論文 / 著書情報
Article / Book Information

題目(和文)	記述方式に基づくモバイル端末連携のためのマッシュアップアプリケーション構成法
Title(English)	Description-Based Composition Methods of Mashup Applications for Cooperation of Mobile Devices
著者(和文)	ﾌﾟﾙｻｼｬｲﾆﾑﾐｯﾄｺﾗｳｨｯﾄ
Author(English)	Korawit Prutsachainimmit
出典(和文)	学位:博士(学術), 学位授与機関:東京工業大学, 報告番号:甲第10255号, 授与年月日:2016年3月26日, 学位の種別:課程博士, 審査員:徳田 雄洋,佐伯 元司,徳永 健伸,権藤 克彦,西崎 真也
Citation(English)	Degree:Doctor (Academic), Conferring organization: Tokyo Institute of Technology, Report number:甲第10255号, Conferred date:2016/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Description-Based Composition Methods of Mashup Applications for Cooperation of Mobile Devices

Thesis submitted
for the
Degree of Doctor of Philosophy

Korawit Prutsachainimmit

Department of Computer Science
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

Advisor: Prof. Takehiro Tokuda

2016

Abstract

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 for single devices. They have limited capability to deal with event-driven and multiple-device mashup composition. A full treatment of data-flow and event-driven mashup composition that utilizes cooperation of mobile devices is not yet achieved. This thesis presents a new methodology for developing data-flow and event-driven mashup applications for single and multiple devices. Our hybrid composition method allows integration of mobile applications and REST Web services in a data-flow and event-driven manner. We propose a device cooperation model that allows mashup applications to take advantages of data sharing among multiple mobile devices. Description-based techniques and application generator tools are applied to reduce development cost. A mashup development system is implemented in Android mobile platform as a first experimental platform. The evaluation results show that our method is expressive and efficient in composing typical mobile mashup applications for single and multiple devices.

Contents

Chapter 1

Introduction

1.1 Motivation

Mobile devices, such as smartphones and tablets, have recently gained popularity and become the most common computing and communication device. As a result, millions of mobile applications are published through major delivery channels, such as the Apple App Store and Google Play Store, covering a variety of user requirements. Even though a huge number and large diversity of mobile applications are available, they are still not covering the long tail of users' requirements [1]. This situation drives the need for users to develop their own mobile applications.

One major problem is that mobile applications are commonly designed for a single purpose. Completing a common problem sometimes requires more than one application. In other words, users have to combine functionalities of multiple applications to solve their specific problems. For example, let us consider a user who wants to translate content of an email into a specific language. Unfortunately, the available email client mobile applications are usually designed for sending and receiving email. The translation features are not included. He/she needs to copy content from an email client application, and use this as an input for a language translation Website or other language translation mobile application. An application that is capable of combining these existing functionalities, i.e., email client and language translation, is needed. However, developing such an application is not practical for non-programmers because extensive knowledge of mobile application development and high-level programming skills are required. Thus, end-users require a tool assisting self-creation of mobile applications. Mobile mashups, which employ the concept of lightweight composition of existing resources, are one of the efficient tools supporting mobile applications development for end-users.

Mobile mashups allow the creation of new mashup applications by using lightweight composition of existing resources. They take advantage of a combination of Web service APIs and device-specific components for enriching mobile services and enhancing user experiences [2]. In particular, this type of integration is able to create new results that cannot be achieved by using conventional mashup composition. For example, location data from a GPS sensor can be integrated with location-based Web services, i.e., Gourmet Navigator API [3], Google Places API [4] or Yelp [5], to create a mobile mashup application that displays points of interest near the current user's location. Furthermore, recent research has succeeded in enabling integration of

functionalities of mobile applications with Web APIs, i.e., Web services and Web applications. Therefore, mobile mashup becomes a capable composition tool for end-user mobile application development [6].

Recently, the usage trend of mobile applications is changing from individual use to collaborative use. Groupware and social network applications, which take advantage of shared information among users, are adapted to mobile platforms. Consequently, the capability of sharing information among multiple mobile devices has become an important feature of commercial mobile applications. For example, modern mobile applications allow users to share their current location among friends, and use shared coordinates to achieve a collaborative purpose, such as finding a central meeting place. In our view, mashup applications can benefit from cooperation of multiple devices by exchanging data and sharing mashup results. The shared data among the participating devices can be integrated with other mashup resources to produce new variety of mashup output. A simple example is the location-based mashup. A mashup application may request the current location from two or more mobile devices, and use this to find the center coordinates among the participating devices. The center coordinates can be given to a Web service API to find the best-ranked restaurant nearest to this centered location. Finally, the output of the mashup, i.e., restaurant name and location, can be shared among all participating devices. However, an efficient method dealing with data sharing among multiple mobile devices and the proper model for integrating the shared data with existing mashup resources is still not proposed. The benefits and challenges of enabling mobile mashups for multiple devices motivates us to explore a new mashup composition approach that is capable of dealing with data cooperation among multiple mobile devices.

Techniques for developing mashup applications for mobile devices mostly originate from Web mashups, which introduces the concept of lightweight integration of existing Web resources. Most mobile mashup approaches inherited application composition techniques from well-engineered Web mashup approaches. As a result, most of them use data-flow mashup composition as a major component integration pattern, similar to that of Web mashups. The mashup resources, such as Web services, Web applications or mobile applications, can be combined as a workflow; passing input/output parameters between connected components to produce mashup output [9]. The data-flow composition pattern has been proven efficient for Web mashups; however, there are obstacles that limit the capability of applying data-flow composition pattern for mobile mashups. For example, let us consider a simple and popular location-aware mashup that displays a point of interest (POI), e.g., restaurant, hotel or ATM, around the current user's location. This mashup passes the device's location as an input to a location-based Web service API, e.g., Google Places API, to find coordinates and additional information of the POIs. After the invocation of the Web service has been completed, coordinates of POIs will be displayed as pins on a map. However, in a real-life situation, mobile devices may move to new locations after the mashup execution is completed. As a result, the displayed pins of POIs are no longer valid. User can update

the mashup result by manually executing the mashup application again to get updated POI locations. One solution of this problem is making the mashup application listen for a specific event and automatically execute the mashup to update the mashup result. We can consider this kind of mashup application as an event-driven mashup application.

In addition to overcoming the limitations of data-flow mashup composition, event-driven mashup composition is valuable composition pattern for mobile mashups. Since, mobile devices monitor their states and report changes in an event-driven manner, the changes of states, such as location or battery status, are notified to other processes as system events. In our view, mobile mashups can take advantage of that event notifications by using them as a trigger to control the execution of additional mashup components. Besides, this event-driven mechanism also improves robustness of the mashup application. In case of data-flow mashup, the component execution will be performed in synchronous manner. When problems, such as a loss of the network connection or Web resources becoming unavailable, have occurred during the execution, the mashup application will be interrupted or terminated. In contrast to the data-flow mashup, the event-driven mashup applications can deal with the network connection and resource availability problems by performing the execution in an asynchronous manner. However, using event-driven mechanism for mashup composition on mobile device is still challenging. Since events in mobile devices are produced from different sources, listening for events in mobile devices requires different techniques and programming models. Another important challenge is realizing a component integration model that is suitable for event-driven mashup composition and execution. Hence, enabling event-driven mashup composition for mobile devices is a challenging topic in mobile mashup research.

Even though the event-driven mechanism and the data cooperation among multiple devices can improve expressivity and robustness of mashup composition, a well-engineered mashup composition approach is still not presented. Therefore, this thesis contribution is the presentation of a new methodology for developing data-driven and event-driven mashup applications for multiple mobile devices. Our approach aims to improve expressivity and efficiency of mobile mashups by enabling the composition of mobile applications, REST Web services, and shared information from multiple devices in both data-driven and event-driven manners.

We first explore efficient methods for data-flow mashup composition for multiple mobile devices. We propose a mashup construction approach that allows data-flow composition of mobile applications, REST Web services, and cooperative data from multiple devices. We present a centralized data cooperation model and mashup execution environment that automates data sharing for multiple devices mashup. We propose a description language called C-MAIDL, which is designed to describe mashup composition logic and device's collaboration behavior. The mashup generator tool is implemented as a quick mashup development tool to reduce development cost. We then implement sample scenarios on Android platform to demonstrate the applicability of our system. Finally, we discuss limitations and improvements of our first approach.

We found that the mobile application integration method in our data-flow composition approach still has limitations and needs improvements. As a result, our goal is to explore a new solution for improving reusability of mobile mashup components. We realized that the key to improve reusability is to enhance the interoperability of mobile applications. We then introduce LIMA, an XML-based modeling language for describing shared functionalities of mobile applications. LIMA enhances interoperability of mobile applications by providing an applicable way to describe abstract and concrete details of shared functionalities. LIMA also encourages reusability of mobile application components and increases flexibility of our mashup composition approach by separating component configuration from mashup composition logic.

To overcome limitations and expand the expressivity of data-flow mashup composition, we then explore an event-driven mashup composition method for multiple devices. We set up practical scenarios that represent usage of data-driven and event-driven mobile mashup applications for single devices and multiple devices. We then derive a suitable component and composition model that supports the composition of our target mashup applications. We apply a description-based mashup development method to facilitate end-user mashup composition. Description languages are defined to represent mashup component configurations and describe the execution sequence of mashup components. To reduce development cost, the description languages are used as an input to mashup generator tools to generate mashup applications. In this way, our approach leverages the development effort and reduces required programming skills. To evaluate the applicability of our approach, we then implement our first prototype on the Android mobile platform. The evaluation results show that our proposed architecture and model improve the efficiency of mobile mashup application composition, particularly the expressivity of user requirements, reusability of mashup components and applications, and robustness of mashup execution.

1.2 Organization of the thesis

The organization of this thesis is as follows. We review the background and related works in Chapter 2. The current constraints of mobile application platforms and solutions for the composition of mashup applications for multiple devices are discussed in Chapter 3. From Chapter 4 to 6, two different composition methods of mashup applications for cooperation of mobile devices are explained. Data-flow composition is presented in Chapter 4. An improved method for enhancing reusability of mobile application components is explained in Chapter 5. Then, event-driven composition for multiple mobile devices is presented in Chapter 6. Next, we give an evaluation of expressivity and reusability, a comparison of our method to other works, and discussions in Chapter 7. Finally, we discuss the possibility of future development and give our conclusion in Chapter 8.

Chapter 2

Background and Related Work

The objective of this research is to present an approach that deals with data-flow and event-driven mashup composition for cooperation of multiple mobile devices. In our view, the current mashup composition methods is targeted to develop data-integration mashup applications for single devices or a domain-specific mashup applications for multiple devices. To have a clear view on our research problems and challenges, we discuss background information about the characteristics and limitations of Web mashup composition, the definition and advantages of mobile mashup composition, and related research on mashup composition for mobile devices.

2.1 Background

A mashup is an application that integrates two or more existing components to create a new kind of service or result [7]. The integration of existing resources, so called mashup components, is the key process of mashup compositions. Mashup composition in Web mashup is the development of Web pages or Web applications that use and combine data, presentation or functionality from existing resources to create new services [8]. Web mashup applications have recently become popular. As a result, many Web mashup construction approaches have been proposed. Most of them focus on various aspects, ranging from data integration to user interface integration [9]. The next generation of Web mashup is mobile mashup. Mobile devices, such as smartphones and tablets, were introduced with the capability to access Web resources by using mobile Internet, and embedded with rich of mobile sensors, such as a GPS sensor, accelerometer and a camera. As a result, mashup applications running on a mobile device and integrating Web resources and making use of device-specific components, are emerged. By considering the architecture of mobile mashup applications, we can categorize them into two major types:

Web-based mobile mashups. This type of mashup is based on standard Web technologies. Mashup applications are developed as Web application and accessed via the mobile Web browser. Given the W3C standards for device APIs, the WAC proposals and browser extension capability, Web-based mobile mashups can make use of a limited set of device-specific capabilities. Similar to conventional Web mashups, Web-based mobile mashups may run by using client-side JavaScript and a back-end Web server that acts as proxy forwarding requests and responses. In general, this type of

mashup is an extended form of Web mashup that uses mobile devices as mashup viewer/consumer.

Native mobile mashups. This type of mashup is based on the native technology of mobile platforms. Each mashup applications are platform-specific applications developed by using a mobile platform SDK. Advantages of native mobile mashups over that of Web-based are they have better execution performance, better user experience and full access to the device-specific features [10]. However, developing this type of mashup requires extensive programming skill along with platform SDK knowledge. Therefore, it is not practical or less possible for end-users to develop this type of mashup. Even though using code-to-code development tools that reduce the required programming skills to Web authoring skills, this task is still not comprehended by end-users.

In this research, we focus on developing native mobile mashup applications that better utilize the device-specific capabilities. Thus, we have to overcome the challenges of developing native mobile mashup applications by reducing development cost and required programming skills.

2.2 Related Work

Mobile mashup approaches have been proposed to facilitate mashup composition for mobile devices. However, the objectives and goals of composing mashup application are different. In this section, we discuss the existing mobile mashup approaches and related technologies that help reducing mashup development cost.

In the beginning, mobile mashup frameworks inherit composition methods from Web mashup. Web mashup editors such as Yahoo Pipes [11], Intel MashMaker [12] and a mashup platform proposed by Kaltofen et al. [13] are capable of creating mobile mashup applications for single and multiple devices. The mobile Web browsers on mobile devices are used as a channel to deliver mashup applications to users. Since these mashup tools create web-based mashup applications, they cannot fully access device-specific data. Moreover, they cannot enable the cooperation of multiple mobile devices.

In academic research, domain-specific mashup approaches that allow composition of native mobile mashup applications for single and multiple devices have been proposed.

The TELAR mashup platform [14] presents a solution to combine location data from GPS sensors with existing Web service APIs to create event-driven mashup applications for single and multiple devices. However, this approach is limited capability to create location-based applications. Multiple client devices are possible as individual execution of mashup applications. The clients cannot perform cooperation.

Telco mashups [15,16,17,18] allow event-driven integration of device APIs and mobile network services. They are also limited to the integration to telephony functions and they do not support cooperation of multiple devices.

MobiMash [19] and its extension [20] allow visual composition of mobile mashup applications for collaboration of multiple mobile devices. SmartComposition [21] and MultiMasher [22], allow the distribution of user interface elements to achieve collaborations of multiple users. These approaches share the same goals as our approach. However, they focus on user interface integration, and not capable for data cooperation among devices.

Our previous research [23] has proposed a mashup construction system for the integration of Web applications, Web services and mobile applications. This approach realizes the reusability of Web information and device-specific capabilities. Data-flow is set as the main composition method, while event-driven composition is possible with manual programming effort. TeWS is also introduced to enable data cooperation among multiple devices. This previous work has limitations about the composition method and still requires manual programming effort. Thus, we use this work as the baseline and aim to make improvements on event-driven mashup composition and cooperation of mobile devices.

Since we use description-based techniques for reducing the mashup composition cost, description-based mashup approaches, i.e., mashup description languages [24,25,26,27,28] and end-user mobile mashup composition [29], must be reviewed. From this existing research, we learn valuable techniques that benefit our mashup description language design, for example, a parameter mapping technique and a flexible mashup development process. In addition, related work on event-driven architecture and mobile device data communication must be studied.

This research aims to explore an efficient and expressive mashup composition method that fulfills following goals.

- *Utilizing data cooperation of mobile devices*: Conventional mobile mashup approaches have limited capabilities to take advantages from shared data among multiple mobile devices. Our approach presents a suitable architecture accommodating data sharing among multiple devices, and realizes the capable component integration model that utilizes the shared data in mashup application composition.
- *Expanding expressivity with event-driven composition pattern*: Most existing approaches employs data-flow as a major component integration pattern. The event-driven approaches are proposed. However, they are designed to address problem of a specific domain, and importantly, less use of device-specific events in mashup composition. Our approach expands expressivity of mashup composition by presenting an efficient method for integrating device-specific feature with other mashup resources in both data-driven and event-driven pattern.

- *Reducing development cost*: End-user mashup composition approaches are presented to reduce required programming skills and development cost. The major challenge is balancing between capability of the composition and complexity of development process. This research presents a description-based mashup development process that maintains the equivalent level of expressivity and usability. We delegate the component development process, which requires higher programming skills, to expert users. Our proposed description languages and mashup generator tools allows novice users to compose mashup applications with less development cost.

Chapter 3

Mashup Composition for Multiple Mobile Devices

In general, mashup composition for mobile devices and conventional mashups of Web resources have different concepts. Mashup composition for mobile devices has its own methodologies and characteristics. Since mobile devices are built with device-specific features, such as a camera, a GPS sensor and an accelerometer, these capabilities bring out the unique characteristics of mobile mashups. A common goal is taking advantages of mobile network services, device-specific features, and context-aware information, by integrating such capabilities with other existing resources. As a result, mobile mashups are able to produce output that cannot be achieved with Web mashups. However, the trend in mobile application usage is evolving from individual to social or collaborative use. In addition, mobile platforms are empowered with technologies that facilitate connectivity among devices. Motivated by these facts, our research aims to discover an architecture that promotes mobile mashups for multiple devices. In this chapter, we discuss the mobile technologies that are necessary for enabling cooperation of mobile devices. We also address challenges of enabling mashup composition for multiple mobile devices.

Recently, composition of mashup applications for multiple mobile devices has become a challenging research topic in Web engineering. Many mashup approaches target to support multiple mobile devices. However, the objectives and characteristics of our research differ from existing approaches. The existing approaches mainly focus on user-interface integration of Web resources, or utilize multiple device mashups for usability purposes. In this research, we focus on the composition of mashup applications that can enable cooperation among multiple mobile devices. We want to achieve data sharing among devices, and process that shared data to produce a cooperative mashup result. In our view, multiple mobile devices can participate in the same mashup application if they share their context information with other devices. This shared information from the participating devices can be collected and integrated with additional mashup components. In this way, the output applications from our mashup approach are able to address the collaborative user's requirements.

To clarify the characteristics and benefits of mashup composition for the cooperation of mobile devices, let us consider one sample scenario, called "meeting point", illustrated in Figure 1.

Figure 1. Multiple Devices Mashup Scenario

3.1 Multiple Device Mashup Scenario and Analysis

3.1.1 Meeting Point Scenario

Suppose that there are four tourists traveling in Japan. They are separately visiting four different places in Tokyo. The travelers want to meet and have dinner together but they are now in different locations. They want to meet at the best-ranked restaurant located near the train station that is closest to the center point among them.

In case of a single traveler, he/she can use mashup technologies to create a mobile mashup application that combines their current location with a Web service to find a restaurant around their locations. The restaurant information, usually written in Japanese, can be translated to a specific language through mashup process with an additional Web service. However, finding a meeting point between four persons is challenging and requires the cooperation of four devices.

To address this problem by using mashup composition, the mashup application first needs to collect the current locations from each traveler's devices, and use them to compute the coordinates that constitute the center point between all four travelers. These coordinates are then used as input to find the nearest train station by querying Google Places Web service APIs. After obtaining the train station data, the best-ranked restaurant near a particular station can be discovered using the Gourmet Navigator APIs. Through this process, the restaurant that will serve as their meeting point is determined. Finally, the meeting point, the restaurant's location and additional information is shared among the four travelers. This kind of mashup composition is intended to be flexible. It can be extended and reused by connecting to various available Web service APIs. For

example, the restaurant information can be translated into a specific language using a translation Web service. Or, instead of finding a restaurant, we can easily apply this mashup composition to find the best hotels or car rental services by integrating with other Web services.

3.1.2 Analysis

Based on the meeting point scenario and other related scenarios, we define a number of characteristics of a mashup composition that our approach should fulfill. These are listed in Table 1.

Characteristics	Our approach
Integration target	Data integration
Cooperative level	Data and mashup results
Scope of component integration	Mobile applications and Web services
Mashup composition	Flow-based and Event-based

Table 1. Characteristic of our approach.

Integration Target. Recently, several mashup composition platforms for multiple mobile devices have been proposed. The existing approaches focus on different integration levels. For example, some approaches use multiple mobile devices for displaying a mashup result to users while some others aim for user interface integration [30]. To make the best use of a mobile device's capabilities, our approach targets the integration of data; we aim to promote data integration. The device-specific data from multiple mobile devices, such as locations from GPS sensors or photos from cameras, can be used as input for other mashup components, and the output from one component can be used as an input to additional components.

Cooperative Level. Our approach focuses on data cooperation and the sharing of mashup results. The participating devices have two responsibilities: sharing information and consuming the shared mashup result. Each device contributes to the cooperative mashup by sending their shared data upon request. When the mashup result is ready, they all receive the same mashup output.

Scope of Component Integration. Our approach targets the integration of mobile applications and Web services. By using mobile applications as mashup components, we can access various kinds of reusable functionalities that are available from installed mobile applications. For example, a barcode-scanning application can be used as a mashup component that scans barcodes by taking photos and returns the barcode number as text. In addition, using mobile applications as mashup components simplifies the usage of system functions. For example, in order to obtain the current location, we have to deal with the location service API of the mobile operating system. However, existing mobile applications are providing more simple mechanisms to retrieve the current location via their application interface, i.e. using inter-application integration features. Similarly, huge numbers of available Web services provide access to various

kinds of reusable functionalities. As a result, we believe that using mobile applications and Web services as mashup components enables our mashup composition to cover many areas of the user's requirements.

Mashup Composition. We design our approach to support two different mashup composition paradigms: data-flow composition and event-driven composition. These two composition paradigms provide different advantages, and each are suited for solving different problems. Creating data-flow mashup is simple and consumes few resources. This makes it suitable for the composition of data-driven mashup applications. In contrast, event-driven mashup composition can represent more complex mashup scenarios with better stability, but it requires more effort and consumes more resources. Thus, we believe that allowing mashup compositions of both paradigms expands the expressiveness and improves the efficiency of mashup composition.

3.2 Constraints of Mashup Composition on Mobile Devices

The conventional mashup applications for mobile devices are composed as Web applications and delivered to users via a mobile Web browser. The major disadvantage is that it tends to make less use of device-specific features. More advanced approaches develop mashup applications using the device's native programming language in order to overcome the limitations of Web-based applications. However, developing a mashup as a native mobile application has many constraints when compare to developing one for a desktop environment. First, each manufacturer of mobile devices tends to use a different operating system, which makes the mashup applications platform-incompatible. Second, there is the limitation of the mobile platform, which has less bandwidth, less memory and CPU performance, restricted user interaction, and limited battery life. Therefore, composing mashup applications for mobile devices is very challenging.

In this section, the constraints related to enabling mashup composition for multiple mobile devices are presented. The study and evaluation of current mobile technologies helps us discover the solutions for composing mashup applications that employ cooperation between multiple devices.

Based on the characteristics of our mashup composition that we defined above, there are technological constrains that we need to take into account for our approach. Each characteristic contains challenges that we have to overcome. In particular, to expand the scope of integration, we have to deal with heterogeneous mashup components. To allow multiple devices to cooperate, the communication channel and the connection topology should be carefully selected. To turn general mobile applications into mashup components, the inter-application communication protocol should be studied. Finally, to enable event-driven mashup composition, an effective method to handle generic notifications in mobile device should be determined.

Scope of the Integration of mashup components. To extend the expressiveness of our mashup composition, the scope is set to the integration of mobile application and Web resources. Integration of heterogeneous mashup components requires a flexible component integration model. In our previous work [31], we have accomplished the integration of existing mobile applications, open source Web services and typical Web applications. Consequently, this research adapts state of the art models to deal with heterogeneous component integration. However, our previous works revolved around flow-based mashup composition for single-device usage. Tethering capability for multiple devices was also developed, however it was designed for cross-platform compatibility. Thus, this research uses this existing model as a baseline to maintain the scope of integration, and extends it to support mashup composition for multiple mobile devices.

Device connectivity. Mashup composition for multiple mobile devices requires a stable communication channel and a proper connection topology. We choose to implement the connectivity between devices in the mashup composition by using a client-server architecture. Since client-server is an architecture that partitions tasks or workloads between the server device and client device [32], our approach applies this model in order to allow cooperation of capable and less capable mobile devices.

Mobile devices are available in various specifications. Many devices are embedded with powerful processing units and rich with built-in sensors. In contrast, some devices are built with limited resources. Our design aims to distribute the resource-consuming tasks to the most capable devices, while the less powerful devices take care of less resource-consuming tasks. For example, let us consider a mashup of three mobile devices. The first is a device with an Internet connection, while the second and third devices have no Internet connection. In general, the execution of mashup applications requires an Internet connection for invoking additional Web resources. Hence, the second and third devices cannot execute a mashup application by themselves. However, by connecting them in a client-server fashion, the second and third devices can send their device-specific information to the first device, which performs some mashup process and sends the result back to both client devices.

Beside the selection of topology, the communication channel is also an important design consideration. There are many communication technologies (i.e., Bluetooth, NFC and Wi-Fi) that are capable of connecting mobile devices. However, some technologies require special hardware or complex configurations. Since most mobile devices are capable of browsing the Web via a mobile Web browser, our approach selects the HTTP protocol as the communication channel. To enable HTTP connections among devices in a client-server architecture, the server device requires Web server functionality. The Web server on mobile devices is suitable for multi-device mashups because it can host Web services to facilitate data sharing and communication among devices. Also, static Web pages can be used for sharing the mashup result. We found that there are several Web server modules available as ordinary mobile applications, i.e. i-Jetty [33] for Android. However, mobile Web servers are unavailable for some mobile

platforms, particularly iOS. Therefore, in this research we use Android as the first experimental platform for the server device. For the clients, we aim to support both Android and iOS devices to experiment with cross-platform compatibility and minimize the constraints of the client devices.

Application integration. In mobile mashups, mobile applications are used as important mashup components because they can act as a gateway to the reuse of the capabilities of mobile devices. Turning a general mobile application into a reusable mashup component requires a mechanism that exposes those shared functionalities. The standard method to achieve this goal is using inter-application communication.

Advanced mobile platforms have implemented inter-application communication into their operating systems. For example, Android uses Intent [34] while iOS uses URL scheme [35] as the standard inter-application communication protocol. However, the capabilities of these integration methods are different in each platform. Table 2 shows a comparison of inter-application communication for Android and iOS. It can be concluded that URL scheme has limitations compared to Intent. However, iOS is currently extending the application integration logic. We found that there is a specification called "x-callback-url" [36] that provides a standard for iOS developers to share functionality among applications. By using x-callback-url, a source application can launch other applications by passing data and providing parameters. The target application can return data after execution is completed. However, there are only a limited number of applications that are currently conforming to x-callback-url. In the Android platform, Intent is a powerful integration protocol that helps turning an ordinary mobile application into a mashup component. Our previous works have shown that mobile mashups can benefit from the integration of Intent-supported mobile applications. In addition, recently, there have been many Android mobile applications supporting the Intent protocol, and are therefore usable as mashup components. Thus, we set the scope of integration for our approach to Intent-supported mobile applications on Android devices, as well as applications that conform to the x-callback-url specification on the iOS platform.

