CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Lancaster E-Prints

Noname manuscript No.
(will be inserted by the editor)

iPOJO Flow: A Declarative Service Workflow Architecture for Ubiquitous Cloud Applications

Xipu Zhang · Choonhwa Lee · Sumi Helal

Received: date / Accepted: date

Abstract The growth of innovative services backed up by various sensors and devices provides an unprecedented potential for ubiquitous computing applications and systems. However, in order to benefit from the recent developments, the current service middleware technology needs a catch-up of being able to fully support interactions among the services. OSGi is considered as a viable service framework solution due to its ability to deal with the dynamism inherent with ubiquitous cloud environments. iPOJO has also emerged as a service component model that simplifies the development of OSGi applications. However, the technology runs short of providing adequate support to foster declarative service compositions of realistic interaction topologies. Noticing this deficiency, we propose an iPOJO component-based service workflow architecture, named iPOJO Flow, where component services can easily be composed together to form realistic, complicated applications. Along with the architectural design, the paper also introduces a new DSL to specify service workflow topologies in a declarative way. The effectiveness of our proposed approach is validated through a prototype demonstration, comparative design analysis, and performance experiments.

Keywords Service composition · iPOJO · smart environment · cloud applications

Xipu Zhang · Choonhwa Lee (✉)
Department of Computer Science, Hanyang University, Seoul, Korea
E-mail: {tidexi,lee}@hanyang.ac.kr

Sumi Helal
School of Computing and Communications, Lancaster University, U.K.
E-mail: s.helal@lancaster.ac.uk

1 Introduction

In recent years, service-oriented pervasive computing systems have been explored in diverse domains such as smart home networks, medical information systems, and business network systems(Botta et al. 2015; Deen 2015; Yousfi et al. 2015). According to SOA (Service-Oriented Architecture) paradigm, services perform certain functionalities representing heterogeneous devices and sensors in pervasive computing environments(Puthal et al. 2015). Since the devices or sensors could come and go at any time, service frameworks for such environments are to be designed to cope with the dynamism and promote flexibility in the usage of services. OSGi is a prominent service execution platform that can deal with the availability and modularity of dynamic services(Pauls et al. 2011). A bundle in OSGi is a deployable unit, which provides modularization and encapsulation for service components. It promotes service dynamism in OSGi environments in that a bundle can be added or removed at runtime without a need to restart the whole framework(Chen and Cao 2010; Rellermeyer and Bagchi 2012; Zhang et al. 2014).

On top of OSGi service framework, iPOJO component model provides a basic means for a service to look for and make interactions with another with desired functionality, serving as a meeting place for rendezvousing services. It is obvious that more realistic functionalities can be enabled by combining service instances with rather simplistic functionalities together. However, iPOJO framework runs short of supporting more complicated interaction patterns beyond linear chaining interaction style. This lack of advanced composition support is a serious drawback to elevate itself as a dominant service platform for the upcoming cloud computing era(Armbrust et al. 2010). Workflow technology might be leveraged to fill this gap, by which an execution order of

participant services is specified in terms of control and data flow, forming a directed graph. WS-BPEL, used to integrate multiple constituent Web Services into a composite application, stands out among other workflow technologies(OASIS Web Services Business Process Execution Language (WSBPEL) et al. 2007). There have been research efforts to employ the BPEL technology for OSGi domains, whose examples include transforming OSGi services into Web Services or wrapping a BPEL engine into a bundle(Anke and Sell 2007; Álamo et al. 2010). However, they do not aim at enabling declarative service compositions, which is a primary contribution of our approach.

In this article, we present a novel service composition framework called iPOJO Flow that supports a directed-acyclic-graph style composition of iPOJO 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 of the current technology.

The remainder of this paper is organized as follows. Section 2 first motivates our research by discussing the shortcomings of iPOJO component-based service compositions of a sample scenario. Then, it presents our architectural design of a workflow-based service composition framework. Section 4 evaluates the proposed service composition framework through a workflow pattern coverage analysis study and use-case scenario demonstration using a prototype implementation. Section 5 discusses previous research efforts with regards to service composition and workflow technologies, clarifying the differences of our approach and the status quo. Finally, Section 6 concludes the paper.

2 iPOJO-Based Service Composition

In order to motivate our approach to a declarative service composition framework for ubiquitous cloud applications, we firstly consider a sample composition scenario for smart home which involves a set of component services representing various sensors and devices at home.

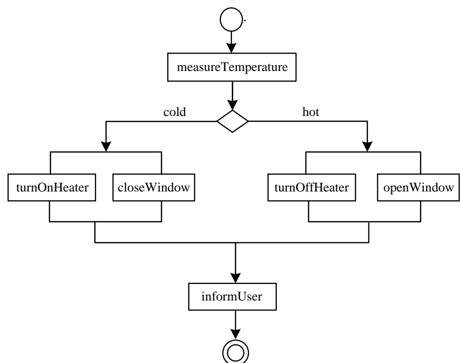

Fig. 1 Service composition scenario for smart home

Bob's house is instrumented with various sensors and actuators, and its smart home application acts as a hub to coordinate their operations and controls. Bob sets out for work at 7 A.M. every morning and comes back in the evening. The home application is tasked with keeping the home condition favorable to the resident, while he is at home. For instance, if the indoor temperature is lower than 22 degrees, the windows are closed and the heater is turned on. If it is too hot for Bob, then the windows are opened to let the cool air in and the heater is stopped. All these events are pushed to Bob's smart phone, which might be logged for later uses.

In this motivational scenario, the smart home application is an orchestrator to facilitate interactions among various sensors and actuators. In order to keep the temperature comfortable, the home automation system would need the following services.

