A Framework for Composing SOAP, Non-SOAP and Non-Web Services

Jonathan Lee, *Senior Member, IEEE Computer Society*, Shin-Jie Lee, and Ping-Feng Wang

Abstract—Recently, there is a trend on developing mobile applications based on service-oriented architecture in numerous application domains, such as telematics and smart home. Although efforts have been made on developing composite SOAP services, little emphasis has been put on invoking and composing a combination of SOAP, non-SOAP, and non-web services into a composite process to execute complex tasks on various mobile devices. Main challenges are two-fold: one is how to invoke and compose heterogeneous web services with various protocols and content types, including SOAP, RESTful, and OSGi services; and the other is how to integrate non-web services, like web contents and mobile applications, into a composite service process. In this work, we propose an approach to invoking and composing SOAP, non-SOAP, and non-web services with two key features: an extended BPEL engine bundled with adapters to enable direct invocation and composition of SOAP, RESTful and OSGi services based on Adapter pattern; and two transformation mechanisms devised to enable conversion of web contents and Android activities into OSGi services. In the experimental evaluations, we demonstrate network traffic and turnaround time of our approach are better than those of the traditional ones.

Index Terms—Heterogeneous service composition, web service, service composition engine

1 Introduction

Recently, there is a trend on developing mobile applications based on service-oriented architecture (SOA) in numerous application domains, such as telematics [1], business [2], smart home [3], [4] and internet of things (IOT) [5]. In IOT, for example, heterogeneous web services are supposed to be integrated to provide composite services. These heterogeneous services may include Simple Object Access Protocol (SOAP) [22] services, RESTful services [29], and OSGi services [30]. SOAP is a protocol specification for exchanging structured information during the implementation of web services, and SOAP services can be employed to build composite web services using Business Process Execution Language (BPEL) [6]. RESTful services are also considered an architectural style that can be used to construct software for clients to request services. An increasing number of Internet services have been developed as RESTful services to provide easier interfaces for displaying object features and communicating with external services. OSGi technology provides an open service platform for service installation, activation, and management in devices. OSGi services support point-to-point remote service delivery programs that enable dynamical binding, assembly, and execution of device services.

- *J. Lee is with the Department of Computer Science and Information Engineering, National Taiwan University, No. 1, Sec. 4, Roosevelt Rd., Taipei 10617, Taiwan. E-mail: jlee@csie.ntu.edu.tw.*
- *S.-J. Lee is with the Computer and Network Center, National Cheng Kung University, Tainan, Taiwan. E-mail: jielee@mail.ncku.edu.tw.*
- *P.-F. Wang is with the Department of Computer Science and Information Engineering, National Central University, Zhongli, Taiwan. E-mail: 93542011@cc.ncu.edu.tw.*

Manuscript received 25 Nov. 2013; revised 13 Feb. 2014; accepted 18 Feb. 2014. Date of publication 10 Mar. 2014; date of current version 10 Apr. 2015. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSC.2014.2310213

Although efforts have been made on developing composite SOAP services [6], [7], little emphasis has been put on invoking and composing a combination of SOAP, non-SOAP, and non-web services into a composite process in order to complete complex tasks on a variety of mobile devices. The challenges can be best explained from the following two perspectives:

- How to invoke and compose heterogeneous web services with various protocols and content types, such as SOAP, RESTful, and OSGi services? A composite process may involve invocations of SOAP, RESTful and OSGi services over SOAP protocol, HTTP and Java method call, respectively. In addition, it also involves a variety of message content types, such as SOAP, JSON, YAML, Protocol Buffer and Java objects, in composing the heterogeneous web services.
- How to integrate non-web services, including web contents and mobile applications, into a composite web service process? Non-web services in a mobile environment typically consist of web contents and mobile applications. Web contents are usually in the form of HTML documents. The program execution entry points of mobile applications are usually developed as GUI components, e.g., Activities in Android platform. Both of these two non-web services are difficult to be composed by a service composition engine, like a BPEL engine.

In this work, we propose a framework for invoking and composing SOAP, non-SOAP and non-web services on mobile devices with two key features: a BPEL engine extended and bundled with adapters to enable the direct invocation and composition of SOAP, RESTful and OSGi services based on Adapter pattern, which provides a flexible mechanism for adding newly developed adapters for

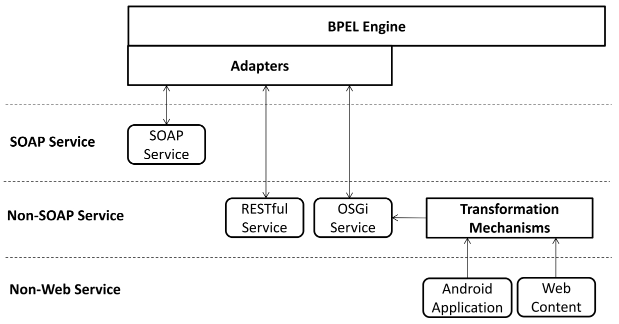

Fig. 1. Conceptual model of the framework for invoking and composing SOAP, non-SOAP and non-web services.

invoking some other kinds of services without modifying the core BPEL engine; and two transformation mechanisms devised to enable the transformation of web contents and Android activities into OSGi services that can be composed by the extended BPEL engine.

Fig. 1 shows the conceptual model of the framework for invoking and composing SOAP, non-SOAP and non-web services. Three adapters for invoking SOAP, RESTful, and OSGi services are developed to extend our BPEL engine, and two transformation mechanisms for converting web contents and Android activities are undergone a conversion into OSGi services before being invoked and composed at runtime by the extended BPEL engine through adapters.

This paper is organized as follows. We discuss the related work in Section 2. Section 3 describes fully the proposed framework. In Section 4, we conduct two experimental evaluations. Finally, in Section 5, we summarize the contributions of the proposed framework.

2 Related Work

It is widely accepted that combining multiple web services into a composite service is more beneficial to users than finding a complex and preparatory atomic service that satisfy a special request [17], [18], [26], [27], [28]. The resulted composite services can be used as atomic services by themselves in other service compositions to satisfy clients requests. BPEL4WS [6] provides a mixture of block-structured and graph-structured process models, and variables associated with message types can be specified as input or output variables to invoke, receive, and reply web services.

Recently, BPEL has been extended to support modeling the composition of heterogeneous web services, such as RESTful and OSGi services. REST [29] (REpresentational State Transfer) is a style of software architecture for distributing hypermedia systems such as the World Wide Web. REST defines a set of architectural principles [29] by which one can design web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages.

In [8], Curbera et al. offer a service-composite development model for composing RESTful services. During the invocations of RESTful services, the messages returned from a service are stored as BPEL process variables. However, how to transform the messages to different content types for follow-up service invocations is unclear.

In [16], He proposes a composite system for integrating both SOAP and RESTful services with a hybrid orchestration based on a BPEL engine and a REST orchestration engine. The composite service workflow is divided into two kinds of sub-workflows according to the types (SOAP or RESTful) of services to be composed. However, the message transformation between the SOAP and RESTful is not discussed.

In [11], Farokhi et al. propose a framework, called MDCHeS, to support dynamic composition and to use both SOAP-based and RESTful web services simultaneously in composite services with three different views: data, process, and component view. However, the approach, like the other approaches, does not discuss the message transformation between the SOAP and RESTful.