Features	**URL Scheme**	**x-callback-url**	**Intent**
Availability	iOS, Android	iOS	Android
Invocation	Yes	Yes	Yes
Input parameters	Yes	Yes	Yes
Output parameters	No	Yes	Yes
Parameter type	Text	Text	Text, Number, Binary
Broadcasting	No	No	Yes
Run as service	No	No	Yes

Table 2. Inter-app communication protocol comparison.

Event listening. The objectives of our approach are different from traditional event-driven mashup compositions. We aim to handle general events that happen across the mobile device, and not just a specific domain or pre-defined events. Therefore, another challenge of our mashup composition is dealing with events produced by multiple sources.

In general, events in mobile devices are one of two types: system events or mobile application events. System events are events that are created by the mobile operating system to notify other processes about a change in their state. A list of example system events is shown in

Table 3. Capturing system events may require different protocols and programming models. For example, system events, such as a change in the user's location, can be detected through platform APIs. In contrast, notifications concerning the battery status, incoming calls and whether the device is "charging" or not are sent through a "broadcast intent" [37]. Additionally, mobile application events are events that are created by an installed mobile application to notify the user or other applications. These kinds of events usually notify the user or other applications via the standard notification channel of the mobile operating system, e.g. the Notification Center in case of Android. For example, incoming email, missed calls or SMS notifications can be detected using the Notification Center. Mobile application events that are produced by custom protocols, however, are out of our scope. In this way, we can create a variety of mashup applications and cover various domains of a user's requirements using system events and standard mobile application events in mashup composition. There are other types of events that we consider out of scope: primarily user interface events that happen when a user interacts with the system or a particular mobile application.

System Event	Description
AIRPLANE_MODE_CHANGED	The user has switched the phone into or out of Airplane Mode.
ANSWER	Handle an incoming phone call.
BATTERY_LOW	Indicates low battery condition on the device.
BATTERY_OKAY	Indicates the battery is now okay after being low.
BOOT_COMPLETED	This is broadcast once, after the system has finished booting.
CALL	Perform a call to someone specified by the data.
CAMERA_BUTTON	The "Camera Button" was pressed.
DATE_CHANGED	The date has changed.
DIAL	Dial a number as specified by the data.
DOCK_EVENT	A sticky broadcast for changes in the physical docking state of the device.
DREAMING_STARTED	Sent after the system starts dreaming.
HEADSET_PLUG	Wired Headset plugged in or unplugged.

INPUT_METHOD_CHANGED	An input method has been changed.
LOCALE_CHANGED	The current device's locale has changed.
MEDIA_REMOVED	External media has been removed.
POWER_CONNECTED	External power has been connected to the device.
POWER_DISCONNECTED	External power has been removed from the device.
SHUTDOWN	Device is shutting down.
TIMEZONE_CHANGED	The time zone has changed.
TIME_CHANGED	The time was set.
TIME_TICK	The current time has changed.

Table 3. Android's system events.

Based on the discussion on the constraints of mashup composition on mobile devices above, this research selects Android as our first experimental platform for mashup composition across multiple mobile devices. Table **4** shows a summary of the scope for our mashup composition approach.

Constraints	**Explanation**
Required components	Pre-installed software component for client devices and mobile a Web server for server devices
Communication style	HTTP request/response on a client-server architecture
Platform constraints	Android for servers and Android or iOS for clients
Mashup component	Mobile applications and Web services
Component restrictions	Intent or x-call-back-url supported mobile applications and REST Web services returning JSON or XML
Integration patterns	Data-flow and event-driven
Event sources	System events and mobile application events
Event restrictions	Intent broadcasting and notification center
Cooperative level	Data cooperation

Table 4. Scope of our approach.

Chapter 4

Data-Flow Composition for Multiple Devices

Mashup technology is initiated by the integration of Web resources, so called Web mashups. To integrate functionalities of mashup components, mashup approaches use composition patterns to capture frequently occurring programming logic. Most existing mashup approaches are using a data-flow (flow-based) style as a major composition pattern as it can represent programming logic in terms of data-flow [38]. This chapter proposes a mashup approach for composition of data-flow applications for multiple mobile devices. We use a description-based approach for the integration of mobile applications, Web services, and Web applications in order to realize cooperation of mobile devices. A description language called C-MAIDL is designed for describing the configuration of mashup components and the dataflow of component compositions. To reduce development costs, the description language is used as an input for a mashup generator intended for generating mashup applications. Finally, we demonstrate that our approach allows users to create mobile mashup applications dealing with cooperation of devices easily and efficiently.

4.1 Single Devices Mashup Composition

In general, data-flow mashup composition is based on the data-flow programming paradigm [38]. The mashup program is defined by the connection of mashup components. Mashup components are connected as a directed graph and communicate via message passing.

Figure 2 shows a simple scenario for a data-flow mashup application that is running on a mobile device. Pictured is a mobile mashup application that scans a barcode of a product, and then searches for product information on online stores. The mashup then translates the product information into a specific language and shares the translated information on Twitter and Facebook. With flow-based 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 the product information. Then, the product information is translated

into a specific language by a translation Web service component. Finally, the translated text will be used as input for Twitter and Facebook applications to create a post on either social network. For a single-device mashup scenario, we realize that the data-flow composition pattern is simple and efficient for a data-driven mobile mashup. Therefore, we extend our ideas by adapting data-flow mashup composition on single devices for use with multiple devices. In other words, our approach aims to allow integration of cooperative data from multiple mobile devices with other mashup components in a data-flow driven manner.

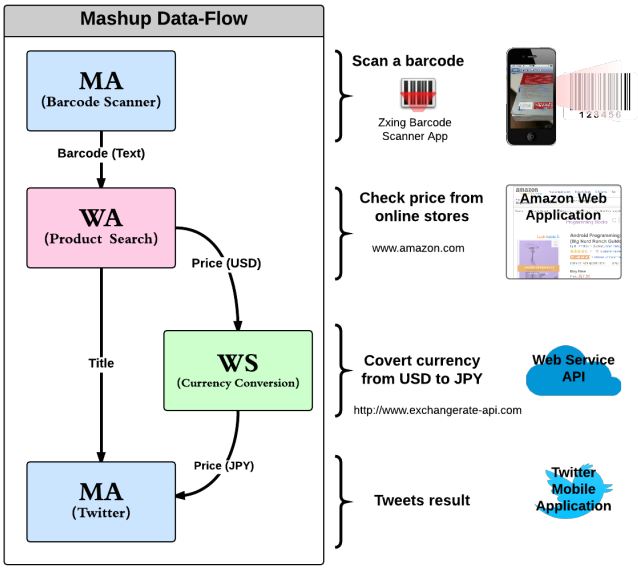


Figure 2. A simple data-flow mashup component integration

4.2 Enabling Cooperation of Mobile Devices

To achieve the goal of a cooperative mashup for mobile devices, we first design the device cooperation model that facilitates the cooperative process among devices, as shown in Figure 3. Our approach uses a client-server architecture to enable cooperation of devices. The participating devices are categorized into one of two roles: a host device (server) and guest devices (clients). The execution of a mashup application requires that a single host device cooperates with one or more guest devices. The guest device is responsible for providing shared information for the execution of the mashup application, while the host device takes care of gathering shared information from each client device and uses the collected information in the mashup process. The mashup process is executed on the host device by integrating the collected information from participating devices with additional mashup components to produce the mashup result. When the mashup result is ready, guest and host devices synchronize and share the result. The cooperation center and a cooperation agent work together to establish a connection between the host and guest devices, and accommodate data sharing among them. This operation involves the interaction between two or more devices. Thus, the

synchronization of requests and responses is an important design consideration. In addition, privacy and permission control must be included in the cooperative process. Based on these requirements, we design the needed steps for cooperation and group them into three distinct phases: initiation, execution and publication. Figure 4 illustrates the coordination phases and the tasks they contain.

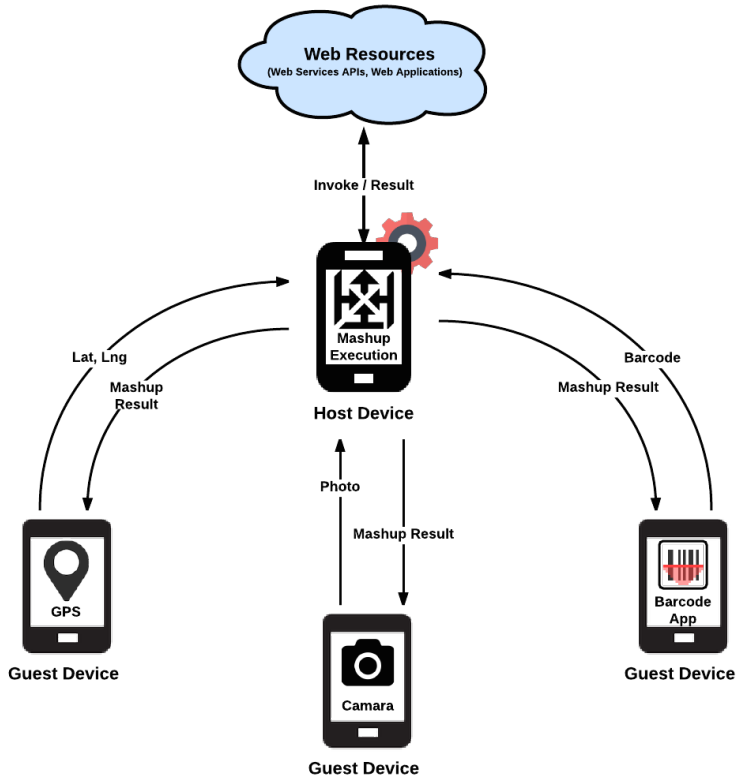


Figure 3. Multiple devices cooperation model

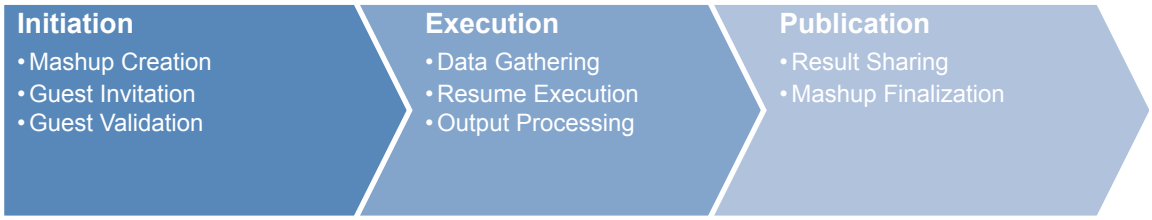


Figure 4. Cooperation phases.

Initiation Phase

The initiation phase consists of following 4 sequential tasks.

i. *Mashup creation.* To start a mashup using cooperation of mobile devices, a cooperative mashup workspace will be created on the host device. The mashup workspace contains a mashup application identifier, a list of guest devices, and a mashup output identifier. This information makes the mashup application unique and allows multiple mashup applications to run on the same host device.

ii. *Guest invitation.* Before starting the mashup process, the host device selects guest devices that will join the created mashup application. Prospective guests can be selected from a list of email addresses or phone numbers available in the host's contact list. After the list of guests is finalized, the host device sends invitation messages to each selected guest and waits for responses.

iii. *Guest validation.* Guest devices can either accept or decline to join the mashup. In case of acceptance, a guest device sends a reply message to notify the host that this particular guest is ready to join the mashup. Alternatively, if a guest declines or does not respond to the invitation, the host device can remove that guest from the participation list and continue listening.

iv. *Mashup execution.* The above invitation processes help our system manage and track availability of guest devices. Once the participation list is completed, the mashup application is started.

To provider a better understanding on the initiation phase, the sequence diagram in Figure 5 shows the step-by-step interaction between a host and a guest (sequence number 1-8).

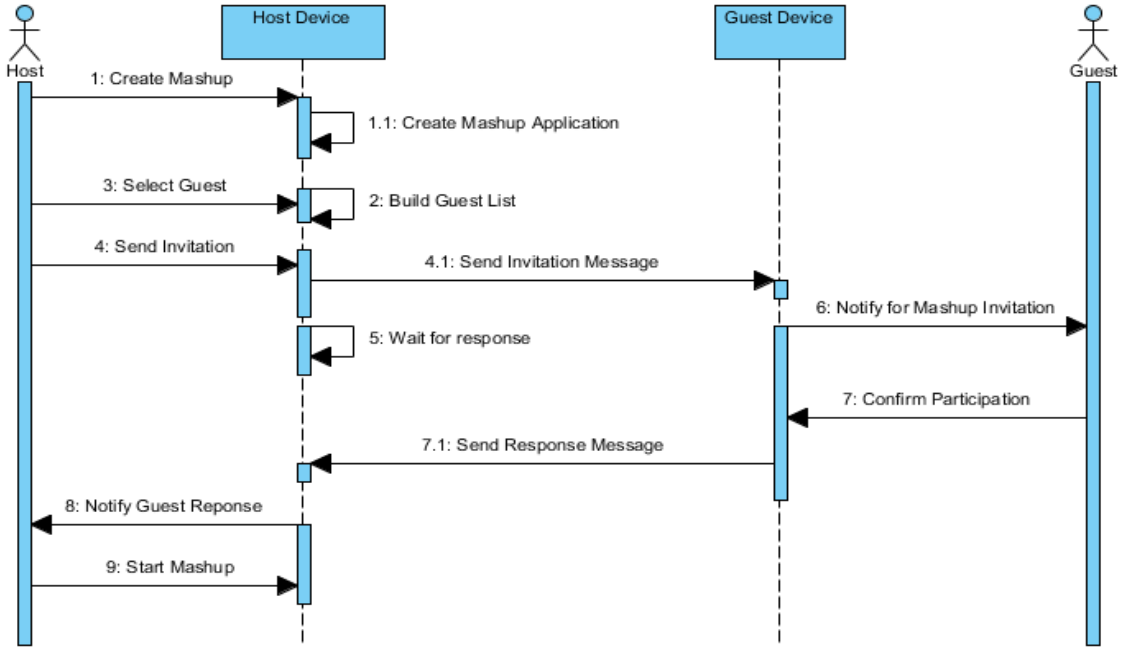


Figure 5. Sequence Diagram of the Initiation Phase

Execution Phase

The execution phase consists of the following 3 sequential tasks.

i. *Data gathering.* During mashup execution, data from guest devices may be required. The host device will send messages to each guest device with a request for data cooperation. The host device then waits for replies. Once the message arrives at a guest device, the user of the guest device will get a notification about

a pending request for data cooperation. If the user chooses to accept the request, the guest device sends the requested information back to the host device.

ii. *Resume execution.* When all requested data has returned from the guest devices, the host device resumes execution of the mashup process. The collected data is integrated with additional mashup components. However, the host device may go back to the information gathering process and resume the mashup execution later in case other data cooperation is needed.

iii. *Output processing.* After all mashup logic has been executed, the host arranges the result and shares it among the participating devices.

The sequence diagram in Figure 6 explains the detailed steps of the execution phase (sequence number 9-13).

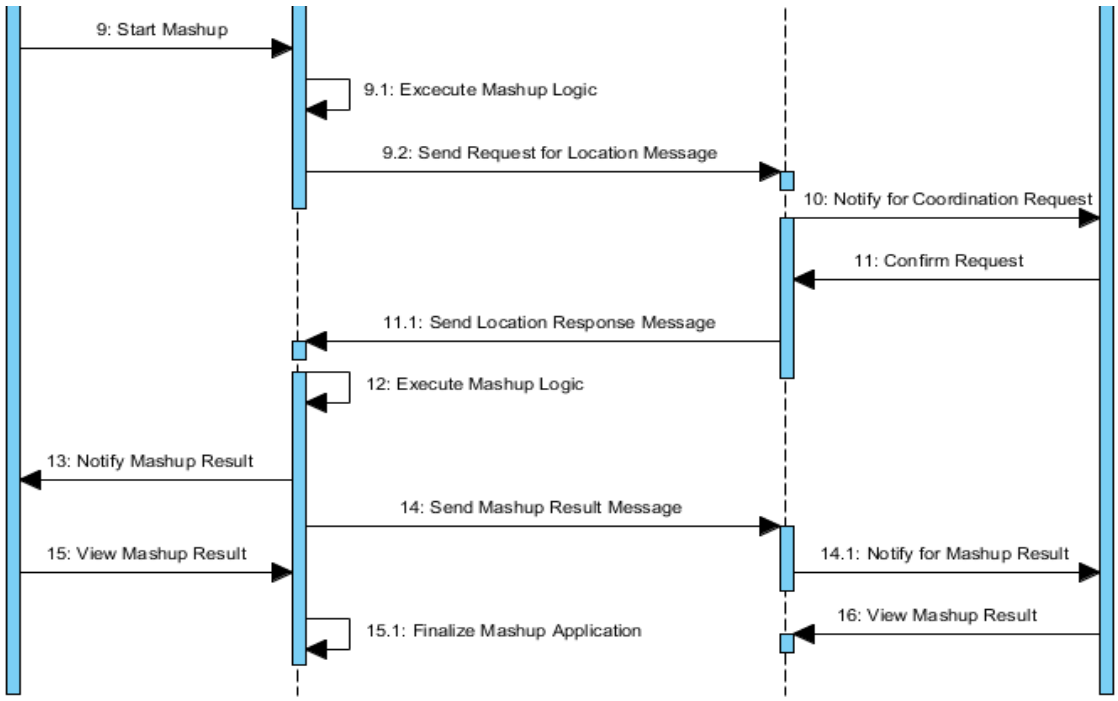


Figure 6. Sequence Diagrams of the Execution and Publication Phases

Publication Phase

The publication phase consists of the following two sequential tasks.

i. *Result sharing.* The users of host devices will get notifications about the status of the mashup result. When the result is ready, the user can start sharing the result by sending messages to inform all participating devices. The notification messages include the location of the mashup output and authentication information. The guest devices then extract the required information from the received message and browse to the mashup result.

ii. *Mashup finalization.* Finally, after all cooperative tasks are finished, the host device terminates the mashup application and releases resources to the system.

The sequence diagram in Figure 6 (sequence number 14-16) shows the steps included in the publication phase.

4.2.1 Mashup Execution Environment

To achieve a mashup for cooperation of multiple devices, the participating devices require the capability to communicate with each other in order to exchange information. We develop a mashup execution environment to automate these tasks. Our mashup execution environment allows the devices to exchange information by using custom mobile applications called cooperation agents and cooperation centers.

In our mashup execution environment, we have categorized the participating devices into two types: the guest device (guest) and the host device (host). The guest device is a mobile device that provides information to be used in mashup applications. The host device is a mobile device which executes mashup applications by using information from the guest devices. The cooperation agent and cooperation center work together to enable information sharing between guests and a host. The cooperation agent will be installed on the guest devices to extract device-specific information required by the mashup process. The cooperation center will be installed on the host device to collect required information from guests. An overview of our mashup execution environment is shown in Figure 7.

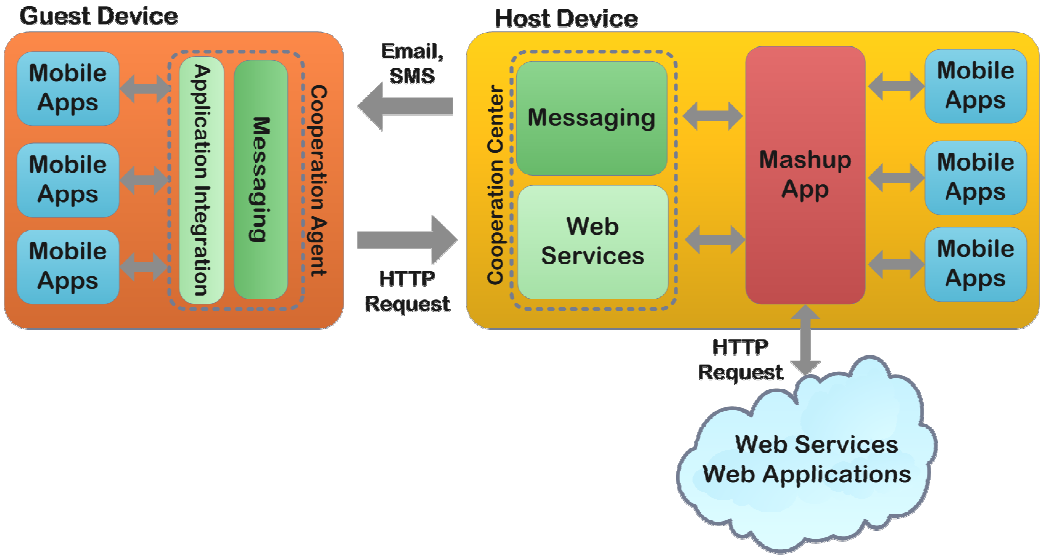


Figure 7. Architecture of mashup execution environment.

Within the mashup execution environment, mobile applications on guest devices can be integrated into mashup. When a mashup application needs information from guest devices, it will interact with the cooperation center. The cooperation center provides programming interfaces for sending request messages to all guests. When a request has arrived at a guest device, the cooperation agent will invoke a mobile application corresponding to the request. For example, if a request for barcode scanning

is received, a barcode application on the guest device will be executed to return the scanned code to the host device. When the host device has received all response messages, it starts integrating the received information with other mashup components according to the mashup logic.

Cooperation Messaging. An important mechanism that enables the cooperation of mobile devices is related to sending and receiving information among devices. An efficient messaging system must be taken into careful consideration. Our approach proposes a messaging system for exchanging cooperative information between mobile devices. In our experiments, we use two different mobile platforms to develop an efficient messaging system. Google's Android device is implemented as the host device while iOS devices are implemented as guest devices. The Android device is selected to be the host device as it has a flexible mobile operating system. Use of special purpose modules is possible, i.e., the mobile Web server. We apply functionalities of the i-Jetty mobile Web server in our messaging system. It is used as a container of Web service APIs that accommodate the communication between different mobile platforms. RESTful Web services and the JSON data format are adapted for better execution performance in a mobile environment [39].

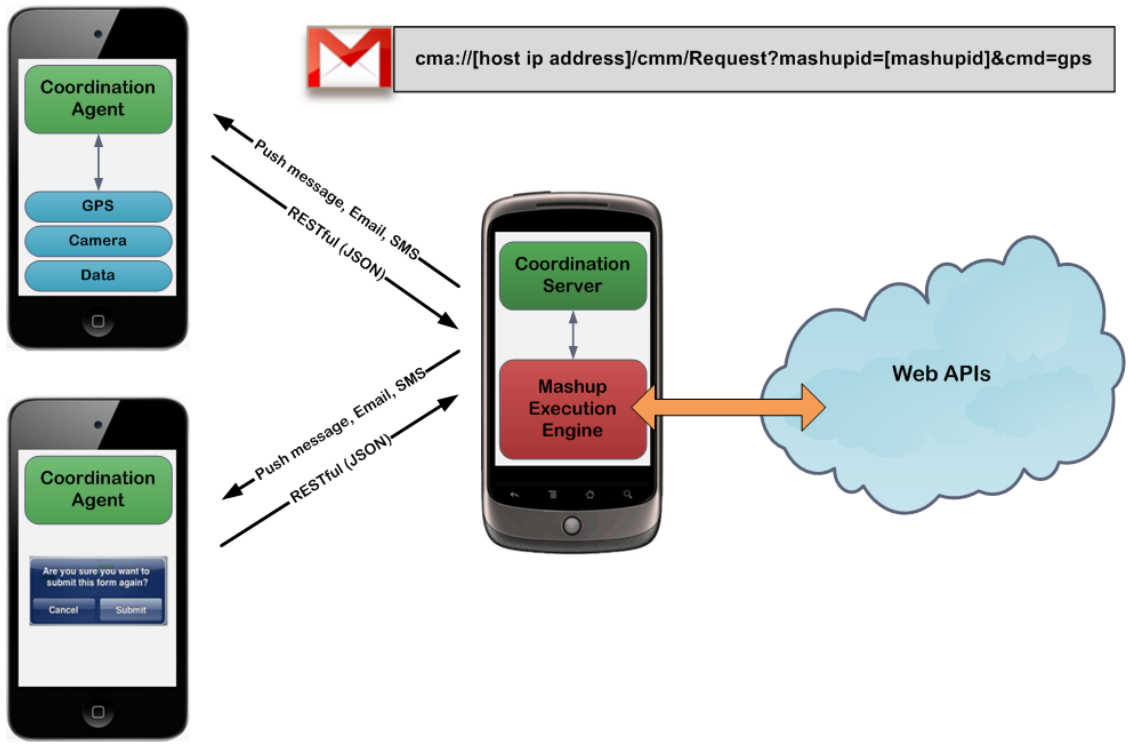


Figure 8 Overview of the architecture for cooperation messaging.

During implementation of the messaging system that targets both Android and iOS devices, we found that there is an important limitation inherent to the iOS platform. iOS does not allow a custom mobile application to run as a background process. Due to this limitation, the capability of communication software listening to incoming requests is

limited. Our messaging system applies different techniques for request and response activities to overcome this limitation. The overview of the architecture for our cooperation messaging system is shown in Figure 8.

Request Message. For collecting cooperative information from guest devices, the cooperation center on the host device creates and sends request messages to all guest devices. The messages will be sent via a standard messaging protocol such as SMS or Email. The cooperation agent on guest devices will receive the messages and reply with the requested information. However, due to iOS limitations, the cooperation agent has to be activated in order to reply to request messages from the host device. To activate the cooperation agent, we use a technique called URL Scheme Mapping. A custom URL scheme can be registered to the iOS device for invoking a particular mobile application. When a user touches a custom URL that is already registered, the corresponding application is brought to active context of the iOS device. In our design, the messages sent to guests include a registered URL Scheme (cma://) and additional parameters. Users of guest devices can invoke the cooperation agent by touching the URL in the received messages. Subsequently, the cooperation agent is brought up and extracts part of the query string in the URL to determine which information is requested. The user interface of the cooperation agent will ask for confirmation before replying to the request. An example of a request URL is shown in Figure 9-A. The parameters of the request message are presented in Table 5.

A. An example of a Request for a Location (URL Scheme)

```
cma://host/cooperation/request?mid=cm001&gid=cma@me.com&cmd=gps&lat=[lat]&lng=
```

B. An example of a Location Response (HTTP)

```
http://host/cooperation/request?mid=cm001&gid=cma@me.com&cmd=gps&lat=36.1551&
```

Figure 9. Examples of request/response messages.

