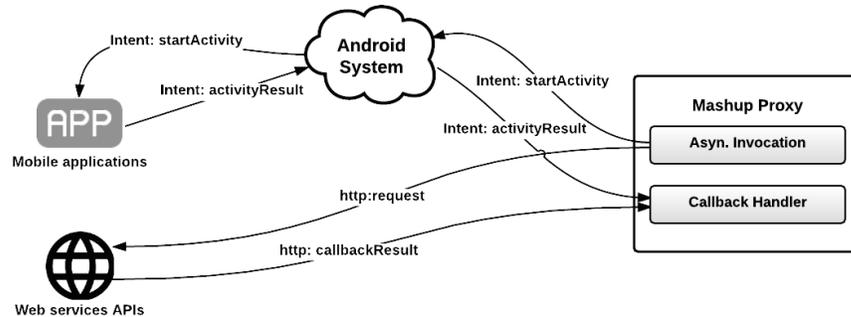


Fig. 6. Functional invocation mechanism of mashup proxy.

### 5.1.2. Dealing with General Event of Mobile Devices

A mashup proxy is designed to handle general events occurring within mobile devices, including system events and mobile application events. In general, Android operating system notifies the system events, such as battery status, call received status or changes of device configurations, to other applications by using Android Intent objects. In addition to providing a mechanism for launching other applications, Intent is also used to broadcast system wide messages to other applications. This requires the implementation of Broadcast Intents and Broadcast Receivers. An application can listen to a specific system event by registering a Broadcast Receivers to the Android system. When that system event is occurred, the registered application will receive the notification as Broadcast Intents. Mobile applications can also produce events by broadcasting Intents. Thus, the same technique can be applied to handle both events produced by Android system and mobile applications. Another major source of events is the “Notification Center”. Many mobile applications report their states to the notification center, such as when a new email is received or a wireless network is detected. Capturing messages from the notification center also allows us to enable an event-driven mechanism. Therefore, we design each mashup proxy to contain an “Event Manager” and an “Event Handler” module. The Event Manager takes care of intercepting system broadcast messages and notification center messages. The Event Handler controls the execution logic corresponding to the listening events. Figure 7 shows the event handling mechanism of the mashup proxy.

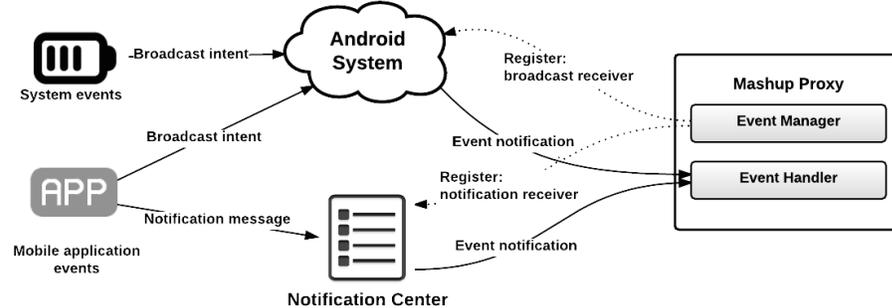


Fig. 7. Event handling mechanism of mashup proxy.

### 5.1.3. Managing Run-Time Parameters of Mashup Proxy

An additional challenge of a mashup proxy is managing run-time configurations of mashup components. In practice, a mashup proxy deals with a specific invocation protocol of actual mashup components by receiving and passing parameters. Some invocation parameters might be constant values while others may be different for each execution. For example, calling a weather forecast Web service requires a constant URI and a data extraction rule, which remains the same in all invocations. On the other hand, a mashup user may change certain parameters, such as geographic coordinates and units of temperature, in every execution. For this reason, we have designed the mashup proxy to contain a module called “Parameter-

ized UI” to provide a user interface for managing invocation parameters of actual mashup components.

#### 5.1.4. Building a Mashup Proxy

To create a mashup proxy, a mashup component developer composes an XML description file (XLIMA) that describes abstract operations, events and technical configurations of the actual mashup components. The composed description file is then used as input for the Proxy Builder tool to generate a mashup proxy as a mobile application. Finally, the generated Android application is installed to the target device. The code generation process of the proxy builder is illustrated in Figure 8.

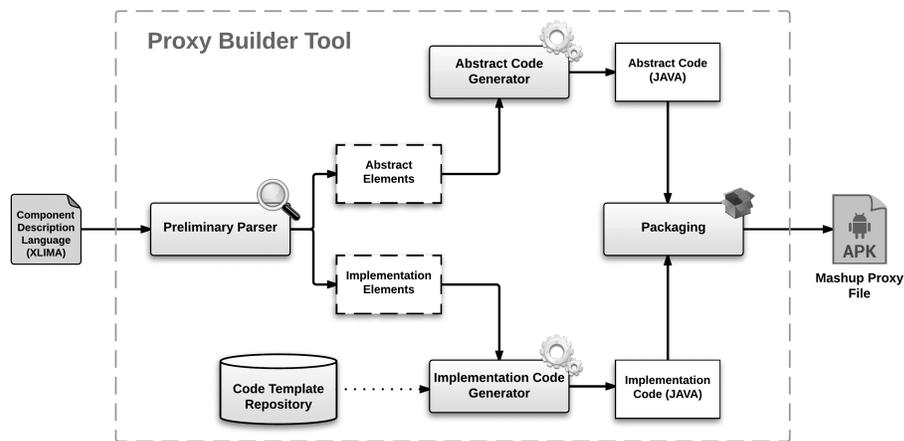


Fig. 8. Code generation process of proxy builder.

The first step of the code generation process is Preliminary Parser reads an XLIMA description file. It then extracts XML descriptions and separates them into two parts: Abstract Elements and Implementation Elements. The abstraction elements contain abstract description of shared functionalities and events, while the implementation elements contain technical details of the invocation. Next, Abstraction Code Generator takes the abstraction elements as an input to generate Abstract Code, i.e. a simplified JAVA programming structure corresponding to abstract functionalities and events described in the description file. Meanwhile, the Implementation Code Generator extracts the implementation elements and uses them to generate Implementation Code, which is JAVA programming codes that contain concrete details of functional invocation and event listening. At this stage, code templates from the Code Template Repository are used. The code templates are matched with specified configurations of target mobile applications or REST Web services in order to generate the corresponding JAVA programming codes. Using code templating technique also provides flexibility for the code generator to create programming codes that support heterogeneous configurations of various mashup components. After the code generation processes have finished, Packaging module combines the abstract code with the implementation code and builds a complete code structure of Android mobile application. It then compiles the combined code as an Android

application package (.apk). Finally, the mashup proxy file is distributed and installed to target devices as an ordinary Android application.

