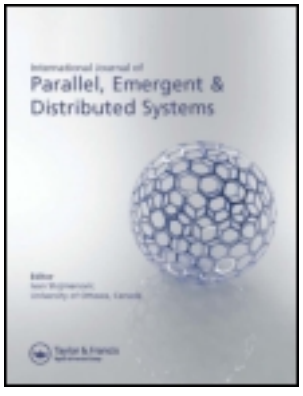

International Journal of Parallel, Emergent and Distributed Systems
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/gpaa20

OpenCOPI: middleware integration for Ubiquitous Computing

Frederico Lopes[a], Flavia C. Delicato[b], Thais Batista[c], Everton Cavalcante[c], Thiago Pereira[c], Paulo F. Pires[b], Paulo Ferreira[d] & Reginaldo Mendes[e]

[a] ECT - Federal University of Rio Grande do Norte, Campus Universitário, Lagoa Nova, NatalRNBrazil

[b] PPGI/DCC-IM - Federal University of Rio de Janeiro, Avenida Athos da Silveira Ramos, 274, Cidade Universitária, Ilha do Fundão, Rio de Janeiro, RJBrazil

[c] DIMAp - Federal University of Rio Grande do Norte, Campus Universitário, Lagoa Nova, Natal, RNBrazil

[d] INESC-ID - Technical University of Lisbon, Rua Alves Redol, 9, Lisbon, Portugal

[e] Federal Service of Data Processing, SGAN Quadra 601, Módulo V, Asa Norte, BrasiliaDFBrazil

Published online: 02 Oct 2013.

To cite this article: Frederico Lopes, Flavia C. Delicato, Thais Batista, Everton Cavalcante, Thiago Pereira, Paulo F. Pires, Paulo Ferreira & Reginaldo Mendes (2014) OpenCOPI: middleware integration for Ubiquitous Computing, International Journal of Parallel, Emergent and Distributed Systems, 29:2, 178-212, DOI: 10.1080/17445760.2013.831415

To link to this article: http://dx.doi.org/10.1080/17445760.2013.831415

International Journal of Parallel, Emergent and Distributed Systems, 2014
Vol. 29, No. 2, 178–212, http://dx.doi.org/10.1080/17445760.2013.831415

OpenCOPI: middleware integration for Ubiquitous Computing

Frederico Lopes[a1]*, Flavia C. Delicato[b2], Thais Batista[c3], Everton Cavalcante[c4], Thiago Pereira[c5], Paulo F. Pires[b6], Paulo Ferreira[d7] and Reginaldo Mendes[e8]

[a]*ECT – Federal University of Rio Grande do Norte, Campus Universitário, Lagoa Nova, Natal, RN, Brazil;* [b]*PPGI/DCC-IM – Federal University of Rio de Janeiro, Avenida Athos da Silveira Ramos, 274, Cidade Universitária, Ilha do Fundão, Rio de Janeiro, RJ, Brazil;* [c]*DIMAp – Federal University of Rio Grande do Norte, Campus Universitário, Lagoa Nova, Natal, RN, Brazil* [d]*INESC-ID – Technical University of Lisbon, Rua Alves Redol, 9, Lisbon, Portugal;* [e]*Federal Service of Data Processing, SGAN Quadra 601, Módulo V, Asa Norte, Brasilia, DF, Brazil*

(*Received 1 August 2012; accepted 23 July 2013*)

In this paper, we present OpenCOPI (*Open COntext Platform Integration*), a Service-Oriented Architecture-based middleware platform that supports the integration of services provided by distinct sources, ranging from services offered by simple systems to more complex services provided by context-provision middleware. OpenCOPI offers selection and composition mechanisms to, respectively, select and compose services provided by different sources, considering applications of both Quality of Service and Quality of Context requirements. It also offers an adaptation mechanism that enables to adapt the application execution due to service failures, service quality fluctuation and user mobility. OpenCOPI allows the definition of applications in a higher abstraction level by the specification of a semantic workflow that contains abstract activities. This paper illustrates the use of OpenCOPI in an application from the Gas & Oil Industry and it also shows the evaluation of the main mechanisms of OpenCOPI: the service selection, composition, adaptation and workflow execution.