- *Thermometer* service monitors the current indoor temperature and returns measured values.
- *HeaterController* service starts and stops heaters at home.
- *WindowController* service controls the operation of windows.
- *OwnerNotifier* service informs the resident of what's happening at home by sending messages to his mobile.

Diagrammed in Fig. 1 is a sample composite application that tries to keep 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 the scenario would be to get hold of an adequate hosting and execution platform where services live to invoke one another. OSGi is widely accepted as an ideal fit to the foundation of cloud service interactions.

On top of the OSGi framework, several component models were proposed, including Declarative Service, Blueprint Container Service, and iPOJO(Escoffier et al. 2007). They aim at simplifying component developments by automating service publication and discovery and inter-service dependency management. Especially, iPOJO has recently gained much popularity in some domains owing to its unique features and better performance in comparison to Declarative Service and Blueprint Container Service(Escoffier et al. 2013; Abras et al. 2014). Benefits of iPOJO component model derive from its design philosophy of separation of concerns; component code focuses only on the implementation of business logic other than non-functional aspects of the component such as service publishing, discovery, and service object creation. These tasks are delegated to iPOJO component container or handlers(Escoffier and Hall 2007).

iPOJO comes with a DSL, named ADL (Architecture Description Language) that allows one to define a service-

based composite application. However, the composition topology of the DSL is limited to consumer-producer link patterns. iPOJO ADL does not include assembling constructs that can be used to describe data passing and control flow among constituent services. As a result, it has a difficulty in declaratively supporting more realistic topologies of composed applications like a directed-acyclic-graph style composition as seen in Fig. 1.

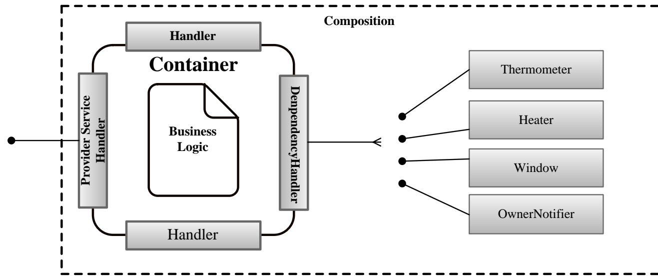

Fig. 2 Structure of iPOJO component

Fig. 2 shows the internal structure of an iPOJO component for the above smart home scenario. The component consists of three parts: POJO (Plain Old Java Object), component container, and handlers. The POJO contains application logic code, while the container is responsible for connecting the POJO with the outside world by plugging in a set of handlers which handle several tasks like instantiating the component itself, solving dependencies, and publishing new services. A corresponding ADL definition in Fig. 3 describes the structural composition and dependencies of the iPOJO component for the smart home. A composition is defined by a <composite> element which contains <subservice> and <instance> elements for services and the component itself. Also, a <provides> element is used to offer the composition as a new service.

It is important to know that this structural composition information must be complemented by some code that expresses logical relationships among sub-services, i.e., service call graphs. The container in a component has to include some code to glue constituent services together to enable inter-service collaborations. In other words, wirings of participating component services must be programmed in a hard-coded way. Sample code in Fig. 4 makes hardwired connections in order to realize the smart home composition scenario presented in Fig 1. The component uses @Requires annotation to import desired services through dependency injection mechanism. Also, developers should write Java code to coordinate sub-service executions. A problem with this approach is that it would be difficult to keep up with changes constantly taking place in the environment. The interconnections of application components may have to be re-programmed or repaired, when a service is newly introduced or removed. Considering the degree of dynamism that our target home automation system is expected to meet, changes at code level can hardly be considered as an acceptable solution.

```
<ipojo>
<!-- Declares a composition -->
<composite name="smartHomeApp">
   <!-- Instantiates services -->
   <subservice action="instantiate"
         specification="home.service.Thermometer"/>
  <subservice action="instantiate"
         specification="home.service.Heater"/>
<subservice action="instantiate"
         specification="home.service.Window"/>
 <subservice action="instantiate"
         specification="home.service.OwnerNotifier"/>
   <!-- Instantiates an instance of the composer -->
 <instance component="home.composer"/>
   <provides action="export"
          specification="home.service.MyComfortableHome"/>
</composite>
<!-- Instantiates an instance of the composition -->
<instance component="smartHomeApp"/>
</ipojo>
```

Fig. 3 Composite service description in ADL

```
@Component(immediate=true)
@Instantiate
public class Composer
{
   @Requries
   private Thermometer thermo;
   @Requries
   private Heater heater;
   @Requries
   private Window window;
  @Requries
   private OwnerNotifier inform;
  public void start()
  {
        ...
         String temp=thermo.readTemperature();
         if(Double.parseDouble(temp) < 22){
                 heater.turnOn();
                 window.close();
         }
         else {
              ...
         }
```

Fig. 4 Sample hardcoding of service composition

With iPOJO ADL that stops short of providing support for control flow over services, application developers are left with no other options than the manual weaving of relationships among component services, specifically using Java code to control a calling sequence of sub-services and determine their invocation conditions. It is far from being a satisfactory solution to our declarative service composition problem. Therefore, we propose to extend the iPOJO component model to enable declarative service compositions based on workflow patterns.