## 5.2. XLIMA

XLIMA is an XML description language that is used for creating mashup proxy. Component developers use XLIMA to describe a mashup component by declaring abstract functions and events, and specifying invocation configurations. The structure of XLIMA files and a brief description of each section are shown in Figure 9.

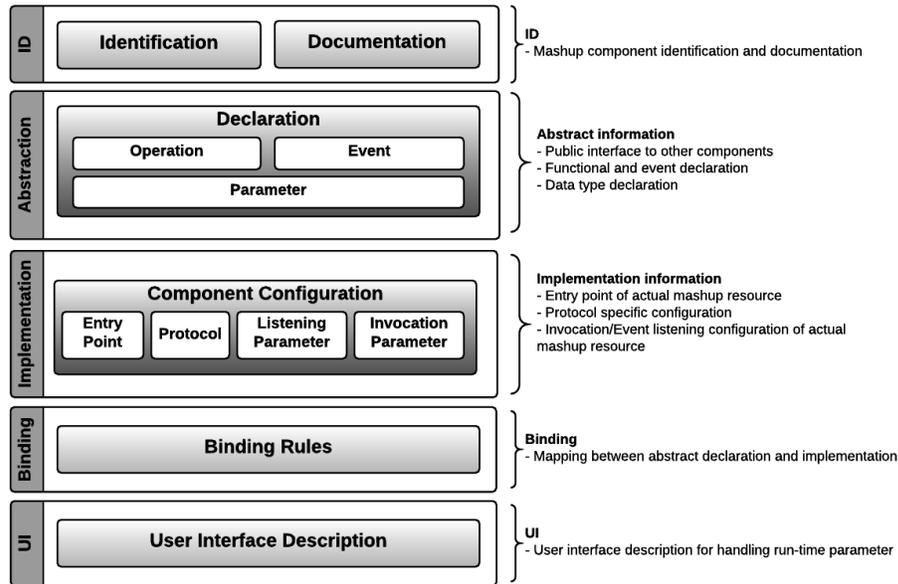


Fig. 9. Structure of XLIMA description language.

**Abstraction.** The abstraction section of XLIMA is designed for describing functions, events, and data types of mashup components. The description is written as technology-independent configurations. In other words, functions and events are described with an identifier, input/output parameters names and data types similar to a function declaration in a programming language. Figure 10-A shows a sample of an event declaration which contains the id of the mashup component, an event name and event parameters. Figure 10-B shows a declaration of a translation function. XLIMA allows defining a return parameter as an event in order to enable asynchronous integration between mashup proxies. In this example, other components can get the result from the “translate” function by listening for the “on\_translated” event, which will occur when the translation function finishes the execution.

**Implementation.** The implementation section of XLIMA is designed for describing the invocation configuration of functions and events that are declared in the abstract information section. The description in this section is written in terms of technology-dependent configurations. In other words, functions and events are described with their specific invocation protocols. Figure 11-A shows a sample of a mobile application component that listens to an

```

A) Event: on_email_received(string from, string subject, string content)
1 <abstraction>
2 ...
3 <component id="jp.ttlab.proxy.email">
4 <event id="on_email_received">
5 <parameters>
6 <param id="from" datatype="string" />
7 <param id="subject" datatype="string" />
8 <param id="content" datatype="string" />
9 </parameters>
10 </event>
11 </component>
12 ...
13 </abstraction>

B) Function: string translate(string text)
1 <abstraction>
2 ...
3 <component id="jp.ttlab.proxy.translator">
4 <operation id="translate">
5 <parameters>
6 <param id="text" datatype="string" />
7 </parameters>
8 </operation>
9 <event id="on_translated">
10 <parameters>
11 <param id="output" datatype="string" />
12 </parameters>
13 </event>
14 </component>
15 ...
16 </abstraction>
    
```

Fig. 10. Sample of XLIMA’s abstraction section.

event (email is received) using the Intent broadcasting protocol. Figure 11-B shows REST Web service configurations, i.e. URI, query string, data format and result extraction path, of a translation Web service component.

```

A) Mobile application component: Email client application
1 <implementation>
2 ...
3 <mobileapplication id="com.fsck.k9">
4 <listener id="K9EmailListener">
5 <protocol name="intent" type="broadcast">
6 <intent-filter>
7 <action>com.fsck.k9.intent.action.EMAIL_RECEIVED</action>
8 <data-scheme>email</data-scheme>
9 <extras>
10 <extra id="subject"
11 name="com.fsck.k9.intent.extra.SUBJECT"
12 datatype="string" />
13 <extra id="from"
14 name="com.fsck.k9.intent.extra.FROM"
15 datatype="string" />
16 <extra id="preview"
17 name="com.fsck.k9.intent.extra.PREVIEW"
18 datatype="string" />
19 </extras>
20 </intent-filter>
21 </protocol>
22 </listener>
23 </mobileapplication>
24 ...
25 </implementation>

B) Web service component: Translation Web service configurations
1 <implementation>
2 ...
3 <webservice id="wsTranslate">
4 <endpoint
5 uri="http://mymemory.translated.net/api/get">
6 <protocol name="http" type="rest">
7 <query>
8 <key name="q" urlencoded="UTF-8"/>
9 <key name="langpair" urlencoded="UTF-8"/>
10 </query>
11 <results>
12 <result id="translateText">
13 <value
14 rule="json"
15 path="/responseData[0]/translatedText" />
16 </result>
17 </results>
18 </protocol>
19 </endpoint>
20 </webservice>
21 ...
22 </implementation>
    
```

Fig. 11. Sample of XLIMA’s implementation section.