In [33], Nitzsche et al. extended BPEL 2.0 with a WSDL-less interaction model, call BPEL^{light}, to enable coupling business logic and web service technology, including WSDL, by introducing a new and single type of interaction activity resuming all BPEL interaction activities. The main focus of the paper is to enable modeling processes or process fragments that can be reused and bound to specific service interfaces in any interface description language. Our approach, however, focuses more on how to model and implement the binding relationships among heterogeneous services.

In [31], [32], Pautasso proposed a process-based composition language for composing RESTful and traditional WSDL-based services based on BPEL. The local adapters (e.g., XSLT, JavaScript) are used to process the data and transform it to make it compatible with what the other service requires. In our approach, SOAP, RESTful and OSGi services are all supported for invocation while executing a composite process. Furthermore, SOAP, JSON, YAML, Protocol Buffer, and Java object messages returned from a service can be transformed into Java objects used as variables of a BPEL process. After the first transformation, the variables can then be transformed again into messages of different content types for follow-up service invocations.

3 Heterogeneous Service Composition Framework

In this work, an extended BPEL engine is developed by applying Adapter pattern [21] to compose SOAP, RESTful and OSGi services. Fig. 2 shows the system architecture of the heterogeneous service composition framework. A concise description of our BPEL engine is given below without going into details of its modules. The adapters and transformers will be further elaborated in the Sections 3.2, 3.3, 3.4 and 3.5.

The entry point of the BPEL engine is Composite Service Activator component with inputs of WSDL and BPEL documents. Deploy component deploys the documents with BPEL properties and uses Service Generator component to transform the BPEL process defined in the BPEL document into Java classes.

Reader component is in charge of generating a BPEL model by analyzing the WSDL and BPEL documents and instantiating the Java class. BPEL Manager component

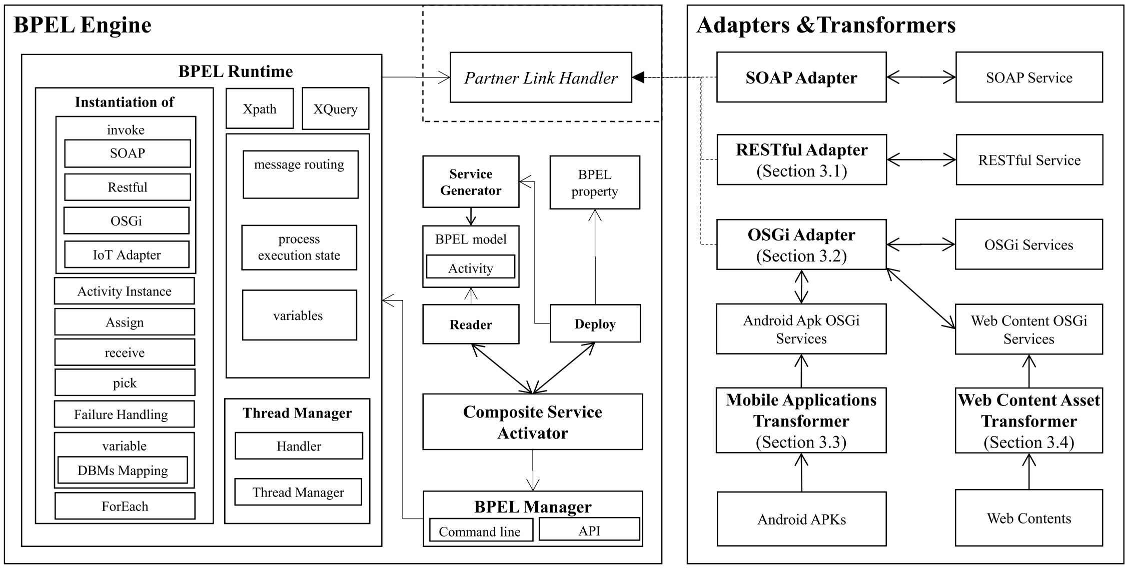

Fig. 2. The system architecture of the heterogeneous service composition framework.

starts the BPEL process by creating a BPEL process instance from the BPEL model with instances of the Java classes.

BPEL Runtime component runs BPEL processes at run time. A BPEL process consists of two kinds of activities: basic and structured. A basic activity describes elemental steps of the process behavior. A structured activity encodes control-flow logic and can contain other basic or structured activities recursively. Thread Manager component handles the multi-threads based on XPath and XQuery.

SOAP, RESTful, and OSGi Adapters are responsible for invoking SOAP, RESTful, and OSGi services, respectively. The three adapters implement the Partner Link Handler interface, which enables the BPEL engine to delegate service invocation behaviors to the services. Mobile Applications Transformer transforms Android APKs into OSGi services, and Web Content Asset Transformer transforms web contents into OSGi services.

Our BPEL engine, complied with BPEL version 2.0, is implemented in Java with 44 packages, 426 classes, 3,750 methods and a total of 93,609 lines of code. The adapters and transformers are implemented with 11 packages, 105 classes, 461 methods, and 6,453 lines of code.

3.1 Experimental Scenario

In this work, a smart living control system with a number of heterogeneous services is developed as the experimental environment for comparing our proposed approach with the traditional approach. The system obtains sensor data in order to calculate the predicted mean vote (PMV) indicator for temperature, humidity, illuminance, etc. The PMV system is supplemented with HVAC (heating, ventilation, and air conditioning) technology, lighting, and motorized roller blinds to provide an energy-saving living environment. We develop two smart living system scenarios by adopting the traditional approach and the proposed approach (see Figs. 3 and 4).

In the traditional approach (Fig. 3), RESTful services and web contents are wrapped as SOAP services and deployed in the server side. As for the Android activity running on Android platform, they cannot be directly invoked by the BPEL engine. Hence, a SOAP service is required as a proxy for retrieving the data generated from the activity. The steps of composing the services are as follows:

1) *Steps 1-3*. The web information about current weather information including temperature, humidity, sunlight, and illumination of the area are retrieved through an Android activity. After that, the activity sends the weather data to a SOAP service. The BPEL engine will retrieve the data through invoking the SOAP service.
2) *Step 4*. The BPEL engine sends the weather data to the calculated PMV SOAP service. Based on the temperature, humidity, and illumination value, the optimal parameter values for the HVAC technology, lighting, and motorized roller blinds will be determined and returned to the BPEL engine.
3) *Steps 5-7*. The RESTful services of Motorized roller blinds controller, Lighting Controller, and HVAC Controller will be invoked with the parameters of the optimal parameter values by the SOAP services.
4) *Step 8*. The BPEL engine invokes a SOAP service to collect the energy saving information. The SOAP

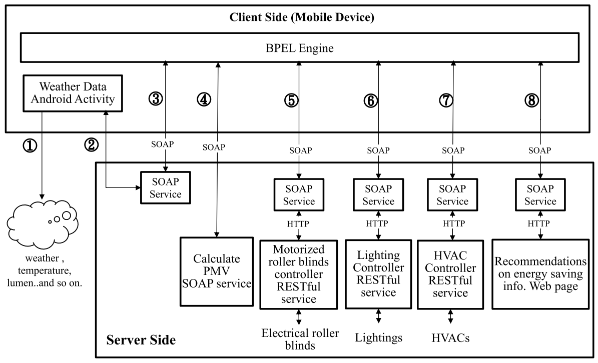
Client Side (Mobile Device)
BPEL Engine
Weather Data Android Activity
①
②
③
④
⑤
⑥
⑦
⑧
SOAP
SOAP Service
HTTP
weather , temperature, lumen..and so on.
Calculate PMV SOAP service
Motorized roller blinds controller RESTful service
Lighting Controller RESTful service
HVAC Controller RESTful service
Recommendations on energy saving info. Web page
Server Side
Electrical roller blinds
Lightings
HVACs

Fig. 3. The heterogeneous service compositions scenario by the traditional approach.