Response Message. In order to return requested data to the host device, the cooperation agent determines the required resources by extracting parameters from the URL in the received message. When the URL is decoded, the cooperation agent invokes the target mobile application and acquires the requested information. For example, given a request for barcode scanning, the cooperation agent invokes a barcode scanner application and obtains the result after the user has finished scanning. For an iOS device, we use x-callback-url to enable integration of existing mobile applications. The x-callback-url is a specification that aims to standardize inter-application communication. However, only very few iOS applications currently support the x-callback-url specification. Therefore, we develop testing applications that conform to the x-callback-url specification to demonstrate how to enable inter-application integration on the iOS platform. To send data back to the host device, the cooperation agent builds a reply HTTP request by adapting the original requested URL, and submits it to the Web service APIs on the host device. The Web service APIs on the host device are implemented using Java Servlets

on the i-Jetty mobile Web server. An example of a response URL is shown in Figure 9-B. The following table shows the composition pattern and parameters of the request message used in the cooperation messaging system.

```
cma://<host>/<spath>/<message>?<mid>=<mashupid>&<gid>=<guestid>&<cmd>=<command>&[params=<params>]
```

Parameter	Description
cma	Registered URL scheme
host	Host device IP Address
spath	Host Service API path
message	Message name (e.g. invite, request or output)
mid	Mashup ID
gid	Guest ID
cmd	Request data (e.g. gps, barcode or rsvp)
params	Other request parameters

Table 5. Parameters of request message.

The following figure (Figure 10) shows a sample of cooperation messages that are sent to guest devices to request for barcode scanning and another message that requests a user's location. As shown, the messages can be sent to guest devices using SMS and email messages.

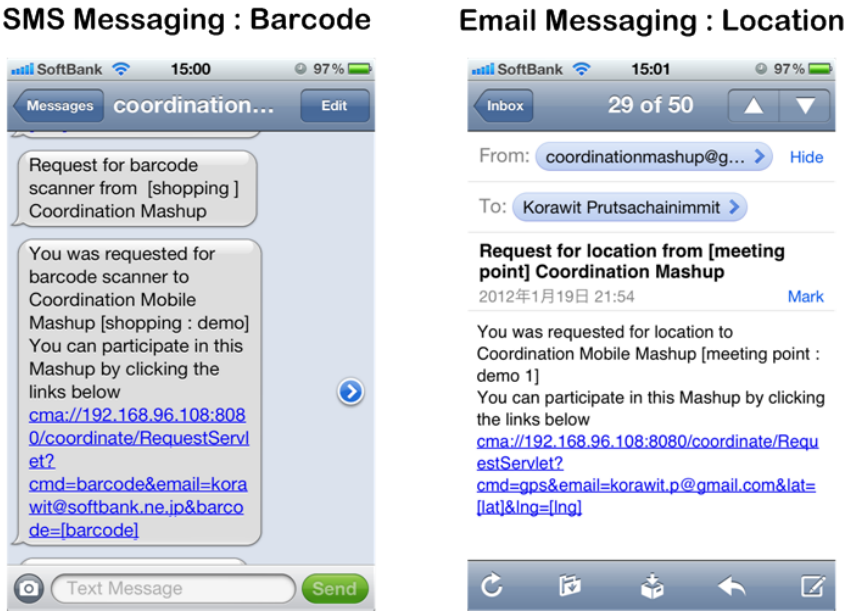


Figure 10. Screenshots of cooperation messages.

4.3 C-MAIDL

C-MAIDL is an XML-based description language, designed for describing the data-flow of mashup applications. It provides ways to describe the configuration of mashup components and the data-flow of component integration. The components can be configured with a set of parameters and connected as a workflow according to the logic of mashup composers. The output of components is shared with the others through a publisher-subscriber model. Results from the components in the upper hierarchical order can be used as input for the lower ordered components. Finally, the composers configure the output component and export the abstracted model as a C-MAIDL description file.

An outline of a C-MAIDL XML description file

```
<project>
     <name> <!--[project name]--> </name>
     <component>
          <name> <!--[component name]--> </name>
          <role>
               <!--[role] tag, use one or both of <publisher><subscriber>-->
               <!--example:<publisher id=002 /><subscriber id=001 />-->
          </role>
          <execution> <!--[single] or [multiple]--> </execution>
          <!--[component configuration] tag chosen from-->
          <!--<mobileapplication><webapplication><webservice><arithmetic>-->
     </component>
     <!-- more components -->
     <output>
          <!--[output configuration] tag chosen from-->
          <!--<mobileapplication><webapplication><webservice>-->
     </output>
</project>
```

C-MAIDL is an extension of our proposed mashup description language MAIDL. The general concept of MAIDL is to provide data flows between mashup components for their execution and output. MAIDL can be used to describe the composition of the following components: the Web Application Component (WA), the Web Service Component (WS), the Mobile Application Component (MA) and the Arithmetic Component (AR). By configuring these components, the mashup composer can extract parts of Web pages, invoke Web services for results, call existing mobile applications and perform arithmetic operations between outputs of components. However, MAIDL does not support the description of data cooperation among multiple devices. Manual programming efforts are required to create cooperative mashup applications. Therefore, we extend C-MAIDL from MAIDL to support cooperative tasks by adding new components to the existing language definition. Additional mashup components, the Cooperation Component and the Output Component, are added to expand the expressiveness. Thus, C-MAIDL mashup components consist of the following.

Web Application Component (WA). Web applications are applicable for integration. This component is used for extracting a part of a Web page or executing queries via an HTML form. Mashup composers are provided with a Web extraction assistant tool [40] to indicate the required information on a Web page. The description of this component will be generated as JavaScript code and executed in the runtime environment on a mobile device. Figure 11 shows the configuration process of the Web application component.

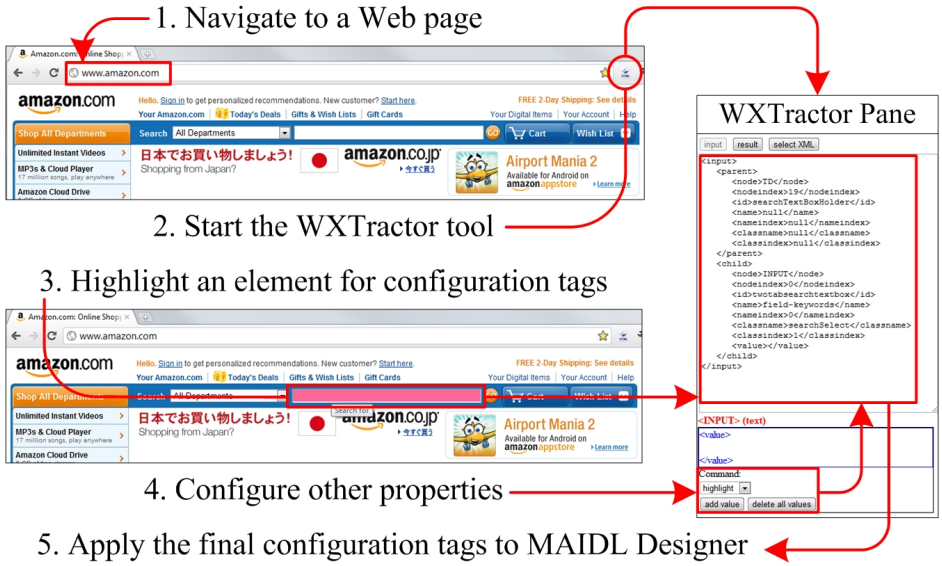


Figure 11. Web extraction process of MAIDL.

Example of Web application component configurations for reusing product search on Amazon.com

```
<webapplication>
      <mode>active</mode>
      <url>http://www.amazon.com/?ie=UTF8&amp;force-full-site=1</url>
      <inputs>
            <input>
                  <input-name>ISBNSearch</input-name>
                  <source>null</source>
                  <parent>
                        <node>table</node>
                        <nodeindex>6</nodeindex>
                        <id>subDropdownTable</id>
                  </parent>
                  <child>
                        <node>input</node>
                        <childindex>0</childindex>
                        <id>twotabsearchtextbox</id>
                        <name>field-keywords</name>
                        <nameindex>0</nameindex>
                        <classname>searchSelect</classname>
                        <classindex>1</classindex>
                        <value>input.ISBN,submit</value>
```

```
            </child>
          </input>
       </inputs>
      <results>...</results>
</webapplication>
```

Web Service Component (WS). This component is used for consuming a RESTful Web service by specifying a URL and query expressions (such as XPath or JSON). The target Web service will be invoked, and a result is returned. We can use the full data or extract a part of it as output.

Example of a Web service component description for currency conversion using an Exchangerate API.

```
<webservice>
    <base>http://www.exchangerate-api.com/</base>
        <paths>
            <path>usd</path>
            <path>jpy</path>
            <path>publisher.results.ProductPrice</path>
        </paths>
        <fields>  <field>k</field>  </fields>
        <values>
            <value>PQkn3-quzTZ-PNDav</value>
        </values>
    <results>
        <result>
            <result-name>YenPrice</result-name>
            <type>single</type>
            <format>SELF</format>
            <query>null</query>
            <index>null</index>
            <filter>null</filter>
        </result>
    </results>
</webservice>
```

Mobile Application Component (MA). A mobile application can be described by using this component description. It allows an application that implements the Intent and Service messaging protocols to work as a mashup component. The Intent Filter and additional parameters are required for the configuration of this component.

Example of a Mobile Application component description for calling the Zxing Barcode Scanner App.

```
<mobileapplication>
     <mode>active</mode>
     <intent>
          <intent-name>com.google.zxing.client.android.SCAN</intent-name>
          <intent-extra>
```

```
                <extra-name>SCAN_MODE</extra-name>
                <datatype>string</datatype>
                <extras>
                     <extra>BAR_CODE_MODE</extra>
                     <extra>...</extra>
                </extras>
           </intent-extra>
     </intent>
     <results>
           <result>
                <result-name>ScanResult</result-name>
                <type>single</type>
                <datatype>string</datatype>
                <value>SCAN_RESULT</value>
                <filter>null</filter>
</result>
           <result>...</result>
     </results>
</mobileapplication>
```

Arithmetic Component (AR). This component provides pre-defined mathematical operations between the results from one or more components. The operation includes addition, subtraction, division, multiplication, summation, comparison, and GPS distance calculation.

Example of an Arithmetic component description for adding a number to an output parameter from another component.

```
<arithmetic>
     <execution>single</execution>
     <operation>add</operation>
     <inputs>
           <input>pulisher.results.YenPrice</input>
           <input>1050</input>
     </inputs>
     <results>
           <result>
                <result-name>TotalPrice</result-name>
                <type>single</type>
           <datatype>number</datatype>
                <value>arithmetic.calculation</value>
                <filter>null</filter>
           </result>
     </results>
</arithmetic>
```

Cooperation Component (CC). This component will be used for cooperation between multiple devices. Required information from participating devices can be described in this component. The description of this component will be converted to code for

communicating with the mashup execution environment to exchange information with other devices.

Example of a Cooperation component description for requesting a barcode scan from participating devices.

```
<cooperation>
      <mobileapplication>
            <app-name>bcapp://x-callback-url/scan?</app-name>
            <app-params>
                  <param-name>formats</param-name>
                  <values>
                        <value>EAN13,EAN8,UPCE,QR</value>
                  </values>
                  <param-name>success</param-name>
                  <values>
                        <value>scanresult</value>
                  </values>
                  <param-name>error</param-name>
                  <values>
                        <value>error</value>
                  </values>
            </app-params>
      <mobileapplication>
      <results>
            <result>
                  <result-name>scannedcode</result-name>
                  <type>single</type>
                  <value>code</value>
                  <filter>null</filter>
            </result>
      </results>
</cooperation>
```

Output Component (OC). The output of a mashup application can be defined with this component. Mashup composers can choose to show the mashup result as pins on the map view or to display it as a Web page in the Web view.

```
<output>
      <cooperation>
            <output-type>web</output-type>
            <output-params>
                  <title>Shopping Compare Coordination Mashup Result</title>
                  <table border=2>
                        <tr>
                              <td colspan=2>Guest</td>
                              <td>cooperation.GuestID</td>
                        </tr>
                        <tr>
                              <td colspan=2>Title</td>
                              <td>publisher[003].results.ProductName</td>
```

```
                        </tr>
                        <tr>
                              <td>Store</td>
                              <td>publisher[003].results.StoreName</td>
                              <td>Link</td>
                              <td>publisher[003].results.StoreURL</td>
                        </tr>
                        <tr>
                              <td>Price(Yen)</td>
                              <td>publisher[004].results.YenPrice</td>
                              <td>Price(USD)</td>
                              <td>publisher[003].results.Price</td>
                        </tr>
                  </table>
            </output-params>
      </cooperation>
</output>
```

To illustrate C-MAIDL, examples of the Cooperation Component and Web Service Component are shown in Figure 12.

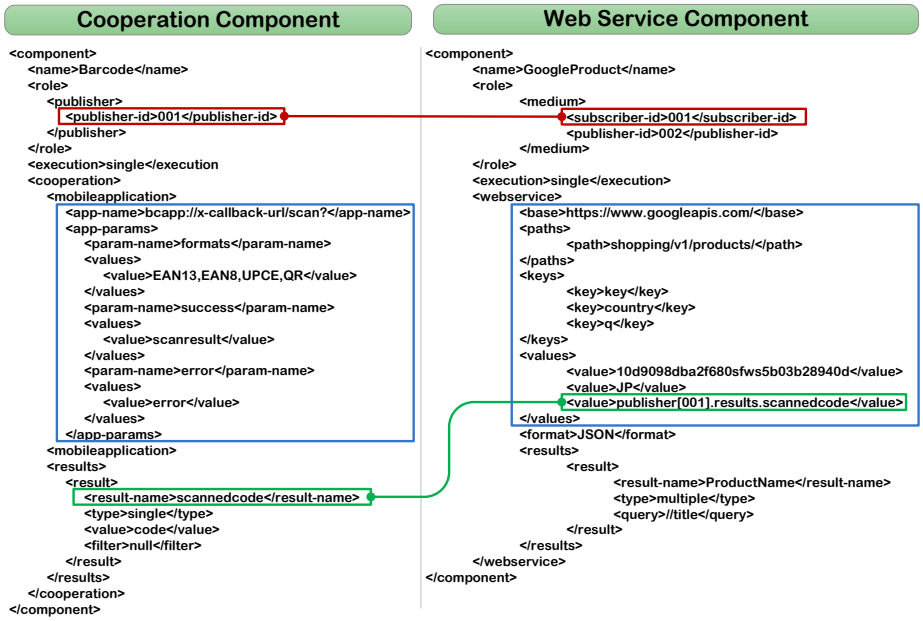


Figure 12. Examples of a C-MAIDL description.

The Cooperation Component is configured as a publisher to provide data to other components. A target mobile application and its launch parameters are specified to activate barcode scanning on the participating devices. Output data from this Cooperation Component is defined (in this case "scannedcode") to later be referred by the other components. The Web Service Component is configured as a subscriber and publisher. As a subscriber, this Web Service Component uses the scanned barcodes from the Cooperation Component as an input to a Web Service API. As a publisher, the result from the Web service execution will be available to the other components.

4.4 Mashup Construction Process

The mashup construction process is shown in Figure 13. To compose a mashup application, a mashup composer creates an abstract model of the mashup by using C-MAIDL to transform the abstract model into a mashup description. The description file will be used as an input for the mashup generator to generate Java source code. This generated code will be compiled into an application, which can be deployed on a target device as an ordinary mobile application.

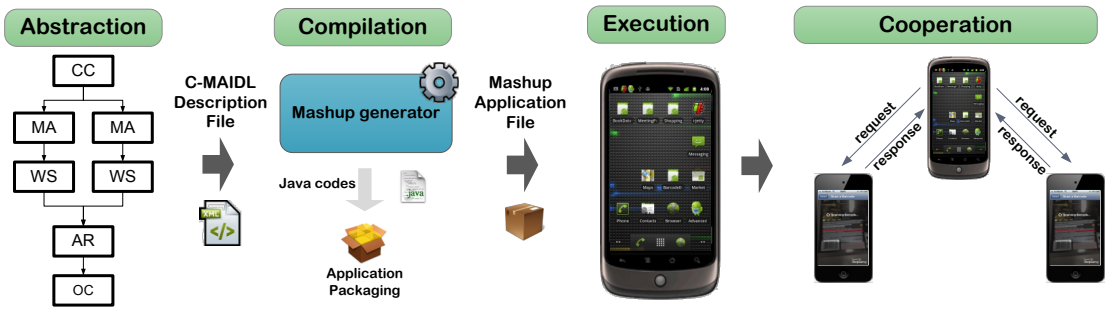


Figure 13. Mashup Construction Process.

This tool takes a C-MAIDL description file as an input to generate a mobile mashup application according to the mashup generator. First, the generator extracts the description of a component from the C-MAIDL file, and then generates Java source code corresponding to the specification. Next, all source code will be manually compiled into an Android package file (.apk). The package file will then be manually installed on the target device by using Android Debug Bridge (adb). Once the generated mashup application is installed and invoked by a mashup user, the flow, which was defined in the C-MAIDL description, will be executed. The connection between participating devices will be established as soon as there is a need for cooperation information. Our mashup execution environment automatically handles the mechanism of requests and responses using pre-installed software components, cooperation agents and the cooperation center.

4.5 Implementation

In order to demonstrate the capabilities of our mashup composition approach, we implemented some sample mobile mashup scenarios. In this section, we present two cooperation mashup scenarios, called *Shopping Assistance* and *Meeting Point*.

To enable host functionalities, some software components are required. The cooperation center and i-Jetty mobile Web server must be installed on the host device. For guest devices, the cooperation agent must be installed to accommodate connectivity among devices. In addition, to demonstrate mobile application integration on guest devices, custom mobile applications (e.g. GPS Locator and Barcode Scanner) have been installed on the guest devices.

4.5.1 Cooperation Mashup Scenarios

Shopping Assistance: Camera and Data Integration Mashup. This sample scenario simulates a shopping situation given three members of a family in a department store. The goal of this mashup is to assist the family in comparing prices of products in a local department store with online stores. It creates a list of selected products and some additional information, such as the product title, prices and links. An Android device works as the host device. Two iOS devices coordinate with the host as guests. The guest devices will scan barcodes of selected products and send it to the host device. The host device then executes the mashup by using the collected barcodes to get information of selected products and creates the summary list. Finally, the list will be shared among the three devices.

The mashup model and screenshots of the mashup application are shown in Figure 14. In this mashup, a host device sends a request for a barcode to all guest devices. The guest devices read the barcodes of selected products and submit it to the host device. The barcode is given to Google's Search API for Shopping [41] to find available online stores and prices. The arithmetic component filters and extracts the lowest price. The price is converted into the designed currency using the Exchange Rate API [42]. Selected products from each guest are processed and combined into a list. Finally, the list of products and a comparison of prices is shared among all devices.

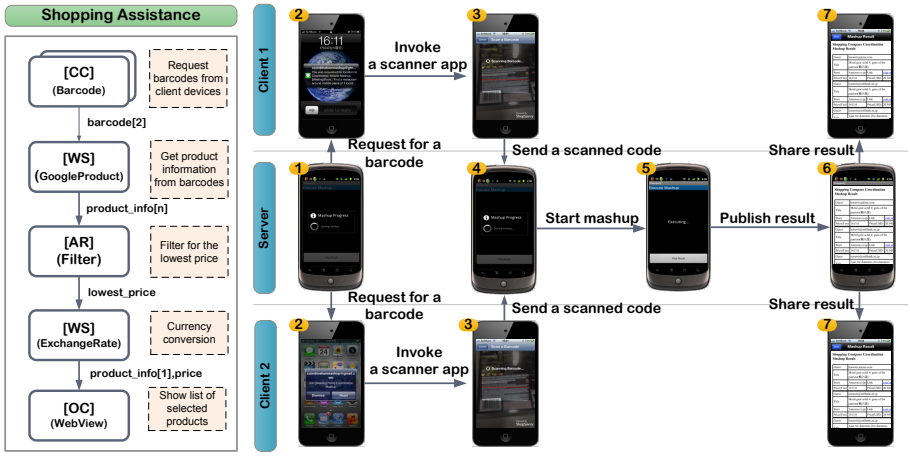


Figure 14. Mashup model and screenshots of Shopping Assistance.
**See Appendix A for the full description of C-MAIDL*

Meeting Point: a Geo-location Mashup. This mashup scenario aims to find the best-ranked restaurant located near the center point of participating devices. Coordinates of three devices are used as input to find the center. The center point obtained from the arithmetic calculation is used to find the nearest train station via the Google Places API Web services. The Gourmet Navigator API is used to find the best restaurant that is nearest to the selected train station. Finally, the details of the meeting point are shared among all devices using map views. The mashup model and screenshots of the mashup application are shown in **Figure 15**.

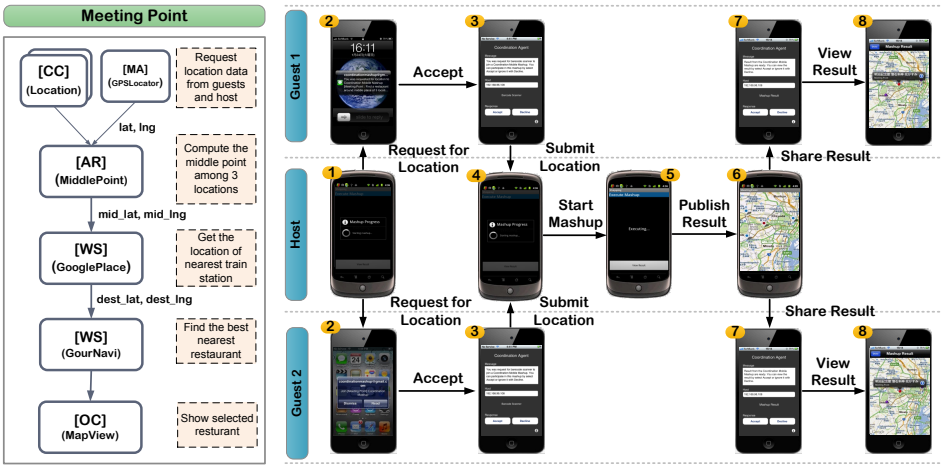


Figure 15. Mashup Model and Screenshots of Meeting Point

See Appendix A for the full description of C-MAIDL

4.5.2 Discussion

Performance of Mashup Execution. From the sample scenario, we found that the major performance factors for the cooperation mashup application lie in the consumption of Web resources and the idle time of the messaging system. By using multiple Web resources in a mashup application, the host device has to create multiple Internet connections to get the results. This task is resource consuming. For instance, in the shopping assistance scenario, the primary workload of the host device is to query the Google Products Web services. Another performance issue is related to cooperation messaging. Since our system applies standard protocols (e.g. Mail and SMS) for sending and receiving cooperation messages, the time it takes to send a message and for a message to arrive is up to the server and network utilization at that moment.

Privacy Protection Trade-offs. Our approach provides a mechanism for privacy protection. The confirmation dialogs of the cooperation agent allow users to verify which information will be shared in the mashup. However, there is a trade-off between mashup execution and privacy protection. When a user enables privacy protection, his/her confirmation becomes a requirement for sharing information; consequently the capability of automatic mashup execution will be disabled. User interaction is required throughout the full process of the mashup.

Robustness of the Messaging System. In the cooperation messaging system, we assumed that we can use global IP addresses to connect participating devices. The guests and the host require an Internet connection to access Web resources and to connect to each other. In some case, losing the network connection may interrupt the mashup execution. Our messaging system, however, avoids failure by using an asynchronous architecture. We use standard messaging systems, such as Email or SMS,

in which host and guest devices wait for cooperation messages asynchronously. Users will be notified about incoming messages via the notification features of the mobile operating system. This allows the guest devices to temporarily disconnect from the network after they have shared the required information. Later, guest devices will need the network connection again when the mashup result is ready. However, some timeout configuration should be implemented in case of a prolonged or permanent disconnection.

Reusability of the Mobile Application Component. The mobile application component is an important mashup component in our mashup composition because it contains a large number of reusable functions. However, using a mobile application as a mashup component is still not efficient enough; namely, the design of the description language is still very technology-dependent. In the current design of C-MAIDL, the mashup description file contains a mobile application component section that describes configurations of mobile applications. The configurations are designed for describing Intent and Intent Filter configurations. In practice, mashup applications should not be aware of a specific technology. To make our approach more extensible, the component description should contain only shared functionalities of components and parameter details in order to remove the dependency on specific technologies from the mashup description.

Mashup Composition. The implementation of the sample scenario indicates that our approach provides an efficient solution for a cooperative mobile mashup. However, our approach is not designed to support an event-driven mashup where mashup components are executed by events. In an event-driven mashup, guest devices may publish their information to the host and update their data when an event is triggered. Host devices have to be aware of changing cooperation information in order to update the mashup result. For instance, our approach will request for locations from guests only once, but in some cases, the participating devices may move to other locations. The host device needs to trace the new locations to update the mashup results.

Scope of Integration. Our system is able to create mashup applications, which integrate various types of mashup components. However, we found that some specific types of resources cannot be included in our mashup composition, e.g., Java Applets, Flash Objects, Web services that require authentication and especially mobile applications that are not implemented using an application integration mechanism. In general, a mobile application is created for a specific purpose. They may not provide the mechanism to collaborate with other applications. Thus, these kinds of mobile applications cannot be used in our system.

Mobile Platform Constraints. In this research, we implemented the functionality of the host device on Android only. Since our messaging system uses Web services, the target platform must be able to function as a Web server and Web services container. We found that Android devices are suitable as host devices because of the availability of several mobile Web server implementations. If there is a new mobile device platform that can be used as a Web service container, it can function as a host device. For guest

devices, the participating devices have to install the cooperation agent that we have provided for both Android and iOS platforms. We can expand coverage of mobile platforms by customizing the cooperation agent software for additional mobile operating systems.

Configuration of the Web Application Component. For the Web application integration, our previous work applies a Web extraction engine [40] to extract information from form-based Web applications and uses a wrapper to integrate this information with other mashup components. However, the process and supporting tools for turning a Web application into a mashup component are still not efficient enough. Figure shows the process of extracting information from a Web application using the WXtractor [6] tool. Recently, several powerful Web extraction platforms that provide Web service interfaces have become available, facilitating the integration with other software components. As a result, it is possible to use these new tools to deal with form-based Web applications instead of using the Web extraction tool from our previous work. Thus, we can limit our integration scope to mobile applications and Web services to minimize the complexity of mashup composition while we still maintain the same level of expressiveness.

Limitations of data-flow composition. Data-flow mashup composition cannot represent all of a user's requirements, and still has some limitations when it comes to mobile mashups. For instance, using the data-flow paradigm, we can compose a simple location-aware mashup that displays a Point of Interest (POI), e.g. restaurants, hotels or gas stations, around the user's current location. This mashup uses the device's location from a GPS sensor as an input to a location-based Web service, such as a restaurant search by location using the Google Places API. When a user executes the mashup application, it will request the current location from the mobile device and use it as an input to invoke the Web service API. Once the result from the Web service becomes available, the locations of POIs will be displayed on a map. The process of this mashup seems perfect; however, in a real life situation, mobile users may move to a new location after the mashup execution has been completed. As a result, the current mashup result becomes invalid. Thus, the user needs to execute the mashup again to get updated POI locations. To solve the problem highlighted by this sample scenario, the mobile mashup requires another execution model that listens to a specific event (e.g. location-changed) and automatically executes the mashup when that specific event has occurred. This mechanism is a so called event-driven execution model. Another drawback of the flow-based pattern is the robustness of the execution model. In flow-based mashup execution, the mashup component is executed in sequence following the mashup logic. The execution sequence from starting point to final output cannot be interrupted; if it is, the mashup execution will fail. Unfortunately, in general mobile device usage, network loss might occur during the execution of mashup. This highlights the robustness problem of flow-based mashup composition.