Keywords: integration platform; service composition; semantic workflow; adaptation mechanism

1. Introduction

Ubiquitous Computing encompasses sensor-instrumented environments, which are often endowed with wireless network interfaces, in which devices, software agents and services are integrated in a seamless and transparent way and cooperate to meet high-level goals of human users [18]. This computational power distribution provides new functionality through support of personalised services and omnipresent applications [29].

Ubiquitous Computing environments are characterised by high heterogeneity and dynamism. In these environments, the execution context is always changing. However, tasks for context gathering, context handling and reaction to context changes should not be tangled with application business logic because this approach tends to be repetitive and error-prone. Instead, a more promising approach is to delegate those tasks to a *context-provision middleware* [29,11,15,17,20,33,22,30] that should support most of the tasks involved in the gathering and manipulation of context information in those heterogeneous, dynamic and distributed environments, unburdening applications of handling contextual information.

*Corresponding author. Email: fred.lopes@gmail.com

Typically, each context-provision middleware addresses different kinds of context and adopts different models for handling and representing contextual information (*context model*). In general, each middleware is suitable to meet different ubiquitous application requirements. For instance, MiddleWhere [29] focuses on users and devices location (addressing location-aware and mobility), while some middleware focus on the management of wireless networks [16,8] and others are centred on the user activities and preferences [18,7]. Moreover, context-provision middleware often use proprietary and non-standardised protocols, thus resulting in isolated islands of devices using non-uniform protocols [28]. The current trend is the emergence of more complex context-aware ubiquitous applications that make simultaneous use of different types of services supported by a myriad of service providers. Such services can encompass context-provision functionalities as well as other functions, not necessarily related to context, but also required for the proper operation/execution of ubiquitous applications. For example, an application can require context-aware services to handle the contextual information as well as conventional services, where conventional services can be database queries services, legacy systems, messaging services, among others. Thus, complex ubiquitous applications, in general, need to use several underlying platforms, each one providing a specific type of context service. This feature increases the need for the integration among these systems in order to provide a value-added service for users of ubiquitous applications and to enable the use of several services provided by the underlying middleware platforms, which can be composed to reach a high-level user goal.

In order to address this problem, we present the OpenCOPI (*Open COntext Platform Integration*), a middleware platform that integrates context-provision services in a transparent and automatic way and provides an environment to facilitate the development of context-aware ubiquitous applications. OpenCOPI enables the integration of services provided by distinct sources, ranging from services offered by simple ad hoc systems to more complex services, such as services provided by context-provision middleware. In addition, OpenCOPI meets many requirements of Ubiquitous Computing that, in general, are not met by isolated context-provision middleware. Among these requirements, we can highlight (i) an *adaptive behaviour*, in which applications dynamically adapt themselves according to available services in the environment and (ii) the fact that applications should be completely independent of specific concrete services, so that OpenCOPI enables to build applications in a high abstraction level by specifying *abstract activities*.

Using OpenCOPI, applications only need to communicate with it to make use of services provided by different underlying context-provision middleware. Moreover, OpenCOPI provides *service selection* and *composition* mechanisms as well as an *adaptation* mechanism. Service selection and composition mechanisms consider the requirements of functional and non-functional applications, e.g. service quality metadata (*Quality of Service*, QoS) and context quality (*Quality of Context*, QoC).[9] The adaptation mechanism enables to adapt the application execution due to service failures, service quality fluctuation, emergence of new services and user mobility. Considering all these characteristics, OpenCOPI enables complex ubiquitous applications to be easily built.