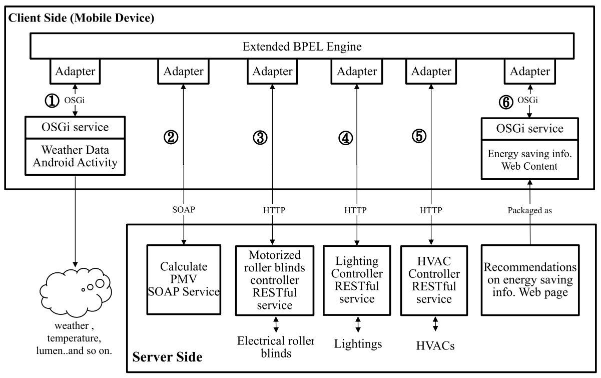
Client Side (Mobile Device)
Extended BPEL Engine
Adapter
Adapter
Adapter
Adapter
Adapter
Adapter
① OSGi
⑥ OSGi
OSGi service
Weather Data Android Activity
②
③
④
⑤
OSGi service
Energy saving info. Web Content
SOAP
HTTP
HTTP
HTTP
Packaged as
Calculate PMV SOAP Service
Motorized roller blinds controller RESTful service
Lighting Controller RESTful service
HVAC Controller RESTful service
Recommendations on energy saving info. Web page
weather , temperature, lumen..and so on.
Server Side
Electrical roller blinds
Lightings
HVACs

Fig. 4. The heterogeneous service compositions scenario by our approach.

service will parse the web page of the information and return the result back to the BPEL engine.

In our proposed approach (Fig. 4), the scenario includes an extended BPEL engine bundled with adapters that enable directly invocations of the heterogeneous services in the experimental environment (see Fig. 4). The steps of composing the services are as follows:

1) *Step 1.* As the Weather Data Android Activity is transformed into an OSGi bundle through the mobile applications transformer, the extended BPEL engine can directly invoke the OSGi service to retrieve the weather information.
2) *Step 2.* The extended BPEL engine sends the weather data to the calculated PMV SOAP service. Based on the temperature, humidity, and illumination value, the optimal parameter values for the HVAC technology, lighting, and motorized roller blinds will be determined and returned to the extended BPEL engine.
3) *Steps 3-5.* The RESTful services of Motorized roller blinds controller, Lighting Controller, and HVAC Controller will be directly invoked with the parameters of the optimal parameter values by the extended BPEL engine.
4) *Step 6.* The energy saving information web page will be transformed into an OSGi bundle through the web content asset transformer, and to be composed by the extended BPEL engine through the OSGi adapter.

3.2 Composing RESTful Services

- *Protocol*. The protocols that a RESTful service supports, e.g., HTTP.
- *Message format*. The message formats that a RESTful service supports, e.g., JSON, YAML, and protocol buffer.

```
<wsdl:service name="WeatherService">
  <wsdlrestful:address protocol="http" format="json"
      url="http://query.yahooapis.com/v1/public/yql?q=*" />
</wsdl:service>
```

Fig. 5. An example of describing a RESTful service.

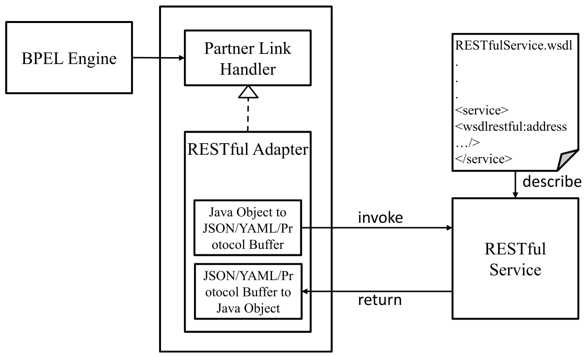

Fig. 6. RESTful adapter architecture.

- *Url*. The URL of a RESTful service.

The extension of WSDL for describing RESTful services includes protocols, urls, and message formats. Fig. 5 shows an example of describing a Weather RESTful service. The service supports HTTP protocol and is with the content format of JSON.

Fig. 6 shows the architecture of the RESTful service adapter. During the execution process, the extended BPEL engine parses the WSDL document of the RESTful service. The properties of the $<$ wsdlrestful:address $>$ elements are obtained from the WSDL. The protocol property determines the protocol of the RESTful service. The url property determines the network address of the RESTful service. The format property returns the content type of the RESTful service messages. When the messages return from the RESTful service, they are transformed into Java objects as runtime variables of the composite process through the JSON/YAML/Protocol Buffer to Java object module based on open source org.json and org.ho.yaml packages. Messages are converted into JSON, YAML or Protocol Buffer messages before they are sent as inputs to another web service.

3.3 Composing OSGi Services

- *type*. The type that an OSGi service locates, e.g., local.
- *ServiceName*. The service name that an OSGi services registers in the OSGi register.
- *Filter*. The filter is an optional property that identifies a concrete OSGi service.

In the framework, we extend WSDL to describe OSGi services including their types, service names and filters. Fig. 7 shows an example of describing an OSGi service. The WSDL describes that an OSGi TimerService is run on a local device and its service name registered is ntu.osgi.Timer.

Fig. 8 shows the architecture of the OSGi service adapter. By referring to the extended WSDL, the extended BPEL engine reads the properties of the OSGi service by using the OSGi adapter, which inherits a partner link handler interface. The adapter searches for the OSGi service through the

```
<wsdl:service name="TimerService">
  <wsdlosgi:address type="local" serviceName="ntu.osgi.Timer" filter="" />
</wsdl:service>
```

Fig. 7. An example of describing an OSGi service.

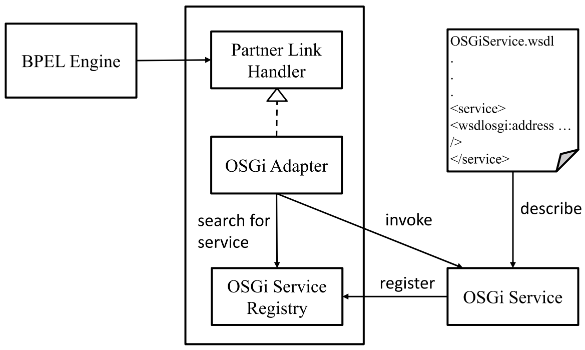

Fig. 8. OSGi adapter architecture.

OSGi registry to get the service's reference, and then invokes the services through method call.

3.4 Composing Mobile Applications

In this section, we describe a mobile application transformation mechanism that enables transforming Android Activities to OSGi bundles. The transformed OSGi bundles can be invoked and composed by the extended BPEL engine.