This chapter has presented a mashup construction approach that enables composition of cooperative mobile mashups. The mashups created by this approach can

enhance cooperation between multiple mobile devices. We proposed a description language called C-MAIDL, which enables the definition of mashup logic and collaboration behavior. A mashup generator is implemented as a fast-paced mashup development tool to aid mashup composition for end-users. We have presented the mashup execution environment that is used to automate cooperation between devices. We have demonstrated our system's applicability for creating cooperative mobile mashups with a sample scenario.

Based on the discussion above, we found that our approach still has limitations and needs improvement. We aim to improve the reusability of the mashup components; specifically the mobile application component. Event-driven mashup composition where mashup components are executed by events is an additional important feature that our approach should fulfill.

Chapter 5

Improving Reusability of Mobile Application Components

One major goal of our mashup approach is turning general mobile applications into reusable mashup components, and integrating them with other existing resources, especially with other mobile applications. Therefore, interoperability among mobile applications is a valuable technique that benefits our mashup composition approach.

The integration of existing mobile applications is important for mashup application development on mobile devices. However, most mobile applications do not yet qualify. A major reason is the lack of an applicable way to describe shared functionalities. Many applications have published a description of their functionalities, but the descriptions are written in different formats. Furthermore, detailed information on how to invoke the offered functionalities, such as the input-output parameters and data type definitions, are not clearly explained. Therefore, developers still require adequate information on where and how to access those functions. Major mobile operating systems, such as Apple iOS and Google Android, have addressed the interoperability issues between mobile applications by providing application integration protocols (e.g., URL Scheme for iOS and Intent for Android). These protocols aim to allow invocation and message passing among applications. However, a standard way to describe the offered functionalities is still unavailable. The stated conditions have limited the interoperability among mobile applications. Hence, it becomes essential to define a structured way for describing the shared functionalities of mobile applications in order to enhance the interoperability and benefit mashup composition on mobile devices.

This chapter presents an XML-based modeling language, called LIMA (Language for Interoperability of Mobile Applications), which allows developers to describe shared functionalities of mobile applications for enhancing interoperability. LIMA benefits developers by accommodating the composition of existing functionalities in the development process. It provides building blocks for sharing functionalities of a mobile application and offers an efficient way to reuse the functionalities of other applications. LIMA provides two levels of modeling, abstraction and implementation, for separating the conceptual description of functionalities from the technical details associated with a specific mobile application integration protocol. The abstraction level describes functions in terms of their signature; functions are given details of input and output parameters including the definition of data types. At the concrete level, the identity of

the mobile application, the configuration of the integration protocol and the invocation parameters are specified.

We demonstrate our modeling language by implementing a parser tool that generates proxy classes for leveraging mobile application development. The parser tool takes a LIMA description file as input to generate a proxy class, i.e., programming code corresponding to the input description. The proxy class simplifies the development process by providing simple programming interfaces that hide the technical details of invoking the target functions. Notably, we apply LIMA with our mashup composition approach to improve integration and reusability of mobile application components. LIMA is applied to the mobile application description process of our previous work (C-MAIDL) to facilitate mobile application integration as well as to abstract platform-specific details out of the mashup description. The LIMA application shows the capability to enhance mobile application integration by providing an efficient way to deal with mobile application components.

5.1 Interoperability of Mobile Applications

The term interoperability is defined as "the ability of two or more systems or components to exchange information and to use the information that has been exchanged" [43]. In the context of mobile application development, interoperability can be seen as the ability of an application to work with other applications by exchanging data and functionalities. To enable interoperability, the applications should ensure that the data and functionalities are managed to promote the exchange and reuse of information [44]. In other words, the standard description of data formats, shared operations and communication protocols are fundamental. In the case of mobile applications, although the basic technologies seem to be in place, we have not yet achieved sufficient interoperability. This section discusses the challenges and our proposed solutions of mobile application interoperability. In addition, we present a sample scenario that realizes the problems and illustrates the contributions of our work.

5.1.1 Challenges

The interoperability of mobile applications seems to be a promising solution to improve mobile application development. However, there are several challenging issues.

i. *Lack of a proper way for describing the shared resources*. Describing the shared resources is necessary for developers to recognize the reusable functionalities. However, in recent mobile technology, an approach to describe these reusable resources is not yet proposed. In practice, developers typically create documents that describe the programming interfaces of their mobile applications. However, the documents created by various developers usually lack consistency. Therefore, the developers who want to use the shared functionalities have to put a lot of

effort into studying the different specifications for each of the reusable components. To address this challenge, we have applied techniques related to Web Service Interoperability as a baseline for our mobile applications interoperability.

Web services use a standard description language, WSDL [45], as a key to succeed in the interoperability of services. WSDL is widely used as the standard language for developers to describe the functionalities of Web services. Similarly, a standard method for describing the functionalities of mobile applications is also essential. Therefore, we have adapted the key ideas of WSDL to introduce LIMA, an XML-based modeling language for enhancing mobile application interoperability. Our proposed language is designed to provide an applicable approach to describe the shared functionalities of mobile applications. Developers can use it to publish the functionalities of their mobile applications and consume the shared resources of other mobile applications.

ii. *Standard data definitions must be defined*. Since the exchanging of information among applications is important for enabling interoperability, the data types must be standardized. There is a possibility that the same data can be differently defined among multiple mobile applications. For example, geographic coordinates, i.e. latitude and longitude, can be used as different data types among applications. One application may represent them as a single string variable with comma separator, while another application may represent it as a complex type of two string variables. To address this challenge, the data definition must be standardized for enhancing interoperability. Therefore, LIMA provides a solution by using primitive and complex data types. The primitive data type is a set of commonly used types, such as Boolean, Text, and Number. Multiple primitive data types can be combined into a complex type to represent a custom data format. For example, the geographic location can be defined as a combination of two numeric data types. Thus, using primitive and complex data types we can cover various data formats, and set standard data types for accommodating interoperability.

iii. *Different technical details of invoking the shared functionalities*. A mobile application is designed to work within their specific environment. Each mobile platform provides exclusive devices, operating systems and application integration protocols. Even within the same platform, the way to access a functionality of an application can be different. Some applications are designed to work as a service, i.e., they allow other applications to call their functions and return a result, while other applications listen for a specific event and activate themselves. To address this challenge, the modeling language must provide multiple configuration sets to support different technical details of invoking the

shared functionalities. The specific configuration sections that correspond to the application integration protocols and the parameters that specify the execution patterns must be defined. In addition, the technical configurations must be separated from the logical descriptions in order to maximize the capability of interoperability. Therefore, the modeling language can be used with any mobile platforms by changing the technical specifications to match them with the infrastructure of the target platforms.

5.1.2 A Sample Scenario

This section illustrates a common usage scenario and the benefits of mobile application interoperability. The sample scenario is about developing a new mobile application that scans a barcode of a product, searches for product information on online stores, translates the product information into a specific language and shares the translated information on Twitter and Facebook. For barcode scanning, a developer has to manually program the devices' camera or use an existing barcode scanning library. In addition, the developer has to deal with Web service APIs of online shopping sites and find a proper translation API. Moreover, the programming interfaces of Twitter and Facebook have to be studied. As a result, developing this kind of application is time-consuming and requires high-level programming skills. In fact, the functionalities that can be used for developing this kind of application are already available in the form of downloadable applications via application stores. For example, we can create this application by using the functionalities of a barcode scanning application, an online shopping application, and social applications, i.e., Twitter and Facebook. A set of sample mobile applications, which offer the required functionalities, and the possibility of interoperability are illustrated in Figure 16. The developer can build the new application by combining the existing functionalities from the set of these mobile applications.

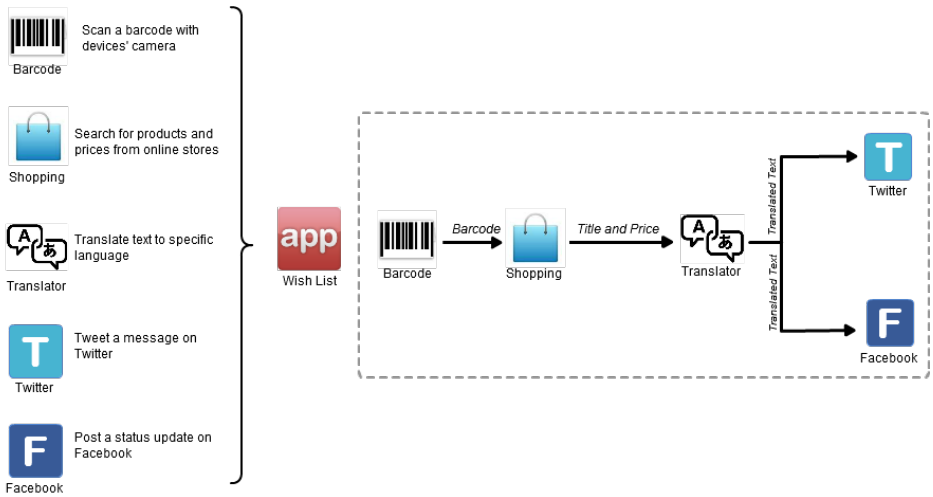


Figure 16. A Sample scenario of mobile application interoperability.

The interoperability process starts with scanning a barcode of a product by using the barcode application. Then, the scanned code is used as an input for the shopping application to search for the product information from online stores. The selected product name and price are translated into a specific language by using the translator application. Finally, the translated information can be published to a social network through the Twitter and Facebook applications. Therefore, developing mobile applications by using interoperability requires less technical knowledge and programming effort than that of a manual development approach.

This scenario has presented the usages and benefits of the interoperability of mobile applications. Instead of manually developing the application, reusing existing functionalities seems to be an efficient approach. However, to achieve interoperability, each of the mobile applications has to expose their functionalities and provide a way to access those functions. Therefore, the contribution of our work is to enhance mobile application interoperability by providing an efficient way to publish and reuse existing functionalities.

5.2 LIMA

In this section, we present our modeling language in three subsections: *Overview*, *Specification,* and *Example*. The overview illustrates the conceptual model of LIMA and explains the relations among its components. The specification presents elements and conventions used for encoding a LIMA description. The example shows a sample of description files and applications of LIMA for different mobile platforms.

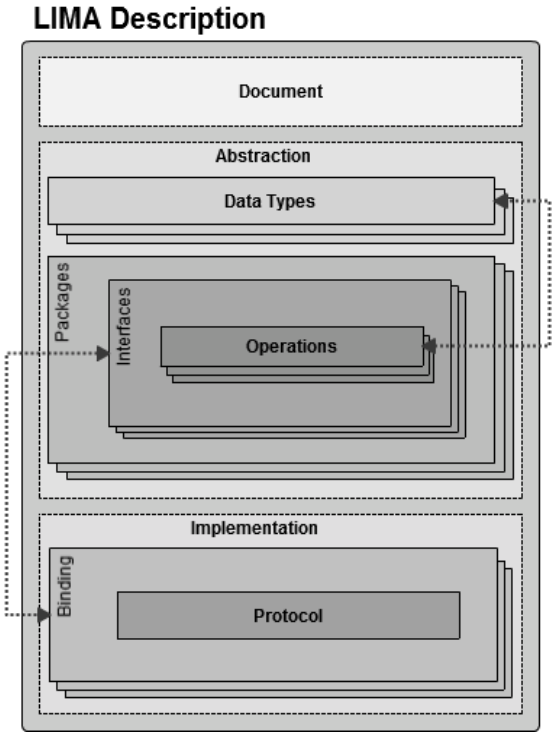


Figure 17. The conceptual model of LIMA.

5.2.1 Overview

LIMA (Language for Interoperability of Mobile Applications) is an XML-based modeling language that provides a way to describe shared functionalities of mobile applications. A major design goal of LIMA is to encapsulate the execution configuration of shared functionality with a simple programming interface that contains only minimal technical aspects. Functionality of a mobile application can be modeled on two levels, abstraction and implementation. The abstraction level describes functions in terms of their signature, i.e. function name, data types and order of parameters. The implementation section describes the invocation information, such as identity of mobile applications, configurations of integration protocols and details of input/output parameters. The conceptual model of LIMA is shown in Figure 17.

The conceptual model of LIMA consists of three major components: *Document*, *Abstraction* and *Implementation*. The document component provides a space for describing target mobile applications in a human-understandable format. The abstraction component consists of four inner components, which are *Data Type*, *Packages, Interface,* and *Operations.* These are used for describing data types, namespaces, abstract interfaces, and logical operations respectively. An operation refers to pre-defined data types for defining a logical operation. Within the implementation component, the *Binding* component is used for linking the execution configuration of a particular functionality with an abstract interface in the abstraction component. The execution configuration in a binding component contains an identity of the target mobile application and a *Protocol* component that is used for specifying details of the application integration protocol.

5.2.2 Specification

LIMA provides a set of XML elements and their associated properties for describing offered functionalities of mobile applications. The following section gives a brief overview of elements and conventions that are used for encoding a LIMA description. Figure 18 shows the structure of a LIMA description.

Figure 18. The structure of the LIMA description.

A LIMA description contains two types of elements: *Scope* and *Definition*. The scope element groups related definition elements together to facilitate the information extraction process. For instance, an XML parser tool can simply extract the definition elements, which contain invocation information, by detecting the "implementation" element. The scope elements consist of the following elements:

- *<lima>*
  This element must be the root element of all LIMA documents. It indicates that the content inside this element is a LIMA description.
- *<document>*
  This element is used to provide human-readable documentation.
- *<abstraction>*
  This element is used to indicate that the content within this element is the abstract description of the target mobile application.
- *<implementation>*
  This element indicates that content within this element describes the concrete details on how to execute the target functionalities.

The definition elements are used to describe details of functionalities in different aspects. For example, a "datatype" element is used for describing a custom data format and a "protocol" element is used for specifying invocation information. In addition, each definition element contains attributes and child elements that are used for specifying related information. Details of functionalities can be defined by using the following definition elements:

- *<datatype>*
  This element is used to define a data type that will be used in interface and operation declarations. The defined data types can be shared among the operations by referring to the data type name. To standardize the data types used

in LIMA, we define primitive data types, which are a set of data types that are commonly used in mobile applications. Table 6 shows primitive datatypes of LIMA.

Data Type Name	Description
bool	A boolean true or false value
number	A number value
text	A string value
byte	An array of binary value represents binary data such as images or sound

Table 6. Primitive data type of LIMA.

The primitive data types are used for defining the input and output parameters of the operation description. However, the primitive data types cannot cover all the data usage in mobile applications. To address this problem, the data type element allows us to define complex data types by combining the existing primitive types. For example, geographic coordinate data that is commonly used in map applications, i.e. latitude and longitude, can be represented as a combination of two numbers. An example of a complex type definition is shown in Figure 19. The description of an operation can refer to a complex type by specifying the complex type name in its "type" attribute.

```
<datatype name="Location" type="complex">
    <element name="latitude" type="number" />
    <element name="longitude" type="number" />
</datatype>
```

Figure 19. An example of a location data definition

- *<package>*
  A package element groups a set of interfaces together in order to make them organized, and prevent naming conflicts. A related set of functionalities can be bundled into a same logical package. Furthermore, in the case of a complex description, it is possible that naming conflicts, i.e. one or more interfaces sharing the same name, will occur. The package element helps to avoid naming conflicts by using the "name" attribute to create a new namespace for the containing interfaces.
- *<interface>*
  An interface element is used to create a logical set of functionalities by grouping related operation elements together. It can promote the usability of our modeling language by providing a flexible way to manage set of functionalities. For

example, the translation features from multiple applications can be grouped into an interface element. In this way, developers can access several translation functions, which are actually offered by different mobile applications, transparently through a single interface description.

- *<operation>*
  An operation element describes a shared functionality offered by a mobile application. The operation name, input parameters and return data type are specified. In other words, an operation element is similar to a method declaration in a programming language. The mapping between LIMA elements and Java programming code is shown in Figure 20.

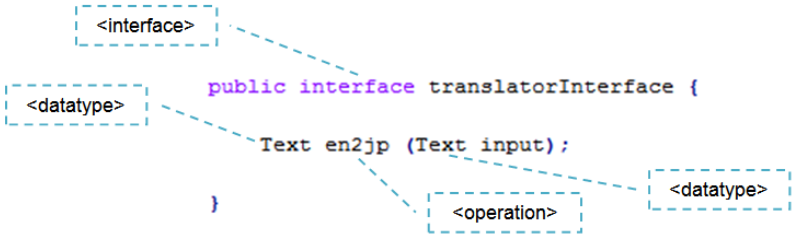


Figure 20. The mapping between LIMA elements and Java programming code

- *<binding>*
  A binding element does the mapping between concrete details of execution and a logical operation in the abstraction section.

- *<protocol>*
  A protocol element is used for specifying required information for invoking a function. This element and its child elements describe configurations of a mobile application integration protocol. For example, a protocol element of an Android application contains child elements that represent configurations of the Intent protocol. Furthermore, the protocol element and its child elements enable loose coupling between operation descriptions and execution details. We can set different execution configurations for each operation element. Given the flexibility of the protocol element, we can describe various application integration protocols used in additional mobile platforms.

5.2.3 An example of LIMA

This section shows samples of LIMA descriptions. The sample descriptions describe functionalities of two mobile applications, from different mobile platforms. The purpose of the samples is to illustrate the basic syntax and the conventions for encoding LIMA, and show that LIMA is flexible enough to be used for different mobile platforms.

The first mobile application is an Android application called “Xzing Barcode Scanner” [46]. The barcode scanner application uses the device’s cameras for scanning barcodes of products, and looks up prices and reviews. The capability to scan barcodes

from cameras and return them as a number is valuable. We can reuse this functionality to extract product numbers and use them as input to additional application logic. The programming interface and related information of this mobile application can be found on the developers' website [47]. Required information for reusing the scan functionality is listed in following table.

Data	Description
Action	com.google.zxing.client.android.SCAN
Intent Extra : Scan Mode	PRODUCT_MODE: Decode only UPC and EAN barcodes. This is the right choice for shopping applications that get prices and reviews for products. ONE_D_MODE: Decode only 1D barcodes (currently UPC, EAN, Code 39, and Code 128). QR_CODE_MODE: Decode only QR codes.
Return	SCAN_RESULT: Return the barcode content.

With the available information, we can compose a LIMA description that describes scanning functionality of the barcode scanner application as is shown in Table. In this description, a "document" element contains overview information of the target functionality. The scan functionality is defined by using an "operation" element. A "pattern" attribute of the "operation" element is set to "InOut" to indicate the execution pattern. The corresponding elements, i.e. "input" and "output", define input and output parameters of the scan functionality. The data type of each parameter is defined within the "datatype" attribute. In the implementation section, configurations of the Intent protocol, such as Intent Filter and Intent Extras are specified. These execution configurations are linked to the scan operation by using a "ref" attribute of a "binding" element. Furthermore, this sample description demonstrates how to define a complex data type. The complex data type, i.e. ScanMode, represents four available scan modes that are used as input parameters of the scan function. Each scan mode is associated with a set of supported barcode formats by defining an initial value within a "value" attribute. The scan operation refers to the supported scan modes by referring the complex type name in its input parameter description.

```
<lima>
<document>
This is LIMA sample file to describe zxing barcode scanning application.
</document>
<!-- Abstraction Level -->
<abstraction>
    <datatype name="ScanMode" type="complex">
        <element name="Product" type="text" value="UPC_A,UPC_E,EAN_8,EAN_13"/>
        <element name="1DCode" type="text" value="UPC_A,UPC_E,EAN_8,EAN_13"/>
        <element name="QRCode" type="text" value="QR_CODE"/>
        <element name="All" type="text" value=""/>
    </datatype>
    <package name="com.google.zxing.client.android">
        <interface name="BarcodeScanner">
```

```
            <operation name="Scan" pattern="InOut">
                <input name="mode" datatype="ScanMode"/>
                <output name="scanResult" datatype="text/plain"/>
            </operation>
        </interface>
    </package>
</abstraction>
<!-- Implementation Level -->
<implementation>
    <binding endpoint="com.google.zxing.client.android" ref="BarcodeScanner">
        <protocol name="intent" type="explicit">
            <activity name="com.google.zxing.client.android.SCAN"
                                        execution="forResult" ref="Scan">
                <param name="SCAN_MODE" datatype="text/plain" ref="mode"/>
                <return name="SCAN_RESULT" datatype="text/plain" ref="scanResult"/>
            </activity>
        </protocol>
    </binding>
</implementation>
</lima>
```

The second mobile application is an iOS version of the first barcode scanner application [48]. This application offers the same functionality as the Android version; however, the application integration protocol and configurations are different. Since the logical functionality of the two sample applications is identical, we can compose a LIMA description for the second application by reusing the abstraction section of the first description. Therefore, the abstraction section will remain the same, while the implementation section is modified to match them with the application integration protocol of iOS. The implementation section for the second sample description is shown in Figure.

```
<!-- Implementation Level -->
<implementation>
    <binding endpoint="zxing" ref="BarcodeScanner">
        <protocol name="urlscheme" type="implicit">
                 <urlscheme name="zxing://scan/callback=[self]/[mode]"
                                              execution="forResult" ref="Scan">
                      <parameter name="mode" datatype="text/plain" ref="mode"/>
                      <callback target="self"/>
                 </urlscheme>
        </protocol>
    </binding>
</implementation>
```

5.3 Implementation

5.3.1 LIMA Parser Tool

In order to demonstrate the applicability of LIMA, we developed a parser tool that converts LIMA descriptions into programming code. The parser tool takes a LIMA description file as input, and then generates a proxy class file as output. The proxy class

helps simplify the mobile application development process by hiding complicated invocation details of the target functionalities. Accordingly, developers can reuse shared functionalities of a mobile application as if calling an ordinary function. The code generation process of LIMA parser is illustrated in Figure 21.

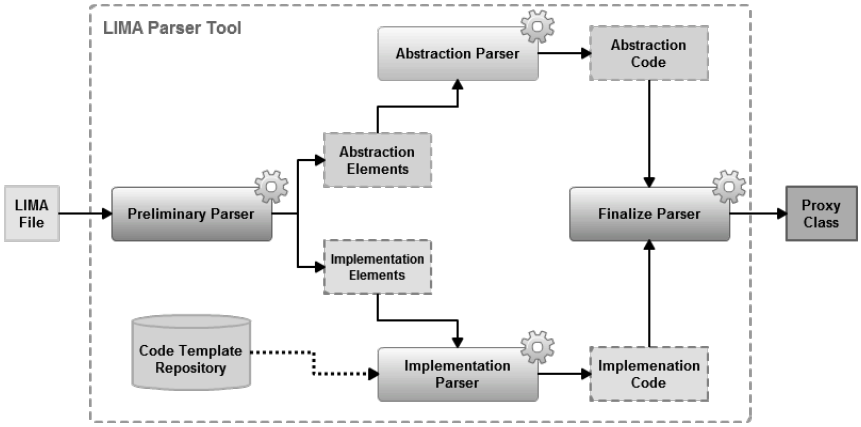


Figure 21. LIMA Parser Code Generation Process

The LIMA Parser tool consists of four components: *Preliminary Parser*, *Abstraction Parser*, *Implementation Parser* and *Finalize Parser*. Each parser component is responsible for a specific task.

- *Preliminary Parser* reads a LIMA description file and separates the description into two parts: *Abstraction Elements* and *Implementation Elements*. The abstraction elements contain abstract specifications of shared functionalities while the implementation elements contain technical details of the invocation. In addition, the preliminary parser validates the syntax of LIMA and provides feedback to users when an error is detected.
- *Abstraction Parser* takes the abstraction elements as input to generate Abstraction Code, i.e. a simple programming structure corresponding to abstract functionalities described in the description file.
- *Implementation Parser* uses the implementation elements to generate programming code, called Implementation Code, which contains concrete details of invoking the target functionalities. Code templates from the Code Template Repository are used to generate output code corresponding to the specified protocols. In addition, using code templates allows the parser tool to create implementation code that supports multiple mobile platforms.
- *Finalize Parser* combines the abstraction code with the implementation code to build a complete proxy class that can be used as a utility to simplify the mobile application development process.

For the current implementation, we built the LIMA Parser Tool for Android platforms. The parser tool generates proxy classes by using the Java programming language. The generated code contains Java packages, classes and operations corresponding to the input description. The parser tool is implemented as an Eclipse Plug-In to provide better integration with the existing Android development environment. Developers can use the plug-in to create proxy classes within Eclipse's Android Development Project. Selecting a LIMA description file, and then choosing

“Generate LIMA Proxy Class” from the context menu can generate a proxy class. A screenshot of using the LIMA Parser Plug-in is shown in Figure 22.

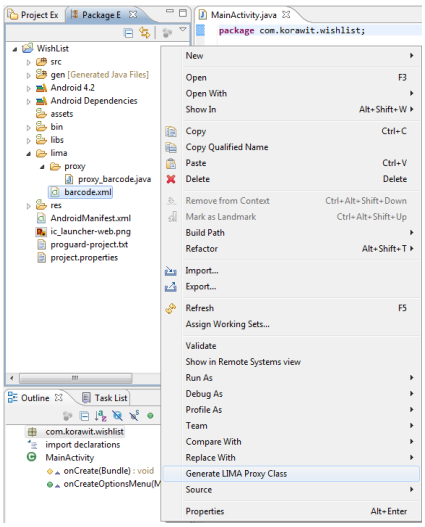


Figure 22. Screenshot of the LIMA Parser Plug-in

5.3.2 Applicability of LIMA in Mobile Mashup Composition

To illustrate the applicability of LIMA for mobile mashups, we apply LIMA to our data-flow composition methods. Our previous work is a description-based mobile mashup approach focused on integration of mobile applications, Web services, and Web applications. We proposed a mashup description language called C-MAIDL, which is designed for describing configurations of mashup components and composition logic of mashup applications. C-MAIDL is used as input to a mashup generator for creating mashup applications. Among the supporting mashup components of C-MAIDL, mobile applications are used as important components to enable integration of mobile devices’ capabilities with Web applications and Web services. To allow the use of a mobile application as a mashup component, our previous work provides a description section for describing execution configurations of selected mobile applications. Since a flexible approach to describe shared functionalities of mobile applications is not yet proposed, our mashup description language uses a configuration method related to a platform-specific protocol, i.e. Android Intent, to model the mobile application components. Embedding concrete technical details in a user-friendly description language can be considered as a demerit. It makes our description language more complicated and reduces the possibility to apply it to other mobile platforms. Therefore, in our previous work, using a mobile application as a mashup component still has limitations due to the lack of a practical way to describe shared functionalities of mobile applications.