**Binding.** The binding section of XLIMA is designed for specifying mappings between the abstract declaration and invocation information. This section is separated from the abstract and implementation sections in order to provide maximum loose coupling. Mashup composers are flexible to create a group of related functions and events that come from different actual mashup components. The description in this section is written as mapping rules that bind each pair of abstract and implementation information together. Figure 12-A shows a sample of an event binding. An abstract event declaration “on\_email\_received” is paired to an actual event of a mobile application called “K9EmailListener”. Figure 12-B shows two samples of binding. The “invoke” binding type is used for binding a component’s function with a Web service API, while the “event” binding type is used for binding an asynchronous result from the Web service with an event of the component. For parameter mapping, we applied the mapping techniques from MCDL [22], which includes three mapping rules: direct, template and parse. In addition, the source parameter of a mapping rule can refer to a user interface element to obtain a run-time parameter from a mashup component.

**User Interface Description.** This section of XLIMA is designed for describing user interface elements that is used for managing run-time parameters of a mashup component. Mashup

```

A) Event binding
1 <bindings>
2 ...
3 <binding type="event">
4   {
5     trigger="com.fack.k9.K9EmailListener"
6     action="jp.ttlab.proxy.email.on_email_received"
7     <mapping rule="direct" source="subject" target="subject" />
8     <mapping rule="direct" source="from" target="from" />
9     <mapping rule="direct" source="preview" target="content" />
10  }
11 </binding>
12 ...
13 </bindings>

B) Function & Event binding
1 <bindings>
2 ...
3 <binding type="invoke">
4   {
5     trigger="jp.ttlab.proxy.email.translate"
6     action="wsTranslate"
7     <mapping rule="direct" source="text" target="q" />
8     <mapping rule="direct" source="userinterface.uiLang" target="langpair" />
9   }
10 </binding>
11 <binding type="event">
12   {
13     trigger="wsTranslate"
14     action="jp.ttlab.proxy.email.on_translated"
15     <mapping rule="direct" source="translatedText" target="on_translated.output" />
16   }
17 </binding>
18 </bindings>
    
```

Fig. 12. Sample of XLIMA’s binding section.

composers can specify type of required user interface and bind the value with a parameter of an existing mashup component (as illustrated in the sample of Binding section). The description in this section will be generated as a configuration screen of the mashup proxy. Figure 13 shows an example of the user interface section and the generated user interface of a mashup proxy.

```

A) User interface description for run-time parameters
1 <userinterface>
2 ...
3 <element id="uiFrom" datatype="string" uitype="EditText">
4   <display-name>From Filter</display-name>
5   <default>korawit@gmail.com</default>
6 </element>
7 <element id="uiSubject" datatype="string" uitype="EditText">
8   <display-name>Subject Filter</display-name>
9   <default-value>""</default-value>
10 </element>
11 ...
12 </userinterface>
    
```

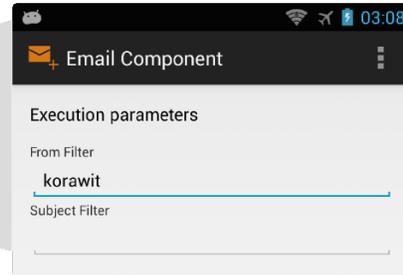


Fig. 13. Sample of XLIMA’s user interface section.

### 5.3. Mashup Orchestration Messaging System

The mashup components integration relies on the orchestration among mashup proxies (MPs) and a mashup orchestration process (MOP) through a delivery channel. Since we create MP and MOP as native mobile applications, an efficient delivery channel that facilitates inter-application communication must be carefully considered. In general, most mobile platforms provide a standard mechanism to allow communication and data exchanging among applications. For Android, several inter-process communication (IPC) techniques, e.g. AIDL, Bound Service or Intent, can be used to facilitate orchestration between MPs and MOP. Although, using low-level IPCs, such as AIDL and Bound Service, is more capable, it requires complex configurations and has less maintainability. For Intent, our previous study on the interoperability of mobile applications shows that Android’s Intent can be efficiently used as the component integration protocol [25]. Thus, we select Intent as integration protocol for the delivery channel.

To realize the way that the mashup orchestration messaging works, the following section explains mechanism of the “Email Translation Scenario”. This scenario is a language trans-

lation mashup that helps translating content of the incoming emails into a specific language. The mashup application listens for new incoming emails. When an email has arrived, it translates the content from Japanese into English, and saves the translated text to a note application. The orchestration mechanism of this mashup scenario is shown in Figure 14, and the required components of this mashup scenario are as follows.

- *Email Client Application.* This is a mobile application component: an open-source e-mail client on Android called K-9 Email [29]. When a new email is received, this application produces an event notification in the form of Intent Broadcasting.
- *Translation Web Service API.* This is a Web service component: an online language translation service called Mymemory Translator [30]. This Web service API uses the REST architecture and JSON data format.
- *Note Application.* This is a mobile application component: an Android mobile application called Evernote [31]. This application provides Intent integration information that allows creating a new note and additional operations.

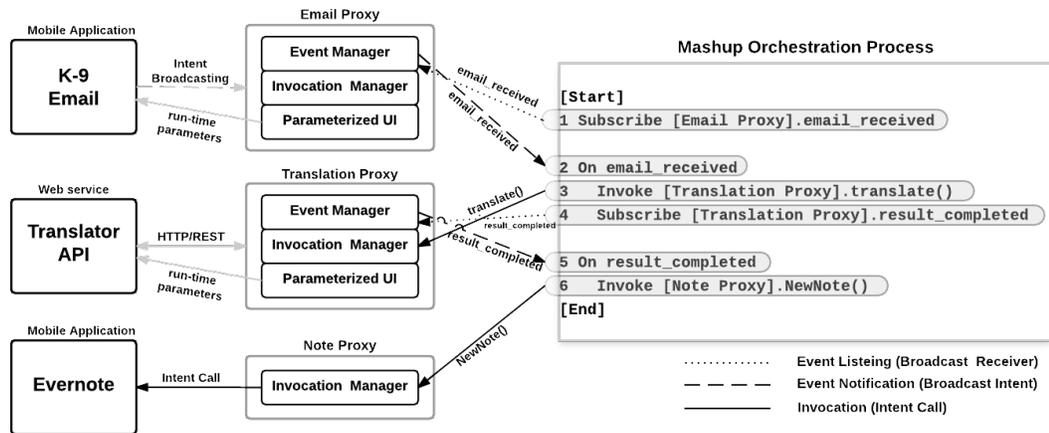


Fig. 14. Component orchestration model of Email Translator scenario.