Fig. 9 shows the architecture of the mobile applications transformer. The inputs of the transformer include the project assets of an Android application that runs on an OSGi platform (Apache Felix) and several existing Android applications. The output is a new Android application together with several OSGi bundles that provide services for invoking the Activities of the existing Android applications. The project assets of developing an Android application include source code files (src), resource files (res), libraries (lib), and a Manifest file. The main functions of the transformer are as follows:

1) *Integrate resources.* In order to prevent the conflicts among the resource file names of the Android applications, all resource files (images, strings, layouts, etc.) will be renamed during the transformation process.
2) *Integrate source code.* The parts of the code that originally refer to the resource files will be modified to refer to the renamed resource files.
3) *Import libraries.* All of the libraries of each Android application will be copied into a new directory, and the new classpath file will include the references to these libraries.
4) *Integrate manifests.* The AndroidManifest.xml is a configuration file that contains Android application information and describes all package components, including the activities, services, and receivers. This function is to integrate the manifest files into a new one.
5) *Modify the Felix export packages.* To deploy the OSGi framework on the new Android application, the Apache Felix library (Felix.jar) was transformed into an OSGi bundle. The Context.startService() function is used to start (bind) the OSGi framework, and the Context.stopService() is used to stop (unbind) the OSGi framework. After the OSGi framework environment is established, OSGi bundles can be installed and uninstalled on the Android platform.

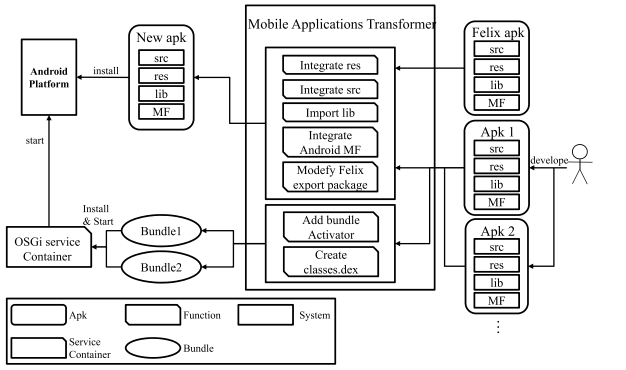

Fig. 9. Mobile applications transformer architecture.

The activities of transforming the Android applications into OSGi bundles are as follows:

1) *Add a bundle activator.* For each Android activity, an OSGi bundle activator class will be automatically generated for the invocation of the Android activity. The activator class includes five methods:
   - *void invokeActivity().* This method involves no input parameter value or returns value. Invoking this method will directly invoke the Android activity using Android Intent mechanism.
   - *void invokeActivity(byte[] bytes).* This method will invoke the Android activity with a parameter of a byte array.
   - *byte[] invokeActicityForResult().* When the method is invoked, it will invoke the Android activity and wait for the returned value from the activity.
   - *byte[] invokeActivityForResult(byte[] bytes).* This method will invoke the Android activity with a parameter of a byte array and wait for the returned value from the activity.
   - *byte[] invokeActivityForResult(Object obj).* This method will invoke the Android activity with a parameter of an object and wait for the returned value from the activity.
2) *Generate manifest.* This activity is to generate a manifest file for each generated OSGi bundle. The following elements must be declared in the manifest file.
3) *Generate and add classes.dex.* The OSGi bundles will be compiled into a dex file to enable their executions on the Android platform.

3.5 Composing Web Contents

As web contents in HTML are usually readable only for human and are with diverse information, the web content asset transformer enables a system designer to identify the assets in the web pages and transform them to OSGi services at design time so as to be composed in a composite process at runtime.

Fig. 10 shows the architecture of the web content asset transformer. The transformer consists of three components: a webpage list manager, a data parser and a bundle packager. The webpage list manager enables a user to identify and select the assets in the web pages, such

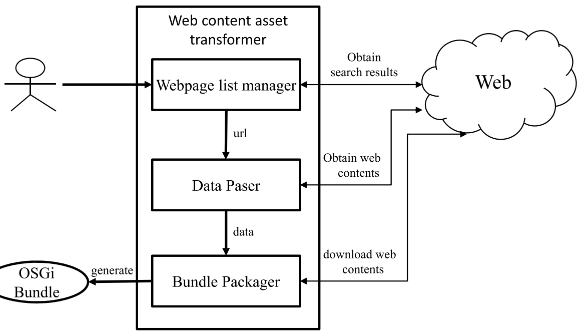

Fig. 10. Web content asset transformer architecture.

as images or texts. The data parser parses the selected web pages and extract the assets. The bundle packager transforms the extracted assets into an OSGi bundle. In the following, we detail how to extract web content assets and how to transform these assets into an OSGi bundle.

As the web pages are selected by the user, the data parser converts the HTML documents into document object model (DOM) trees, and parses the following four types of assets with their corresponding HTML tags:

- Text: <h1> ... <h6>, <p>, <textarea>, <blockquote>, <pre>
- Image: <img>
- Video: <video>
- Audio: <audio>

These assets are stored in four Java bean objects (Fig. 11). These Java beans all inherit the same parent class MultimediaInfo. The attributes of each class are as follows:

- MultimediaInfo: id, title
- TextInfo: content
- ImageInfo: alt, width, height, filename, src, content
- VideoInfo: poster, width, height, filename, src, content
- AudioInfo: filename, src, content

Attribute id keeps the identifier of each asset by following the universally unique identifier (UUID) standard. Attribute title stores any ”title”-related data attribute. Attribute content stores data with a byte array. Attributes width and height record the resolution of an image or a video. Attribute filename specifies the file name of each asset.

After the web content assets are extracted by the data parser, the assets together with the serializations of the Java bean objects will be stored into a folder. Meanwhile, an OSGi bundle activator class that implements the interface showed in Fig. 12 will be generated in the folder as well. The class provides OSGi services after the folder is packaged into an OSGi bundle. The exposed OSGi services include:

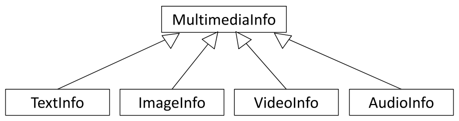

Fig. 11. Java bean objects for storing the web content assets.

```
pubilc interface DataBundle {
    String[]             getAllTexts();
    MultimediaInfo[]     getText(String query);
    String[]             getAllImages();
    MultimediaInfo[]     getImage(String title);
    String[]             getAllVideos();
    MultimediaInfo[]     getVideo(String title);
    String[]             getAllAudios();
    MultimediaInfo[]     getAudio(String title);
}
```

Fig. 12. The interface of web content bundle.

- getAllTexts(): String[], to retrieve the titles of the TextInfo Java bean objects as a string array.
- getText(String query): MultimediaInfo[], to retrieve the assets through keyword matchings for the titles.
- getAllImages(): MultimediaInfo[], to retrieve all images.
- getImage (String title): MultimediaInfo[], to retrieve the images of the specified title.
- getAllVideos(): MultimediaInfo[], to retrieve all videos.
- getVideo(String title): MultimediaInfo[], to retrieve the videos of the specified title.
- getAllAudios(): MultimediaInfo[], to retrieve all audio files
- getAudio(String title): MultimediaInfo[], to retrieve the audio files of the specified title.

After the folder is ready for packaging, the Bundle Packager automatically generates the manifest file into the folder and starts compressing the folder into a jar file as an OSGi bundle. The manifest file contains the information of the exported OSGi service interfaces of the bundle.