OpenCOPI architecture is based on *Service-Oriented Architecture* (SOA) [12] and on the semantic Web services technology [5]. SOA has become a popular approach in Ubiquitous Computing domains due to its characteristics such as loose-coupling, stateless and platform independence, thus making SOA an ideal candidate for integrating ubiquitous services [38]. In our work, according to the SOA approach, context-provision middleware are *providers* that provide context services to ubiquitous applications (*clients*), so that OpenCOPI works as a mediator between these players. In turn, the

semantic Web services technology is an SOA implementation and therefore enables to build ubiquitous systems with loose-coupling and platform-independence characteristics. Moreover, OpenCOPI describes services through *semantic descriptions* that allow the discovering and composition of services through inference mechanisms. Such semantic descriptions are used to build *semantic workflows*, which are abstract descriptions of applications that specify the order in which a set of (abstract) activities is performed by various services to achieve the application goals [1]. Semantic workflows completely decouple the workflow activities from the underlying services in order to enable the development of applications independently of the available services or context-provision middleware.

In this paper, we extend our previous work about OpenCOPI on the following aspects by (i) detailing the composition and selection services, which were briefly presented in [13]; (ii) including a running example and using it to illustrate all the presented mechanisms and showing how the abstract workflow is defined; (iii) performing a more comprehensive evaluation of the proposed mechanisms, namely service composition, selection and adaptation, as well as workflow execution, by using execution plans more complex than those evaluated in our previous work and (iv) including the evaluation of the implementation effort to build drivers to different context-provision middleware, which is a task required by OpenCOPI. In addition, in this paper, we extend the adaptation algorithm used in OpenCOPI and assess such algorithm by using different configuration strategies to show that the overhead introduced by the adaptation is not significant.

This paper is organised as follows. Section 2 presents the case study explored in this paper and used to illustrate the OpenCOPI facilities. Section 3 describes the OpenCOPI middleware platform. Section 4 discusses *AdaptUbiFlow*, the component responsible for the OpenCOPI's selection and workflow adaptation algorithms. Section 5 presents an evaluation focused on service composition, service selection, workflow adaptation and OpenCOPI's overhead analysis. Section 6 discusses related work. Finally, Section 7 contains the final remarks.

2. Running example

This section presents a case study that is used along this paper. This case study is an application from the Gas & Oil Industry domain that monitors the oil well in production through a pumping unit machine in order to detect the need of changing the pumping unit settings. Modifications may be necessary to increase the oil production and/or to decrease the abrasion of some equipment of the pumping unit. Depending on the situation, the application can trigger actions to make changes or directly notify the human responsible for taking decisions about the pumping unit reconfiguration. This application was chosen because it uses different types of context information provided by many sources.

To exemplify the use of OpenCOPI in the context of this application, we have selected the *burden* variable to be monitored, which denotes the charge of oil extracted from a well at each cycle of movement of a pumping unit. Each pumping unit has a specific maximum value of burden (*maxBurden*) for its correct operation. If this value is abruptly reached, the pumping unit operation must be stopped quickly to prevent its damage (a reactive strategy). Furthermore, there is an intermediary value (*intermValue*), in which actions may be taken (in a proactive way) to prevent the pumping unit to be stopped, consequently avoiding loss of production and risks to the equipment. The complete case study description, including the service providers, possible service compositions and services

metadata can be found at the following URL: http://consiste.dimap.ufrn.br/projects/opencopi/.

Figure 1 shows the workflow that represents the case study application in which each activity is performed at least by one service. The execution starts in the first activity, *SubscribeBurden*, which is related to a subscription to monitor the *burden* value of the pumping unit. If the current *burden* value is between the pumping unit's intermediary *burden* value and the maximum value, then the workflow follows *Flow1*. If the *burden* value is greater than the maximum value, then the workflow follows *Flow2*. *Flow1* encompasses activities to automatically change the *regime* (relation between the length of pumping unit's stem and its frequency, in cycles per minute) of the pumping unit operation. First, the *SearchRegimeOptions* activity looks for possible regimes of the pumping unit operation, in which each regime variable is composed of a stem length value and cycles per minute value. Then, the *SearchPreviousChanges* activity finds the regimes previously used in this pumping unit. The next step is to change the regime (*ChangeRegime* activity) and update this information (*UpdateRegime* activity) in the registry of changes. Finally, a search is performed for technicians available (*SearchTechnicians* activity) in the vicinity of the oil well and a message is sent to them (*SendMsgToEmployees* activity). Thus, they can check whether everything is running as expected. In turn, *Flow2* describes the situation in which the *burden* value is greater than the maximum limit of the pumping unit. To avoid damage to the pumping

