IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received May 31, 2017, accepted August 16, 2017, date of publication September 4, 2017, date of current version September 27, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2748622

Blueprint Flow: A Declarative Service Composition Framework for Cloud Applications

CHOONHWA LEE[1], CHENGYANG WANG[1], EUNSAM KIM[2], AND SUMI HELAL[3]
[1]Division of Computer Science and Engineering, Hanyang University, Seoul 133-791, South Korea
[2]Department of Computer Engineering, Hongik University, Seoul 121-791, South Korea
[3]CISE Department, University of Florida, Gainesville, FL 32611, USA

Corresponding author: Eunsam Kim (eskim@hongik.ac.kr)

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea, Funded by the Ministry of Science and ICT, under Grant 2017R1A2B4010395 and Grant 2016R1D1A1A09917396.

ABSTRACT Cloud applications provides users with services that can be accessed on demand through the Internet. Fertile service frameworks are considered one of the most critical ingredients for the envisaged benefits so as to further interactions among cloud computing resources and application components. Such foundations should lead to the proliferation of new innovative services and applications. The research community has been exploring the Open Service Gateway initiative's (OSGi) potential as a top candidate for cloud application platforms. Although the current OSGi specification provides some level of support for dynamic service discovery, tracking, and composition, more should be done to be able to adequately address the need for diverse interaction patterns for cloud applications. This paper introduces a novel service framework built upon OSGi platforms that supports a directed-acyclic-graph style composition of constituent services. Given a declarative blueprint of service interconnections and interactions, the framework can find and assemble corresponding component services to form a real application. Our proposal can enable a realistic topology of service component interlinkings beyond linear chaining interactions as supported by the status quo. The design, implementation details, and validation results of our workflow-based service composition framework architecture are discussed in the paper.

INDEX TERMS Distributed computing, middleware, service computing, service-oriented systems engineering, software as a service.

I. INTRODUCTION

Cloud computing has been hailed as a great innovation to change the way computing resources and services are provided and used. According to the paradigm, computing resources, such as computation, storage, and applications, can now be delivered over the network on an on-demand basis [1], [2]. While some early cloud service offerings turned out to be very successful, there remain several technical challenges for a wider acceptance. One main obstacle is securing a suitable service execution platform upon which services can be discovered and used by one another. In other words, the proliferation of diverse cloud applications must be backed up by a fertile foundation where value-added services can emerge out of existing ones with rather limited functionality.

Recently, there has been a research move to explore the possibility of using OSGi (Open Service Gateway initiative) framework as the base of cloud computing platform [3]. The OSGi platform is a Java-based component framework in which components can be installed, updated, and uninstalled at runtime [4]. A basic component in OSGi is called a bundle. A bundle is a standard Java archive with a manifest file containing metadata. All OSGi services are supposed to be implemented as a regular Java interface inside bundles and registered with the interface and metadata in the OSGi service registry. A cloud application can be viewed as a collection of services that offer some computing resources or functionalities. The framework facilitates a service to look up and interact with other services. However, when it comes to the issue of service composition, the OSGi platform falls short of full-fledged composition features. This lack of composition support is viewed as a serious drawback to elevate itself as a dominant service platform for the upcoming cloud era [5], considering the fact that service reusability is key to promoting the scalability, productivity, and innovation expected to enable cloud computing vision. In this article, we present a novel service composition framework that supports a directed-acyclic-graph style composition of OSGi component services. Based on a declarative workflow definition language, our framework can model and instantiate a topology of versatile service interconnections beyond the linear interactions supported by the current OSGi specification.

The remainder of this paper is organized as follows. Section II first motivates our research for service composition frameworks and reviews the state-of-the-art technology for the topic. Then, the paper presents our architectural design of a workflow-based service composition framework in Section III. The effectiveness of the proposed architecture is evaluated via the validation and experimental study presented in Section IV. Section V discusses previous research efforts relevant to service composition and workflow technologies, clarifying the differences of our approach and others. Finally, Section VI concludes the paper.

II. MOTIVATIONAL SCENARIO AND STATE-OF-THE-ART TECHNOLOGY

In order to start presenting our approach to a service composition framework for cloud applications, let us first consider a sample composition scenario for smart homes which involves a number of component services representing various sensors and actuators.

Crispin's home is instrumented with various sensors and actuators, and smart home application acts as a hub to coordinate their operations and controls. On a sunny day, the smart home application opens windows, turns off dehumidifiers, and informs Crispin of the temperature and UV intensity by pushing a message to his mobile device. In the case of a rainy day, the application closes windows. Additionally, it turns on dehumidifiers, and then informs Crispin of the weather condition.

In this scenario, the smart home application is a coordinating hub to facilitate interactions among various services representing the sensors and devices installed at home, including the followings.

- *WeatherSensor* service that provides weather-related information.
- *WindowOpen* service and *WindowClose* service that control the window switch.
- *DehumidifierOn* service and *DehumidifierOff* service that reduces and maintains indoor humidity level.
- *InformUser* service to inform the resident by sending a message.

Fig. 1 represents the sample workflow scenario that keeps comfortable settings for the resident. The pivotal point of the scenario is the orchestrating ability that governs the execution of individual services representing sensors and devices at home. The first step to realize this scenario would be to get hold of an adequate service hosting and execution platform where services can find and call one another. Having been one of the most prominent service frameworks, OSGi is now being poised as an ideal fit for the foundation of cloud service executions and interactions [3], [6]. To explore its potential as the future cloud computing platform, we have based our service workflow system architecture on the

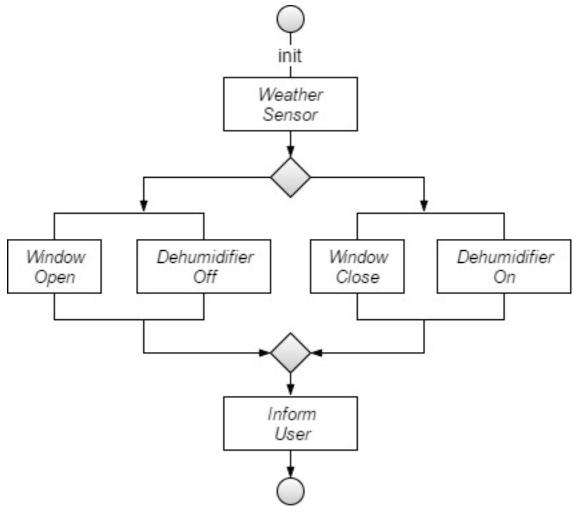


FIGURE 1. Service composition scenario for smart home.