LIMA enhances our previous approach by providing an efficient way to describe shared functionalities, separating technical details of invocation from the mashup description language. We modify a description section that is used for describing mobile application components by using a LIMA description. The platform-specific details are removed from the mashup description language, and replaced with an abstracted LIMA description. The mashup generator tool can extract abstract and

concrete information of a mobile mashup component to enable automatic application integration.

Figure 23 is a comparison between the previous description and the new description. The former description describes a mobile application by using configurations of the Intent protocol. In contrast, the new description specifies abstraction level description and location of the full LIMA description. The abstract information about offered functionalities will be used in the process of mashup component integration, while the concrete details of invocation will be used at a runtime of mashup application. The mashup generator tool can uses the URI that locates a full description to obtain technical details of invoking, and then use them in the mobile application generation process. This method enables loose coupling between the mashup description and mobile application components. Since the execution configurations of those components are separated, the mashup application will not be affected when the configurations are changed.

C-MAIDL

```
<intent>
    <intent-name>com.prach.GPSLocator</intent-name>
    <intent-extra>
        <extra-name>MODE</extra-name>
        <extras>
            <extra>PASSIVE</extra>
        </extras>
        <extra-name>TYPE</extra-name>
        <extras>
            <extra>null</extra>
        </extras>
    </intent-extra>
</intent>
```

C-MAIDL with LIMA

```
<lima>http://www.tt.cs.titech.ac.jp/~korawit/lima/gpslocator.xml</lima>
<package name="com.prach.GPSLocator">
    <interface name="Location">
        <operation name="Get" pattern="Out">
            <output name="Latitude" datatype="location"/>
        </operation>
    </interface>
</package>
```

Figure 23. Comparison between the former C-MAIDL and new C-MAIDL with LIMA

5.3.3 Discussion

With our modeling language, the interoperability among mobile applications seems to be a promising solution to accommodate mobile mashup application development. However, our language still has some limitations. This section discusses limitations of our work and states related work that must be considered.

LIMA modeling language is designed to support the data interoperability. The target functionality must work in form of a service-oriented function, i.e., software components that can be reused for different purposes. In other words, LIMA cannot deal with functionalities that require human interaction. Second, there are only a limited number of published functionalities. The reason is the different points of view on sharing functionalities. For example, some developers have no intention to share functionalities of their applications due to commercial issues. To address this problem,

a mechanism to gain profit from sharing functionalities is required. According to the concept of Software as a Service [49], software providers benefit from selling parts of software as a service. Similarly, in the context of mobile application, once mobile operating systems provide a mechanism that allows developers to benefit from sharing functionalities, the number of shared functionalities will increase. Finally, LIMA should be promoted. Since our modeling language introduces a new paradigm for mobile application development, we have to encourage developers to use our language as a standard approach for modeling a mobile application.

Chapter 6

Event-Driven Composition for Multiple Device

Mobile devices, e.g. smart phones and tablets, are context-aware devices. The devices keep track of context information by monitoring sensors and report the changes to users. Users are informed about events such as a change of location, receipt of mail, and low battery with notifications. These notifications can be viewed as events that can be used as triggers to develop applications in an event-driven manner. Event-driven mobile applications are becoming popular. There are many applications that take advantage of events in mobile devices. They help users executing pre-defined tasks when a specific event occurs. For example, a popular commercial application for Android called "Taskers" [50] listens to changes in time, location, or hardware/software state, and can perform about 200 actions ranging from launching an app to automatically making a phone call when its listening events are fired. While many applications can use the event notification mechanism of mobile devices to expand coverage of the user's requirements, mobile mashups still cannot take advantage of these features as much as they should; especially considering the limitation of flow-based mobile mashups that we have discussed before. As such, we aim to explore how to use the lightweight integration concept of mobile mashups to develop event-driven mobile applications.

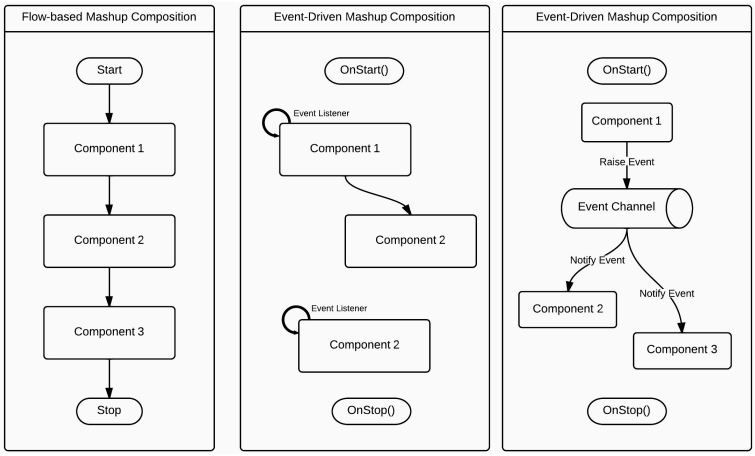


Figure 24. Data-flow and even-driven component integration.

Event-driven mobile mashups are based on event-driven programming. This can be defined as an architectural style in which one or more components are executed in response to receiving event notifications [51]. Thus, we can define event-driven mashup composition as a mashup composition style in which one or more components in the application logic executes in response to receiving one or more event notifications.

Figure 24 shows the characteristics of event-driven mashup composition compared to conventional data-flow (flow-based) composition. Event-driven mobile mashup applications work as a set of individual components that monitor changes of their internal state and notify other components as events, as well as listen to events produced by other components. When an event notification occurs, the components that are listening for this event invoke a function or service of other components. Thus, we can compose typical event-driven applications by mapping events to existing functionalities.

In this chapter, we propose a mashup approach that offers a full treatment of event-driven mashup composition on mobile devices. Our approach aims to enable the integration of mobile applications and Web services as well as to utilize data cooperation among multiple mobile devices. We set up sample scenarios that illustrate the different aspects of event-driven applications for mobile devices. From these scenarios we realize suitable components and a composition model for event-driven mashup composition. Our approach uses description languages and generator tools to leverage development efforts and reduce the required programming skills. To evaluate the applicability of our approach, we implement our first prototype on the Android mobile system.

6.1 Characteristics of Event-Driven Mashups

6.1.1 Motivating Scenarios

In this section, we present three scenarios that illustrate the benefits of event-driven mobile applications. These scenarios represent different aspects that reflect unique problems and usages of event-driven mobile applications. The first scenario is an "event-trigger mashup". This type is a typical mashup application that is similar to the common problem that most existing approaches aim to solve. It starts out by listening for an event and performs additional mashup logic when the event is fired. The second scenario is a "data streaming mashup". This mashup deals with handling the stream of data that is continuously produced from events, and automatically executes additional logic to update mashup results. Finally, for scenario "event-driven mashup for multiple devices", we study how to use the event-driven composition model for cooperation of multiple devices. Given these scenarios, we derive requirements needed to achieve event-driven mashup composition and discover challenges that need to be solved.

1. Email Translator Scenario. 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 logic. This scenario simulates a requirement for translating 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. Unfortunately, in

general, email client applications do not provide translation functions. 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 application that helps translating email from Japanese to English is needed. In addition, the application can benefit from an event-driven architecture by listening to new incoming emails and automatically translating the title and content into a specific language. It can then save the translated content in 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.

2. Around Me Scenario. The second scenario addresses a data streaming mashup in which the system is continuously notified of events as a stream of data. In this scenario, we reuse the common scenario of a location-aware mashup that continuously monitors the "location-changed" event and integrates location with a Web service to display a POI on a map [52]. This scenario simulates a situation where a tourist is travelling in Japan. While he is travelling in a city, he 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 application should automatically update the result when the "location-changed" event is fired.

3. Barcode Book Review Scenario. Our 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 our previous work [23]. The scenario simulates a mobile mashup application that helps users find information about selected books from online services by using barcode scanning. 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.

4. Library Assistant Scenario. This scenario focuses on mashup composition for cooperation of mobile devices. The scenario simulates a situation in a library where staff wants to create a list of all available books in a library room. The information that is required in the list are, among others, title, author, price and publisher. If done manually, the staff would have to take each book from the shelf to get a title or ISBN, and use that information to search for additional information from an online bookstore, and record the information to a list. Multiple staff could work together to speed up this task, but this kind of manual operation requires huge effort. As a result, the staff needs a

mobile application that allows a device's camera to be used as a barcode scanner to get the ISBN from the books. The scanned code can then be used to obtain the required information from online bookstores. Finally, the book information is combined into a list. It is important to note that staff should be able to work together by using multiple mobile devices. The information that is scanned from each device must be combined to create a shared list of available books.

6.1.2 Analysis

Based on the scenarios in Section 2, we derive objectives that our approach should fulfill 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 2) and a "photo-taken" (in Scenario 3) event, are used as triggers for the mashup process. In addition, mobile applications act as event producers. They make changes in their states known by sending notifications. Mobile applications, such as email clients (in Scenario 1), social networks, etc., inform users and other applications about their state through events. 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. However, the major challenge of using an event-driven architecture in mashup composition is that we have to deal with events that are produced from different sources, already discussed in Section 3.2.

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 1 and 3 use the translation Web services for translating Japanese to English and English to Japanese. In practice, the general configurations of the translation Web service in both mashup applications are identical. However, the source and target language parameters that are passed to the translation Web service are changed. Thus, to enhance reusability of mashup components, mashup components used in our approach should be flexible enough to allow for a change in execution parameters for different mashup compositions.

Similar to other existing mashup platforms, our approach aims at reusing mashup components in multiple compositions. Our proposed techniques use a component descriptor to save and maintain changes in component configurations at design-time. However, we notice from sample scenarios that providing runtime configurations for each of the components makes the components more flexible, and reduces the required user effort on component development. To enable this feature, the mashup components should not only provide the common interfaces for integration but also a user interface

for managing component configurations at run-time. The boundary between technical configurations and the execution parameters must be carefully considered.

Support an event-driven execution model. In a conventional data-flow mashup, users start the mashup application and wait for the result. However, our sample scenarios require a more efficient execution model. In Scenario 2, users use the mashup application and move to other locations. The mashup application should handle this execution automatically. In addition, users should be able to switch to others applications and use their mobile devices as usual. The mashup applications should automatically activate themselves when new email is received. Similarly, robustness of scenario 3 (data-flow composition with synchronous execution) should be improved. According to Scenario 4, when a participating device has finished scanning a barcode, it should notify other devices to trigger additional mashup processes and add the new information to the list. To address these problems, our approach should support event-driven mashup execution, in which the components are executed asynchronously in response to specific events.

Since we design our mashup composition output for mobile applications, an execution model should be considered that optimizes performance on the target device. Event driven execution is different from data-flow in terms of the execution model. Data-flow mashups execute components following the composition logic to produce a result, and then terminate the process. In contrast, event-driven mashups keep listening for events, holding up the mashup execution, and stopping 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. In addition, to improve the robustness of mashup applications, component interaction should be done in an asynchronous manner. The data and logical components, such as Web service components, should contain both functionality and events that notify others of their execution state.

Simplify the mashup development. We targeted all sample scenarios to users who do not have skills in programming. Therefore, we assume that our targeted users cannot deal with device-specific configurations or the mashup composition process that requires manual programming. The technical knowledge on developing mashup components, as well as the composition of multiple components should be leveraged.

Since recent research shows that end-users cannot deal with a configuration process that requires programming skills [23], our approach has to provide tools that simplify the configuration and composition process. However, when we increase simplicity, expressive capability will be limited. Therefore, another challenge is to strike a balance between simplicity of configuration and capabilities of the approach.

6.2 Event-Driven Mashup Composition

To achieve our objectives and address the problems that we have stated in the previous section, we proposed an approach that allows for the composition of event-driven mashup applications for multiple devices. We handle the integration of mobile applications and Web services in event-driven manner. The general concept of our approach is the orchestration of mashup components, which work as a proxy for actual mashup resources and provide a uniform event-driven 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 is simplified by an event bus system adapted from the state of the art in event-driven architecture [18]. To utilize data cooperation among multiple devices, we apply the concept of using a mobile Web server and Web services from our previous approach. A mashup composer can use the XML description language to serialize configurations of mashup components and record the composition logic. The description language is used as an input to the code generator tool to generate the mashup components and the composite application. The final output is 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 overall architecture of our approach. We also highlight the key ideas and techniques that we have applied to deal with challenges in enabling event-driven mashups on mobile devices.

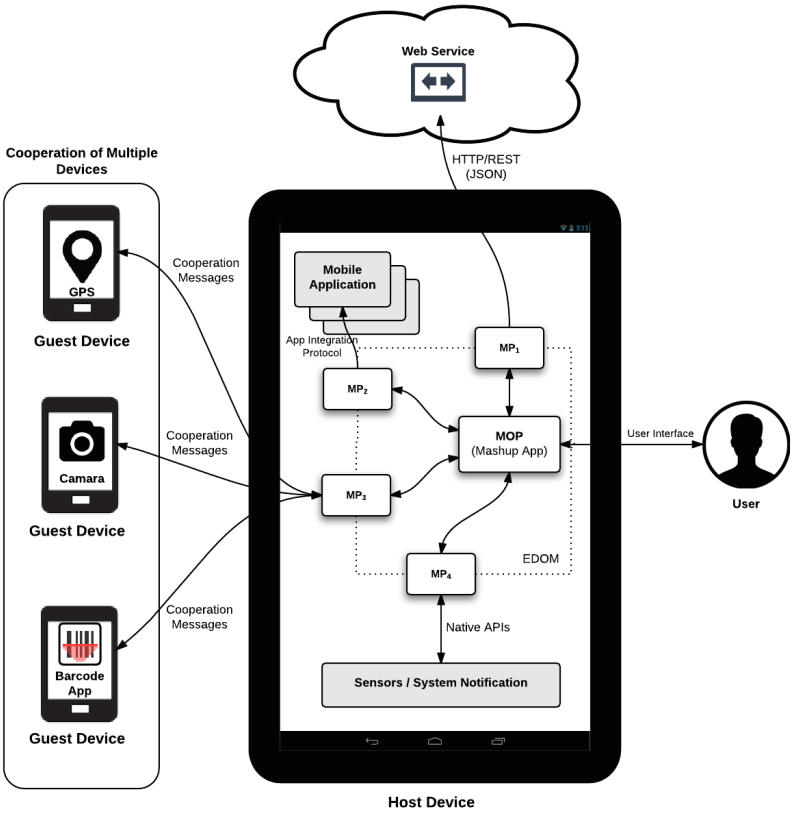


Figure 25. Overview architecture of event-driven mashup composition.

6.2.1 Overview

An overview of the architecture we use in our approach is shown in Figure 25. There are four types of mashup resources (Web services, mobile applications, sensors/system notification and data cooperation) that can be combined to create mashup applications. The Mashup Orchestration Process (MOP) is responsible for controlling the execution of mashup logic and managing the orchestration of mashup components. It provides a user interface that a mashup user can use to start/stop the mashup application, and view the mashup result. A Mashup Proxy (MP) is used to facilitate component integration. An MP acts as an intermediary between the MOP and the actual mashup components by transforming a technology-specific programming interface into a common integration protocol. They also augment the mandatory event-driven mechanism to 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 triggered events to the MOP. The MOP 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. The target MP then translates the message into the proper programming interface, regarding the specification of the actual component, and invokes the target functionality for a result. Finally, the result from the invocation is transformed to a standard format and sent back to the MOP. In this way, the mashup process is completed by cooperation between an MOP and MPs.

To enable cooperation between devices, the MOP can request data from a guest device by subscribing to the events of a cooperation MP, designed for cooperation tasks. The cooperation MP interacts with the cooperation center to collect data from all guest devices. Once the data from a guest device is received, the cooperation MP will send it to the MOP as an event with parameters. In this way, the MOP is totally separated from the cooperation task. Replacement of the cooperation center to support different connectivity architectures will not affect the mashup process.

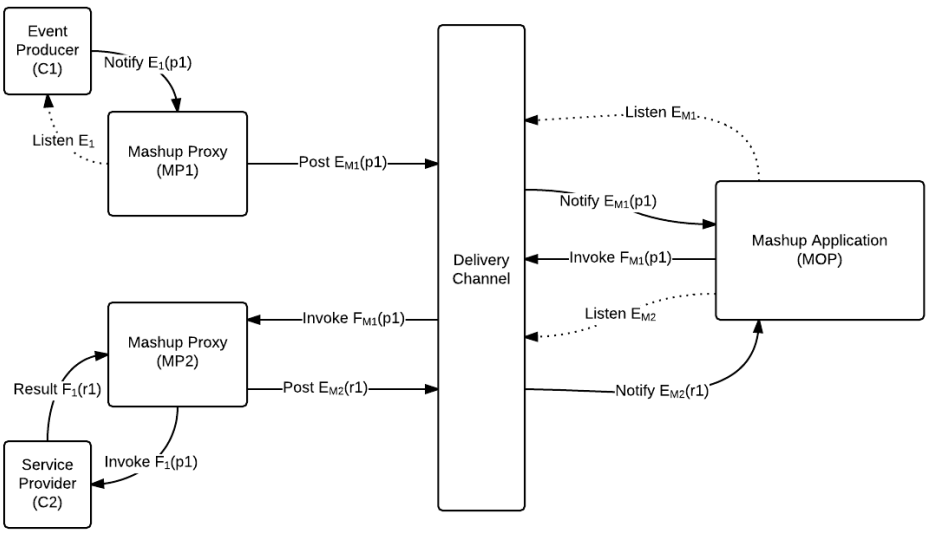


Figure 26. Event-driven mashup component integration

6.2.2 Even-Driven Mashup Component Integration

This section presents the cooperation 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 provide 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 26.

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.

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: for example, system events or notification messages from installed mobile applications.

Service Provider (C2). This component represents a mashup component that provides functionality that other mashup components can invoke for a result. For example, Web services or mobile applications which provide programming interfaces.

Mashup Proxy (MP1, MP2). This component works as intermediary 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 events to facilitate the cooperation between MOP and MPs.

The model describes the minimum building blocks of event-driven mashup component integration; that is, the invocation of a mashup component from an event. The goal is to use an event E_1 of event source C1 as a trigger to invoke function F_1 of service provider C2. In order to get a result from F_1, the mashup application has to extract parameters from event E_1 and use them as input parameters for invoking F_1.

i. MP1 is used as an intermediary component between C1 and MOP. It listens to an actual event $E_1(p1)$, E_1 with parameter p1, and transforms that technology-specific event into an event message $E_{M1}(p1)$, E_{M1} with parameter p1.
ii. MP1 posts the transformed event onto the delivery channel to notify the MOP. Instead of listening for notifications from actual components, the MOP listens for E_{M1} event messages via the delivery channel. Once the MOP receives the event notification, it extracts parameter p1 and uses it to invoke function F_1.

iii. The MOP sends an invocation message $F_{M1}(p1)$ through the delivery channel and then listens for a callback event E_{M2} to process the result.

iv. MP2 receives the invocation message, transforms the message into the proper programming interface, and forwards the call and parameters $F_1(p1)$ to actual component C2.

v. After C2 has finished the execution, it informs MP2 as a callback F_1 with a result r1. MP2 then transforms F1(r1) into an event message $E_{M2}(r1)$ and posts the message onto the delivery channel.

vi. Finally, the MOP receives the $E_{M2}(r1)$ notification, extracts r1 and uses it for the rest of the mashup process.

6.2.3 Mashup Proxy

The mashup component 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 other mashup components to an actual mashup resource as well as passing the event notifications from an actual 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 conceptual model of a mashup proxy illustrated in Figure 27.

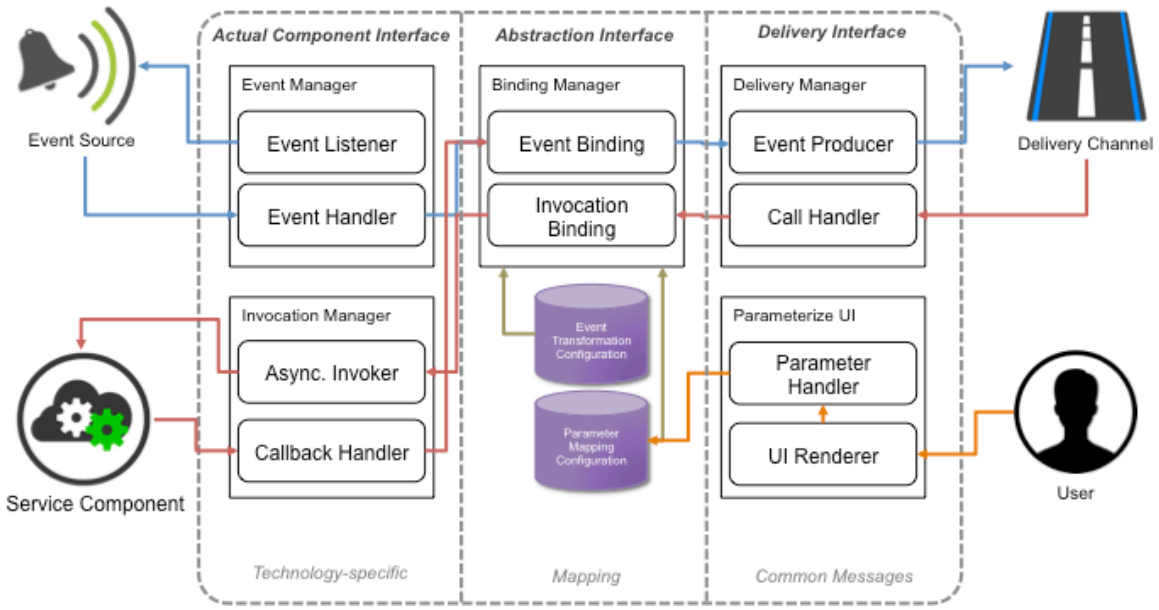


Figure 27. Architecture of Mashup Proxy.

To fulfill the requirements of an MP, the design contains 4 groups of components: Actual Component Interface, Abstraction Interface, Delivery Interface and Parameterize UI.

Actual Component Interface. This group of components is connected to the Event Source and Service Component. They handle the heterogeneous events and invocation protocols by supplying corresponding technical configurations. An Actual Component Interface contains two major components: an Event Manager and an Invocation Manager. An Event Manager is responsible for listening to events that are produced from actual mashup components. It uses an Event Listener module to register itself to a specific event channel and handles listening events by using an Event Handler module. To support multiple kinds of event channels and notification protocols, both the Event Listener and Handler are configurable during MP development. On the other hand, an Invocation Manager is used to deal with functional invocation of service components, e.g. REST Web services, mobile applications and system functions. An Asynchronous Invoker module creates invocation commands and submits them to the actual mashup components. A Callback Manager module receives the returned result and extracts the required information. Similarly, an Asynchronous Invoker and a Callback Handler are also configurable during MP development to deal with various the types of invocation protocols.

Abstraction Interface. This component group is used for defining abstract events and functions of an MP, and links them with actual events and functions from an Actual Component Interface. Our architecture allows one MP to bind its events and function to multiple mashup components. Within a Binding component, the Event Binding module takes care of the mapping between abstract and actual events while the Invocation Binding module is responsible for function mapping. There are two additional components that facilitate the binding process: Event Transformation and Parameter Mapping.

We found that raw events (events directly produced from a mashup component) are not efficient for event-driven mashup composition. For example, in a location-aware mashup, the “location-changed” event is used as the trigger for executing additional mashup logic. In practice, a GPS sensor produces the “location-changed” notifications even if the device does not move to new location, or has moved only within close distance. This situation leads to a performance problem because the additional mashup logic is frequently executed even though the mashup result does not require updating. As a result, we designed an Event Transformation to help transforming raw events into more usable events by defining conditional filters and corresponding actions. With Event Transformation, we can define a filter that ignores small changes in the location, e.g. changes within 100 meters will not affect the mashup result.

In our mashup component integration, each event and function contains input and output parameters. To link them together, the parameters have to be converted from a technology-specific style into an abstract style and vice versa. We use a Parameter Mapping module to facilitate this task by providing mapping rules. The mapping rules are used to specify how a parameter should be mapped from a source to its destination. Our architecture applies this mapping rule technique using a novel mashup component description language (MCDL) [24], which includes 3 rules: direct, template and parse.

Delivery Interface. Components in this group take care of delivering events and function calls to the Delivery Channel. An Event Producer component reads the binding configuration from an Event Binding and posts the abstract event onto the Delivery Channel. A Call Handler listens to the Delivery Channel for a function call. When a function call is received, the Call Handler reads the binding configuration and passes the abstract function call to an Asynchronous Invoker.

Parameterize UI. An additional role of a mashup proxy is to manage runtime configurations of the corresponding mashup component to enhance reusability. In practice, a mashup proxy deals with a specific invocation protocol and parameters by receiving and passing them among mashup components. 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 data extraction rule that remains the same in all invocations. On the other hand, a mashup user may change certain parameters, such as geographic coordinates and units, in every execution. For this reason, we have designed the mashup proxy to be able to provide a user interface for managing invocation parameters of actual mashup components. Component developers can use our description language to select which parameters to publish as runtime parameters and define a specific type of user interface to handle the selected parameters.

6.2.4 Mashup Application Development Process

In general, mashup development consists of two major processes: component development and mashup composition. The component development process is describing functionality and properties of a component that will be used in the composition, while the composition process is defining how components interact with each other. However, we have learned from current research that developing a high quality mashup component requires high technical knowledge that end-users cannot accomplish by themselves [53]. For example, developing a Web service component may require a good understanding of the REST protocol and the JSON data format in order to extract information from a Web service specification. Therefore, our approach divides the mashup development process into 3 stages, which are 1) Component development, 2) Mashup Composition and 3) Mashup Execution. Each state is designed for different roles and levels of technical skill in order to optimize the efficiency of the overall development process. An overview of our approach is shown in Figure 28.

1. **Component Development.** The goal of this process is transforming actual mashup resources into a reusable mashup component. To develop a component, the component developer composes a component description file, which contains configurations of functionalities and events. This file is then used as an input for the component builder tool to generate the output mashup component. Finally, the generated component is published to the component repository. This process requires computer skills, such as an understanding of Web service APIs, mobile

application integration protocols, XML or JSON, to describe the functionalities and events that are available in each component. Thus, we recommend that this process should be performed by an IT specialist or hobby programmer who understands the stated technology.

2. **Mashup Composition.** This is the process of building a new mashup application using existing mashup components. The process starts with a mashup composer, which describes composition logic of mashup applications and saves it as a mashup composition file. During the composition process, 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 a mashup application. The output application is stored in mashup application repository. This process requires less technical knowledge than composition development. Since the component description in this stage is abstracted to be technologically independent, the mashup composer can be a user who understands how to compose an XML description.

3. **Mashup Execution.** In the mashup execution state, the target mashup application is installed to a target device. The execution engine then validates the required components of the installed application. In case a required mashup component is not installed, the execution engine will ask the mashup user to confirm the installation of additional mashup components. Once all validation is complete, the mashup application can be run as an ordinary mobile application.