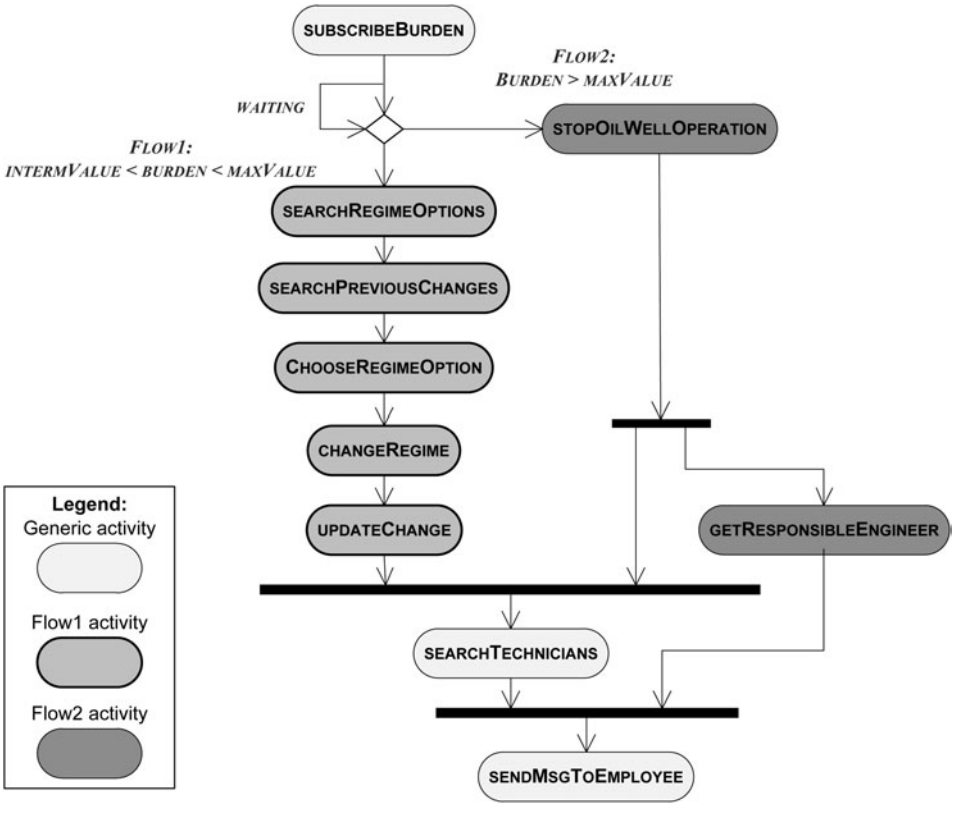

Figure 1. Case study workflow.

unit, the operation of the well is stopped (*StopOilWellOperation* activity). Next, a search is performed to find the engineer responsible for this oil well (*GetResponsibleEngineer* activity) and the technicians near to the oil well (*SearchTechnicians* activity). Finally, warning messages are sent to them (*SendMsgToEmployees* activity).

Figure 2 presents the interaction among the case study application, OpenCOPI, and the underlying service providers (including context-provision middleware). In this application, we are considering nine service providers that are detailed as follows: *WellDatabase*, *ChangeControlSystem*, *BMDimensioner*, *GPSLocalizationMiddleware*, *WifiLocalizationMiddleware*, *CellularLocalizationMiddleware*, *HRDatabase*, *GSMPlatform* and *MailPlatform*. These different service providers offer services that are consumed by the application through OpenCOPI. In this approach, the application is not aware of the underlying service providers that supply the services consumed by the application. In OpenCOPI, each activity of the workflow can be performed by one or more services,[10] and a service is able to perform an activity if such service satisfies the functional and non-functional requirements of an activity.