4 Experimental Evaluation

In this section, we conduct experimental evaluations to demonstrate that both network traffic and turnaround time of the proposed framework are better than the traditional frameworks.

4.1 Definition of Evaluation Metrics

To evaluate the performance of our approach and the traditional framework, two metrics are adopted: (1) network traffic: data transmitted over a network; and (2) turnaround time: the total time between the submission of a program for execution and the return of the complete output to the customer.

The network traffic metric is formally defined in Definition 1.

Definition 1. *Given a set of heterogeneous services* $S = \{\{ss_1, \ldots, ss_n\}, \{rs_1, \ldots, rs_m\}, \{os_1, \ldots, os_p\}\}$, *where a SOAP service is denoted as* ss_i, *a RESTful service is denoted as* rs_i *and an OSGi service is denoted as* os_i. *In the traditional approach, the network traffics generated in invoking SOAP, RESTful and OSGi services are denoted as* N^{tb}_{soap}, N^{tb}_{rest} *and* N^{tb}_{osgi}, *respectively. The total network traffic is denoted as* N^{tb}_{total} *and is calculated by equation (1). The adapter that sends and receives messages between RESTful services and SOAP services, denoted as* $r2s_i$, *and the*

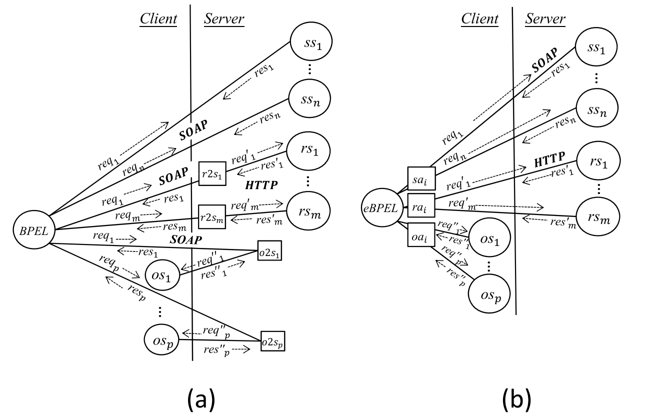

Fig. 13. The network traffic generated between client and server through (a) the traditional framework, and (b) the proposed framework.

adapter that sends and receives messages between OSGi services and SOAP services, denoted as $o2s_i$. In the proposed approach, the network traffics generated in invoking SOAP and RESTful services are denoted as N^{eb}_{soap} and N^{eb}_{rest}, respectively. Because OSGi services are locally invoked, no network traffics are generated. The adapters that sends and receives messages between the extended BPEL engine and SOAP services, RESTful services and OSGi services are denoted as sa_i, ra_i and oa_i, respectively. The total network traffic is denoted as N^{eb}_{total} and is calculated by equation (1):

$$N^{tb}_{total} = N^{tb}_{soap} + N^{tb}_{rest} + N^{tb}_{osgi}, \tag{1}$$

$$N^{tb}_{soap} = \sum_{i=1}^{n} req_i + \sum_{i=1}^{n} res_i, \tag{2}$$

$$N^{tb}_{rest} = \sum_{i=1}^{m} req_i + \sum_{i=1}^{m} res_i, \tag{3}$$

$$N^{tb}_{osgi} = \sum_{i=1}^{p} req_i + \sum_{i=1}^{p} res_i + \sum_{i=1}^{p} req''_i + \sum_{i=1}^{p} res''_i, \tag{4}$$

$$N^{eb}_{total} = N^{eb}_{soap} + N^{eb}_{rest}, \tag{5}$$

$$N^{eb}_{soap} = \sum_{i=1}^{n} req_i + \sum_{i=1}^{n} res_i, \tag{6}$$

$$N^{eb}_{rest} = \sum_{i=1}^{m} req'_i + \sum_{i=1}^{m} res'_i. \tag{7}$$

req_i is the size of the request message sent to a SOAP service ss_i, and res_i is the size of the reply message returned from the SOAP service. req'_i is the size of the request message sent from the adapter $r2s_i$ to the RESTful service rs_i, and res'_i is the size of the reply message returned from the RESTful service. req''_i is the size of the request message sent to the OSGi service os_i from the adapter $o2s_i$, and res''_i is the size of the reply message returned by OSGi service.

As shown in Fig. 13a, the BPEL engine sends request messages to remote SOAP services, RESTful services, and OSGi services and receives response messages from the invoked serviced through the traditional approach. SOAP services located at server-side are built as adapters for the service invocation by the BEPL engine. The total network traffic, N^{tb}_{total}, is the sum of network traffics in between the BPEL engine and SOAP services (Equations (2) and (3)), and in between OSGi services and adapters (Equation (4)).

As depicted in Fig. 13b, the extended BPEL engine directly invokes remote heterogeneous services through the bundled adapters sa_i, ra_i and oa_i. The total network traffic N^{eb}_{total} is the sum of the network traffics in between the BPEL engine, SOAP services, and the RESTful services.

Definition 2 defines turnaround time that is a metric derived from the transmission time and processing time. The transmission time is the amount of time for sending a message from the start node of a link to its destination node of a link. The processing time is the amount of time for executing a service after the service receives an invocation request.

Definition 2. *Given a set of heterogeneous services $S = \{\{ss_1, \ldots, ss_n\}, \{rs_1, \ldots, rs_n\}, \{os_1, \ldots, os_n\}\}$, where a SOAP service is denoted as ss_i, a RESTful service is denoted as rs_i and an OSGi service is denoted as os_i. In the traditional approach, the total turnaround time is denoted as R^{tb}_{total}. The turnaround time for invoking the SOAP, RESTful and OSGi services are denoted as R^{tb}_{soap}, R^{tb}_{rest}, and R^{tb}_{osgi}, respectively. In the proposed approach, the total turnaround time is denoted as R^{eb}. The turnaround time for invoking the SOAP, RESTful and OSGi services are denoted as R^{eb}_{soap}, R^{eb}_{rest} and R^{eb}_{osgi}, respectively:*

$$R^{tb}_{total} = R^{tb}_{soap} + R^{tb}_{rest} + R^{tb}_{osgi}, \tag{8}$$

$$R^{tb}_{soap} = \sum_{i=1}^{n} \frac{req_i + res_i}{b_{c,s_i}} + \left(\sum_{i=1}^{n} \frac{sp_i}{c_{s_i}} + \sum_{i=1}^{n} \frac{sa_i}{c_c} \right), \tag{9}$$

$$\begin{aligned} R^{tb}_{rest} = &\left(\sum_{i=1}^{m} \frac{req_i + res_i}{b_{c,s_i}} + \sum_{i=1}^{m} \frac{req'_i + res'_i}{b_{s,s_i}} \right) \\ &+ \left(\sum_{i=1}^{m} \frac{sp_i}{c_{s_i}} + \sum_{i=1}^{m} \frac{r2s_i}{c_{s_i}} \right), \end{aligned} \tag{10}$$

$$\begin{aligned} R^{tb}_{osgi} = &\left(\sum_{i=1}^{p} \frac{req_i + res_i}{b_{c,s_i}} + \sum_{i=1}^{p} \frac{req''_i + res''_i}{b_{s,s_i}} \right) \\ &+ \left(\sum_{i=1}^{p} \frac{sp_i}{c_c} + \sum_{i=1}^{p} \frac{o2s_i}{c_{s_i}} \right), \end{aligned} \tag{11}$$

$$R^{eb}_{total} = R^{eb}_{soap} + R^{eb}_{rest} + R^{eb}_{osgi}, \tag{12}$$

$$R^{eb}_{soap} = \sum_{i=1}^{n} \frac{req_i + res_i}{b_{c,s_i}} + \left(\sum_{i=1}^{n} \frac{sp_i}{c_{s_i}} + \sum_{i=1}^{n} \frac{sap_i}{c_c} \right), \tag{13}$$

$$R^{eb}_{rest} = \sum_{i=1}^{m} \frac{req'_i + res'_i}{b_{s,s_i}} + \left(\sum_{i=1}^{m} \frac{sp_i}{c_{s_i}} + \sum_{i=1}^{m} \frac{rap_i}{c_c} \right), \tag{14}$$