OSGi platform. We first review the status quo of the technology advancements especially for promoting and managing service interactions, based on which our system is built.

On top of its service execution platform, OSGi provides a facility to foster service compositions and interactions at different abstraction levels. Early fundamental composition supports are found at Service Tracker Service and Wire Admin Service. Using these, application developers have to manage service-level dependencies manually. Wire Admin Service is a system service that is used to control a wiring topology of OSGi applications. Using the service, OSGi services can be chained together to form a pipeline over which data flow from one end to the other. One limitation of Wire Admin Service is that the producer-consumer design pattern can only be supported in a hard-coded way, which would require changes at the code level later in case of a wiring topology change. Service Tracker Service provides a means for service availability tracking beyond the low-level event mechanism of OSGi platform, notifying potential consumers of when a service in question becomes available, unavailable, or modified.

OSGi specification has evolved to provide more advanced service management mechanisms such as Blueprint Container Service and Declarative Service. They are intended to ease the complexity of building a composite application and managing dependencies among its components. Declarative Service allows developers to manage service dependencies in a declarative way [7], [8]. Thanks to the system service, service binding does not have to be performed imperatively in advance. In contrast to the hard-wring of Wire Admin Service, service compositions can now be declaratively described in an XML file. A service component in the description is annotated with *binder* and *unbinder* methods that are called for its wiring and unwiring in the event of changes in service availability. The Blueprint Container specification defines a dependency injection framework for OSGi environments. It provides features for the lifecycle managements and configurations of component instances as well as interactions with the service registry. In other words, it is a Spring-like framework with extra support for OSGi services. Blueprint Container Service also uses an XML composition file which describes dependencies between component services without requiring any hard-writing. A primary difference between Blueprint Container Service and Declarative Service is that the former is an injection solution that uses proxies to mask OSGi service dynamism, while the latter uses a cascading approach which is more like a component model with dependency management responsibility. Neither of them supports complex application topologies like directed-acyclic-graph topologies. Lastly, iPOJO is a service component model aiming to simplify OSGi applications development [9], [10]. An iPOJO component is a Plain Old Java Object encapsulated in a container into which different handlers can be plugged at runtime. The handlers take care of non-functional aspects such as service dependencies and instance configurations.

To realize the scenario in Fig. 1, the aforementioned support of the OSGi specification may prove helpful to a certain extent. However, it is hardly regarded as sufficient to handle diverse composition scenarios inherent in cloud computing environments. For instance, a shortcoming of the OSGi Blueprint Container specification is found at its application component topology modeling. The Blueprint Container Service is unable to orchestrate the workflow of the smart home scenario in Fig. 1, leaving hard-wiring of component services as the only option; hardcode is required to build flow paths among service procedures and handle data transfers along the paths. Therefore, we propose Blueprint Flow Service that is a workflow-based composition engine to support DAG-style service compositions in OSGi environments which will help us overcome hard-wiring of component services.

III. WORKFLOW-BASED SERVICE COMPOSITION FRAMEWORK

To enable declarative service compositions in OSGi environments, we have designed a workflow-based composition middleware architecture named Blueprint Flow on top of the Blueprint Container specification. The specification is a dependency injection framework which is similar to Spring Dynamic Module with extra support for OSGi services [7]. It was designed to cope with the dynamism of OSGi environments where services can come and go at any time. Applications are associated with Blueprint XML definitions which are materialized and managed by a Blueprint container. The container has several managers whose primary role is to handle application component's explicit and/or implicit dependencies on other components of the application. Being responsible for managing the lifecycle of backing component instances, the managers are the centerpiece of the Blueprint Container specification. The system service supports an assembly of component instances into an application without any hard-wired, procedural code to wire up constituent services. Also, dependencies among the constituents are managed automatically.

As depicted in Fig. 2, there are three major types of managers in the Blueprint Container specification. A Bean manager provides Java objects representing a component instance with all the dependencies properly injected and configured. A Reference manager provides proxies to services registered with the OSGi service repository, while a Service manager handles the registration of component instance's services into the OSGi service repository.

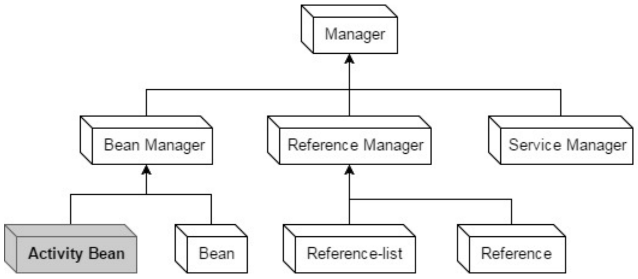


FIGURE 2. **Blueprint container managers.**

A. ACTIVITY BEAN

A new type of Activity Bean, which is our extension of the Blueprint Bean, is introduced in Fig. 2 where it is shown alongside the original bean. Activity Beans are a basic building block that comprises a workflow of services, collectively playing the role of an orchestrator that controls and manages a composition and execution process. Fig. 3 diagrams the relationship of various types of activities that are defined in our Blueprint Flow middleware architecture. Activity beans can be further classified as Component Activity Beans, Process Activity Beans, and Logic Activity Beans. Process Activity Beans include Sequence Process Activity and Parallel Process Activity, while Logic Activity Beans include Branch Activity and Loop Activity.