- 1 When the mashup application is started, the MOP subscribes to event “email\_received” of Email Proxy and waits for the event. Email Proxy then uses Intent configurations, specified in the XLIMA file, to register itself to receive event notifications from K-9 Mail application. Once the K-9 Mail application receives a new email, it notifies the listening processes of the “email\_received” event by using Intent Broadcasting. It also provides two pieces of information, subject and content, of the new email as event parameters. The Email Proxy then creates a mashup event as a Broadcast Intent and posts to Android system.
- 2 The MOP then receives the event notification and executes the specified mashup logics (3 and 4).
- 3 The MOP invokes “translate” function of the Translation Proxy by sending an invocation message and parameters as Intent. When the Translation Proxy receives the

- invocation message and parameters, it creates a HTTP request message by using configurations specified in the XLIMA file, and sends the request message to the translation Web service.
- 4 The MOP subscribes to the event “result\_completed” of the Translation Proxy to receive the translation result. Once the Translation Proxy receives the HTTP response message, it extracts the translated text and creates an event by using the translated text as a parameter. Then, the created event is posted to Android system as Intent Broadcasting.
  - 5 The MOP receives the notification of “result\_completed” event. It extracts the translated text parameter to use as input of additional components.
  - 6 The MOP then invokes the Evernote application through the Note Proxy by using the translated text as a parameter for Evernote’s Intent.

Finally, the translated text is saved to the note application, and the mashup process waits for the next event notification. The screenshots of this scenario are presented in Figure 15.<sup>a</sup>



Fig. 15. Screenshots of Email Translator mashup application.

The development process of this mashup application starts from creating XLIMA description files of three mashup components. The created description files are used as input for the mashup proxy builder to generate three mobile applications that work as mashup proxies. Then, the generated mashup proxies are installed to the target device. The next step is creating a mashup application by composing a MEDAL description file. The created MEDAL

<sup>a</sup>K-9 Email and Evernote must be installed and logged in with a valid user account before the mashup application is started. The video demonstrating Email Translator mashup can be viewed at <https://www.youtube.com/watch?v=Iut0UcmgTCY>

file is used as an input for the mashup application generator to generate a mobile application that works as the mashup application. The generated mashup application is then installed to the target device. Finally, the user can run the mashup application as an ordinary mobile application.

### 5.4. MEDAL

MEDAL is an XML description language for describing the composition logic of mashup applications. It is designed for representing hybrid mashup composition, which is the integration of mashup components in data-flow and event-driven manners. The structure of MEDAL files and a brief description of each section are shown in Figure 16.

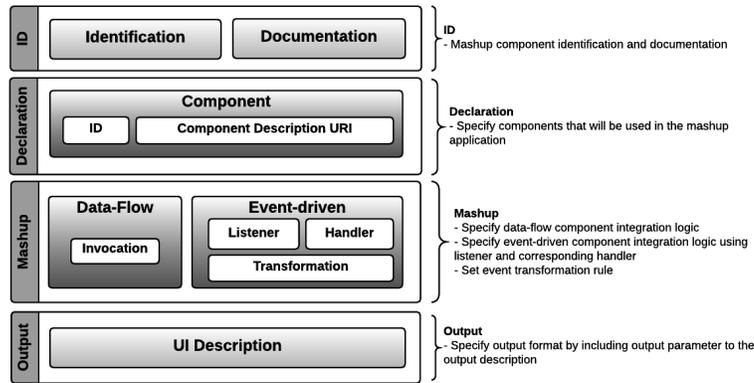


Fig. 16. Structure of MEDAL description language.

**Declaration.** This section is designed for the declaration of mashup components that will be used in the mashup application. A mashup component can be declared by specifying the URI of an XLIMA description file. The declaration allows other parts of the description refer to events and functions of the declared mashup component. The XML script in Figure 17 shows a sample declaration section that declares four mashup components.

```

1 <declaration>
2   <components>
3     <component
4       id="jp.ttlab.proxy.email"
5       alias="K9Email"
6       url="http://localhost/mobilemashup/cpr/k9email_proxy.xlima">
7     </component>
8     <component
9       id="jp.ttlab.proxy.translate"
10      alias="Translator"
11      url="http://localhost/mobilemashup/cpr/translate_proxy.xlima">
12    </component>
13    <component
14      id="jp.ttlab.proxy.note"
15      alias="Evernote"
16      url="http://localhost/mobilemashup/cpr/note_proxy.xlima">
17    </component>
18    <component
19      id="jp.ttlab.proxy.twitter"
20      alias="Twitter"
21      url="http://localhost/mobilemashup/cpr/twitter_proxy.xlima">
22    </component>
23  </components>
24 </declaration>

```

Refer to an XLIMA file

Fig. 17. Sample of MEDAL's description for mashup component declaration.

**Mashup.** This section of MEDAL is designed for describing hybrid mashup component integration. Mashup composers can define a sequence of component execution as a conventional data-flow style or define an event-driven execution that intercepts a target event and executes corresponding mashup components. Figure 18 shows a sample of mashup description for the hybrid integration of four mashup components.



Fig. 18. Sample description of a hybrid mashup composition with MEDAL.

The essential properties for this section are as follow.

- *Listener*: Define an event that will be listened for.
- *Handler*: Define actions that will be performed when the listening event occur.
- *Data-flow*: Define a data-flow component execution.
- *Invocation*: Specify a function that will be called.

An event listener can contain one or more event handlers. Each event handler can contain one or a data-flow of actions that will be performed after the event has occurred. Component C1 and C2 are executed in an event-driven way, which is listening for a new email and translating the content to a specific language. C3 and C4 are composed as a data-flow style, which saves the translated content to a note application and publishes it to a social network.

**Event transformation.** We found that sometimes events that are directly produced from mashup components are less usable. For example, in Android devices, every changes of the remaining battery level are alerted to mobile users as system events. However, a mashup application may only interested in low battery situation, such as the remaining battery is less than 5 percent. This section of MEDAL is designed for describing event transformation rules for filtering uninterested events. Figure 19 shows a description of an event transformation rule that allows a mashup application to define the low battery event.