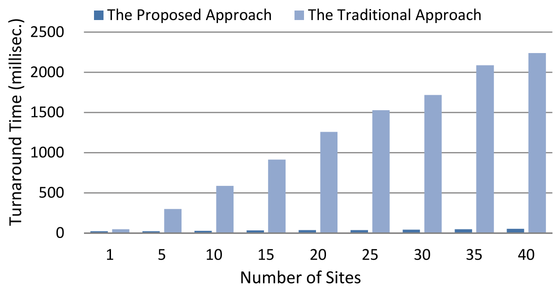

Fig. 14. The turnaround time of invoking weather data android activity.

$$R^{eb}_{osgi} = \sum_{i=1}^{p} \frac{req''_i + res''_i}{b_{s,s_i}} + \left(\sum_{i=1}^{p} \frac{sp_i}{c_c} + \sum_{i=1}^{p} \frac{oap_i}{c_c} \right). \qquad (15)$$

b_{c,s_i} is the bandwidth between the client and the server s_i. b_{s,s_i} is the bandwidth between the servers. sp_i is the number of time units required for invoking the services on s_i. ap_i is the number of time units needed to process the results obtained from service invocations. sap_i, rap_i, oap_i are the numbers of time units required for invoking SOAP, RESTful and OSGi services, respectively. c_c is the computing capability of the client, and c_{s_i} is the computing capability of the service s_i.

In the traditional approach, the turnaround time of SOAP service, R^{tb}_{soap}, refers to the amount of time spent in sending and receiving messages for service invocations, $\sum_{i=1}^{n} \frac{req_i + res_i}{b_{c,s_i}}$, that is, it counts the time of execution process of the related services in server and adapters in client, $\sum_{i=1}^{n} \frac{sp_i}{c_{s_i}} + \sum_{i=1}^{n} \frac{sa_i}{c_c}$. The turnaround time of RESTful service, R^{tb}_{rest}, is the amount of time spent in sending and receiving messages for service and adapter invocations, $\sum_{i=1}^{m} \frac{req_i + res_i}{b_{c,s_i}} + \sum_{i=1}^{m} \frac{req'_i + res'_i}{b_{s,s_i}}$, that is, it counts the time of execution process of the related adapters and services, $\sum_{i=1}^{m} \frac{sp_i}{c_{s_i}} + \sum_{i=1}^{m} \frac{r2s_i}{c_{s_i}}$. The turnaround time of OSGi service, R^{tb}_{osgi}, is the amount of time spent in sending and receiving messages for service and adapter invocations, $\sum_{i=1}^{p} \frac{req_i + res_i}{b_{c,s_i}} + sum_{i=1}^{p} \frac{req''_i + res''_i}{b_{s,s_i}}$, that is, it counts the time of execution process of related adapters and services, $\sum_{i=1}^{p} \frac{sp_i}{c_c} + \sum_{i=1}^{p} \frac{o2s_i}{c_{s_i}}$.

In our approach, the turnaround time of SOAP service, R^{eb}_{soap}, consists of the time in sending and receiving messages for service invocations, $\sum_{i=1}^{n} \frac{req_i + res_i}{b_{c,s_i}}$, and of the time of execution process of related adapters and services, $\sum_{i=1}^{n} \frac{sp_i}{c_{s_i}} + \sum_{i=1}^{n} \frac{sap_i}{c_c}$. The turnaround time of RESTful service, R^{eb}_{rest}, consists of the time in sending and receiving messages for service invocations, $\sum_{i=1}^{m} \frac{req'_i + res'_i}{b_{s,s_i}}$, and of the time of execution processes of related adapters and services, $\sum_{i=1}^{m} \frac{sp_i}{c_{s_i}} + \sum_{i=1}^{m} \frac{rap_i}{c_c}$. The turnaround time of OSGi service, R^{eb}_{osgi}, consists of the time in sending and receiving messages for service invocations, $\sum_{i=1}^{p} \frac{req''_i + res''_i}{b_{s,s_i}}$, and of the time of execution processes of related adapters and services, $\sum_{i=1}^{p} \frac{sp_i}{c_c} + \sum_{i=1}^{p} \frac{oap_i}{c_c}$.

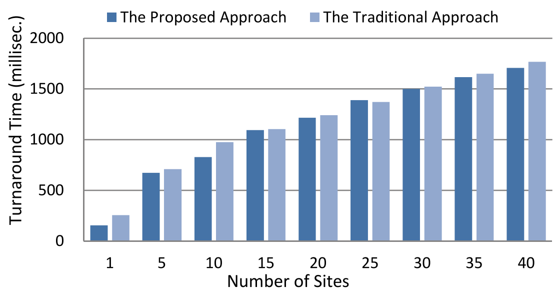

Fig. 15. The turnaround time of invoking PMV soap service.

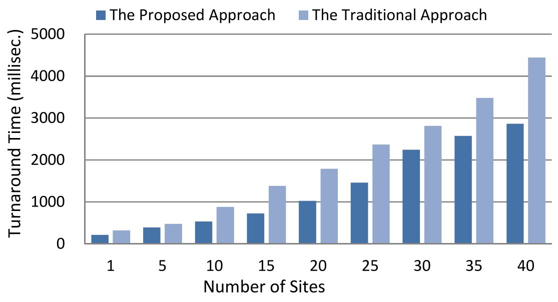

Fig. 16. The turnaround time of invoking motorized roller blinds controller RESTful service.

4.2 Experimental Results

In the experimental environment, the client-side mobile device is Android 2.2 platform with 1 GHz Cortex A8 CPU and 512 MB memory. The server runs Java virtual machine 6.0 and the OSGi platform is Apache Felix. We calculate the turnaround time and the network traffic by adopting our approach and the traditional approach over a wireless network with a 256 KB/s bandwidth which is estimated and controlled by a bandwidth controller software. A site denotes a smart living control environment as described in Section 3.1. There are up to 40 sites in the experiment. For each kind of service with a certain number of sites, the turnaround time is the sum of the time gathered for all the sites. Each approach is applied and evaluated 2,000 times, and the average of the turnaround time is calculated and depicted in Figs. 14, 15, 16, 17, 18, and 19.