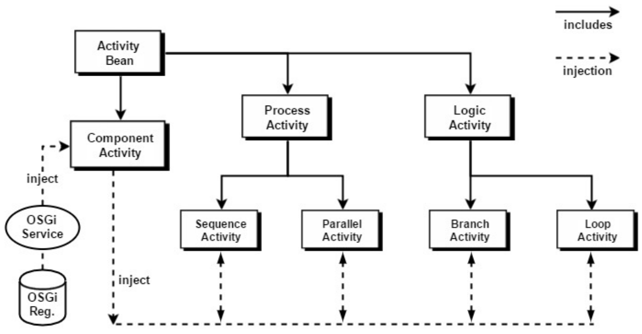


FIGURE 3. **Activity Beans in Blueprint Flow.**

1) COMPONENT ACTIVITY

A Component Activity (CA) is used to represent a component node of service composition DAGs. It is a basic building block of the Blueprint Flow composition graph. Several required items of the component should be injected into a CA, including OSGi service instance, required service interface, and the context of the component instance. The CA offers a typed interface for accessing the injected service instance. It also provides carry-on values, which corresponds to service context, being passed to the next stage of the workflow. Since it can be injected into other activities, a CA can be viewed as a basic construct for service workflows.

2) PROCESS ACTIVITY

A Process Activity (PA) provides a means to describe how components are wired and composed together. Basically, there are two patterns of the wiring: Sequence Process Activity (SPA) and Parallel Process Activity (PPA). An SPA allows a set of Activity Beans to be invoked in a sequential manner. An activity in the SPA set is executed right after the completion of its previous activity in the same set. A PPA describes a set of activities that are to be invoked simultaneously. It is used to model a point in a workflow process where a single thread of control forks into multiple flows in parallel, allowing activities to be executed simultaneously. All required activities should be imported into a PA via dependency injection, and the PA itself can be exported to and injected into other activities.

3) LOGIC ACTIVITY

A Logic Activity (LA) is flow control constructs including conditional branching and looping. An LA is further extended to define Branch Logic Activity (BLA) and Loop Logic Activity (LLA). A BLA controls the flow path of a workflow based on a certain condition. Among two or more branching alternatives, it chooses one as the next execution step. Modeling a while loop, an LLA repeats a set of activities, until a specified criterion is met. LAs can also be exported to other activities and injected into part of other PAs and LAs to form a workflow skeleton.

B. BLUEPRINT-BASED SERVICE WORKFLOW ENGINE

Our Blueprint Flow architecture is designed to implement DAG-style workflow patterns. The structural composition of an application is defined by an XML-based description file in which dependencies and interactions among constituent CAs, PAs, LAs, and OSGi services are stated. When instantiating a workflow, each node of it is to be backed up by a service component, whether simple or composite [11], [12]. According to OSGi Blueprint Container specification, component metadata hold all the necessary configuration information used by its manager. Fig. 4 depicts key components of our architectural design for Blueprint Flow. The diagram also illustrates a set of steps through which a workflow is instantiated into a real service composition.

(Step 1) Application developers define a service workflow using our XML-based DSL which is discussed in the next sub-section. The composite service definition file describes the application topology in terms of local Java objects, OSGi services, and Blueprint components. Given the definition file,

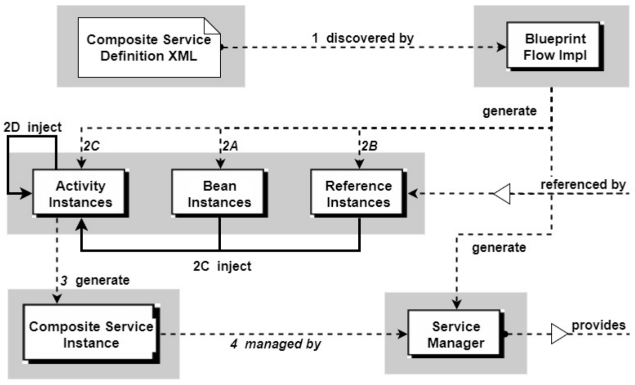


FIGURE 4. Workflow-based service composition of Blueprint Flow.

our Blueprint Flow engine (referred to as Blueprint Flow Impl in the figure) initiates the workflow creation.

(Step 2A) Native Java service objects in the workflow definition, can be instantiated and managed by a Blueprint Bean Manager.

(Step 2B) If the component in question is a service discovered from the OSGi service registry, our Blueprint Flow container sets up a service tracker for the OSGi service. When it becomes available, a proxy to the service is created and provided by a Reference Manager.

(Step 2C) Activity Beans may be involved in the workflow definition. An instantiation of activities may entails that components, including Java services created by Bean Managers and OSGi services tracked by Reference Managers, need to be injected into the activities.

(Step 2D) Blueprint Flow engine assembles Activity Beans together via dependency injection to form flow control and structural workflow patterns such as PAs and LAs.

(Step 3) As the next step, our Blueprint Flow container makes sure that all the necessary components are readily available and there is no dependency problem for them. It, then, binds and engages all the constituent components into the workflow skeleton, materializing the workflow definition.

(Step 4) At the end, the generated workflow-based composite service is registered back into the OSGi service registry with a canonical interface by Blueprint Service Manager. Consequently, others can make use of the composite service without having any knowledge about its components and structure.