**Output.** Some mashup applications require user interface elements, such as text, tables or maps, to display the mashup result. This section of MEDAL is designed for formatting and displaying the mashup output as a Web pages or pinpoints on a map. Mashup composers

```

1 <listener id="batteryLife" alias="OnBatteryStatusChanged"
2   publisher="BatteryNotification" event="status_changed">
3   <transformation>
4     <rule repeat="once" parameter="percentage" operator="less" value="5" action="raise" >
5   </transformation>
6   <handler>
7     <invocation component="Twitter" operation="tweet" >
8       <mapping mode="single">
9         <template
10          src="percentage"
11          pattern="My battery is % left, I will contact you later"
12          dest="tweet.message" />
13       </mapping>
14     </invocation>
15   </handler>
16 </listener>

```

Raise an event when batter is less than 5%

Fig. 19. Sample description of event transformation in MEDAL.

can create a Web page style output by composing HTML tags embedded with output data from the mashup components. A map style output requires set of locations and data that will be displayed on a map as pins. The output acts as a built-in mashup proxy that receives invocation and parameters from other mashup components. The received parameters are formatted and displayed on the output screen of mashup application. The XML script in Figure 20 shows a sample of displaying a mashup result using in a map. Location-based results from a Web service can be displayed as pins on a map by calling “showPOI” function of the output proxy. The configuration of the map, such as mode, zoom level or pin icon can be specified in the “mapview” tag.

```

1 <mashup>
2   ...
3   <listener id="queryResultListener" alias="OnResultCompleted"
4     publisher="GooglePlace" event="callbacked">
5   <handler>
6     <invocation component="output" operation="showPOI" >
7       <mapping mode="collection">
8         <direct src="lat" dest="latitude" />
9         <direct src="lng" dest="longitude" />
10        <direct src="name" dest="title" />
11      </mapping>
12    </invocation>
13  </handler>
14 </listener>
15 </mashup>
16 <output>
17   <mapview mode="map" zoom="16" userlocation="on">
18     <pinpoints mode="multiple" icon="red_simple">
19   </pinpoints>
20 </mapview>
21 </output>

```

Display out put as a map



Fig. 20. Sample description of mashup output formatting in MEDAL.

## 6. Evaluation

The evaluation of our approach is divided into 3 subsections. We report the empirical evaluation results regarding the usability of our mashup development system in Section 6.1. We illustrate the bounds on expressiveness and explain the extensibility of our composition model in Section 6.2. In section, 6.3, we discuss problems and limitations found in the current composition method and suggest possible solutions.

### 6.1. Usability Evaluation

The main purpose of our mashup composition method is reducing effort and required skills in developing mobile mashup applications. We aim to minimize the required skills of a mashup composer to the level of advanced user or novice programmer, i.e. a user who has experienced in composing XML documents and understand basic programming concept. In order to evaluate applicability of our method, a usability evaluation by human composers is conducted.

The evaluation focuses on the mashup composition process. We used a pre-questionnaires to select 10 mashup composers who have background knowledge in composing XML documents and understand the concept of operation, event, and parameter. The selected composers were asked to use MEDAL to complete two mashup compositions, a tutorial and a freestyle composition. For the tutorial, composers followed a step-by-step guide to build a mashup example called “email translator” that is described in Section 3.1. The tutorial also explains concepts of hybrid mashup composition and describes the specification of the MEDAL language. Output of the tutorial is a complete MEDAL description file, which will be verified by the mashup application generator. We then explained the composition model and demonstrated the output mashup application before the composers continued to the freestyle composition.

Table 1. Mashup Components for Usability Evaluation

Name	Type	Description
Input Components		
Barcode Scanner	MA	Scan barcodes and return as a text by using device’s camera.
Location Provider	MA	Get current user location
Photo	MA	Take photos using device’s camera
System	MA	Device’s status monitoring
Email	MA	Monitor incoming email and get the content
Location-based processing components		
GooglePlace	WS	Search for places around a location
GourNavi	WS	Search for restaurant around a location
OpenWeatherMap	WS	Search for weather information by location
GeoName	WS	Search for place name of a location
Yelp	WS	Search for local business around a location
Flickr Location	WS	Retrieve photos from the Flickr photo sharing service
Text-based processing components		
Flickr Search	WS	Retrieve photos from the Flickr photo sharing service
Online Shopping	WS	Search for product information from online stores by using barcode or keyword
YouTube	WS	Retrieve video link from the YouTube
Translate	WS	Translate text to a specific language
OCR	WS	OCR Scan function that converts text in an image to a text
Exchange Rate	WS	Do currency conversion
Train Schedule	WS	Search Japanese train schedule information
Output components		
Facebook	MA	Update status with current logged in Facebook account
Twitter	MA	Tweet a message with current logged in Twitter account
Evernote	MA	Create note messages on a note application
SMS	MA	Send SMS
Email	MA	Send Email
Text2Speech	MA	Read input text out as speech
Dropbox	MA	Save a file to Dropbox
Launcher	MA	Launch an application

In the freestyle composition, composers were asked to create a mashup application from existing mashup components using concepts they have learned in the prior task. The mashup components are created using XLIMA and the mashup proxy builder. The list of mashup components and their descriptions are shown in Table 1.

26 mashup components are generated. The mobile application components were selected from commonly used applications that contain reusable functions and available in the top charts of the Google Play Store. Web service components were selected from the popular APIs listed by ProgrammableWeb [32]. The components are categorized into input, processing, and output components for better understanding of the composers. Composers were allowed to study a component specification document before the composition began. The document contains details of available components, including list of functions, events and their parameters.

We then observed three elements in the evaluation: Composition Pattern, Planning Time and Composition Time. We measured the complexity of the composition by considering the component integration model and the number of components used in each composition. For planning time, we measured the time that composers used to finish the planning document, which contains the description of the mashup application, a list of selected mashup components and a component integration model. Finally, the composition time was measured from the time spent in using a text editor to create the MEDAL description file. Table 2 shows result of the evaluation.