Figs. 14 and 19 show the turnaround time of invoking weather data Android activity and receiving recommendation web contents, respectively. Noted that there is no need to transfer the data by the server-side SOAP service in our approach, which gives us an advantage of less turnaround time over the traditional one. Fig. 15 indicates that the

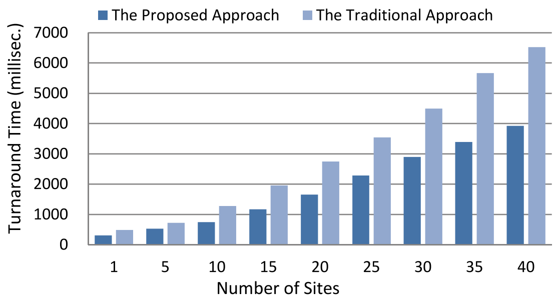

Fig. 17. The turnaround time of invoking lighting controller RESTful service.

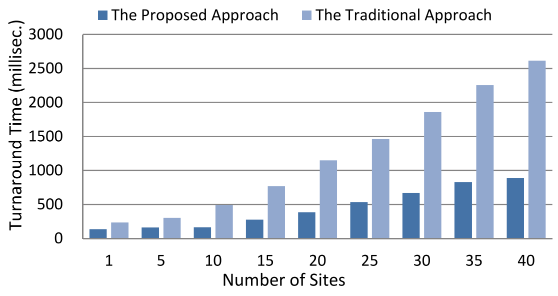

Fig. 18. The turnaround time of invoking HAVC controller RESTful service.

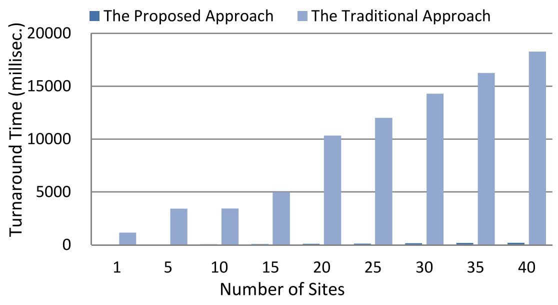

Fig. 19. The turnaround time of retrieving energy saving recommendations web content.

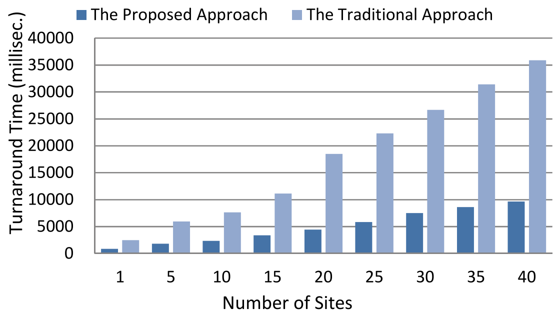

Fig. 20. Total turnaround time.

TABLE 1
Paired Samples T-Test of the Total Turnaround Time

Number of sites	The Traditional Approach (T1)	The Proposed Approach (T2)
1	2501.66	873.32
5	5945.74	1818.43
10	7658.31	2347.08
15	11125.02	3368.5
20	18510.11	4427.86
25	22278.28	5837.8
30	26699.12	7520.16
35	31390.8	8629.89
40	35867.63	9634.35
T1-T2	Using α = 0.05 and degrees offreedom=8, t=4.463 (>2.306)	

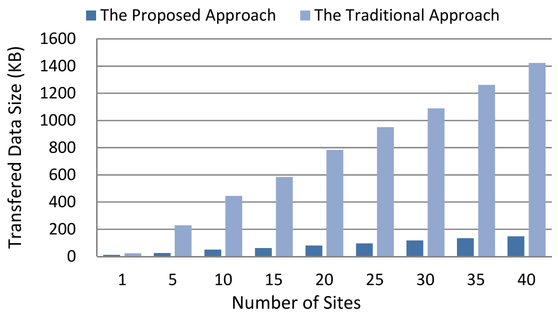

Fig. 21. Network traffic generated between client and servers.

TABLE 2
Paired Samples T-Test of the Network Traffics

Number of sites	The Traditional Approach (N1)	The Proposed Approach (N2)
1	24.72827	11.78827
5	230.51311	25.6406
10	445.47998	50.76045
15	584.72878	62.7459
20	783.23135	81.49608
25	951.99891	95.89873
30	1089.49562	117.6038
35	1262.91215	134.16151
40	1423.30293	148.17036
N1-N2	Using α = 0.05 and degrees offreedom=8, t=4.753 (>2.306)	

differences between our approach and the traditional one are small in invoking PMV SOAP service.

The experimental results shown in Figs. 16, 17 and 18 point out that the turnaround time of invoking the controller RESTful services in our approach is less than those of the traditional one due to the request message data are directly transferred in between the extended BPEL engine and the RESTful services without being transferred through server-side SOAP services. Fig. 20 shows the total turnaround time of the two approaches. In the experiment, the turnaround time is largely influenced by the transferred data size. Table 1 shows that the t value for comparing the total turnaround time is 4.463 (>2.306) with degrees-of-freedom $= 8$ and $\alpha = 0.05$, which indicates that there is a significant difference between the total turnaround time of the two approaches.

The evaluation results shown in Fig. 21 indicate that our approach generates less network traffic than the traditional approach. The traditional approach requires server-side soap services to connect to the RESTful services and OSGi bundles. The traditional approach requires server-side soap services located in a server site (room) to connect to the RESTful services and OSGi bundles. As the number of the server sites (rooms) increases, our approach is significantly superior to the traditional approach on the network traffics. Table 2 shows that the t value for comparing the network traffic is 4.753 (>2.306) with degrees-of-freedom $= 8$ and $\alpha = 0.05$, which indicates that there is a significant difference between the average values of the network traffics of the two approaches.

5 Conclusion

This paper presents a framework for composing SOAP, RESTful, OSGi services, web contents and Android Activities. A BPEL engine is extended and bundled with adapters to enable the direct invocation and composition of SOAP, RESTful and OSGi services with heterogeneous content types: SOAP, JSON, YAML, Protocol Buffer and Java objects. Additionally, two transformation mechanisms are devised to enable the transformation of web contents and Android activities that are two typical assets on mobile devices into OSGi services so as to be composed by the extended BPEL engine.

We also conducted an experiment on a smart living control system to evaluate our approach. In our approach, the request and response messages of the invocations of RESTful services do not need to be transferred via server-side SOAP services. These messages are directly transferred in between the extended BPEL engine and the RESTful services. The experimental results show that our proposed approach generates less network traffic and spends less turnaround time than those of the traditional approaches.

With the extended BPEL engine based on Adapter pattern, SOAP, non-SOAP and non-web services can be integrated through a systematic way. Furthermore, the engine can be considered as a more suitable framework for composing heterogeneous services than the traditional ones from the perspective of resource consumptions in mobile environments.

ACKNOWLEDGMENTS

This research was sponsored by National Science Council (Taiwan) under the grant NSC 103-2622-E-002-017-CC1.

REFERENCES

[1] J. Lee, S.-J. Lee, H.-M. Chen, and K.-H. Hsu, "Itinerary-Based Mobile Agent as a Basis for Distributed OSGi Services," *IEEE Trans. Computers*, vol. 62, no. 10, pp. 1988-2000, Oct. 2013.

[2] Y. Natchetoi, V. Kaufman, and A. Shapiro, "Service-Oriented Architecture for Mobile Applications," *Proc. ACM First Int'l Workshop Software Architectures and Mobility*, 2008.