C. SERVICE WORKFLOW LANGUAGE

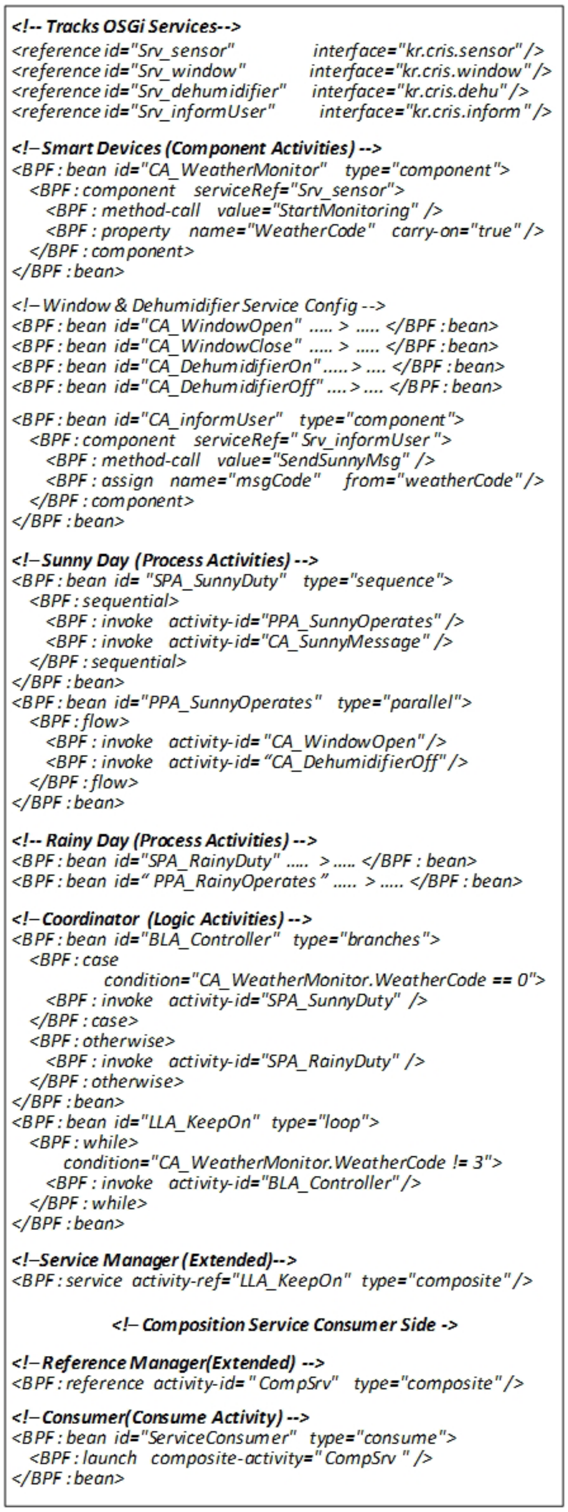
We have designed an XML-based orchestration language, named OSCL (OSGi Service Composition Language), that is used to specify service compositions in OSGi environments. As summarized in Table 1, the language has been designed to easily describe interactions among different kinds of component elements. As mentioned earlier in the paper, the Bean Manager of the Blueprint Container specification has been extended to be able to handle a new type of Blueprint Bean, i.e., Activity Bean. The Activity Bean is further subclassed to Component Activity, Process Activity, and Logic Activity.

TABLE 1. OSCL service composition language features.

	Element	Attributes	Description
CA	Bean	id, type	Describes a Component Activity when type = component
	Component	serviceRef	Declares the component details
	Property	name, value, carry-on	For setting and delivering context
	Assign	Name, from	Retrieves and assigns values from repository
	Method-call	Value	Accepts method name and provides strongly typed interface
SPA	Bean	id, type	Describes a Sequence Process Activity when type = sequence
	Sequential	-	Declares a sequential process
	Invoke	activity-id	Executes injected activity entities
PPA	Bean	id, type	Describes a Parallel Process Activity when type = parallel
	Flow	-	Declares a parallel process
	Invoke	activity-id	Executes injected activity entities
BLA	Bean	id, type	Describes a Branches Logic Activity when type = branches
	Case	Condition	Case construct for implementing branches logic
	Otherwise	-	Otherwise construct for implementing branches logic
	Invoke	activity-id	Executes injected activity entities
LLA	Bean	id, type	Describes a Loop Logic Activity when type = loop
	While	Condition	While construct for repeating
	Invoke	activity-id	Executes injected activity entities
Cons.	Bean	id, type	Describes an Activity Consumer when type = consume
	Launch	entry-activity	Launches a composite activity (application)
	Extended Service Manager		
	Service	activity-ref, type	Registers a composite service when type = composite
	Extended Reference Manager		
	Reference	activity-id, type	Provides a proxy to a registered composite service

The readers are referred to Table 1 for our OSCL design and to Fig. 5 for a sample workflow description.

Component Activity (CA) is the most basic element for a workflow description. A CA is an extension of Activity Bean as diagrammed in Fig. 3, consisting of a component instance (native Java services and OSGi services), a service interface, a context repository, and a CA executor. The component instance can be injected into a CA by defining the *serviceRef* attribute, and the properties of the service instance are injected using the *property* element. Interface information is injected by defining the *method-call* element. The *assign* element is used for fetching the property value from an upper-level repository. Finally, a CA executor will be created as an entry point of this CA.



```
<!-- Tracks OSGi Services-->
<reference id="Srv_sensor"          interface="kr.cris.sensor" />
<reference id="Srv_window"          interface="kr.cris.window" />
<reference id="Srv_dehumidifier"    interface="kr.cris.dehu" />
<reference id="Srv_informUser"       interface="kr.cris.inform" />

<!–Smart Devices (Component Activities) -->
<BPF : bean  id="CA_WeatherMonitor"   type="component">
  <BPF : component   serviceRef="Srv_sensor">
    <BPF : method-call   value="StartMonitoring" />
    <BPF : property   name="WeatherCode"   carry-on="true" />
  </BPF : component>
</BPF : bean>

<!–Window & Dehumidifier Service Config -->
<BPF : bean  id="CA_WindowOpen" ..... > ..... </BPF : bean>
<BPF : bean  id="CA_WindowClose" ..... > ..... </BPF : bean>
<BPF : bean  id="CA_DehumidifierOn" ..... > .... </BPF : bean>
<BPF : bean  id="CA_DehumidifierOff" .... > .... </BPF : bean>

<BPF : bean  id="CA_informUser"   type="component">
  <BPF : component   serviceRef=" Srv_informUser ">
    <BPF : method-call   value="SendSunnyMsg" />
    <BPF : assign   name="msgCode"    from="weatherCode" />
  </BPF : component>
</BPF : bean>

<!–Sunny Day (Process Activities) -->
<BPF : bean  id= "SPA_SunnyDuty"   type="sequence">
  <BPF : sequential>
    <BPF : invoke   activity-id="PPA_SunnyOperates" />
    <BPF : invoke   activity-id="CA_SunnyMessage" />
  </BPF : sequential>
</BPF : bean>
<BPF : bean  id="PPA_SunnyOperates"   type="parallel">
  <BPF : flow>
    <BPF : invoke   activity-id= "CA_WindowOpen" />
    <BPF : invoke   activity-id= “CA_DehumidifierOff" />
  </BPF : flow>
</BPF : bean>

<!-- Rainy Day (Process Activities) -->
<BPF : bean  id="SPA_RainyDuty" .....  > ..... </BPF : bean>
<BPF : bean  id=“ PPA_RainyOperates ” ..... > ..... </BPF : bean>

<!–Coordinator  (Logic Activities) -->
<BPF : bean  id="BLA_Controller"   type="branches">
  <BPF : case
           condition="CA_WeatherMonitor.WeatherCode == 0">
    <BPF : invoke   activity-id="SPA_SunnyDuty" />
  </BPF : case>
  <BPF : otherwise>
    <BPF : invoke   activity-id="SPA_RainyDuty" />
  </BPF : otherwise>
</BPF : bean>
<BPF : bean  id="LLA_KeepOn"  type="loop">
  <BPF : while>
      condition="CA_WeatherMonitor.WeatherCode != 3">
    <BPF : invoke   activity-id="BLA_Controller" />
  </BPF : while>
</BPF : bean>

<!–Service Manager (Extended)-->
<BPF : service  activity-ref="LLA_KeepOn"  type="composite" />

            <!– Composition Service Consumer Side ->

<!–Reference Manager(Extended) -->
<BPF : reference  activity-id=" CompSrv"   type="composite" />

<!–Consumer(Consume Activity) -->
<BPF : bean  id="ServiceConsumer"   type="consume">
  <BPF : launch   composite-activity=" CompSrv " />
</BPF : bean>
```