Table 2. Usability evaluation result

Composer ID	Complexity Comp. Count	Planning Time (min)		Composition Time (min)		Total Time(min)	
		Total	Avg.	Total	Avg.	Total	Avg.
C001	2	0:12	0:06	0:10	0:05	0:22	0:11
C002	4	0:14	0:03	0:13	0:03	0:27	0:06
C003	5	0:13	0:02	0:21	0:04	0:34	0:06
C004	4	0:11	0:02	0:12	0:03	0:23	0:05
C005	4	0:13	0:03	0:15	0:03	0:28	0:07
C006	3	0:23	0:07	0:13	0:04	0:36	0:12
C007	4	0:22	0:05	0:11	0:02	0:33	0:08
C008	3	0:05	0:01	0:05	0:01	0:10	0:03
C009	3	0:22	0:07	0:11	0:03	0:33	0:11
C010	4	0:15	0:03	0:17	0:04	0:32	0:08
Avg. Time/Component			0:04		0:03		0:07
Avg. Total Time			0:15		0:12		0:27

It appears that all users succeeded in developing a mashup application using our method. All 10 composers have finished their mashup composition with a little support, e.g. concerning the detailed specification of MEDAL and configuration of particular mashup components. The result shows that the total composition time is related to complexity of composed mashup applications, i.e. the number of used components. The average time to compose a simple mashup application, using 3 mashup components as input, processing, and output, is less than 30 minutes. This is significantly lower when compared to manual development. The planning and composition time is also related to the complexity of the mashup application. However, for some particular users, i.e. C006 and C008, it can be seen that the total planning time is quite different even if the complexity is identical. In this case, we found that U008

stated in the pre-questionnaire that he/she has experience in mobile mashup development. It could be inferred that that planning time may have related to user's experiences in mashup composition, especially data-flow or event-driven styles. Therefore, we can assume that the planning and composing time might be reduced when the composers are more familiar with the hybrid composition model and the MEDAL specification.

After the freestyle task, the composers were asked to fill out post-questionnaires to evaluate satisfaction of the mashup approach and comprehension of MEDAL description language. The post-questionnaires consist of 10 of 5-points Likert scale questions and additional questions about personal opinions. Figure 21 shows the 5-points Likert scale result of the post-questionnaires.

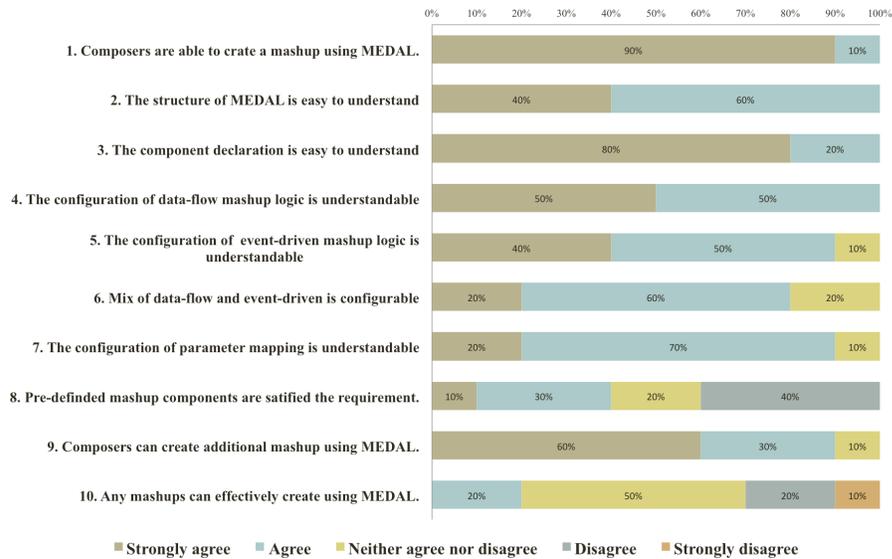


Fig. 21. Post-questionnaires result.

- Most composers gave high ratings concerning the ability of MEDAL to assist their compositions and the comprehensibility of configurations for defining data-flow and event-driven mashup logic. They also indicated that they are able to create additional mashup applications using our approach. The result also shows that composers understand the expressivity limitation of our approach.
- Composers have given additional comments about our method. They requested more mashup components, configuration sections supporting conditional statements, and assistant tools such as visual mashup composition or a MEDAL editor.
- Composers are interested in developing mashup components. In our approach, developing mashup components requires XML editing skills and additional knowledge on Web service specifications and mobile application configurations. The evaluation results of a previous study shows that even novice composers can deal with component configuration using a description language and a generator tool [19]. Thus, we believe that mashup

composers should be capable of developing mashup components using our approach. However, a component development evaluation should be conducted.

### 6.2. Expressivity Evaluation

In order to evaluate the expressivity, we simulated the possible mashup compositions that can be built by using our approach. To the best of our knowledge, the number of possible mashup compositions can express the capability to deal with a variety of user requirements and bounds on expressiveness of our mashup composition method.

The simulation uses two common composition patterns with the set of mashup components used in Section 6.1. The result from 6.1 indicated that the commonly used composition patterns are input-process-output (IPO) and input-process-process-output (IPPO), which is consistent with the common patterns found in the evaluation result of previous study [19]. By using a simple composition model such as IPO or IPPO, our method can create mashup applications that cover broad areas of requirements. Let us consider an example of creating location-based mashup applications in Figure 22.

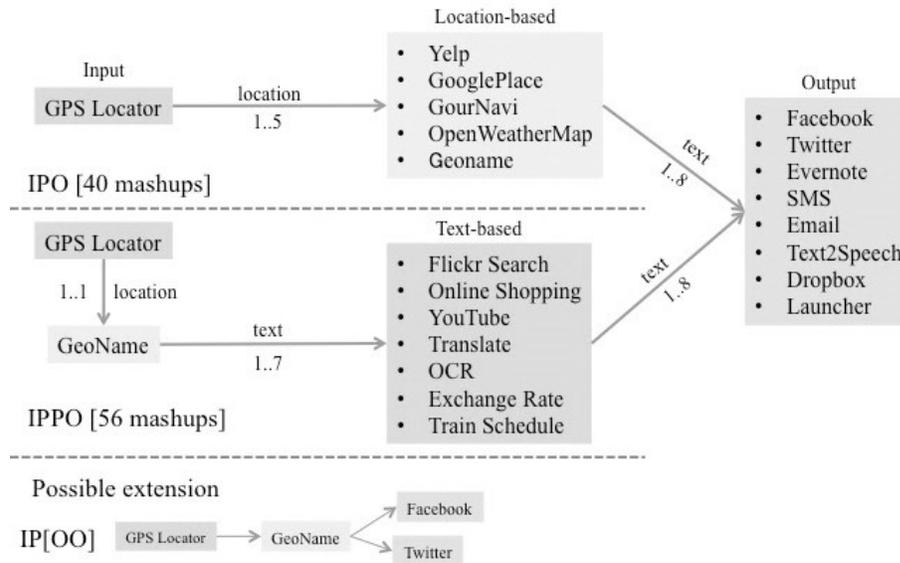


Fig. 22. Simulations on generating location-aware mashup applications.