[3] D.-M. Han and J.-H. Lim, "Design and Implementation of Smart Home Energy Management Systems Based on Zigbee," *IEEE Trans. Consumer Electronics*, vol. 56, no. 3, pp. 1417-1425, Aug. 2010.

[4] M. Darianian and M.P. Michael, "Smart Home Mobile RFID-Based Internet-of-Things Systems and Services," *Proc. Int'l Conf. Advanced Computer Theory and Eng.*, 2008.

[5] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Computer Networks*, vol. 54, no. 15, pp. 2787-2805, 2010.

[6] T. Andrews et al., "Business Process Execution Language for Web Service (BPEL4WS) 1.1," technical report, BEA Systems and International Business Machines Corporation and Microsoft Corporation and SAP AG and Siebel Systems, 2003.

[7] "W3C Member Submission, "*OWL-S: Semantic Markup for Web Services*, http://www.w3.org/Submission/OWL-S/, 2004.

[8] F. Curbera, M. Duftler, R. Khalaf, and D. Lovell, "Bite: Workflow Composition for the Web," *Proc. Fifth Int'l Conf. Service-Oriented Computing (ICSOC '07)*, pp. 94-106, 2007.

[9] R.P. Diaz Redondo, A. Fernandez Vilas, M. Ramos Cabrer, J.J. Pazos Arias, and M.R. Lopez, "Enhancing Residential Gateways: OSGi Service Composition," *IEEE Trans. Consumer Electronics*, vol. 53, no. 1, pp. 87-95, Feb. 2007.

[10] M. zur Muehlen, J.V. Nickerson, and K.D. Swenson, "Developing Web Services Choreography Standards: The Case of REST versus SOAP," *Decision Support System*, vol. 40, 2005.

[11] S. Farokhi, A. Ghaffari, H. Haghighi, and F. Shams, "MDCHeS: Model-Driven Dynamic Composition of Heterogeneous Service," *Int'l J. Comm., Network and System Sciences*, vol. 5, pp. 644-660, 2012.

[12] J. Niemoller, K. Vandikas, R. Levenshteyn, D. Schleicher, and F. Leymann, "Towards a Service Composition Language for Heterogeneous Service Environments," *Proc. 15th Int'l Conf. Intelligence in Next Generation Networks*, 2011.

[13] J. Lee, S.P. Ma, S.J. Lee, Y.C. Wang, and Y.Y. Lin, "Dynamic Service Composition: A Discovery-Based Approach," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 18, no. 2, pp. 199-222, Mar. 2008.

[14] J. Lee, S.J. Lee, H.M. Chen, and C.L. Wu, "Composing Web Services Enacted by Autonomous Agents through Agent-Centric Contract Net Protocol," *Information and Software Tech*, vol. 54, p. 951-967, 2012.

[15] Y.Y. Peng, S.P. Ma, and J. Lee, "REST2SOAP: A Framework to Integrate SOAP Services and RESTful Services," *Proc. IEEE Int'l Conf. Service-Oriented Computing and Applications (SOCA)*, 2009.

[16] K. He, "Integration and Orchestration of Heterogeneous Services," *Proc. IEEE Joint Conf. Pervasive Computing (JCPC)*, pp. 467-470, Dec. 2009.

[17] S. Dustdar and W. Schreiner, "A Survey on Web Services Composition," technical report, Distributed Systems Group, Technical Univ. Vienna, 2004..

[18] H. Cervantes and R.S. Hall, "Service Oriented Concepts and Technologies," *Service-Oriented Software System Engineering: Challenges and Practices*. Idea Group Publishing, 2005.

[19] R.P.D. Redondo, A.F. Vilas, M.R. Cabrer, J.J.P. Arias, J.G. Duque, and A. GilSolla, "Enhancing Residential Gateways: A Semantic OSGi Platform," *IEEE Intelligent Systems*, vol. 23, no. 1, pp. 32-40, Jan./Feb. 2008.

[20] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly, May 2007.

[21] E. Freeman, E. Robson, B. Bates, and K. Sierra, *Head First Design Patterns*. O'Reilly Media, Oct. 2004.

[22] M. Gudgin, M. Hadley, N. Mendelsohn, J.J. Moreau, and H.F. Nielsen, "Simple Object Access Protocol (SOAP) 1.2," *W3C*, http://www.w3.org/TR/soap12, Apr. 2007.

[23] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) Version 1.1," *W3C*, http://www.w3.org/TR/wsdl, Mar. 2001.

[24] "Knopflerfish 3," *OSGi R4*, http://www.knopflerfish.org/, 2014.

[25] *Eclipse*, http://www.eclipse.org/, 2014.

[26] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana, "The Next Step in Web Services," *Comm. ACM- Service-oriented computing*, vol. 46, no. 10, pp. 29-34, 2003.

[27] M.P. Papazoglou and D. Georgakopoulos, "Service-Oriented Computing: Introduction," *Comm. ACM- Service-oriented computing*, vol. 46, no. 10, pp. 24-28, 2003.

[28] J. Lee, S.-P. Ma, and A. Liu, *Service Life Cycle Tools and Technologies: Methods, Trends and Advances*. IGI Global, 2011.

[29] R. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," PhD thesis, Univ. of California, Irvine, 2000.

[30] "OSGi Alliance," *OSGi Service Platform, Core Specification, release 4*, http://www.osgi.org/, 2014.

[31] C. Pautasso, "A Flexible System for Visual Service Composition," PhD dissertation (No. 15608), Swiss Federal Institute of Technology in Zurich, July 2004.

[32] C. Pautasso and G. Alonso, "From Web Service Composition to Megaprogramming," *Proc. Fifth Int'l Conf. Technologies for E-Services (TES '04)*, Aug. 2004.

[33] J. Nitzsche, T. van Lessen, D. Karastoyanova, and F. Leymann, "BPEL$^{\text{Light}}$," *Proc. Fifth Int'l Conf. Business Process Management*, pp. 214-229, 2007.

Jonathan Lee received the PhD degree in computer science from Texas A&M University in 1993. He is a professor in the Department of Computer Science and Information Engineering at National Taiwan University (NTU) in Taiwan. He was the department chairman from 1999 to 2002 and was the director of Computer Center at National Central University from 2006 to 2012. His research interests include software engineering, service-oriented computing, and software engineering with computational intelligence. He has authored more than 100 journal articles and refereed conference papers. He was awarded IBM Shared University Research Award (2010), CIEE Electrical Engineering Outstanding Professor Award, NCU Distinguished Professor Award (2006-2013), and NCU Distinguished Research Award (2004). He also served as the program chairs of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005) and the 8th International Fuzzy Systems Association World Congress (IFSA 1999). He is a senior member of the IEEE Computer Society.

Shin-Jie Lee received the PhD degree in computer science and information engineering from National Central University in Taiwan in 2007. He is an assistant professor in the Computer and Network Center at National Cheng Kung University (NCKU) in Taiwan and holds joint appointments from the Department of Computer Science and Information Engineering at NCKU. His current research interests include agent-based software engineering and service-oriented computing.

Ping-Feng Wang is currently working toward the PhD degree in computer science and information engineering from National Central University, Taiwan. His current research interests include software engineering and service-oriented computing.