FIGURE 5. A sample Blueprint Flow description of smart home scenario.

Sequence Process Activity (SPA) is one sub-type of Process Activity. Activities defined inside the *sequential* element form an activity list that comprises a sequential process. The Executor runs the sequential process and invokes activities one by one in the defined order. Parallel Process Activity (PPA) is another sub-type of Process Activity. As shown in the table, activities inside the *flow* element form an activity set. A main difference from an SPA is that PPA Executor runs activity tasks in parallel to invoke the activities simultaneously.

Branch Logic Activity (BLA) is responsible for controlling the execution path of workflows based on branch conditions. As seen in the sample of Fig. 5, *BLA_Controller* represents a ternary operator, i.e., expression in *expression ? statement 1 : statement 2*. Process Generator analyses activities and condition to complete an executable ternary operator. It, then, determines which activity to invoke next.

Loop Logic Activity (LLA) is used to support a while loop. The way an LLA works is similar to a BLA. Process Generator creates an executable activity depending on the evaluation result of its ternary operator.

D. SERVICE & REFERENCE MANAGER EXTENSION

Once having been built, a composite service can be offered as a service to others, meaning that the service composition may be registered with the OSGi service registry, so that it can be consumed by others without knowing its internal details. In the original Blueprint Container specification, Service Manager is responsible for the registration of component services with the OSGi registry. We have extended Service Manager to support and manage the advertisement of workflow-based composite services. An example of it can be seen in the *service* element in Fig. 5.

Reference Manager of the Blueprint Container specification manages a proxy that represents a service object from the OSGi service registry. A composite service published to the registry by our extended service manager should also be handled as an OSGi service. The Reference Manager has also been extended to provide and manage a proxy to the workflow-based composite service as shown in the *reference* element at the end of Fig. 5.

Activities in the Blueprint Flow architecture provide strongly typed interfaces to make a connection to the executor within the activities. We introduced a Consume Activity as an entry to a composite service, as can been seen in the *launch* element in Fig. 5. Service consumers who want to run a workflow-based application do not have to understand the internal details. All consumers should do is to get a proxy of the composite service with the help of our extended reference manager, and associate a Consume Activity with the proxy.

IV. FRAMEWORK VALIDATION AND EXPERIMENTS

We have prototyped the architecture of Blueprint Flow composition framework based on Eclipse Equinox (http://www.eclipse.org/equinox) that provides a certified implementation of the OSGi Core specification and Apache Aries (http://aries.apache.org) which implements the Blueprint Container Specification [7].

In order to help to define workflow descriptions, we have also developed a workflow designer for Blueprint Flow by which application developers can easily specify service compositions. It allows the developers to build workflow diagrams using a combination of graphical building blocks as shown in Fig. 6. The blocks are connected together using links and forks, and the workflow being built is displayed graphically. After the necessary configuration of the building blocks, a XML-based OSCL definition file is generated to be fed into the Blueprint Flow engine.

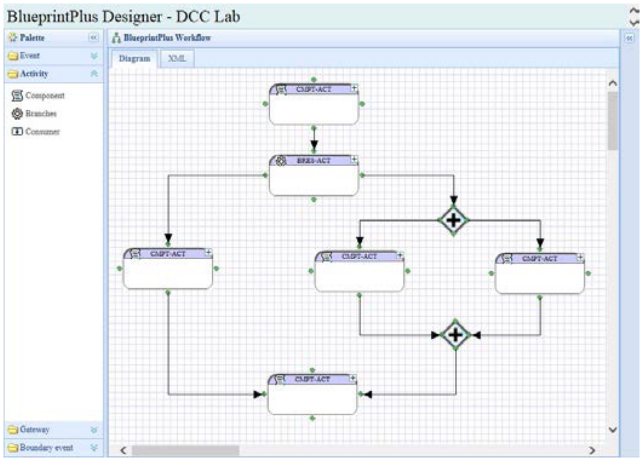


FIGURE 6. Screenshots of Blueprint Flow designer implementation.

A. WORKFLOW PATTERN EVALUATION

WS-BPEL (Web Services Business Process Execution Language) is an XML-based workflow language that enables process-oriented service composition [13]. As the standard language for Web Service composition, its workflow definition is centered around the notion of business processes used as the glue between interacting services. Programmers define how a business process that involves Web Services is executed using WS-BPEL language. BPEL primitives are used to invoke remote services, orchestrate process execution, and manage events and exceptions.

The task of a workflow language is to define case-driven business process by specifying, executing, and monitoring workflow models. We compare our OSCL language and BPEL from the standpoint of workflow pattern coverage. In the literature, workflow patterns are defined as a means of categorizing recurring problems and solutions in modeling business process [14]. Using the same evaluation criteria, we have checked what pattern coverage our OSCL offers. The criteria specifies a group of conditions that a workflow language in question has to fulfill in order to support a pattern. A pattern is said to be *supportive* (marked as "+" in Table 2), if the workflow language fully satisfies the evaluation criteria for the pattern. Otherwise, it is *unsupportive* (marked as "–" in the table). Table 2 also summarize the coverage of workflow patterns by our OSCL language in comparison with BPEL; Our Blueprint Flow framework provides better support for advanced branching and synchronization patterns and structural patterns. In contrast, it does not support cancellation patterns. The result shows that our OSCL languages provides better coverage for major patterns.