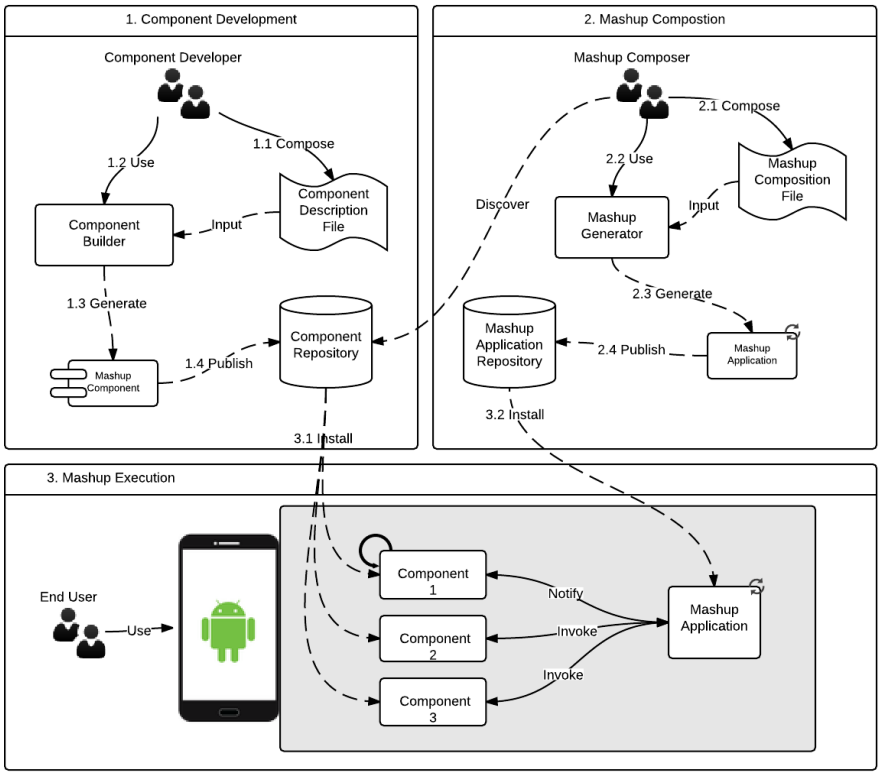


Figure 28. Mashup Composition Process.

6.3 Description Languages

In our approach, we separate the mashup development process into two major tasks: component development and mashup composition. For the component development process, we design a description language called XLIMA (eXtended Language for Interoperability of Mobile Applications). XLIMA inherits the concepts and design from LIMA, a description language from our work on interoperability of mobile application that we presented in Chapter 5. A mashup component developer can use XLIMA for describing abstract functionalities and technical configurations of mashup components, mobile application components and Web services 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, we design another description language called MEDAL, Mashup Event-Driven Annotation Language, to represent event-driven mashup composition logic. MEDAL inherits concepts and design from our proposed description language C-MAIDL, and extends the capability to represent event-driven mashup composition. However, instead of including mashup component configurations together with mashup composition descriptions as C-MAIDL does, MEDAL contains only the description of mashup component integration but provides a mechanism to link to URIs of mashup component configuration files (XLIMA files). Thus, developing event-driven mashup applications with our approach requires both XLIMA for component description and MEDAL for mashup application description. Similarly, MEDAL description files are used as input for the mashup application builder to generate mashup applications as native mobile applications. The following sections present the specification and features of XLIMA and MEDAL.

6.3.1 XLIMA

XLIMA is an XML description file that is required for mashup component development. It contains configurations for: component identification, abstract declaration of functions and events, invocation configuration, mapping abstract and invocation configuration, and user interface description for parameterize user interfaces. Composers describe a mashup component by declaring abstract functions and events and specifying invocation configurations in the file. The binding section is used for mapping the declared abstract and invocation information together. The parameterize UI section describes user interface elements that will assist mashup users in run-time configuration editing of the mashup component. The structure of XLIMA files and a brief description of each section are shown in Figure 29.

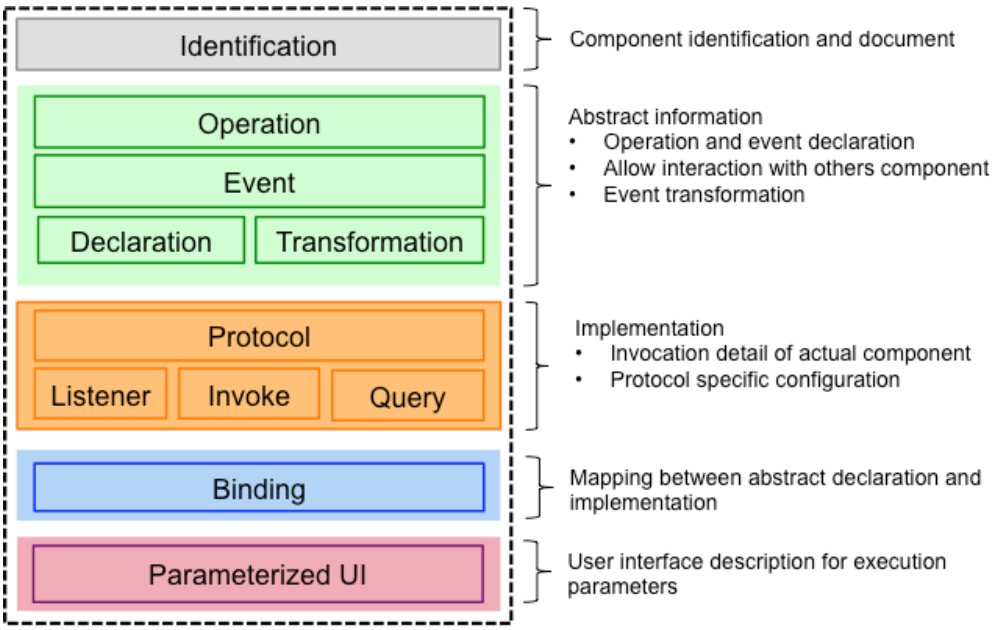


Figure 29. Components of XLIMA.

Abstract information. The abstract information section of XLIMA is designed for describing functions, events, and data types of mashup components. The description is written in terms of technology-independent configurations. In other words, functions and events are described with an identifier, input/output parameters names and data types or function names in a programing language. The script below shows a sample description of a language translator component, which contains one function and one event.

An example of function and event declaration in XLIMA

```
<abstraction>
    <component id="com.korawit.mashup.component.translate">
        <operation id="translate">
            <parameters>
                <param id="text" datatype="string" />
            </parameters>
        </operation>
        <event id="on_translated">
            <parameters>
                <param id="output" datatype="string" />
            </parameters>
        </event>
    </component>
 </abstraction>
```

Implementation information. The implementation information section of XLIMA is designed for describing 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 word, functions and events are described with their specific invocation protocols. Since our approach focuses on the composition of mobile applications, Web services and cooperation among data among

devices, this section can contain descriptions of mashup components using three sets of XML tags: <mobileapplication>, <webservice>, and <cooperation>.

i. *Mobile application component*. This set of tags describes Android's Intent configurations as inter-application communication protocols. Intent and Intent filter information is used in configurations of functionalities while Intent Broadcasting configurations are used for event listening. The examples below show a sample description of a mobile application component that sends a tweet message through a Twitter client mobile application.

An example of the SendTweet function description in XLIMA

```
<mobileapplication id="com.twitter.android">
    <function id="SendTweet">
        <protocol name="intent" type="implicit">
            <intent-filter>
                <action mode="query">com.twitter</action>
                <settype>application/twitter</settype>
                <extras>
                   <extra id="text" name="android.intent.extra.TEXT" datatype="string" />
                   <extra id="photo" name="android.intent.extra.STREAM" datatype="uri" />
                </extras>
                <flags>
                    <add>FLAG_GRANT_READ_URI_PERMISSION</add>
                </flags>
            </intent-filter>
        </protocol>
    </function>
  </mobileapplication>
```

In order to describe a mashup component that listens to events in mobile devices, the Intent Broadcasting configuration is used. The examples below show a sample description of a mobile application component that listens to new incoming Email and triggers an event as Intent Broadcasting.

An example of OnEmailReceived event description of XLIMA

```
<mobileapplication id="com.fsck.k9">
    <listener id="K9EmailListener">
        <protocol name="intent" type="broadcast">
            <intent-filter>
                <action>com.fsck.k9.intent.action.EMAIL_RECEIVED</action>
                <data-scheme>email</data-scheme>
                <extras>
                    <extra id="subject" name="com.fsck.k9.SUBJECT" datatype="string" />
                    <extra id="from" name="com.fsck.k9.FROM" datatype="string" />
                    <extra id="preview" name="com.fsck.k9. PREVIEW" datatype="string" />
                </extras>
            </intent-filter>
        </protocol>
     </listener>
  </mobileapplication>
```

From the above examples, we list the essential properties and their descriptions of mobile application components in the following table.

Tags	Description
function	Set of properties for describing a function
listener	Set of properties for describing an event
protocol	Inter-application communication protocol name: name of selected inter-application communication protocol type: type of inter-application communication protocol
intent-filter	Configuration of Android's Intent protocol action: identifier of target mobile applications Extra: collection of invocation parameters

ii. *Web service component*. This set of tags describes Web service APIs and their required parameters. Web service components in XLIMA support the REST architecture Web service with JSON or XML data format. The description contains the location of the target Web service, required query parameters and the output extraction pattern. The examples below show a description of a translation Web service component.

An example of Web service component description of XLIMA

```
<implementation>
    <webservice id="wsTranslate">
        <entrypoint url="http://mymemory.translated.net/api/get">
            <protocol name="http" type="rest">
                <query>
                    <key name="q" urlencode="UTF-8"/>
                    <key name="langpair" urlencode="UTF-8"/>
                </query>
                <results>
                    <result id="translateText">
                        <value rule="json" path="/responseData[0]/translatedText" />
                    </result>
                </results>
            </protocol>
        </entrypoint>
    </webservice>
</implementation>
```

From the above examples, we list the essential properties and their descriptions of mobile application components in following table.

Tags	Description
entrypoint	URI of a Web service
protocol	Invocation protocol, i.e., http or https
query	Set of required invocation parameters key: parameter name **value of each parameter will be determined in the binding description*

results	Set of return results from the target Web service result: define name of result to be integrated with other components value: configuration for extraction of required information from total result

iii. *Cooperation component.* Required information from participating devices can be specified in this component. We use the cooperation mechanisms of our C-MAIDL approach to facilitate data exchange among participating devices. The request message will be sent to all participating devices. The client software, cooperation agent, on the guest device invokes the local component to extract the requested data, and sends it back to the host device. Once data from a guest device has been sent back to host device, the cooperation component notifies the mashup application by embedding cooperation data as an event parameter, and posts the event onto the delivery channel. In this way, the mashup orchestration process receives the cooperation data from multiple guest devices as ordinary events; it is not aware about connectivity and the cooperation process. As a result, this set of tags describes required data from guest devices as well as the target mashup component on guest devices. The examples below show a description of data cooperation, in which the host device requests a barcode scan from all guest devices. The guest devices will invoke a mobile application, ScannerGo which support x-callback-url, to scan a barcode and return the barcode number.

An example of a cooperation component description in XLIMA

```
<cooperation>
    <mobileapplication id="com.levelup.scannergo">
        <function id="BarcodeScan">
            <protocol name="urlscheme" type="x-callback">
                <urlscheme scheme="ilu://x-callback-url/scanner-go">
                  <x-query>sg-camera=BACK&sg-history=NO</x-query>
                     <x-error>error</x-error>
                </urlscheme>
             </protocol>
         </function>
    </mobileapplication>
    <results>
          <result id="scancode" datatype="text" />
    </results>
</cooperation>
```

Binding. The binding section of XLIMA is designed for describing the mapping between the abstract declaration of function and events and invocation information in the implementation section. This section is separated from the abstract and implementation sections in order to provide maximum loose coupling. Mashup composers are free to create a group of related functions and events that come from

difference actual components. The description in this section is written in term of mapping rules. Each pair of abstract and implementation information requires a rule to bind them together, as well as a specification of how parameters of the pairs are mapped. The script below shows a sample of a binding section that maps an abstract event declaration to invocation details.

An example of a function and event binding description of XLIMA

```
<bindings>

    <binding type="invoke" trigger="translate" action="wsTranslate">

        <mapping rule="direct" source="translate.text" target="wsTranslate.q" />

        <mapping rule="direct" source="userinterface.uiLang" target="wsTranslate.langpair" />

    </binding>

    <binding type="event" trigger="wsTranslate" action="on_translated">

      <mapping rule="direct" source="wsTranslate.translatedText"

      target="event.on_translated.output" />

 </binding>
```

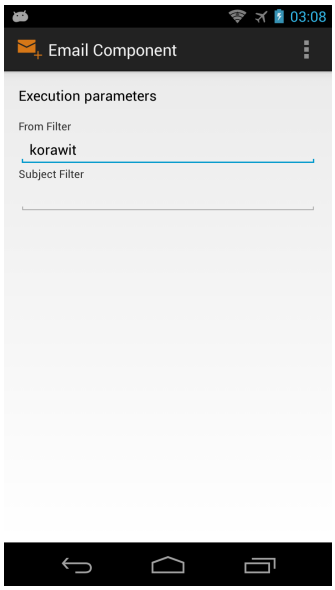
From the above examples, we list the essential properties and their description of the binding section in Table.

Tags	Description
binding	Function and event mapping type: type of mapping (invoke or event) trigger: reference to function or event id in the implementation section action: reference to function or event id in the abstract information section
mapping	Parameter mapping rules rule: type of mapping method (direct or template) source: source parameter target: destination parameter

Parameterize UI. This section of XLIMA is intended for the description of user interface elements and the mapping between those elements and invocation parameters. The description in this section will be generated as a screen of the mashup proxy component. The script below shows an example of the parameterize UI section.

An example user interface description in XLIMA

```
<userinterface>
    <element id="uiFrom" datatype="string" uitype="EditText">
        <display-name>From Filter</display-name>
        <default>korawit@gmail.com</default>
    </element>
    <element id="uiSubject" datatype="string" uitype="EditText">
        <display-name>Subject Filter</display-name>
        <default-value>""</default-value>
    </element>
</userinterface>
```



Given the above examples, we list the essential properties and their description of the parameterize UI section in following table. The screen that was generated from the sample description is shown above.

Tags	Description
element	Specification of a user interface component datatype: Data type of the parameter uitype: UI type (EditText, Password or List) display-name: Caption for the UI element default: default value **Abstract information section can directly refer to parameters of this section*

6.3.2 MEDAL

MEDAL is an XML description file that is required for the composition of event-driven mashup applications. It is designed for representing event-driven mashup composition logic; in other words, for describing the integration of mashup components in an event-driven manner. MEDAL contains a set of XML tags for declaring the required mashup components, specifying which events will be listened for, and configuring the corresponding action to those events. An event listener description contains properties for linking an abstract event of a mashup component with one or more event handlers. The event handler contains invocation descriptions that specify which functions will be invoked corresponding to the event. The structure of MEDAL files and a brief description of each section are shown in Figure 30.

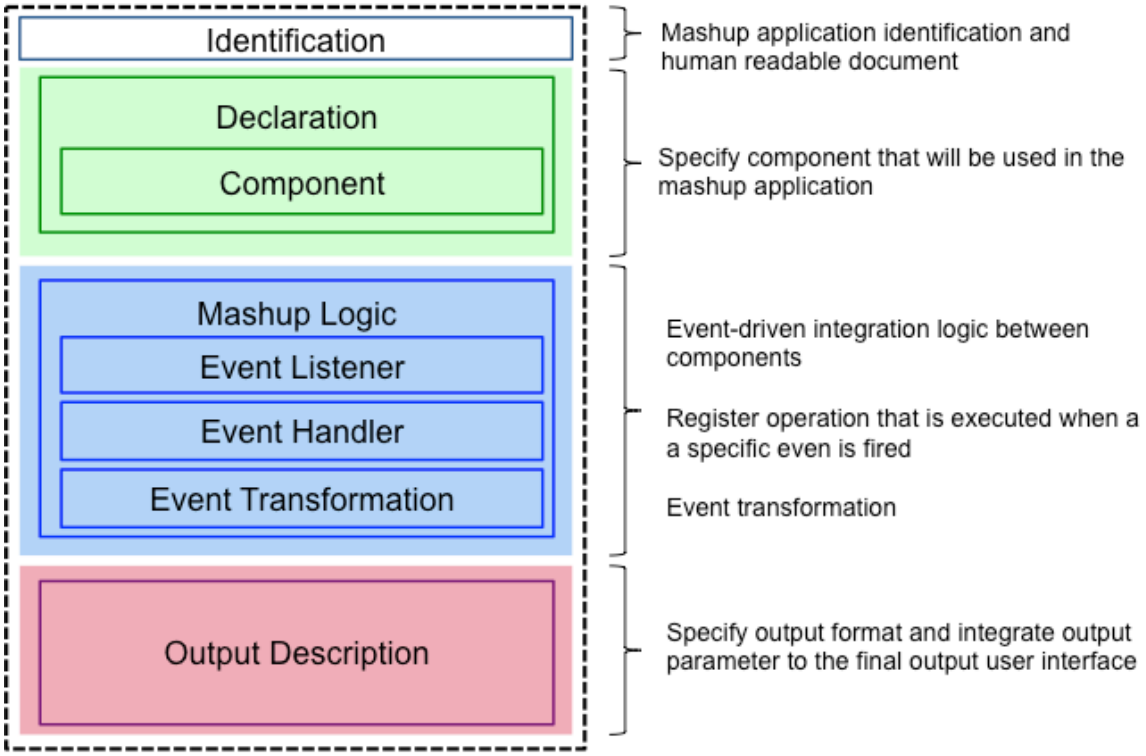


Figure 30. Components of MEDAL

Declaration. This section of MEDAL is intended for the declaration of mashup components, which are required for the composition of a mashup application. A mashup component can be declared by specifying the URI of an XLIMA description file. Once the declaration is complete, another part of the MEDAL description can refer to events and function of that mashup component. The script below shows a sample declaration section that declares three mashup components.

An example of component declaration in MEDAL

```
<declaration>
      <components>
            <component
                  id="com.korawit.mashup.component.email"   alias="K9Email"
                  url="http://localhost/mobilemashup/cpr/k9email_proxy.xlima">
            <component
                  id="com.korawit.mashup.component.translate" alias="Translator"
                  url="http://localhost/mobilemashup/cpr/translate_proxy.xlima">
            <component
                  id="com.korawit.mashup.component.note" alias="Evernote"
                  url="http://localhost/mobilemashup/cpr/note_proxy.xlima">
      </components>
</declaration>
```

Given the above examples, we list the essential properties and their descriptions for the declaration section in Table.

Tags	Description
component	Reference information of mashup component in the composition alias: alias name of mashup component url: URI pointing to the XLIMA description file of the component

Mashup Logic. This section of MEDAL is intended for the description of component integration in an event-driven manner. Mashup developers can create multiple event listeners to intercept target events. An event listener contains one or more event handlers, where each event handler specifies one or more actions to be performed after the event has occurred. For example, actions such as invocation of a Web service or call to a mobile application for a result can be defined in the event handler. The script below shows a sample mashup logic section.

An example description of event-driven integration in MEDAL

```
<mashup>
      <listener id="emailListener" alias="OnEmailReceived"
            publisher="K9Email" event="on_email_received">
            <handler>
                  <invocation component="Translator" operation="translate" >
                        <mapping mode="single">
                              <direct src="OnEmailReceived.content" dest="translate.text" />
                        </mapping>
                  </invocation>
            </handler>
      </listener>
      <listener id="translationListener" alias="OnTranslated"
            publisher="Translator" event="on_translated">
            <handler>
                  <invocation component="Evernote" operation="addnew" >
                        <mapping mode="single">
                              <direct src="OnTranslated.output" dest="addnew.text" />
                        </mapping>
                  </invocation>
            </handler>
      </listener>
</mashup>
```

Given the above examples, we list the essential properties and their descriptions for the mashup logic section in the following table.

Tags	Description
listener	Define which event will be listened for alias: alias name of event that will be referred to by other elements publisher: mashup component that produces this event event: event name

handler	Define action or set of actions that will be performed given this event
invocation	Define target functionality and parameters component: target component name operation: function name mapping:
mapping	Parameter mapping rules rule: type of mapping method (direct or template) source: source parameter target: destination parameter

Event transformation. As discussed above, we found that sometimes events that are directly produced from mashup components are less useful. Thus, this section of MEDAL is designed for describing event transformation rules for turning less useful events into more interesting events. For example, battery status information frequently alerts other processes in Android mobile devices. Mashup applications may be interested in a notification that the remaining battery is less than 5 percent. The script below shows an event transformation description of the battery status example.

An example description of event transformation in MEDAL

```
<listener id="batteryListener" alias="OnBatteryStatusChanged"
            publisher="BatteryNotification" event="battery_status_changed">
            <transformation>
                  <filter parameter="percentage" operator="less" value="5" action="raise" >
            </transformation>
            <handler>
            …
            …
            …
</listener>
```

In the above example, the mashup application is listening to “battery_status_changed” events. The event provides a parameter called “percentage” that indicates the current remaining battery. If the value of the percentage parameter is more than 5, the mashup application will ignore this event by not invoking the actions defined in its event handler. On the other hand, the mashup application will perform actions in the event handler in case the battery status is less than 5 percent. We list the essential properties and their descriptions of mashup logic section in Table.

Tags	**Description**
transformation	Define event transformation rule
filter	Specify filter condition of event parameter parameter: event parameter that will be used as a condition operator: filter operator (less, equal, more) value: criteria value action: action to perform after meeting the filter condition (raise or ignore)

Output description. This section of MEDAL is designed for formatting the display of the mashup output. Some mashup applications require user interface elements, such as textboxes, tables or maps to display the mashup result. This section provides two types of output description: Web view and Map view. Mashup composer can create Web view by composing HTML tags embedded within output data from the mashup components. A Map view is a configuration of data to display on a map as pins. The output acts as a built-in mashup component. The final mashup component can display the result to the mashup user by invoking the output component with parameters. The script below shows a sample of showing a mashup result in Map View, in which search results from the Google Places Web service is displayed as pins. The configuration of maps, such as mode, zoom level or pin icon can be specified in the <output> tag.

An example description of the Map View output in MEDAL

```
<mashup>
...
<listener id="queryResultListener" alias="OnResultCompleted"
            publisher="GooglePlace" event="Callbacked">
            <handler>
                  <invocation component="output" operation="showPOI" >
                        <mapping mode="collection">
                                    <direct src="Callbacked.lat" dest="output.latitude" />
                                    <direct src="Callbacked.lng" dest="output.longitude" />
                                    <direct src="Callbacked.name" dest="output.title" />
                        </mapping>
                  </invocation>
            </handler>
      </listener>
</mashup>
<output>
      <mapview mode="map" zoom="16" userlocation="on">
            <pinpoints mode="multiple" icon="red_simple">
      </mapview>
</output>
```

6.4 Implementation

In order to demonstrate the capabilities of our event-driven mashup composition approach, we implemented sample mashup scenarios. In this section, we present the *Email Translation Scenario,* one of the event-driven mashup scenarios presented in 6.1.1. This scenario simulates a language translation requirement that helps translating the content of emails. The mashup application listens for new incoming emails. When an email has arrived, it translates the content from Japanese into English, and save the translated text to a note application. The required components of this mashup application are as follows.

i. *Email Client Application*. This is a mobile application component: an open-source e-mail client on Android called K-9 Email [54]. When new email is received, this application produces event notifications in the form of Intent Broadcasting.

ii. *Translation Web Service API*. This is a Web service component: an online language translation API called Mymemory Translator [55]. This Web service API uses the REST architecture and JSON data format.

iii. *Note Application*. This is a mobile application component: a popular mobile application on Android called Evernote [56]. This application provides Intent integration information for creating a new note and other operations.

The mashup application development process of this scenario starts from developing mashup components. XLIMA description files of two components, an Email Proxy for K-9 Email and a Translator Proxy for the translation Web service, must be created. Since the Evernote application is used an output component, a mashup proxy is not required. Therefore, three description files are used as input for the mashup component generator to generate a mashup proxy as three mobile applications, which are installed on the target device. At this point, we have three ready-to-use mashup components that can be reused in many mashup compositions. The next step is to create a mashup application by composing a mashup application description file with MEDAL, and generating the output application with the mashup application generator tool. The final step is installing the mashup application on the target device. However, since this mashup application integrates two existing mobile applications which require authentication, the K-9 Email and Evernote applications must be installed and we must be logged in with a user account before the mashup application is started. Finally, the user can run the mashup application as an ordinary mobile application. The composition model of this mashup scenario is shown in Figure 31.

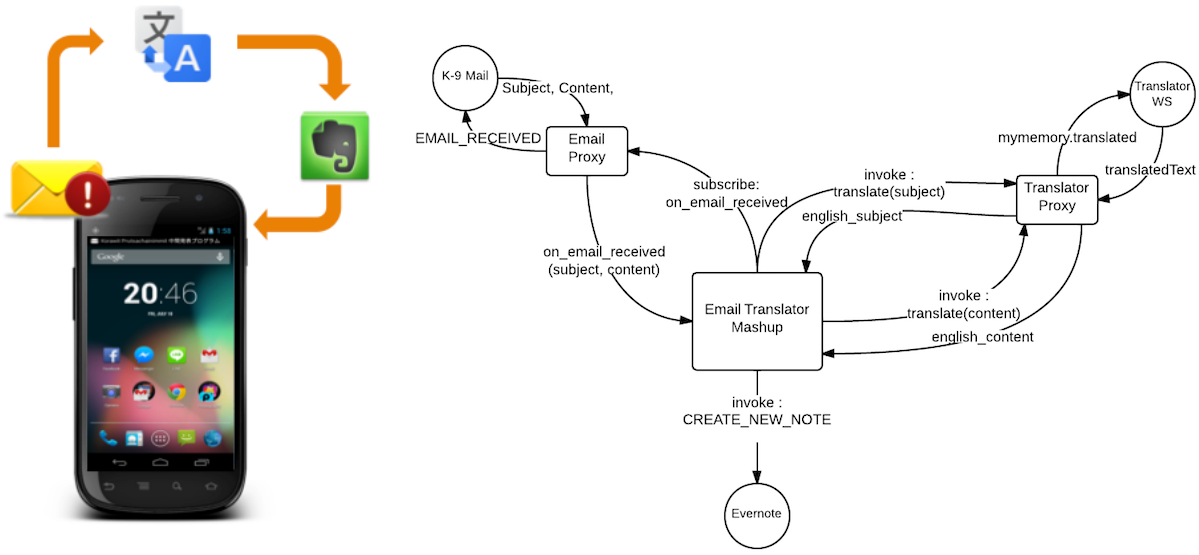
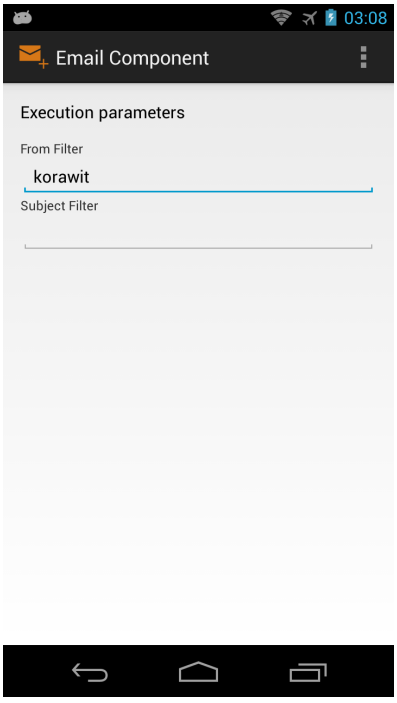


Figure 31. Mashup composition model of automatic email translation scenario.