3 iPOJO Flow Composition Architecture

To enable declarative service compositions in OSGi environments, we have designed a workflow-based composition middleware architecture, named iPOJO Flow, on top of the iPOJO component model. As illustrated in Fig. 5, iPOJO Flow framework architecture consists of three layers: physical, platform, and application layer. Physical layer serves as an interface to the physical world, collecting data from the home-instrumented sensors and passing commands to the actuators. Platform layer is the core of our architectural design, which is based on iPOJO component model. The centerpiece of this layer is the workflow engine that performs service compositions according to a given workflow description. Application layer at the top contains applications composed by using services from the platform layer. This top layer also includes iCDL (iPOJO Composition Description Language) files to describe an application composition and topology. iCDL is a DSL designed for our iPOJO Flow architecture.

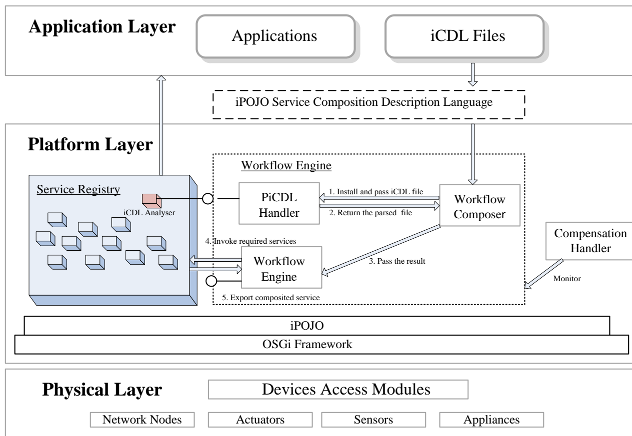

Fig. 5 iPOJO Flow service composition architecture

Our work focuses mainly on the platform layer, extending the existing iPOJO to enable workflow-based declarative service compositions. Our research efforts are made up of three key parts: (1) creating iCDL, a new XML-based DSL, to describe workflow-based compositions, (2) architecting an iPOJO-based workflow engine to take care of service composition processes, and (3) extending iPOJO handler architecture to instantiate service workflow definitions in the new composition language.

Fig. 6 depicts the steps of service composition process performed at the platform layer. A composition is started by the workflow engine on top of OSGi service framework, consisting of PiCDL (Parsed iPOJO Composition Language) Handler, Workflow Composer, and Workflow Engine. It all starts with Workflow Composer that application developers use to define control and data flow over a set of services. Then, iCDL Analyzer service invoked by PiCDL Handler parses the description file and passes the result on to Workflow Engine through Workflow Composer. Workflow Engine executes the actual composition, instantiating a directed graph of iPOJO components as described in the declarative workflow definition, and publishes a newly created composite service with OSGi service registry.

The engine is made up of the following three components: PiCDL Handler, Workflow Composer, and Workflow Engine. PiCDL Handler is an iPOJO handler that we extended. The primary function of this handler is to parse a service composition description into a Java object at runtime by invoking iCDL Analyser based on SAX. Tags in the composition description are converted into a hierarchical structure which is used by Workflow Engine at a later stage. Workflow Composer is used to plug in PiCDL Handler into the framework. PiCDL Handler makes use of the dependency injection mechanism of iPOJO model to pass parsed descriptions to Workflow Composer. Workflow Engine is a central entity that performs an actual build-up of service workflows and handles the execution of the instantiated workflow. It also publishes the resultant composite service to the framework.

In order to keep composed workflows available all the time, our architecture also includes Compensation Handler that monitors the health of workflows and their component services. On detection of failures in component services, the handler triggers the repair of damaged workflows, so that the faulty component can be replaced with a healthy one.

Table 1 Elements and attributes used in iCDL

Type	Element Name	Attribute	Description
SE	partnerService	remoteAcess	Includes specified service
	service	serviceType	Describes individual service
		form	
		name	
	variables		Includes intermediate variable
	variable	name	Describes auto intermediate variable
	subprocess	seq	Defines logical structure
	iCDL	id	Indicates root element
	process	name	Includes the entire composition structure
CE	switch		Models multiple branches operation
	case	Cobject	Models a single branch
		Coperation	
		Cvalue	
	while		Used to support loop operation
	if	Iobject	Defines conditional operation
		Ioperation	
		Ivalue	
AE	invokeOperation	service	Invokes a method of the service
		method	

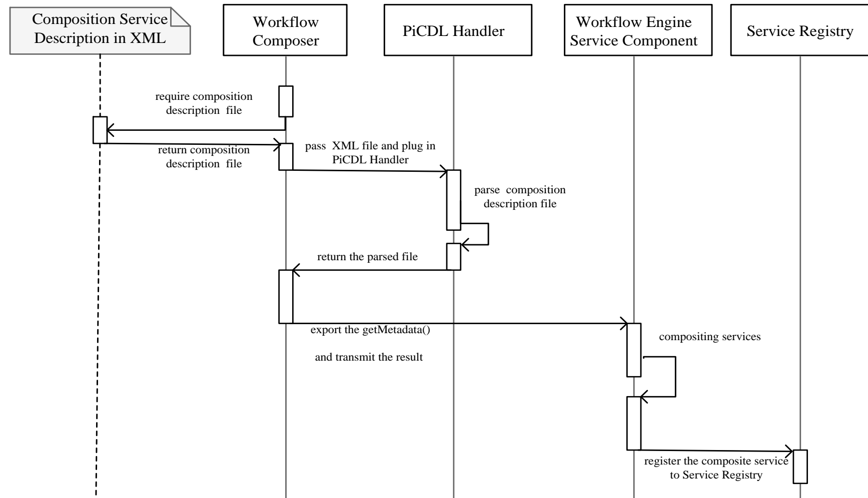

Fig. 6 Sequence diagram of service composition process

In order to enable workflow-based service compositions, we have introduced a service composition language named iCDL (iPOJO Composition Description Language). Elements and attributes of the DSL are summarized in Table 1. Three types of elements are defined, including SE (Structure Element), AE (Action Element), and CE (Control Element). First, an SE element describes the basic structure of workflow compositions. All component services indicated by <service> tag are contained in <partnerServices> element. <variable> tag represents an intermediate variable and <subprocess> element includes various control flow tags. An AE element is used to invoke services. <invokeOperation> tag has service and method attributes for individual services. Lastly, a CE element is used to model various control flow of workflow executions, which includes <if>, <switch>, <case>, and <while>.