TABLE 2. Blueprint Flow workflow pattern comparison.

Workflow Pattern	BPEL	OSCL	Note
Basic Control Flow Patterns			
Sequence	+	+	*Supported by SPA*
Parallel Split	+	+	*Supported by PPA*
Synchronization	+	+	*Supported by PPA within SPA*
Exclusive Choice	+	+	*Supported by BLA*
Simple Merge	+	+	*Supported by BLA with PPA*
Advanced Branching and Synchronization Patterns			
Multi-choice	+	+	*Supported by BLA within BLA*
Synchronizing Merge	+	+	*Supported by PPA with SPA*
Multi-merge	-	+	*Supported by prototype bean*
Discriminator	-	-	*Not supported*
Structural Patterns			
Arbitrary Cycles	-	+	*Supported by PA with BLA*
Implicit Termination	+	+	*Supported by PA*
Patterns Involving Multiple Instances			
M.I. Without Synchronization	+	+	*Supported by prototype in CA*
M.I. With a Priori Design Time Knowledge	-	-	*Not supported*
M.I. With a Priori Runtime Knowledge	-	-	*Not supported*
M.I. Without a Priori Runtime Knowledge	-	-	*Not supported*
State-based Patterns			
Deferred Choice	+	+	*Supported by BLA*
Interleaved Parallel Routing	-	-	*Not supported*
Milestone	-	-	*Not supported*
Cancellation Patterns			
Cancel Activity	+	-	*Not supported*
Cancel Case	+	-	*Not supported*

B. SAMPLE SCENARIO DEMONSTRATION

To demonstrate the feasibility and effectiveness of our Blueprint-based composition framework, we choose a use-case scenario of smart home as presented in Section II. Being built on top of OSGi service platforms, all devices in this environment are registered with the OSGi service registry as services. Fig. 7 shows a screenshot from our prototype implementation of the smart home scenario involving a number of smart sensors and devices: a weather sensor, window controllers, door locks, and dehumidifiers. A workflow-based service composition can be specified by describing interactions of smart home devices and services in OSCL language. The OSCL description of the scenario is given in Fig. 5.

According to the scenario in Fig. 1, if *WeatherCode* from the weather sensor service is 0, it is a sunny day. Otherwise, it is rainy. First, the weather service checks the weather source and sets *WeatherCode* accordingly. A BLA accepts the code value and selects a PPA as the next executable node which contains two CAs (*WindowOpen* and *DehumidifierOff*). Finally, *informUser* CA is responsible for informing the resident of what has been done by sending a notification message.

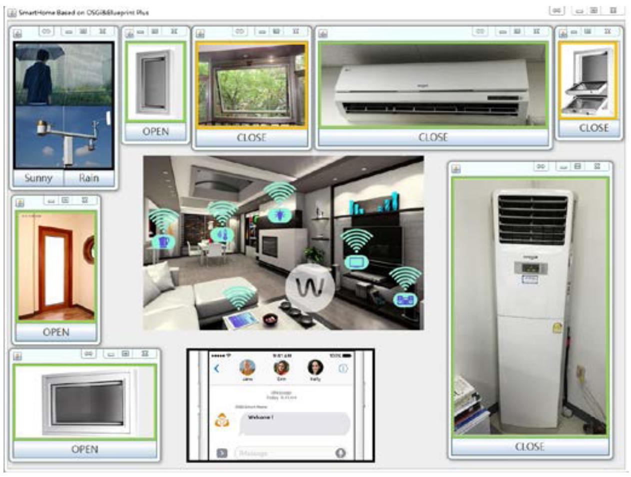


FIGURE 7. Screen shots of smart home scenario.

C. PERFORMANCE EXPERIMENTS

In order to evaluate the performance of our Blueprint Flow framework, we use a simple workflow scenario which involves two sequence and one multi-choice patterns. The test composite application is a calculator which consists of *Tokenizer* service, *Add* service, *Subtract* service, and *Print* service. The tokenizer service accepts an arithmetic expression, determines whether it is an addition or subtraction, and prepares operand values. The results of *Add/Subtract* service are passed on to *Print* service, so that it can deliver final results to users.

For BPEL experiments, we have chosen Apache Tomcat as the base container and Apache ODE (http://ode.apache.org) as a WS-BPEL engine. It is also noted that Eclipse Equinox (http://www.eclipse.org/equinox) and Apache Aries (http://aries.apache.org) are used as the base OSGi platform and Blueprint Container implementation, respectively. We measured the performance of both cases of WS-BPEL and Blueprint Flow in terms of CPU load, memory usage, and composition time.

CPU load is a measurement of the amount of computational work that a workflow requires. We ran the above experimental composition scenario 10 times using a machine with Intel Core i5-3470 CPU running at 3.2GHz. As shown in Fig. 8, the results show that CPU load is much lower and stable, when using our Blueprint Flow composition engine. The x axis of the graph indicates the separate runs of the workflow execution.

Lower memory usage should be desired, because it allows a faster execution of the workflow and permits the system to have room for other applications. Again, we have repeated the composition scenario 10 times again using WS-BPEL and Blueprint Flow engines. Fig. 9 compares memory usage for the two cases; the BPEL case shows the memory usage that varies from 8.37MB to 17.9MB. In the case of Blueprint

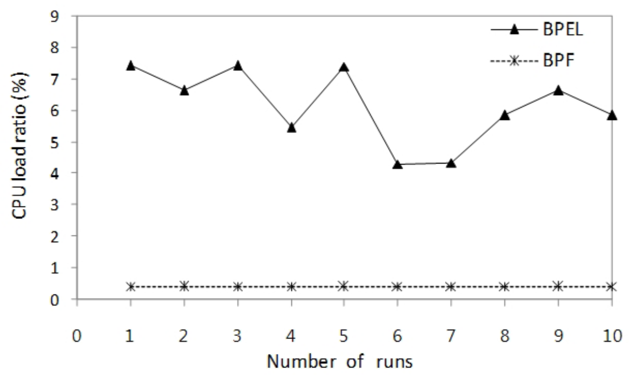


FIGURE 8. CPU load evaluation.