The event-driven mashup process starts from the Email Translator Mashup (mashup application), which subscribes to event "on_email_received" of the Email Proxy component. The Email Proxy then uses Intent configurations, specified in the XLIMA file, to register an actual event notification of the K-9 Mail application. When the K-9 Mail application receives a new email, it notifies the Email Proxy and supplies two parameters, subject and content of the new email. The Email Proxy then creates a mashup event and posts it onto the delivery channel, which is also implemented using Intent Broadcasting. The mashup application then receives the event notification with attached parameters. Next, the subject parameter is used as an input to invoke the "translate" function of the Translator Proxy. At the same time, the mashup application subscribes to the event "on_translated" of the Translator Proxy to receive the translation result. Once the result from the Web service returns to the Translator Proxy, a mashup event is created, with the translated text as a parameter, and the event is posted onto the delivery channel. As a result, the mashup application receives the translated text as a parameter from such an event. Next, the mashup application invokes the Evernote application by using the translated content as parameter for Evernote's Intent. The same process can be repeated for translating content data. Finally, the translated text is saved to the note application, and the mashup process waits for the next event notification. The full XLIMA and MEDAL descriptions are shown in Appendix. The screenshots of mashup components for this scenario are presented in Figure 32. With the parameterize UI feature, users can use the Email Proxy application to customize target email to be translated by editing filter values. Similarly, the Translator proxy provides a language pair parameter that allows changing the target language.



i. Run-time parameters to filter email from specific person or subject

ii. Run-time parameters to select language pair for translation

Figure 32. Screenshots of mashup components

Chapter 7

Evaluation

The evaluation of this research is divided into three sections. We explain the expressivity of our mashup composition in Section 7.1 and show result of usability evaluation in section 7.2. The comparison of our approach to other related mashup approaches is discussed in section 7.3. In the final section, 7.4, we discuss problems and limitations found in the current composition method as well as further improvements.

7.1 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 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”. The tutorial also explains concepts of our mashup composition approach 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.

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 7.

Name	Type	Description
Input components		
Barcode Scanner	MA	Scan barcodes and return as a text by using device’s camera.
Location	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

Table 7 Generated mashup components for usability evaluation

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 [57]. 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 8 shows result of the evaluation.

Composer	Composition Pattern	Mashup Planning Time (minutes)		MEDAL Composition Time (minutes)		Total Time (minutes)	
	No. of Component	Total Usage	Comp. Avg.	Total Usage	Comp. Avg.	Total Usage	Comp. 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**	

Table 8 Usability evaluation result

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 33 shows the 5-points Likert scale result of the post-questionnaires.

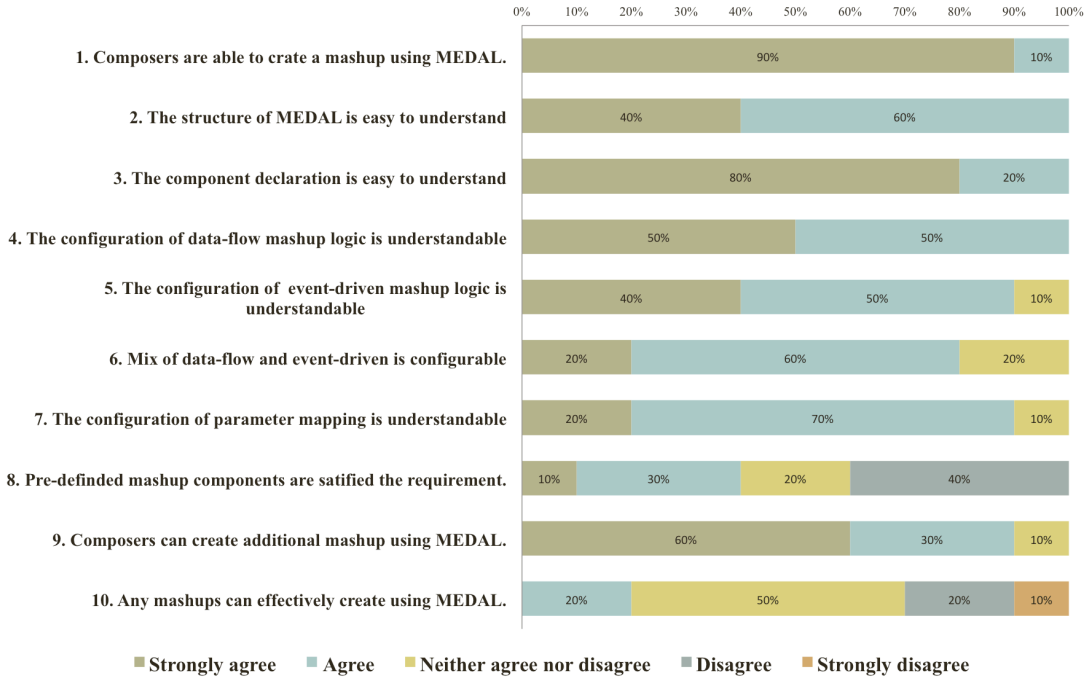


Figure 33 Post-questionnaire result

- Most composers gave high ratings concerning the ability of MEDAL to assist their compositions and the comprehensibility of configurations for defining data-flow and even-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.

7.2 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 7.1. The result from 7.1 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. 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 34. 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.

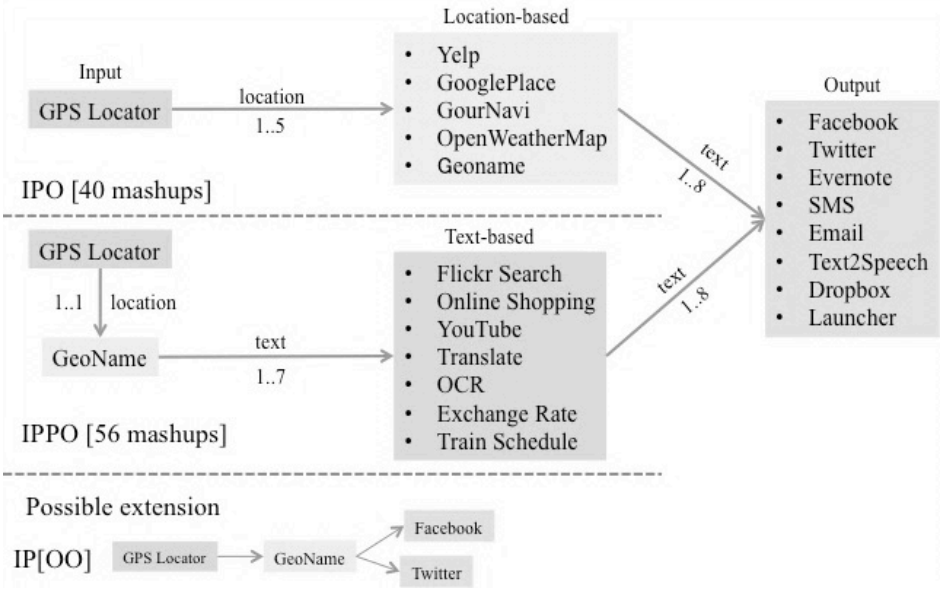


Figure 34 Simulations on generating location-aware mashup applications

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. We define three sets of reusable components: mobile applications, Web services and mobile events. We then use our mashup description and generators to develop mashup applications from these sets of components. The number of applications that we can generate expresses the capability

to deal with a variety of user requirements and expressivity of our mashup composition method.

The selected mobile application components are taken from the frequently used functionalities in mobile devices. We survey frequently used functionalities from popular applications on two major application delivery channels, Google's Play Store and Apple's App Store. To create our sample list, we select reusable functionalities, available as existing applications or system APIs that support Intent integration. We divide the list of functionalities into several categories as shown in Table 9.

Category	Name	Description
Sensor	GPSLocator	Get current device's location
	Location notifier	Detect an enter/exit action of the registered location
	Date and Time*	Get current date and time
Camera	Zxing Barcode	Read QR code using the camera
	Photo*	Take a photo
Language	Dictionary	Look up the definition of a word
	Text2Speech	Speak the received text
	Speech input	Translate the user's speech into text
	Translate Intent	Translate the given text into another language
Telecom	Phone dialer*	Call a phone number
	SMS*	Send an SMS to the receiver's phone number
	Email*	Send an Email to the receiver's address
Map	Google Maps	Show a pin on a map and navigate to it
Storage	ES File Explorer	Open files
	Dropbox	Download or save a file on cloud storage
Social Network	Facebook	Update Facebook status, send a message
	Twitter	Tweet a message
	YouTube	Play a YouTube video from the given URL
Utility	App Launcher*	Launch other mobile applications
	Evernote	Add a text as a new note
	Web Search*	Perform a Web search of the given keyword
	Music player*	Launch a music player with the given file URI

**Built-in Android application*

Table 9. List of sample mobile application components.

There are large numbers of Web service APIs available for mashups as listed in API directory Websites, i.e. Programmable Web [57]. We select frequently used Web service APIs that are compatible with our approach: those offering a REST interface with JSON or XML output. Since our approach focuses on the integration of data, some

popular Web service APIs, such as Google Maps which allows the embedding of maps onto web pages, are not selected. The list of Web services is shown in

Table 10.

Category	Name	Description
Ecommerce	Google Shopping	Search for product information, e.g. title, price or picture, from online stores using keywords or barcode numbers [58].
Video	YouTube	Search for videos by keywords [59].
Photos	Flickr	Retrieve photos from the Flickr photo sharing service using a variety of parameters [60].
Social	Facebook Graph	Query Facebook data using its graph API [61].
Music	Last.fm	Search for music related information, e.g. artist, album or track name [62].
Conversion	MyMemory Translate	Translate the given text into another language [55].
	ExchangeRate API	Currency exchange rate Web service [42].
Reference	Wikipedia	Search for Wikipedia content by keywords [63].
	FlightStat	Search for flight status and tracking information [64].
	GeoName	Resolve the place name from a geographic location [65].
Location-based Recommendations	Yelp	Retrieve business reviews and rating information for a particular geographic region or location [5].
	Google Places	Search for POIs using particular geographic coordinates [4].
	GourNavi	Retrieve restaurant reviews and rating information for a particular geographic location [3].
	OpenWeatherMap	Retrieve weather information for a particular geographic location [66].
Web Extraction	Import.io	A partial Web extraction engine that provide Web service APIs to access the extraction result [67].

Table 10. List of sample Web services

Given a set of mobile components, Web services components and system events listed in section 3.2, we are able to generate various data-flow and event-driven mashup applications by using simple composition logic. The sample applications are listed in

Table 11.

Application	Component	Pattern	Description
Last Train *Display last train schedule using current location*	[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 for the last train.
Photo Diary *Create a diary in a note application including photo and related information about the place it was taken*	[MA] Camera [M] GPSLocator [WS]GeoName [WS] Wikipadia [MA] Evernote	Data-Flow	Use the camera to take photos and find related information about that photo by using the current location to find the place name. The place name is used to find a Wikipedia article. Finally, the photo and related information is saved to Evernote.

Wish List *Tweet a message about interesting items to buy*	[MA]Xzing Barcode [WS]GoogleShopping [WS] Translate [WS] ExchangeRate [MA]Twitter	Data-Flow	Use the Xzing application to scan barcodes of interesting items and find title, price and a link from online stores. Then translate the information into a desired language. Finally, tweet the translated information as a wish list.
Check In *Automatically send a message to friends when you arrived at a place.*	[E]Wi-Fi Connected [MA] Facebook	Event-Driven	When connected to a Wi-Fi network, use the network name to determine arrival status, and send a message to a friend using the Facebook application.
Battery Tweet *Tweet a message when the remaining battery status becomes critical*	[E] Battery Status [MA] Twitter	Event-Driven	When battery status is lower that 5 percent, tweet a message to notify friends
Blind Buddy *Assist the blind by speaking the current place name out loud*	[E] Location Changed [WS] GeoName [WS] Translate [MA] Text2Speech	Event-Driven	When the location has changed, speak the new place name out loud.

[MA]: mobile application component, [WS]: Web service component, [E]: Event
**The APIs to find the last train schedule of two stations are not available as a Web service. We can use a Web extraction tool to get the required information from the Website Jorudan [], and access this information via the Web service API of the Web extraction tool.*

Table 11. List of generated applications

With a small set of mashup components, our approach can generate various kinds of mashup applications. The simple integration model that integrates a mobile application component with a Web services component can express broad areas of requirements. For example, the Web service component of the Last Train application, which finds the schedule for the last train, can be replaced with other Web services, such as a restaurant and hotel search, weather information, photo retrieval or some other location-based service to address different user's requirements. 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. For example, we can change the event and mobile application component to other pair of components, such as when an SMS is received the message is spoken out loud, or when an email is received the text is translated or when the location has changed, another Web service component is called in order to find nearby POIs.

The result from this experiment shows that our mashup composition method is expressive and flexible enough to generate a variety of mashup applications. Integration of exiting mobile applications and Web service APIs can address problems of various domains. Supporting both data-flow and event-driven composition is another key to enhancing expressivity. Notably, the capability to integrate data from multiple devices makes our approach unique and more expressive than other existing approaches.

7.3 Comparison to other mashup approaches

Many mobile mashup approaches have been proposed. Some of them share the same goals as our approach. Table 12 shows a comparison of our approach with other approaches. We select mashup approaches with similar objectives; that is, development of event-driven mashup applications for mobile devices. The first selected approach is TELAR [14], which is a context-aware mobile mashup platform that integrates Web services and context information of local sensors. With TELAR, users can create event-driven mashup applications that show POIs around their current location. Next is telco mashup, which is mobile mashup for multiple devices. Telco mashup allows integration of Telco services and/or device APIs to encourage collaboration among multiple users [17]. Another one is MobiMash [19], which is a mashup composition approach that generates hybrid mobile applications based on data integration and service orchestration. We also added our previous work MAIDL to illustrate the improvement of the approach.

Features	TELAR	TELCO	MobiMash	MAIDL	XLIMA+MEDAL
Component Integration	Location[P]/ WS[P]	Telephony[P]	WA / WS	MA / WS / WA / TeWS	MA + WS
Mashup Output	Web-based	Web-based	Hybrid App	Native App	Native App
Composition	Event-Driven	Event-Driven	Event-Driven (UI events)	Data-flow / Event-Driven[P]	Data-flow / Event-Driven
Event Coverage	Location	Telephony	User Interface	Pre-defined	System / Mobile Apps
Event Handler	Manual	Manual	Automatic (UI events)	Manual	Automatic
Component Execution	Sync	Sync	Sync	Sync	Async
Resource Consumption Level	Browser	Browser	Native App	Native App	Native App
Cooperation Characteristic	Multiple Clients	Multiple Clients	User Interface Synchronization	Tethered Web services	Data Cooperation

[P] *denotes pre-defined components*

Table 12. Comparison with other related mashup approaches.

7.4 Discussion

The results of the expressivity evaluation and the comparison to other mashup approaches indicate that our mashup composition provides an efficient solution to mobile mashup composition for multiple mobile devices. The scope of integration and expressivity of our approach is expanded from existing mashup approaches as shown in Figure 35. Our approach achieves integration of device-specific features by using mobile applications as a proxy to access sensor and context-aware data. REST Web services and device-specific data from multiple mobile devices are also usable as

mashup components. The mashup composition can be composed in an event-driven or data-flow manner. However, our approach is still unable to address some challenges. This section discusses problems and limitations found in our current composition method and a further plan for improvement.

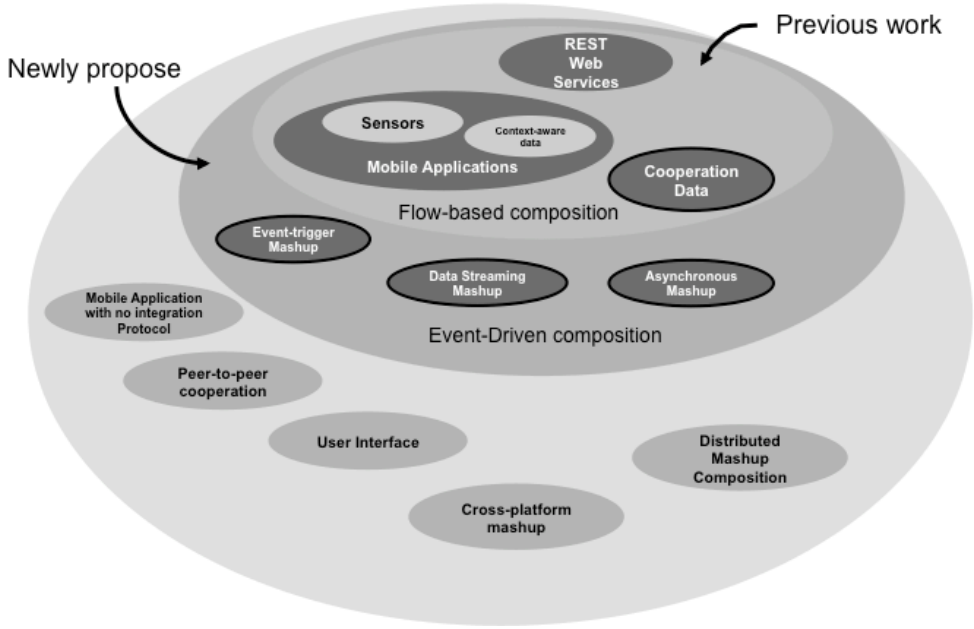


Figure 35. Scope of integration of our approach.

Constraints of mobile application components. Our approach uses the capabilities of Android Intents to turn mobile application into mashup components. However, not all applications support Intent integration. Furthermore, the level of supporting differs per application. We can use Intent to invoke functionality of some mobile applications as service functions, sending input parameters and receiving output, such as barcode reader or translator applications. On the other hand, some mobile applications take inputs from Intents and bring themselves to active context without returning a value. As a result, the composer should consider supporting the Inter feature of mobile application components. Some applications can work as a service function, while some applications can work as a final component to display or receive the output. Recently, the Android operating system has implemented a Shared Intent [68] feature, which allows applications to send data to other 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, most recent mobile applications can be used as output components in our mashup composition.

Limitations of event-driven composition. Our approach allows using events from the Android system via Intent Broadcasting and events of mobile applications through the notification center. Multiple events can be listened for and handled in a mashup application. One event may have multiple handlers to execute multiple mashup logic. However, the current composition model still does not support simultaneous events with one or multiple handlers. This composition style becomes a limitation because most event notifications of the Android system using a single event model. However,

Android has a mechanism called Sticky Broadcast Intent [37], which holds the Intent Broadcasting as a background service. Sticky 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.

Extension to Mesh Networking. We use a mobile Web server, i-Jetty, for enabling connectivity between a host device and guest devices. The communication request messages are sent through Email or SMS while the response message uses HTTP requests to Web service interfaces. We found that execution time of our mashup is heavily dependent on the efficiency of the network that delivers Email and SMS messages. In addition, the i-Jetty Web server is not compatible with more recent versions of the Android operating system. As a result, these problems reduce efficiency of the cooperation process of our approach. Improved mobile device connectivity technology should be considered.

Recently, we found that there is research and commercial software aimed at creating mesh networking for mobile devices; for instance, OpenGarden [69] and Serval Project [70]. By using mesh networking, mobile devices are connected to other devices in a peer-to-peer fashion through various channels, e.g., Bluetooth or Wi-Fi. Since the mesh network topology allows connections to reach every node, cooperation of data on this type of network is more flexible. The cooperation mashup scenario can be more complex and expressive. In addition, the available mesh networking frameworks show that they provide simple configuration and satisfactory performance. Thus, we plan to use mesh networking as a new communication infrastructure for execution of cooperative mashup applications.

Mashup Development Process. This research aims to reduce the development cost by using description languages and mashup generator tools. We inherit concepts from our previous work, MAIDL, and improve the development process in order to reduce the required programming skills and efforts. The evaluation results of MAIDL show that managing configurations of mashup components requires technical knowledge and effort. This is a common problem for mashup approaches that aim to support end-users. In facts, end-users cannot deal with complex configurations of Web service APIs or Intent parameters. In some cases, they cannot comfortably edit the XML description file. As a result, state-of-the-art mashup approaches apply two techniques to address this problem [30]. The first technique is to delegate the component development process to users with higher programming skills; so called component developers. Our approach applies this concept to improve our mashup development process. We separate the component description language from the mashup composition language to allocate this task to skilled users or developers. XLIMA, a technology-dependent language, is used by component developers while MEDAL, a technology-independent language, is designed for users who are familiar with editing XML tags. 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. Notably, the run-time configuration feature of our mashup components is a key to improving component reusability. In this way, our approach can reduce the development cost in comparison to our previous approach. However, we plan to implement a visual composition tool for supporting end-user mashup composition.

Execution Performance and Resource Consumption. In general, mobile mashup applications are created as Web applications, hybrid mobile applications or native mobile applications. Native mobile applications can be considered to be the most resource-consuming. The conventional mobile mashup approaches build mashup output as a single native mobile application. The modules, which work as mashup components, and the orchestration process, are placed in one mobile application. In case of a complex mashup composition that uses many mashup components, the mashup process execution requires more resources from mobile devices. In addition, let us consider multiple mashup applications that use the same mashup components but composed using different logic. During the execution of these mashup applications, each identical mashup component allocates their own resources from devices even if they perform the same task.

To address this problem, a unique idea of our mashup composition is that mashup components and mashup applications are built as mobile applications, and integrated using an inter-application communication protocol. One reason behind this idea is that we want to utilize the automatic resource management of the Android system and reduce resource consumption. Separating the processes of mashup components and mashup applications can enhance reusability and reduce resource consumption. The mashup components run as ordinary mobile applications, which will be kept as inactive mobile applications when there is no component interaction activity. In this way, multiple mashup processes, which contain only integration logic, can invoke the mashup components in different orders while keeping the resource consumption level equal to that of one mashup application.

Chapter 8

Conclusion

This research proposed description-based composition methods for data-flow and event-driven composition for cooperation between mobile devices. The methods allow for the composition of intent-supported mobile applications, REST architecture Web services, and cooperation data from multiple mobile devices. The output mashup can be created as a mobile application running on single devices, or as a data cooperation application running on multiple devices. The composition methods also allow the creation of both data-flow and event-driven mashups in order to increase expressivity. The integration of cooperation data from multiple devices with other existing resources also expands the coverage and expressivity to the higher level when compared to conventional multiple-device mobile mashup approaches.

To realize mashup composition methods for multiple mobile devices, we studied constraints of mobile technologies and discussed related design considerations. As a result, we selected Android as our first experimental platform. The HTTP protocol with a client-server architecture is selected as the connectivity model. A mobile Web server module is used as a Web service container to host the communication interface on the server device. To allow cooperation between devices, pre-installed software for the clients and server are the minimum requirements.

We first explored data-flow mashup composition for cooperation of mobile devices by using our previous mobile mashup approach, MAIDL, as the baseline. We proposed a mashup construction system that allows composition of mobile applications and Web services in data-flow driven manner. The mashup created by this approach was targeted towards data cooperation among multiple mobile devices. We proposed an XML-based description language called C-MAIDL, which is designed for representing mashup component configurations and mashup application logic in data-flow driven manner. We built mashup generator tools for aiding end-users in mashup application development. Custom agents are pre-installed on the guest devices and a custom communication center is pre-installed on the host device to accommodate the communication processes. To develop a mashup application, mashup composers specify the intended mashup configuration in a C-MAIDL file. A mashup generation engine takes the description file as an input to generate Java code for the cooperating mashup applications. The generated code is packed into an Android package file and installed on the host device. We demonstrated the applicability for our system for cooperative mobile mashups with sample scenarios. Finally, the limitations and required improvements were discussed.

Our first approach took advantage of the integration of mobile applications with other resources. However, the mobile application integration method still has limitations

and needs improvement. As a result, we introduced LIMA, an XML-based modeling language for describing shared functionalities of mobile applications. LIMA aims to enhance interoperability of mobile applications by providing an applicable way to describe abstract and concrete details of shared functionalities. The design of LIMA has adopted successful ideas from other contexts of software interoperability. Developers can use LIMA to share functionalities of their mobile applications, and to reuse functionalities of other mobile applications. We illustrated a conceptual design and specification including examples of LIMA descriptions. The most important application of LIMA is to enhance integration capability of mobile application components. In other word, LIMA encourages reusability of mobile application components and increases flexibility of our mashup composition approach by separating component configuration from mashup composition logic.

Given the limitations of data-flow based mashup composition, we proposed another mashup composition method for developing event-driven mashup applications. This method aims to allow the composition of mobile applications, REST Web services, and shared information from multiple devices in both data-driven and event-driven ways. We set up sample scenarios that illustrate real situations of both data-driven and event-driven mashups for single devices and for multiple devices. We realize the design and characteristics of component configurations and integration by analyzing the sample scenarios. We used description languages and code generator tools to leverage mashup development cost. The XLIMA description language is defined to represent mashup component configurations, and MEDAL for describing execution logic of mashup components. To evaluate the applicability of our approach, we implemented our first prototype on the Android mobile platform and used the mashup system to create a mashup application following the sample scenarios.

We then evaluated the expressivity of our mashup composition and compared our approach to other related mashup approaches. The evaluation results showed that our proposed architecture improves expressivity and reduces the development cost of mobile mashup application composition. Finally, we discussed problems and limitations found in our current composition methods. In the future, we aim to expand the expressivity of mashups by supporting more types of mashup components and reducing mashup composition effort by implementing a visual composition tool. Improving device connectivity with better mobile network technology, i.e. mesh networking on mobile devices, is also one of our future goals.

Acknowledgement

I would like to express my gratitude to my academic supervisor, Prof. Takehiro Tokuda for his kindness and encouragement throughout the course of my study. I would also like to express my sincerest thanks to Prof. Motoshi Saeki, Prof. Takenobu Tokunaga, Prof. Katsuhiko Gondow, and Asst. Prof. Shin'ya Nishizaki, who together with my supervisor constitute as my thesis committees, for their valuable comments and suggestions.