A typical iCDL file is composed up of three sections: services definition, intermediate values declaration, and logical structure description. Services definition part lists up services that take part in the composition by using <parternerServices> and <service> elements. The second part features <variables> and <variable> elements to define what intermediate values are used in the workflow. The last logical structure part focuses on expressing control flow by using various control elements in <subprocess> element. Fig. 7 displays an iCDL definition for the sample smart home scenario in Section 2. For this scenario, there are four services brought up by the engine as shown in <partnerService> element. Then, a <variable> element indicates that one variable temperature is used to hold intermediate data items used in the workflow. Finally, the <subprocess> element defines flow control constructs (i.e., conditional branches) such as <switch> element and <if> element.

```
<? xml version="1.0" encoding="UTF-8">
< iCDL xmlns: xsi=http://www.w3.org/2001/XMLSchema-instance>
<process name="smartHomeApp" accessLevel="public">
   <partnerService remoteAccess="true">
    <service name="Thermometer" serviceType="readThermometer"
     form="entity"/>
    <service name="Heater" serviceType="controlHeater" form="entity"/>
   <service name="Window" serviceType="controlWindow" form="entity"/>
   <service name="UserNotifier" serviceType="informUser" form="entity"/>
  </partnerServices>
  <variables>
    <variable name="temperature"/>
 </variables>
 <subprocess seq="true">
<invokeOperation service="ThermometerMonitoring"
          method="readTemperature"  outputVariable="temperature"/>
         <if Iobject="temperature" Ioperation="lessThan" Ivalue="22">
           <invokeOperation service="Heater" method="turnOn"/>
           <invokeOperation service="Window"method="close"/>
         </if>
        <if Iobject="temperature" Ioperation="moreThan" Ivalue="22">
          <invokeOperation service="Heater" method="turnOff"/>
          <invokeOperation service="Window" method="open"/>
        </if>
      <invokeOperation service="UserNotifier" method="inform"/>
</subprocess>
</process>
</iCDL>
```

Fig. 7 iCDL description of smart home application

The composition process executed by the workflow engine consists of two main phases as shown in Fig. 8. At the first step, Workflow Engine obtains a service composition description from Workflow Composer. It also imports component services indicated by <service> tags in the iCDL file. The engine then stores imported services and intermediate variable values in HashMaps structure. Resulted from the precedent workflow node invocation, intermediate variable values might be used for flow control decisions for the following step or as parameters for the subsequent node. The second step takes care of the workflow execution. Each sub-element in <subprocess> element corresponds to a

particular control element such as <switch> or <if>. Workflow Engine uses Java reflection mechanism to call operations corresponding to control flow elements in the iCDL file.

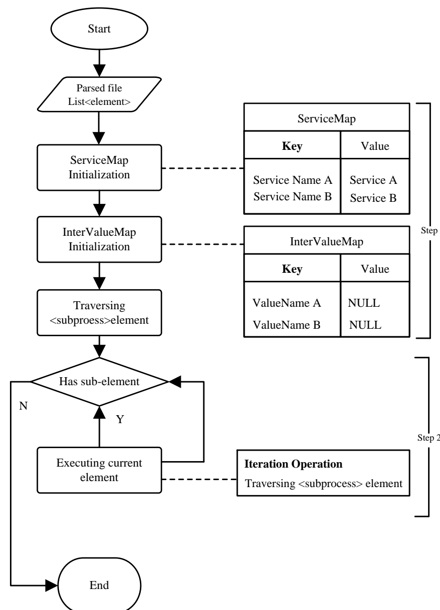

Fig. 8 Composition steps by Workflow Engine

Since our iPOJO Flow architecture is built upon OSGi framework and iPOJO component model, it primarily targets the small and medium-sized ubiquitous computing environments like home/office networks. However, we'd like to add that the coverage of the workflow system may expand further than that to include cloud services hosted in a remote network. R-OSGi (Remote Services for OSGi) provides a way to access remote services from other OSGi domains (Cheng et al. 2012). It provides a technical base for our iPOJO Flow to discover and make use of distributed services. Moreover, OSGi framework can serve as an effective foundation to construct cloud computing services and systems (Zhang et al. 2013; Houacine et al. 2013; Neto et al. 2015). Therefore, our iPOJO Flow can support workflow descriptions and compositions involving cloud services from remote domains.

4 Framework Validation and Evaluation

We have prototyped the architecture of iPOJO Flow service composition framework based on Eclipse Equinox that provides a certified implementation of the OSGi Core specification and Apache Felix iPOJO implementation. Our proof-of concept prototype has been demonstrated by using the smart home scenario presented in Section 2.

In order to help to define workflow descriptions, we have also developed a workflow designer tool for iPOJO Flow, which is part of Workflow Composer. Application developers can easily specify service compositions using the tool, as shown in Fig. 9, building blocks of the smart home application are laid down and linked together to form a service workflow. After the necessary configuration of the workflow nodes, a XML-based iCDL definition file is generated to be fed into the iPOJO Flow engine in charge of materializing the workflow.