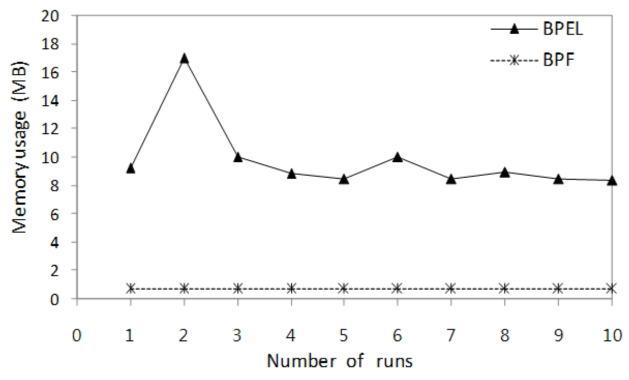


FIGURE 9. Memory usage evaluation.

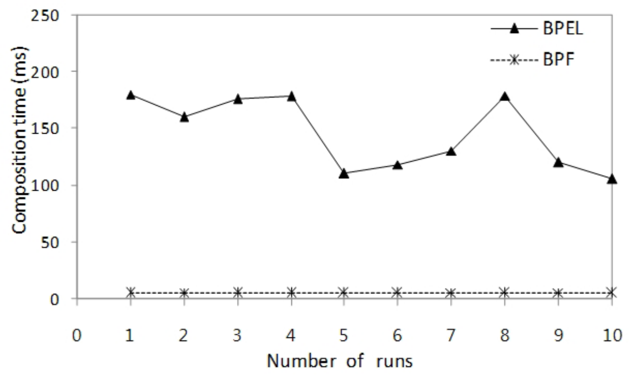


FIGURE 10. Composition time comparison.

Flow, the memory footprint is much lower, ranging from 0.620MB to 0.662MB.

A composition time indicates how fast a composition engine is able to compose and execute a workflow. We have run the same scenario repeatedly 10 times using WS-BPEL and Blueprint Flow frameworks to compare their workflow composition time. As plotted in Fig. 10, the average composition time for WS-BPEL engine is 145 ms, which is 30 times higher than the average time 4.58 ms for our Blueprint Flow engine.

V. RELATED WORK

When it comes to services technologies, Web Services technology might be viewed as the most mature and prominent [15]. Its discovery and delivery protocols along with a service description language can adequately describe, locate, and invoke individual services as an atomic unit. But the core technology itself does not provide functionalities for rich behavioral support for managing collaborations and interactions among services.

It is well known that there are two ways to build business processes: service orchestration and service choreography [16]. Service orchestration has a central orchestrator that coordinates interactions among component services. The orchestrator is responsible for invoking and combining the individual components. According to service choreography, there is no central orchestrator. Collaborations are specified by a global description of participating component services and their interactions. In other words, it is said that service choreographies are not executed by a central coordinator but enacted, when individual component services play their role. There have been several research efforts on service choreography which focus on service modeling, synthesizing of models, and process flow modeling [17]–[19].

As briefed in Section II, OSGi technology provides basic features for promoting service interactions and compositions. However, it lacks full-fledged service modeling and composition language supports. This void may be filled by WS-BPEL (Web Services Business Process Execution Language) that has establised itself as the standard for modeling and execution of Web Service orchestration [13]. WS-BPEL processes consist of Web Services components with a limited functionality that implement process activities. There have also been a number of recent efforts to extend the BPEL technology. AO4BPEL is an aspect-oriented extension to BPEL which addresses the issue of limited modularity in BPEL technology [20], [21]. AdaptiveBPEL is a technology which intends to meet the need of differentiated Web Services compositions [22]. An adaptation process is driven by policies, and a policy mediator is used to negotiate a composite policy. There is also some research on the development of REST Web Services composition using BPEL [23], [24]. RESTful interfaces can be directly used within a BPEL process and RESTful Web Services APIs can be implemented using BPEL with declarative constructs for publishing resources. Also interesting is a dialogue protocol proposed to support interactions among Web Services [25]. A dialogue can be viewed as a sequence of message exchanges between intelligent agents.

Lastly, there have been some efforts to enhance OSGi's capability to support service compositions beyond the built-in packages such as Declarative Service and Blueprint Container Service. For instance, a WS-BPEL engine is incorporated into OSGi platforms as a service [26]. It encapsulates the logic information of the composite service using BPEL technology inside a virtual bundle which is not need to be implemented. The BPEL Engine is responsible for registering and invoking OSGi composite service. The engine creates one process for each OSGi composite service and each composition process is the real implementation of the virtual bundle. One downside of this approach is that they may not be an ideal match. A WS-BPEL engine might be too heavy to be hosted in OSGi framework which is positioned as a technolgy for lightweight service platforms. In contrast, our proposed approach effectively enables lightweight and rapid service compositions for OSGi environments with full supports for major workflow patterns.

VI. CONCLUSION

This paper presents an architecture of a novel service framework that supports directed-acyclic-graph style compositions in OSGi environments along with its workflow description language. The framework is capable of instantiating a declarative blueprint of service interconnections into wirings of corresponding component instances available in the environment. Our proposal has been prototyped to demonstrate the effectiveness of its architectural design in promoting service composition and usage. We have also evaluated our approach in comparison with WS-BPEL, which is the most prominent workflow technology today. Especially, a comparison has been made with regards to the coverage of workflow patterns that the two workflow systems support. The result shows that our composition language provides better support for major workflow patterns. A subsequent performance study reveals that our composition framework is much more streamlined with lesser composing time, CPU load, and memory usage. Hence, a well-suited match for small-sized environments like OSGi platforms. In conclusion, our evaluation study confirm that our framework architecture indeed achieves its primary design goal which is a lightweight workflow engine targeting OSGi environments without negatively affecting the coverage of workflow features.

REFERENCES

[1] R. Buyya, C. Shin, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generat. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, Jun. 2009.