By using GPS locations combined with one of 5 possible location-based Web services and 8 alternative mobile applications, we can generate 40 mashup applications using the IPO pattern. In addition, if we added one more component that converts location into text, i.e. GeoName, the location name can be integrated to additional text-based components to create more 56 mashup applications with the IPPO pattern. Moreover, our integration model also supports multiple output components that help increase the number of possible compositions.

Even though the number of possible mashup compositions is considerably large, some compositions might not practical. Therefore, instead of only doing simulations, real mashup

application development is conducted. Table 3.<sup>a</sup> shows a sample of practical mashup applications generated by our approach.

Table 3. Sample of generated mashup applications

Application	Components	Pattern	Description
<b>Last Train</b> <i>Display last train schedule</i>	[MA]GPSLocator [WS]GooglePlace [WS]Train Schedule	Data-Flow	Use the current location to find nearest train station and use the station name to find the schedule of the last train.
<b>Photo Diary</b> <i>Create a diary in a note application including photo and related information</i>	[MA]Camera [M]GPSLocator [WS]GeoName [WS]Wikipedia [MA]Evernote	Data-Flow	Use the camera to take photos and find information of the photos. Finally, the photos and information are saved to Evernote.
<b>Wish List</b> <i>Tweet a message about interested shopping items</i>	[MA]Xzing Barcode [WS]GoogleShopping [WS]Translate [WS]ExchangeRate [MA]Twitter	Data-Flow	Use Xzing application scans barcodes of interested items. Then, finds title and price from the online stores. Finally, translates and tweet as a wish list.
<b>Check In</b> <i>Send a message when you arrived at a place.</i>	[E]Wi-Fi Connected [MA]Facebook	Event-Driven	When connected to a Wi-Fi network, send a message to a friend via Facebook application
<b>Battery Tweet</b> <i>Tweet when battery is low</i>	[E]Battery Status [MA] Twitter	Event-Driven	When battery status is lower than 5%, tweet a message to friends
<b>Blind Buddy</b> <i>Assist the blind by speaking the current place's name out loud</i>	[E]Location Changed [WS]GeoName [WS]Translate [MA]Text2Speech	Hybrid	When the location has changed, speak the new place name out loud.

The composition of practical mashup applications is extensible. For example, the Web service component of “Last Train” application, which finds the schedule of last train, can be replaced with other Web services, such as a restaurant or hotel search, weather information, photo retrieval or some other location-based services. Similarly, the simple integration of an event and a mobile application component, i.e. Arrival Status and Battery Tweet, can be easily customized to address different problems. We can change the event and mobile application component to different pairs such as speak the message out loud when an SMS is received or call another Web service component when the location has changed.

### 6.3. Discussion

#### 6.3.1. Constraints of mobile application components

Since our approach relies on the capabilities of Android Intent, mobile applications that are capable to be used as mashup components must support the Intent integration as well as publishing information about their Intent configurations. In fact, not all applications support Intent integration. Moreover, the level of Intent supporting is different for each application. As a result, the component developer should be aware that some applications can work as a service function, while some applications can only work as a final component to display or receive the output. However, modern mobile operating systems, such as Android and iOS,

<sup>a</sup>MA:Mobile application, WS:Web service, E:Event trigger

have extended capability of inter-application communication to accommodate data sharing among applications. For instance, we can share a photo with social network applications by pressing the share button on the photo gallery, and selecting a target from a list of compatible applications. As a result, we believe that most recent mobile applications can be used as mashup components in our approach. This also increases the possibility of implementing our approach for other mobile platforms.

### 6.3.2. *Limitations of event-driven composition.*

In our approach, a mashup application can listen for multiple events. One event can have multiple handlers to execute multiple mashup logics. However, the current composition model still does not support simultaneous events with one or multiple handlers. The possible solution is using a technique called “Sticky Broadcast Intent”, which holds the Intent Broadcasting as a background service. This kind of Intent allows other processes or applications to access the notification at any time. However, we did not include this mechanism in the scope of our mashup composition because listening for an event requires a background process, which consumes more resources. More importantly, simultaneous events composition is rarely used in typical event-driven applications.

### 6.3.3. *Mashup Development Process.*

This research aims to improve the mashup composition by reducing the programming efforts and required technical skills. A common challenge is that managing configurations of mashup components requires technical knowledge. As a result, state-of-the-art mashup approaches delegate the component development process to users with higher programming skills; so called component developers [33]. Our approach applies this concept by separating the component description language from the mashup composition language. Component developers use XLIMA, a technology-dependent language, while MEDAL, a technology-independent language, is designed for mashup composers. In addition, we enhance reusability and maintainability of our mashup components by building them as mobile applications. Updating the configuration of mashup components does not affect the mashup application. Importantly, the run-time configuration feature of our mashup components is a key to improving component reusability.

### 6.3.4. *Component Reusability and Resource Consumption.*

The conventional mobile mashup approaches build mashup output as a single mobile application. Mashup components are placed in the same application context with the mashup orchestration process. As a result, a mashup application can become a resource-consuming process because of less component reusability. Let us consider multiple mashup applications that use the same mashup components. During the execution of these mashup applications, each identical mashup component allocates their own resources from the device even if they perform the same task. To address this problem, the unique idea of our mashup composition is building mashup components and mashup applications as separated mobile applications. In this way, we can take advantage of the automatic resource management of the mobile operating system. Separating the processes of mashup components and mashup applications

also enhances reusability and reduce resource consumption. The mashup components that run as ordinary mobile applications will be kept as inactive process when there is no mashup execution. In this way, multiple mashup processes can reuse the existing mashup components in different composition logics while the resource consumption level is equal to that of one mashup application.