Validation efforts of our workflow-based service composition framework include a comparative workflow pattern coverage study and the performance evaluation of our approach against iPOJO ADL and BPEL.

4.1 Workflow Pattern Evaluation

Our iCDL language design was inspired by WS-BPEL (Web Services Business Process Execution Language). It is an XML-based workflow language that enables process-oriented service composition. 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 Web Services. The language defines a set of primitives that are used to invoke remote services, orchestrate process execution, and manage events and exceptions. We compare our iCDL language and BPEL with regards to workflow pattern coverage. In the literature, workflow patterns are defined as a means of categorizing recurring problems and solutions in modeling business process(Yang et al. 2014). Workflow pattern coverage should indicate how effectively complex composition scenarios can be modeled by the workflow language(Gupta et al. 2015).

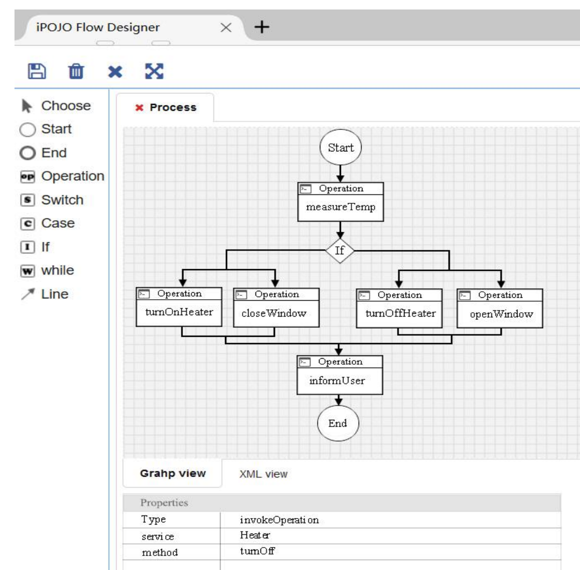

Fig. 9 Screenshot of iPOJO Flow designer

Table 2 compares the workflow pattern coverage of iCDL and BPEL. A pattern is said to be supportive (marked as "+" in the table), if the workflow language fully satisfies the evaluation criteria of the pattern. Otherwise, the pattern is unsupportive (marked as "-"in the table.) Our iPOJO Flow framework provides the same level of supports for basic control-control patterns, advanced branching and synchronization patterns, and structural patterns. But it does not support cancellation patterns. Therefore, it can be argued that our iCDL language provides the same coverage as BPEL for major workflow patterns.

Table 2 iPOJO Flow workflow pattern comparison

Workflow Pattern	*BPEL*	*iCDL*	*Note*
Basic Control-Flow Patterns			
Sequence	+	+	Supported by <subprocess> *seq* attr
Parallel Split	+	+	Supported by <subprocess> *seq* attr
Synchronization	+	+	Supported by <subprocess> *seq* attr
Exclusive Choice	+	+	Supported by <switch> element
Simple Merge	+	+	Supported by <switch> element
Advanced Branching and Synchronization Patterns			
Multi-Choice	+	+	Supported by <switch><case> seq attribute
S-Sync-Merge	+	+	Supported by <switch><case> seq attribute
Structural Patterns			
Implicit Termination	+	+	Supported by <subprocess> *seq* attribute
Mi-No-Synchronization	+	+	Supported by <invokeOperation> Included in <while> element
Mi-dt	-	-	Not supported
Mi-rt	-	-	Not supported
State-based Patterns			
Deferred Choice	+	-	Not supported
Cancellation Patterns			
Cancel Activity	+	-	Not supported
Cancel Case	+	-	Not supported
New Control-Flow Patterns			
Structured Loop	+	+	Supported by <while> element
Transient Trigger	-	-	Not supported
Recursion	-	-	Not supported

4.2 Performance Experiments

In order to evaluate the performance of our iPOJO Flow framework for workflow-based service composition, we have used the smart home scenario in Section 2. The composition workflow involves sequences and branch patterns. Our prototype implementation is compared against the original iPOJO component model implementation and BPEL engine ODE (http://ode.apache.org). Performance results are measured in terms of memory usage and workflow materialization time. Our experiment uses a machine with Intel Core i5-4200 CPU with memory of 4GB running at 2.8GHz. Eclipse Equinox implementation of the base OSGi framework was used along with Apache Tomcat 8.0.

We ran the smart home composition scenario 10 times to obtain an average using our iPOJO Flow, iPOJO component model, and WS-BPEL language. Performance results from each case are presented as iCDL, ADL, and BPEL case, respectively. Fig. 10 compares memory usage from the three cases. The iPOJO model shows the memory usage that varies from 4.43MB to 4.77MB. Memory utilization in our iCDL case is more stable and a little lower, whose average is measured at 4.60MB. Also, we can see that the BPEL case exhibits higher usage with some level of fluctuations.

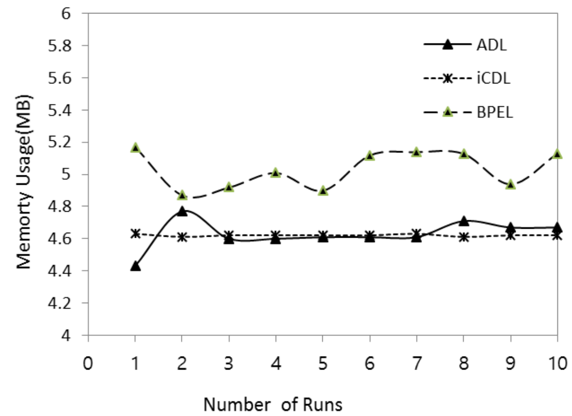

Fig. 10 Memory usage evaluation

Another set of experiments aims to compare workflow instantiation time for the three cases. A composition time indicates how fast composition engines can materialize and execute a workflow. Again, we repeat the smart home scenario 10 times using iPOJO Flow framework, iPOJO model, and BPEL engine. As plotted in Fig. 11, iCDL and ADL approaches excel themselves by showing the fast composition speed of less than 2.6ms. But BPEL case suffers from composition time ranging from 538ms to 129ms.