[2] M. A. Vouk, "Cloud computing—Issues, research and implementations," *J. Inf. Technol. Interface*, vol. 16, no. 4, pp. 235–246, 2008.

[3] "RFP 133 cloud computing," OSGi Alliance, San Ramon, CA, USA, Tech. Rep., 2011. [Online]. Available: https://osgi.org/bugzilla/attachment.cgi?id=46

[4] *OSGi Core Release 6*, OSGi Alliance, San Ramon, CA, USA, Jun. 2014.

[5] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web services Web: An introduction to SOAP, WSDL, and UDDI," *IEEE Internet Comput.*, vol. 6, no. 2, pp. 86–93, Apr. 2002.

[6] P. Bakker and B. Ertman, *Building Modular Cloud Applications With OSGi*. Sebastopol, CA, USA: O'Reilly Media, Sep. 2013.

[7] *OSGi Compendium Release 6*, OSGi Alliance, San Ramon, CA, USA, Jul. 2015.

[8] H. Cervantes and R. S. Hall, "Automating service dependency management in a service-oriented component model," in *Proc. ICSE/CBSE Workshop*, Portland, OR, USA, 2003, pp. 1–6.

[9] C. Escoffier and R. S. Hall, "Dynamically adaptable applications with iPOJO service components," in *Proc. Softw. Compos.*, Paris, France, 2007, pp. 113–128.

[10] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: An extensible service-oriented component framework," in *Proc. Int. Conf. Services Comput.*, Salt Lake City, UT, USA, Jul. 2007, pp. 474–481.

[11] M. C. Jaeger, G. Rojec-goldmann, and G. Muhl, "QoS aggregation for Web service composition using workflow patterns," in *Proc. 8th IEEE Int. Enterprise Distrib. Object Comput. Conf.*, Monterey, CA, USA, Sep. 2004, pp. 149–159.

[12] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan, "Adaptive and dynamic service composition in Eflow," in *Proc. Adv. Inf. Syst. Eng.*, Stockholm, Sweden, 2000, pp. 215–234.

[13] *Web Services Business Process Execution Language Version 2.0*, OASIS Standard, 2007.

[14] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. A. Mulyar, "Workflow control-flow patterns: A revised view," BPM Center, Tech. Rep. BPM-06-22, 2006.

[15] E. Newcomer, *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Reading, MA, USA: Addison-Wesley, May 2002.

[16] F. Daniel, P. Milano, B. Pernici, and P. Milano, "Insights into Web service orchestration and choreography," *Int. J. E-Bus. Res.*, vol. 2, no. 1, pp. 58–77, Mar. 2006.

[17] M. Broy, I. H. Krüger, and M. Meisinger, "A formal model of services," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, pp. 1–40, Feb. 2007.

[18] S. Uchitel, J. Kramer, and J. Magee, "Negative scenarios for implied scenario elicitation," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, Charleston, SC, USA, 2002, pp. 109–118.

[19] M. Brambilla, S. Ceri, I. Manolescu, and P. Fraternali, "Process modeling in Web applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 4, pp. 360–409, Oct. 2006.

[20] A. Charfi and M. Mezini, "AO4BPEL: An aspect-oriented extension to BPEL," *World Wide Web*, vol. 10, no. 3, pp. 309–344, Sep. 2007.

[21] A. Charfi and M. Mezini, "Aspect-oriented web service composition with AO4BPEL," *Lecture Notes in Computer Science*, vol. 3250, pp. 168–182, 2004.

[22] A. Erradi and P. Maheshwari, "AdaptiveBPEL: A policy-driven middleware for flexible Web services composition," in *Proc. Middleware Web Services*, Enschede, The Netherlands, 2005, pp. 5–12.

[23] C. Pautasso, "RESTful Web service composition with BPEL for rest," *Data Knowl. Eng.*, vol. 68, no. 9, pp. 851–866, Sep. 2009.

[24] C. Pautasso, "BPEL for rest," in *Proc. Bus. Process Manage.*, Milan, Italy, 2008, pp. 278–293.

[25] C. D. Walton, "Model checking multi-agent Web services," in *Proc. AAAI Symp. Semantic Web Services*, Palo Alto, CA, USA, 2004, pp. 68–75.

[26] R. P. D. Redondo, A. F. Vilas, M. R. Cabrer, J. J. P. Arias, and M. R. López, "Enhancing residential gateways: OSGi service composition," *IEEE Trans. Consum. Electron.*, vol. 53, no. 1, pp. 87–95, Apr. 2007.

CHOONHWA LEE received the B.S. and M.S. degrees in computer engineering from Seoul National University, South Korea, in 1990 and 1992, respectively, and the Ph.D. degree in computer engineering from the University of Florida, Gainesville, FL, USA, in 2003. He is currently a Professor with the Division of Computer Science and Engineering, Hanyang University, Seoul, South Korea. His research interests include cloud computing, peer-to-peer and mobile networking and computing, and services computing technology.

CHENGYANG WANG received the B.S. degree in computer engineering from Hanyang University, South Korea, in 2015, where he is currently pursuing the degree with the Division of Computer Science and Engineering. His research interests include cloud and distributed computing systems and computer networking protocols.

EUNSAM KIM received the B.S. and M.S. degrees in computer engineering from Seoul National University, South Korea, in 1994 and 1996, respectively, and the Ph.D. degree in computer and information science and engineering from the University of Florida, in 2006. He was with the Digital TV Research Laboratory. LG Electronics, South Korea, from 1996 to 2002. He is currently an Associate Professor with the Department of Computer Engineering, Hongik University, South Korea. His research interests include distributed computing, P2P streaming, cloud computing, mobile computing, and networked storage systems.

SUMI HELAL received the Ph.D. degree in computer science from Purdue University, West Lafayette, IN, USA. He is currently a Professor with the Computer and Information Science and Engineering Department, University of Florida, Gainesville, FL, USA, where he is also the Director of the Mobile and Pervasive Computing Laboratory. He directs the Gator Tech Smart House, an experimental facility for developing and validating assistive technology in support of aging, disability, and independence. His research interests include pervasive and mobile computing, smart health and wellbeing, and cloud-sensor systems.