I would like to give my thanks to Mr. Prach Chaisatien, Mr. Tomoya Noro and Mr. Kristian Slabbekoorn for their supportive assistance to my research. My gratitude also goes to all Tokuda laboratory members for their warmest regards. This research will not be successfully finished without the greathearted support from Thai students in Tokyo Institute of Technology.

Finally, to my family, for giving me strength and support to continue my study.

Korawit Prutsachainimmit

Department of Computer Science

Graduate School of Information Science and Engineering

Tokyo Institute of Technology 2016

Bibliography

1. Anderson, C. The Long Tail: Why the Future of Business Is Selling Less of More by Chris Anderson. *Journal of Product Innovation Management*, *24*, (2007), 1–30.

2. Maximilien, E. M. (2008, August). Mobile Mashups: Thoughts, Directions, and Challenges. In *ICSC* (Vol. 8, pp. 597-600).

3. Gourmet Navigator API. http://api.gnavi.co.jp/api/manual.htm.

4. Google Places API. https://developers.google.com/places/documentation/.

5. Yelp Serach API. http://www.yelp.com/developers/documentation/v2/search_api.

6. Chaisatien, P. and Tokuda, T. A Description-based Approach to Mashup of Web Applications, Web Services and Mobile Phone Applications. *Artificial Intelligence and Applications*, (2011).

7. Wong, J. and Hong, J. What do we mashup when we make mashups? *Proceedings of the 4th international workshop on End User Software Engineering* (2008), 35–39.

8. Zang, N. and Rosson, M.B. What's in a mashup? And why? Studying the perceptions of web-active end users. In *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*. 2008, pp. 31–38.

9. Daniel, F., Matera, M., Milano, P., and Weiss, M. Next in Mashup Development : Apps on the Web. *Current*, *13*, October (2011), 22–29.

10. Xu, K., Zhang, X., Song, M., & Song, J. (2009, September). Mobile mashup: Architecture, challenges and suggestions. In *Management and Service Science, 2009. MASS'09. International Conference on* (pp. 1-4)

11. Yahoo Pipe. https://pipes.yahoo.com/.

12. Intel Corp.: Mash maker (2007). http://mashmaker.intel.com/web/.

13. Kaltofen, S., Milrad, M., and Kurti, A. A cross-platform software system to create and deploy mobile mashups. *Web Engineering*, (2010), 518–521.

14. Brodt, A. and Nicklas, D. 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*, (2008), 700.

15. Chudnovskyy, O. and Weinhold, F. Integration of telco services into enterprise mashup applications. *Current Trends in Web* (2012).

16. Sanders, R.T. End-user Configuration of Telco Services Putting the end user in the loop – without losing enterprise control. (2012), 72–74.

17. Gebhardt, H., Gaedke, M., Daniel, F., et al. From mashups to telco mashups: A survey. *IEEE Internet Computing 16*, 2012, 70–76.

18. Stecca, M. and Maresca, M. An execution platform for event driven mashups. *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services - iiWAS '09*, (2009), 33.

19. Cappiello, C., Matera, M., and Picozzi, M. MobiMash: end user development for mobile mashups. *Proceedings of the 21st* (2012), 473–474.

20. Cappiello, C., Matera, M., and Picozzi, M. End-User Development of Mobile Mashups. (2013), 641–650.

21. Krug, M., Wiedemann, F., and Gaedke, M. SmartComposition: A Component-Based Approach for Creating Multi-screen Mashups. In *Web Engineering*. Springer, 2014, pp. 236–253.

22. Husmann, M., Nebeling, M., Pongelli, S., and Norrie, M.C. MultiMasher: Providing Architectural Support and Visual Tools for Multi-device Mashups. In *Web Information Systems Engineering--WISE 2014*. Springer, 2014, pp. 199–214.

23. Chaisatien, P. and Tokuda, T. A description-based composition method for mobile and tethered mashup applications. *Journal of Web Engineering*, *12*, 1&2 (2013), 93–130.

24. Aghaee, S., & Pautasso, C. (2011, December). The mashup component description language. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services* (pp. 311-316).

25. Pérez, I., Herranz, Á., Munoz, S., and Moreno-Navarro, J. Modeling Mash-Up Resources. (2008).

26. EMML. http://www.openmashup.org/.

27. Sabbouh, M., Higginson, J., Semy, S., and Gagne, D. Web mashup scripting language. *Proceedings of the 16th international conference on World Wide Web WWW 07*, *pages*, (2007), 1305–1306.

28. Maximilien, E.M., Wilkinson, H., Desai, N., and Tai, S. A Domain-Specific Language for Web APIs and Services Mashups. *ServiceOriented Computing–ICSOC 2007*, *4749*, (2007), 13–26.

29. Cappiello, C., Matera, M., and Picozzi, M. DashMash: a mashup environment for end user development. *Web Engineering*, (2011), 152–166.

30. Daniel, F., Casati, F., Benatallah, B., and Shan, M. Hosted universal composition: Models, languages and infrastructure in mashart. *Conceptual Modeling-ER* (2009), 428–443.

31. Chaisatien, P. and Tokuda, T. A description-based composition method for mobile and tethered Mashup applications. *Journal of Web Engineering*, *0*, 0 (2013).

32. Berson, A. *Client-server architecture*. McGraw-Hill, 1992.

33. i-Jetty. https://code.google.com/p/i-jetty/.

34. Intent. http://developer.android.com/reference/android/content/Intent.html.

35. iOS URL Scheme. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html.

36. x-callback-url. http://x-callback-url.com.

37. Intent Broadcasting. http://developer.android.com/reference/android/content/BroadcastReceiver.html.

38. Chowdhury, S.R.O.Y., Birukou, A., Daniel, F., and Casati, F. Composition Patterns in Data Flow Based Mashups. *1*, 212 (2011).

39. Tsai, C. L., Chen, H. W., Huang, J. L., & Hu, C. L. (2011, March). Transmission reduction between mobile phone applications and RESTful APIs. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (pp. 445-450).

40. Guo, J., Chaisatien, P., Han, H., Noro, T., and Tokuda, T. Partial Information Extraction Approach to Lightweight Integration on the Web. *Current Trends in Web*, *6385s*, (2010), 372–383.

41. Google Search API for Shopping. http://code.google.com/apis/shopping/search/.

42. Exchange Rate API. http://www.exchangerate-api.com/.

43. Miller, P. Interoperability. What is it and Why should I want it. *Ariadne*, *24*, 10 (2000).

44. Gasser, U. and Palfrey, J. When and How ICT Interoperability Drives Innovation. (2007).

45. Chinnici, R., Moreau, J.-J., Ryman, A., and Weerawarana, S. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. *Interface 26*, 2007, 1–103. http://www.w3.org/TR/wsdl20/.

46. Xzing Barcode Scanner (Android). https://play.google.com/store/apps/details?id=com.google.zxing.client.android&hl=en.

47. Xzing API. https://code.google.com/p/zxing/.

48. Xzing Barcode Scanner (iOS). https://itunes.apple.com/en/app/barcodes-scanner/id417257150?mt=8.

49. Cusumano, M.A. The Changing Software Business: Moving from Products to Services. *Computer*, *41*, 1 (January 2008), 20–27.

50. Tasker. https://play.google.com/store/apps/details?id=net.dinglisch.android.taskerm&hl=en.

51. Bruns, R. and Dunkel, J. *Event-Driven Architecture*. Springer Berlin Heidelberg, 2010.

52. Voulodimos, A.S. and Patrikakis, C.Z. Using personalized mashups for mobile location based services. In *IWCMC 2008 - International Wireless Communications and Mobile Computing Conference*. 2008, pp. 321–325.

53. Cappiello, C. and Daniel, F. Information quality in mashups. *Internet Computing, …*, (2010).

54. K-9 Email. https://play.google.com/store/apps/details?id=com.fsck.k9&hl=en.

55. MyMemory Translation API. http://mymemory.translated.net/doc/spec.php.

56. Evenernote (Android). https://play.google.com/store/apps/details?id=com.evernote&hl=en.

57. ProgrammableWeb. http://www.programmableweb.com.

58. Google Conent API for Shopping. https://developers.google.com/shopping-content/.

59. YouTube APIs. https://www.youtube.com/yt/dev/api-resources.html.

60. Flickr API Garden. https://www.flickr.com/services/api/.

61. Facebook Graph API. https://developers.facebook.com/docs/graph-api.

62. Last.fm REST API. http://www.last.fm/api/rest.

63. MediaWiki API. http://www.mediawiki.org/wiki/API:Main_page.

64. FlightStat API. https://developer.flightstats.com/api-docs/how_to.

65. GeoName API. http://www.geonames.org/export/.

66. OpenWeatherMap API. http://openweathermap.org/api.

67. import.io. https://import.io.

68. Android Shared Intent. http://developer.android.com/reference/android/content/BroadcastReceiver.html.

69. Open Garden. https://opengarden.com.

70. Serval Project. http://www.servalproject.org.

A : XML Description Files

C-MAIDL

To demonstrate the syntax of C-MAIDL, we provide three C-MAIDL XML description files. There are two example description files of Shopping Assistance and Meeting Point scenarios that we presented in section 4.5.

Shopping Assistance C-MAIDL Mashup Descriptions

```
<project>
      <component>
            <name>Barcode</name>
            <role>
                  <publisher>
                        <publisher-id>001</publisher-id>
                  </publisher>
            </role>
            <execution>single</execution>

            <cooperation>
                  <mobileapplication>
                        <app-name>bcapp://x-callback-url/scan?</app-name>
                        <app-params>
                              <param-name>formats</param-name>
                              <values>
                                    <value>EAN13,EAN8,UPCE,QR</value>
                              </values>
                              <param-name>success</param-name>
                              <values>
                                    <value>scanresult</value>
                              </values>
                              <param-name>error</param-name>
                              <values>
                                    <value>error</value>
                              </values>
                        </app-params>
                  <mobileapplication>
                  <results>
                        <result>
                              <result-name>scannedcode</result-name>
                              <type>single</type>
                              <value>code</value>
                              <filter>null</filter>
                        </result>
                  </results>
            </cooperation>
      </component>
```

```
<component>
      <name>GoogleProduct</name>
      <role>
            <medium>
                  <subscriber-id>001</subscriber-id>
                  <publisher-id>002</publisher-id>
            </medium>
      </role>
      <execution>single</execution>
      <webservice>
            <base>https://www.googleapis.com/</base>
            <paths>
                  <path>shopping</path>
                  <path>search</path>
                  <path>v1</path>
                  <path>public</path>
                  <path>products</path>
            </paths>
            <keys>
                  <key>key</key>
                  <key>country</key>
                  <key>q</key>
                  <key>alt</key>
            </keys>
            <values>
                  <value>10d9098dba2f680c748de5b03b28940d</value>
                  <value>JP</value>
                  <value>publisher[001].results.scannedcode</value>
                  <value>json</value>
            </values>
            <format>JSON</format>
            <results>
                  <result>
                        <result-name>ProductName</result-name>
                        <type>multiple</type>
                        <query>//title</query>
                        <index>null</index>
                        <filter>null</filter>
                  </result>
                  <result>
                        <result-name>StoreName</result-name>
                        <type>multiple</type>
                        <query>//author/name</query>
                        <index>null</index>
                        <filter>null</filter>
                  </result>
                  <result>
                        <result-name>Price</result-name>
                        <type>multiple</type>
                        <query>//inventories/price</query>
                        <index>null</index>
                        <filter>null</filter>
                  </result>
```

```
                  <result>
                        <result-name>StoreURL</result-name>
                        <type>multiple</type>
                        <query>//link</query>
                        <index>null</index>
                        <filter>null</filter>
                  </result>
            </results>
      </webservice>
</component>

<arithmetic>
      <name>BestPrice</name>
      <role>
            <medium>
                  <subscriber-id>002</subscriber-id>
                  <publisher-id>003</publisher-id>
            </medium>
      </role>
      <execution>multiple</execution>
      <operation>lowest-filter</operation>
      <input>
            <value>GoogleProduct.results.Price</value>
      </input>
      <results>
            <result>
                  <result-name>ProductName</result-name>
                  <type>single</type>
                  <value>arithmetic.results.ProductName</value>
            </result>
            <result>
                  <result-name>StoreName</result-name>
                  <type>single</type>
                  <value>arithmetic.results.StoreName</value>
            </result>
            <result>
                  <result-name>Price</result-name>
                  <type>single</type>
                  <value>arithmetic.results.Price</value>
            </result>
            <result>
                  <result-name>StoreURL</result-name>
                  <type>single</type>
                  <value>arithmetic.results.StoreURL</value>
            </result>
      </results>
</arithmetic>

<component>
      <name>ExchageRate</name>
      <role>
            <medium>
                  <subscriber-id>003</subscriber-id>
```

```
                <publisher-id>004</publisher-id>
            </medium>
        </role>
        <execution>single</execution>
        <webservice>
            <base>http://www.exchangerate-api.com/</base>
            <paths>
                <path>usd</path>
                <path>jpy</path>
                <path>publisher.results.Price</path>
            </paths>
            <keys>
                <key>k</key>
            </keys>
            <values>
                <value>PQkn3-quzTZ-PNDav</value>
            </values>
            <format>XML</format>
            <results>
                <result>
                    <result-name>YenPrice</result-name>
                    <type>single</type>
                    <query>null</query>
                    <index>null</index>
                    <filter>null</filter>
                </result>
            </results>
        </webservice>
</component>

<output>
        <cooperation>
            <output-type>web</output-type>
            <output-params>
                <title>Shopping Compare Coordination Mashup Result</title>
                <table border=2>
                    <tr>
                        <td colspan=2>Guest</td>
                        <td>cooperation.GuestID</td>
                    </tr>
                    <tr>
                        <td colspan=2>Title</td>
                        <td>publisher[003].results.ProductName</td>
                    </tr>
                    <tr>
                        <td>Store</td>
                        <td>publisher[003].results.StoreName</td>
                        <td>Link</td>
                        <td>publisher[003].results.StoreURL</td>
                    </tr>
                    <tr>
                        <td>Price(Yen)</td>
                        <td>publisher[004].results.YenPrice</td>
```

```
                                    <td>Price(USD)</td>
                                    <td>publisher[003].results.Price</td>
                              </tr>
                        </table>
                  </output-params>
            </cooperation>
      </output>
</project>
```

Meeting Point C-MAIDL Mashup Descriptions

```
<project>
      <component>
            <name>HostLocator</name>
            <role>
                  <publisher>
                        <publisher-id>001</publisher-id>
                  </publisher>
            </role>
            <execution>single</execution>
            <mobileapplication>
                  <mode>passive</mode>
                  <intent>
                        <intent-name>com.prach.GPSLocator</intent-name>
                        <intent-extra>
                              <extra-name>MODE</extra-name>
                              <extras>
                                    <extra>PASSIVE</extra>
                              </extras>
                              <extra-name>TYPE</extra-name>
                              <extras>
                                    <extra>null</extra>
                              </extras>
                        </intent-extra>
                  </intent>
                  <results>
                        <result>
                              <result-name>Latitude</result-name>
                              <type>single</type>
                              <value>LAT</value>
                              <filter>null</filter>
                        </result>
                        <result>
                              <result-name>Longitude</result-name>
                              <type>single</type>
                              <value>LNG</value>
                              <filter>null</filter>
                        </result>
                  </results>
            </mobileapplication>
      </component>
      <component>
```

```
        <name>GuestLocation</name>
        <role>
              <publisher>
                    <publisher-id>002</publisher-id>
              </publisher>
        </role>
        <execution>single</execution>
        <cooperation>
              <mode>passive</mode>
              <mobileapplication>
                    <app-name>gpsapp://x-callback-url/getLocation?</app-name>
                    <app-params>
                          <param-name>mode</param-name>
                          <values>
                                <value>current</value>
                          </values>
                    </app-params>
              </mobileapplication>
              <results>
                    <result>
                          <result-name>Latitude</result-name>
                          <type>single</type>
                          <value>lat</value>
                          <filter>null</filter>
                    </result>
                    <result>
                          <result-name>Longitude</result-name>
                          <type>single</type>
                          <value>lng</value>
                          <filter>null</filter>
                    </result>
              </results>
        </cooperation>
  </component>
  <arithmetic>
        <name>MiddleLat</name>
        <role>
              <medium>
                    <subscriber-id>001</subscriber-id>
                    <subscriber-id>002</subscriber-id>
                    <publisher-id>003</publisher-id>
              </medium>
        </role>
        <execution>single</execution>
        <operation>average</operation>
        <values>
              <value>HostLocator.results.Longitude</value>
              <value>GuestLocator.results.Longitude</value>
        </values>
        <results>
              <result>
                    <result-name>MiddleLatResult</result-name>
                    <type>single</type>
```

```
                <value>arithmetic.calculation</value>
                <filter>null</filter>
            </result>
        </results>
    </arithmetic>
    <arithmetic>
        <name>MiddleLng</name>
        <role>
            <medium>
                <subscriber-id>001</subscriber-id>
                <subscriber-id>002</subscriber-id>
                <publisher-id>004</publisher-id>
            </medium>
        </role>
        <execution>single</execution>
        <operation>average</operation>
        <values>
            <value>HostLocator.results.Longitude</value>
            <value>GuestLocator.results.Longitude</value>
        </values>
        <results>
            <result>
                <result-name>MiddleLngResult</result-name>
                <type>single</type>
                <value>arithmetic.calculation</value>
                <filter>null</filter>
            </result>
        </results>
    </arithmetic>
    <component>
        <name>GourNavi</name>
        <role>
            <medium>
                <subscriber-id>003</subscriber-id>
                <subscriber-id>004</subscriber-id>
                <publisher-id>005</publisher-id>
            </medium>
        </role>
        <execution>single</execution>
        <webservice>
            <base>http://api.gnavi.co.jp/</base>
            <paths>
                <path>ver1</path>
                <path>RestSearchAPI</path>
            </paths>
            <keys>
                <key>keyid</key>
                <key>input_coordinates_mode</key>
                <key>coordinates_mode</key>
                <key>latitude</key>
                <key>longitude</key>
                <key>hit_per_page</key>
                <key>range</key>
```

```
            </keys>
            <values>
                  <value>10d9098dba2f680c748de5b03b28940d</value>
                  <value>2</value>
                  <value>2</value>
                  <value>publisher[002].results.MiddleLatResult</value>
                  <value>publisher[003].results.MiddleLngResult</value>
                  <value>8</value>
                  <value>2</value>
            </values>
            <format>XML</format>
            <results>
                  <result>
                        <result-name>Name</result-name>
                        <type>multiple</type>
                        <query>//name</query>
                        <index>null</index>
                        <filter>null</filter>
                  </result>
                  <result>
                        <result-name>Category</result-name>
                        <type>multiple</type>
                        <query>//category</query>
                        <index>null</index>
                        <filter>null</filter>
                  </result>
                  <result>
                        <result-name>Latitude</result-name>
                        <type>multiple</type>
                        <query>//latitude</query>
                        <index>null</index>
                        <filter>null</filter>
                  </result>
                  <result>
                        <result-name>Longitude</result-name>
                        <type>multiple</type>
                        <query>//longitude</query>
                        <index>null</index>
                        <filter>null</filter>
                  </result>
            </results>
      </webservice>
</component>
<arithmetic>
      <name>DistanceCalculator</name>
      <role>
            <subscriber>
                  <subscriber-id>003</subscriber-id>
                  <subscriber-id>004</subscriber-id>
                  <subscriber-id>005</subscriber-id>
            </subscriber>
      </role>
      <execution>multiple</execution>
```

```
            <operation>gpsdistance</operation>
            <values>
                  <value>publisher[002].results.MiddleLatResult</value>
                  <value>publisher[003].results.MiddleLngResult</value>
                  <value>publisher[004].results.Latitude</value>
                  <value>publisher[004].results.Longitude</value>
            </values>
            <results>
                  <result>
                        <result-name>Distance</result-name>
                        <type>multiple</type>
                        <value>arithmatic.calculation</value>
                        <filter>null</filter>
                  </result>
            </results>
      </arithmetic>
      <output>
            <cooperation>
                  <feature-id>map</feature-id>
                  <feature-extras>
                        <points>
                              <array>
                              <loop>
                                    <object>
                                          <name>name</name>
                                          <value>GourNavi.results.Name</value>
                                          <name>category</name>
                                          <value>GourNavi.results.Category</value>
                                          <name>latitude</name>
                                          <value>GourNavi.results.Latitude</value>
                                          <name>longitude</name>
                                          <value>GourNavi.results.Longitude</value>
                                          <name>distance</name>
                                          <value>DistanceCalculator.results.Distance</value>
                                          <name>unit</name>
                                          <value>km</value>
                                    </object>
                              </loop>
                              </array>
                        </points>
                  </feature-extras>
            </cooperation>
      </output>
</project>
```

XLIMA and MEDAL XML

To demonstrate the syntax of XLIMA and MEDAL, we provide two XLIMA and one MEDAL XML description files used in the sample scenario of event-driven mashup composition that we presented in section 6.4.

Automatic Email Translation Mashup Application

XLIMA Description file for Email Proxy Component

```
<XLIMA>
    <name>New Email Notification Proxy</name>
    <description>
        This is a XLIMA sample file to describe event-driven mobile mashup component.
        When a new email received, K9 email client will send intent broadcasting.
        This component intercept that event and transform to  on_email_received event.
    </description>

    <abstraction>
        <component id="com.korawit.component.email">
            <event id="on_email_received">
                <parameters>
                    <param id="from" datatype="string" />
                    <param id="subject" datatype="string" />
                    <param id="content" datatype="string" />
                </parameters>
                <trasformation>
                    <filter parameter="from" type="contain"
                        value="userinterface.uiForm" action="raise" />
                    <filter parameter="subject" type="contain"
                        value="userinterface.uiSubject" action="raise" />
                </trasformation>
            </event>
        </component>
    </abstraction>
    <implementation>
        <mobileapplication id="com.fsck.k9">
            <listener id="K9EmailListener">
                <protocol name="intent" type="broadcast">
                    <intent-filter>
                        <action>com.fsck.k9.intent.action.EMAIL_RECEIVED</action>
                        <data-scheme>email</data-scheme>
                        <extras>
                            <extra id="subject"
                              name="com.fsck.k9.intent.extra.SUBJECT" datatype="string" />
                            <extra id="from"
                              name="com.fsck.k9.intent.extra.FROM" datatype="string" />
                            <extra id="preview"
                              name="com.fsck.k9.intent.extra.PREVIEW" datatype="string" />
                        </extras>
                    </intent-filter>
```

```
                </protocol>
            </listener>
        </mobileapplication>
        <event-binding trigger="K9EmailListener" event="on_email_received">
            <mapping rule="direct" source="K9EmailListener.subject"
                  target="event.on_email_received.subject" />
            <mapping rule="direct" source="K9EmailListener.from"
                  target="event.on_email_received.from" />
            <mapping rule="direct" source="K9EmailListener.preview"
                  target="event.on_email_received.content" />
        </event-binding>
    </implementation>
    <userinterface>
        <element id="uiFrom" datatype="string" uitype="EditText">
            <display-name>From Filter</display-name>
            <default>korawit@gmail.com</default>
        </element>
        <element id="uiSubject" datatype="string" uitype="EditText">
            <display-name>Subject Filter</display-name>
            <default-value>""</default-value>
        </element>
    </userinterface>
</XLIMA>
```

XLIMA Description file for Translate Proxy Component

```
<XLIMA>
    <identification>Translation Proxy</identification>
    <description>
        This is a XLIMA sample file to describe event-driven mobile mashup component.
        Translation proxy is a Web service proxy component. It receive input text and
        translate
        into designed language.
    </description>
    <abstraction>
        <component id="com.korawit.mashup.component.translate">
            <operation id="translate">
                <parameters>
                    <param id="text" datatype="string" />
                </parameters>
            </operation>
            <event id="on_translated">
                <parameters>
                    <param id="output" datatype="string" />
                </parameters>
            </event>
        </component>
    </abstraction>
    <implementation>
        <webservice id="wsTranslate">
            <entrypoint url="http://mymemory.translated.net/api/get">
                <protocol name="http" type="rest">
```

```
                    <query>
                        <key name="q" urlencode="UTF-8"/>
                        <key name="langpair" urlencode="UTF-8"/>
                    </query>
                    <results>
                        <result id="translateText">
                            <value rule="json" path="/responseData[0]/translatedText" />
                        </result>
                    </results>
                </protocol>
            </entrypoint>
        </webservice>
    </implementation>
    <bindings>
        <binding type="invoke" trigger="translate" action="wsTranslate">
         <mapping rule="direct" source="translate.text" target="wsTranslate.q" />
         <mapping rule="direct" source="userinterface.uiLang" target="wsTranslate.langpair" />
        </binding>
        <binding type="event" trigger="wsTranslate" action="on_translated">
            <mapping rule="direct" source="wsTranslate.translatedText"
                  target="event.on_translated.output" />
        </binding>
    </bindings>
    <userinterface>
        <element id="uiLang" datatype="string" uitype="EditText">
            <display-name>Language Pair</display-name>
            <default>ja|en</default>
        </element>
        <element id="uiPreview" datatype="string" uitype="EditText">
            <display-name>Preview</display-name>
            <default>ja|en</default>
            <value rule="direct" source="wsTranslate.translatedText" trigger="wsTranslate" />
        </element>
    </userinterface>
</XLIMA>
```

MEDAL Description file for Automatic Email Translation Mashup

```
<MEDAL>
<identification>
      <mashupid>com.korawit.mashup.emailtranslation</mashupid>
      <name>Automatic Email Translation</name>
</identification>
<document>
A sample scenario of even-driven mobile mashup that listens to new incoming email, translate
the email content into a apecific language, finally save the translated text to a note app.
</document>
<declaration>
      <components>
            <component id="com.korawit.mashup.component.email"
                  alias="K9Email"
                        url="http://localhost/mobilemashup/cpr/k9email_proxy.xlima">
```

```
            <component id="com.korawit.mashup.component.translate"
                  alias="Translator"
                        url="http://localhost/mobilemashup/cpr/translate_proxy.xlima">
            <component id="com.korawit.mashup.component.note"
                  alias="Evernote" url="http://localhost/mobilemashup/cpr/note_proxy.xlima">
      </components>
</declaration>

<mashup>
      <listener id="emailListener" alias="OnEmailReceived"
            publisher="K9Email" event="on_email_received">
            <handler event="OnEmailReceived">
                  <invocation component="Translator" operation="translate" >
                        <mapping mode="single">
                              <direct src="OnEmailReceived.content" dest="translate.text" />
                        </mapping>
                  </invocation>
            </handler>
      </listener>
      <listener id="translationListener" alias="OnTranslated"
            publisher="Translator" event="on_translated">
            <handler event="OnTranslated">
                  <invocation component="Evernote" operation="addnew" >
                        <mapping mode="single">
                              <direct src="OnTranslated.output" dest="addnew.text" />
                        </mapping>
                  </invocation>
            </handler>
      </listener>
</mashup>
</MEDAL>
```