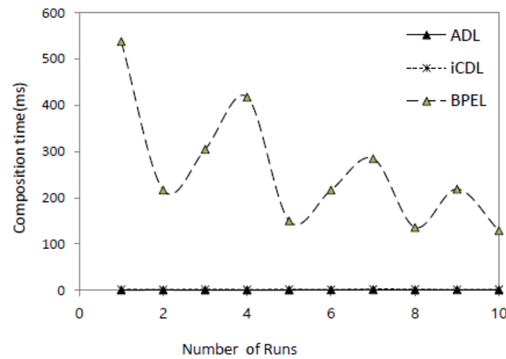

Fig. 11 Composition time comparison

We then measure service availability which indicates the robustness of our iPOJO-based workflow system to unpredictable changes and failures in the environments. Participant component services in the environment are likely to come and go anytime without any notice. As presented in Section 3, our iPOJO Flow architecture is designed to deal with such dynamism. The service compensation mechanism incorporated in iPOJO Flow system monitors the health of instantiated workflows and triggers recovery actions on detection of component service failures. A workflow can be repaired by replacing a failed service with an equivalent instance, if there exists an alternative in the environment.

We intentionally injected failures into the component services in the scenario given in Fig. 1. Then, a mean time between service component failures and workflow repairs is measured. This experiment is repeated for a different number of failures. Table 3 tabulates recovery times when four participating services get down for the smart home scenario in Section 2. The results show that the average workflow repair time in the case of four failures is about 10.7 ms.

Fig. 12 compares the workflow recovery times for a varying number of failed component services. The average is 1.30 ms for one failed service, 5.10 ms for two component service failures, 8.30 ms for three failures, and 10.70 ms for four failed services. With the increase of failed services, the workflow recovery time grows accordingly, as depicted in the graph. From the experiments, we can see that our iPOJO Flow architecture is capable of quickly adapting to changing situation to maintain high availability of composed workflows.

Table 3 Failure recovery time

Failure Detection	Workflow Repair	Elapsed Time (ms)
1519042709435	1519042709451	16
1519042861676	1519042861685	9
1519042960035	1519042960049	14
1519043054406	1519043054415	9
1519043124828	1519043124840	12
1519107039417	1519107039427	10
1519107404741	1519107404749	8
1519107531043	1519107531051	8
1519107624602	1519107624612	10
1519107694451	1519107694462	11

5 Related Work

Service composition allows us to combine component services of primitive functionalities to form a composite, value-added service. Basically, it aggregates and reuses existing services to build up an application to handle more complicated tasks (Moghaddam and Davis 2014). After having been explored in the last decades, the technology is being spotlighted again for heterogeneous cloud computing environments(Baker et al. 2012, 2013, 2017). Previous research efforts for service composition generally aimed at providing schemes to innovate a new service through the re-use of existing services for cost/time reduction and improved efficiency (Vakili and Navimipour 2017). Among the research topics of particular interest to us is workflow-based service compositions like WS-PBEL where a complex business process can be defined in terms of component service invocations and their interactions. Especially, applications in large-scale distributed environments can benefit from employing workflow technologies to cope with the diversity and heterogeneity inherent to such environments(Li and Liu 2012; Viriyasitavat et al. 2012; Xu et al. 2012).

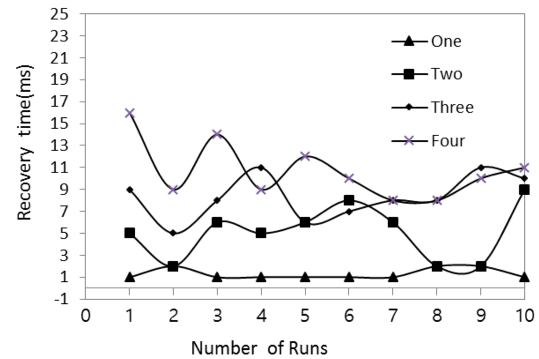

Fig. 12 Workflow availablity measurement

The current OSGi specification runs short of fully supporting complicated composition topologies like workflow-based orchestrations. Several research efforts were made to overcome such limitation by tapping WS-BPEL technology for complicated orchestration capacity. For instance, an OWL-S/OSGi framework was proposed to support BPEL-style services compositions on top of OSGi platforms(Díaz Redondo et al. 2007). According to the proposal, service matchmaking is made possible based on semantic descriptions of OSGi services. OSGi services can be packaged and offered to the outside world as Web Services, so that they can take part in a BPEL workflow(Anke and Sell 2007). These OSGi-backed Web Services can be combined into a business process described in BPEL, and its materialized workflow is executed by a BPEL engine. On the contrary, a BPEL service can be brought to the OSGi domain to become part of a workflow(Álamo et al. 2010). Another noteworthy is a BPEL-based service composition framework that is made capable of cross-breeding SOAP, RESTful, and OSGi services by employing adapter patterns(Liu et al. 2013). For adapting OSGi services into BPEL equivalents, the research extended WSDL description for OSGi services including service types, service names, and filters.