## 7. Conclusions and Future Work

This research proposed a new methodology for developing mashup applications for mobile devices. The existing methods have succeeded in developing data-flow mashup applications. They, however, have limited capability to create event-driven mashup applications. We presented a full treatment hybrid mashup composition method that utilizes both data-flow and event-driven composition. This method allows the composition of mobile applications and REST Web services in both data-flow and event-driven ways. In order to realize the novel architecture of component configurations and component integration, practical scenarios for both data-flow and event-driven mashups were set and analysed. The concept of using an inter-application communication protocol to facilitate orchestration of mashup proxies and a mashup orchestration process is discovered and applied. The description languages and code generator tools are used to reduce mashup development cost, and to separate component development process from the mashup composition process. To demonstrate the applicability of our approach, we implemented our first prototype in the Android mobile environment and conducted usability evaluations.

The evaluation results have shown that the proposed approach improves expressivity and reduces the cost of developing mobile mashup applications. Finally, we discussed limitations found in our current composition methods and suggested possible solutions. In the future, we aim to implement our approach for other mobile platforms, and reduce the required programming skills by applying an end-user programming paradigm such as visual programming.

## References

1. C. Anderson (2007), *The Long Tail: Why the Future of Business Is Selling Less of More* by Chris Anderson. Journal of Product Innovation Management, Vol. 24, , pp. 130.
2. K.Xu, X.Zhang, M.Song, and J.Song (2009), *Mobile mashup: Architecture, challenges and suggestions*. Management and Service 2008 , pp 25.
3. E.Maximilien (2008). *Mobile mashups: Thoughts, directions, and challenges*. Semantic Computing, IEEE International (2008), pp. 614617.
4. P.Chaisatien, and T.Tokuda (2011), *A Description-based Approach to Mashup of Web Applications, Web Services and Mobile Phone Applications*. Information Modelling and Knowledge Bases XXII, Frontiers in Artificial Intelligence and Applications, Vol. 225, pp 174-193.
5. V.Agarwal, S.Goyal and S.Mittal (2012), *Towards Enabling Next Generation Mobile Mashups*. Mobile and Ubiquitous, pp. 1325.
6. S.Chowdhury, Roy, et al, (2011) *Composition patterns in data flow based mashups*. Proceedings of the 16th European Conference on Pattern Languages of Programs (EuroPLoP'11). 2011.
7. Michelson, M.Brenda (2006), *Event-driven architecture overview* Patricia Seybold Group Vol. 2.
8. Tasker. <https://play.google.com/store/apps/details?id=net.dinglich.android.taskerm>.
9. Yahoo Pipe. <https://pipes.yahoo.com/>.
10. R.Ennals and M.Garofalakis (2007), *MashMaker: mashups for the masses* Proceedings of the 2007

- ACM SIGMOD international conference on Management of data (SIGMOD '07), pp. 1116-1118.
11. S.Kaltofen, M.Milrad and A.Kurti (2010), *A cross-platform software system to create and deploy mobile mashups*. Springer Berlin Heidelberg, pp. 518-521.
  12. A.Brodt and D.Nicklas (2008), *The TELAR mobile mashup platform for Nokia internet tablets*. Proceedings of the 11th international conference on Extending database technology Advances in database technology - EDBT '08, pp. 700.
  13. F.Nyrhinen, A.Salminen, T.Mikkonen, and A.Taivalsaari (2010), *Lively mashups for mobile devices*. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, 35 LNICST, pp. 123-141.
  14. V.Realinho, T.Romo, and A.Dias (2012), *An event-driven workflow framework to develop context-aware mobile applications*. International Conference on Mobile, pp. 22.
  15. H.Gebhardt, M.Gaedke, F.Daniel, et al (2012), *From mashups to telco mashups: A survey*. IEEE Internet Computing 16, pp. 70-76.
  16. Sanders, R.Torbjorn, F.Mbaabu, and M.M.Shiaa (2012) *End-user configuration of telco services*. 16th International Conference on Intelligence in Next Generation Networks.
  17. Trigger. <https://play.google.com/store/apps/details?id=com.jwsoft.nfcactionlauncher>.
  18. C.Cappiello, M.Matera, M.Picozzi, A.Caio, and M.Guevara (2012), *MobiMash: end user development for mobile mashups*. In Proceedings of the 21st international conference companion on World Wide Web, ACM, pp. 473-474.
  19. P.Chaisatien and T.Tokuda (2013), *A description-based composition method for mobile and tethered Mashup applications*. Journal of Web Engineering Vol. 12.1-2, pp. 93-130.
  20. WAC Specification. <http://specs.wacapps.net>.
  21. W3C's Device API Working Group. <http://www.w3.org/2009/dap/>
  22. S.Aghae, and C.Pautasso (2011), *The mashup component description language*. Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, ACM, pp. 311-316.
  23. EMMML. <http://www.openmashup.org/>.
  24. M.Sabbouh, J.Higginson, S.Semy, and D.Gagne (2007), *Web mashup scripting language*. Proceedings of the 16th international conference on World Wide Web WWW 07, pp. 1305-1306.
  25. K.Prutsachainimmit and T.Tokuda (2014), *LIMA: A Modeling Language for Enhancing Mobile Application Interoperability*. Information Modelling and Knowledge Bases XXV 260, pp. 98.
  26. A.S.Voulodimos and C.Z.Patrikakis (2008), *Using personalized mashups for mobile location based services*. In Wireless Communications and Mobile Computing Conference, IWCMC'08. International, pp. 321-325.
  27. C.Cappiello, F.Daniel, M.Matera, and C.Pautasso (2010), *Information quality in mashups*. Internet Computing, IEEE, Vol. 14(4), pp. 14-22.
  28. Intent. <http://developer.android.com/reference/android/content/Intent.html>.
  29. K-9 Email. <https://play.google.com/store/apps/details?id=com.fsck.k9>
  30. MyMemory Translation API. <http://mymemory.translated.net/doc/spec.php>.
  31. Everernote (Android). <https://play.google.com/store/apps/details?id=com.evernote>
  32. ProgrammableWeb. <http://www.programmableweb.com>.
  33. F.Daniel and M.Matera (2009), *Turning web applications into mashup components: issues, models, and solutions*. Springer Berlin Heidelberg, pp. 45-46.