OSGi's potential as distributed service platforms has actively been explored over the past years. Distributed OSGi

extends the original OSGi framework, so that services can be discovered and invoked across neighboring platforms(Chen and Cao 2010; Roelofsen et al. 2010; Zhang et al. 2014).With the support of distributed OSGi services, building a flexible and adaptive pervasive cloud infrastructure has been a focus by several recent researches. One research group proposed a component migration scheme to develop a pervasive cloud infrastructure, called OSGi-PC, which supports flexible migrations among various cloud nodes (Zhang et al. 2013). Similarly, a R-OSGi based cloud architecture was proposed to facilitate inter-framework service exchanges and invocations (Cheng et al. 2012).s In addition, an OSGi-based mobile cloud service model, named MCC-OSGi, was presented with a focus on lightweight mobile cloud services (Houacine et al. 2013). The architecture identified different service models depending on varying roles played by mobile platform or the cloud. This trend of technological developments towards OSGi framework as cloud service platforms and the ensuing proliferation of OSGi services is our main motivation to pursue solid technological solutions for declarative service workflow composition framework for OSGi environments.

Lastly, besides its basic abstraction of services, the OSGi specification has introduced declarative service supports such as Blueprint Container Service and Declarative Service(The OSGi Alliance 2014). iPOJO component model is another such effort. They all intend to ease the complexity of building a composite application and managing its dependencies based on a composition description. A number of advanced platforms have emerged with the advent of iPOJO to facilitate the development of dynamic pervasive applications based on OSGi frameworks in distributed environments. For instance, the SmartComponent Framework is introduced to build sensor clouds with the purpose to manipulate, access, and visualize information distrusted sensors (Neto et al. 2015). Also, two iPOJO-based middleware, named Cilia and iCasa, is proposed to help construct pervasive health and smart manufacturing applications which require a high degree of flexibility at runtime (Lalanda et al. 2017; Lalanda et al. 2015). However, their emphasis is not on service composition support to enable iPOJO component-based pervasive applications. Specifically, Cilia essentially focuses on device integrations but not services. iCasa is mostly concerned about context-aware service composition instead of workflow-based orchestrations. In general, these existing works lack proper support for declarative service composition of diversified topologies, which had led us to develop iPOJO Flow service composition framework.

Our proposed workflow architecture effectively enables lightweight and rapid service compositions for iPOJO component environments with support for major workflow patterns. To the best of our knowledge, we are the first one who have come up with an iPOJO component-based workflow system, and such iPOJO workflow support has not been reported in the literature. Some recent research focuses on leveraging iPOJO component model to build applications in ubiquitous cloud environments (Neto et al. 2015; Lalanda et al. 2017; Lalanda et al. 2015). But they do not consider workflow support for iPOJO compoents. Through experimental performance evaluations in terms of composition time, CPU utilization, and service availability, we have validated that our iPOJO Flow architecture achieves better performance than other existing approaches. However, we acknowledge that there exists a need of further performance experiments comparing our architecture with others from the perspective of response time, latency, and cost per customer. Such efforts should be able to help us understand the strength and weakness of our approach to a deeper extent.

6 Conclusion

This paper proposes our iPOJO Flow architecture that is designed to enable workflow-based service compositions for ubiquitous cloud applications. The novel service composition framework extends the iPOJO component model, so that component services can interact with one another in a much more diverse composition topologies beyond conventional producer-consumer patterns. The paper also presents our design of iCDL language to describe a workflow and its participating services. 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 workflow pattern coverage of their workflow definition languages. A subsequent performance study reveals that our composition framework is far more streamlined with lesser composing time and memory usage. Hence, a well-suited match for small-sized environments like OSGi platforms. In conclusion, these evaluation results confirm that our iPOJO Flow architecture design has achieved its primary goal that is a lightweight workflow engine targeting OSGi services without negatively affecting its ability to model diverse composition topologies and scenarios for cloud computing era.

Acknowledgements This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No. 2017R1A2B4010395).

References

Abras S, Calmant T, Ploix S, et al (2014) Developing dynamic heterogeneous environments in smart building using iPOPO. In: Smart grids and green IT systems (SMARTGREENS), 2014 3rd International conference on, pp 1-7

Álamo JMR, Yang HI, Wong J, Chang CK (2010) Automatic service composition with heterogeneous service-oriented architectures. In: Smart homes and health telematics (ICOST), 2010 8th International conference, pp 9–16

Anke J, Sell C (2007) Seamless Integration of Distributed OSGi Bundles into Enterprise Processes using BPEL. In: Communication in distributed systems (KiVS), 2007 ITGGI conference on, pp 1–6

Armbrust M, Stoica I, Zaharia M, et al (2010) A view of cloud computing. Communications of the ACM 53(4):50-58. doi:10.1145/1721654.1721672

Baker T, Asim M, Tawfik H, et al (2017) An energy-aware service composition algorithm for multiple cloud-based IoT applications. J Netw Comput Appl 89:96–108. doi: 10.1016/j.jnca.2017.03.008

Baker T, Rana OF, Calinescu R, et al (2013) Towards autonomic cloud services engineering via intention workflow model. In: Grid economics and business models (GECON), 2013 10th International conference, pp 212–227

Baker T, Taleb-Bendiab A, Randles M, Hussien A (2012) Understanding elasticity of cloud services compositions. In: Utility and cloud computing (UCC), 2012 5th International conference, pp 231–232

Botta A, de Donato W, Persico V, Pescapé A (2015) Integration of cloud computing and Internet of Things: A survey. Futur Gener Comput Syst 56:684–700. doi: 10.1016/j.future.2015.09.021

Chen H, Cao C (2010) Research and application of distributed OSGi for cloud computing. In: Computational intelligence and software engineering (CISE), 2010 International conference on, pp 1-5

Cheng HC, Lee WT, Wei XW, Sun TW (2012) A novel service oriented architecture combined with cloud computing based on R-OSGi. Lecture Notes in Electrical Engineering 182 291–296. doi:10.1007/978-94-007-5086-9_38

Deen MJ (2015) Information and communications technologies for elderly ubiquitous healthcare in a smart home. Pers Ubiquitous Comput 19(3-4):573–599. doi: 10.1007/s00779-015-0856-x

Díaz Redondo RP, Fernández Vilas A, Ramos Cabrer M, et al (2007) Enhancing residential gateways: OSGi service composition. IEEE Trans Consum Electron 53(1):87–95. doi: 10.1109/TCE.2007.339507

Escoffier C, Bourret P, Lalanda P (2013) Describing dynamism in service dependencies: Industrial experience and feedbacks. In: Services computing (SCC), 2013 IEEE International conference on, pp 328–335

Escoffier C, Hall RS (2007) Dynamically adaptable applications with iPOJO service components. In: Software composition (SC), 2007 6th International conference on, pp 113–128

Escoffier C, Hall RS, Lalanda P (2007) IPOJO: An extensible service-oriented component framework. In: Services computing (SCC), 2007 IEEE International conference on, pp 474–481

Gupta IK, Kumar J, Rai P (2015) Optimization to Quality-of-service-driven web service composition using modified genetic algorithm. In: Communication and Control (IC4), 2015 International conference on, pp 1–6

Houacine F, Bouzefrane S, Li L, Huang D (2013) MCC-OSGi: An OSGi-based mobile cloud service model. In: Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on. pp 1–8

Lalanda P, Chollet S, Aygalinc C, Gerbert-Gaillard E (2015) Service-based architecture and frameworks for pervasive health applications. In: Emerging Technologies and Factory Automation (ETFA) IEEE 20th International Conference on.pp 1-8

Lalanda P, Morand D, Chollet S (2017) Autonomic mediation middleware for smart manufacturing. IEEE Internet Comput 21(1): 32-29.doi: 10.1109/MIC.2017.18

Li L, Liu J (2012) An efficient and flexible web services-based multidisciplinary design optimisation framework for complex engineering systems. Enterp Inf Syst 6(3):345–371. doi: 10.1080/17517575.2011.651627

Liu J, Wang D, Chen Y, Lv Y (2013) Anything is service: Using LIR-OSGi and R2-OSGi to construct e service network. J Softw 8(1):236–242. doi: 10.4304/jsw.8.1.236-242

Moghaddam M. b, Davis JG. (2014) Service selection in web service composition: A comparative review of existing approaches. In: Web services foundations. Springer, New York

Neto L, Reis J, Guimarães D, Gonçalves G (2015) Sensor cloud: SmartComponent framework for reconfigurable diagnostics in intelligent manufacturing environments. In: Industrial Informatics(INDIN), 2015 IEEE International Conference on, pp 1706–1711

OASIS(2007) Web Services Business Process Execution Language Version 2.0

Pauls K, McCulloch S, Hall RS, Savage D (2011) OSGi in Action. Manning Publications Co, Greenwich

Puthal D, Sahoo BPS, Mishra S, Swain S (2015) Cloud computing features, issues, and challenges: A big picture. In: Computational Intelligence and Networks (CINE), 2015 1st International Conference on, pp 116–123

Rellermeyer JS, Bagchi S (2012) Dependability as a cloud service - A modular approach. In: Dependable systems and networks workshops (DSN-W), 2012 IEEE/IFIP 42nd International conference on, pp 1–6

Roelofsen R, Bosschaert D, Ahlers V, et al (2010) Think large, act small: An approach to web services for embedded systems based on the OSGi framework. In: Exploring services science (IESS), 2010 Internationl conference, pp 239–253

The OSGi Alliance (2014) OSGi Core Release 6 Specification.

Vakili A, Navimipour NJ (2017) Comprehensive and systematic review of the service composition mechanisms in the cloud environments. J Netw Comput Appl 81:24–36, doi: 10.1016/j.jnca.2017.01.005

Viriyasitavat W, Xu L Da, Martin A (2012) SWSpec: The requirements specification language in service workflow environments. IEEE Trans Ind Informatics 8(3):631–638. doi: 10.1109/TII.2011.2182519

Xu L Da, Viriyasitavat W, Ruchikachorn P, Martin A (2012) Using propositional logic for requirements verification of service workflow. IEEE Trans Ind Informatics 8(3):639–646 . doi: 10.1109/TII.2012.2187908

Yang P, Xie X, Ray I, Lu S (2014) Satisfiability analysis of workflows with control-flow patterns and authorization constraints. IEEE Trans Serv Comput 7(2):237–251. doi: 10.1109/TSC.2013.31

Yousfi A, de Freitas A, Dey A, Saidi R (2015) The use of ubiquitous computing for business process improvement. IEEE Transactions on Services Computing 9(4):1–1, doi:10.1109/TSC.2015.2406694

Zhang W, Chen L, Lu Q, et al (2013) Towards an OSGi based pervasive cloud infrastructure. In: Green Computing and Communicationsand IEEE Internet of Thingsand IEEE Cyber, Physical and Social Computing, (GreenCom-iThings-CPSCom), 2013 IEEE International Conference on 2013, pp 418–425

Zhang WS, Chen LC, Liu X, et al (2014) An OSGi-based flexible and adaptive pervasive cloud infrastructure. Sci China Inf Sci 57:1–11. doi: 10.1007/s11432-014-5070-3