WellDatabase provides information about oil wells, being responsible for asynchronously providing the current oil burden in each pumping unit. This provider abstracts a widget of Context Toolkit (CT) [11], a context-provision middleware. This widget monitors the pumping unit operation and triggers events to OpenCOPI through a CT driver. This service performs the first workflow activity (*SubscribeBurden*).

BMDimensioner is a platform that provides services to manage the configuration of the pumping unit's operation regime. The services provided by this platform are responsible for presenting the possible pumping unit configurations and changing them (e.g. switching an operation regime to another one or stopping the pumping unit operation). This platform provides services corresponding to the *SearchRegime*, *ChangeRegime* and *StopOilWell-Operations* activities (presented in Figure 1).

ChangeControlSystem stores and recovers configuration changes performed at the oil exploration equipment. In this case study, this system recovers which regimes were

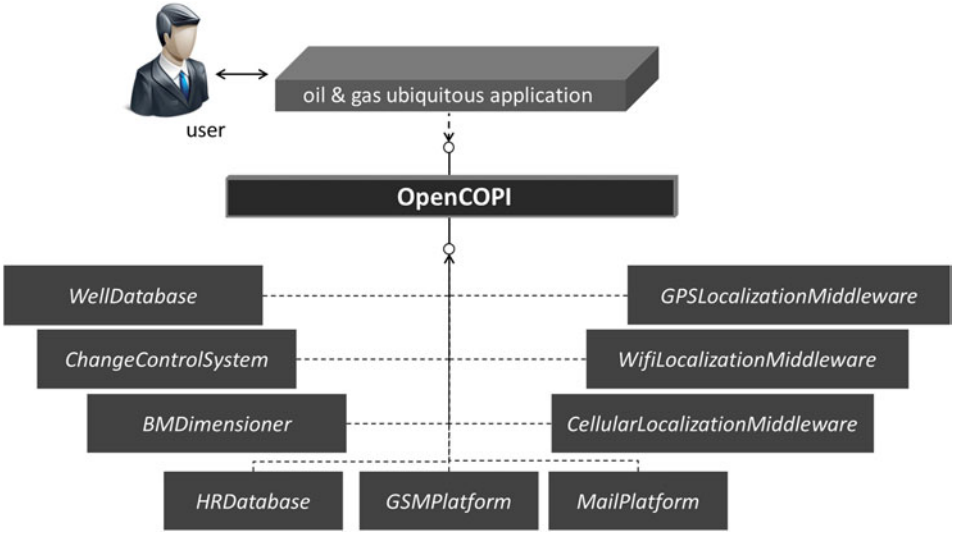

Figure 2. Service providers of the case study.

already used in each pumping unit. This system performs the *SearchPreviousChanges* and *UpdateChange* activities (presented in Figure 1).

The *GPSLocalizationMiddleware*, *WifiLocalizationMiddleware* and *CellularLocalizationMiddleware* platforms are responsible for providing the location of technicians spread over the oil field. Although they have the same functionality, each platform has a different quality level (QoS and QoC) and their quality will influence the selection of the service composition. Services provided by these platforms perform the *SearchTechnicians* activity (presented in Figure 1).

HRDatabase is a system that provides employees information, e.g. which employees are working at a given time. This system performs the *GetResponsibleEngineer* activity (presented in Figure 1).

Finally, *GSMPlatform* and *MailPlatform* provide services used to send messages to employees. As well as the location platforms, these platforms provide services with different quality level, thus performing the *SendMessageToEmployee* activity (presented in Figure 1).

3. OpenCOPI

This section presents OpenCOPI, a middleware platform, that integrates different service providers including, but not limited to, context-provision middleware platforms to make easier the task of developing context-aware adaptive ubiquitous applications. OpenCOPI enables the collaboration of different service providers to reach a high-level goal, which is to supply value-added services and contextual data to ubiquitous applications.

OpenCOPI provides its own communication model and an OWL (Web Ontology Language) [36] ontology-based context model, in which context is handled by adopting the semantic Web services perspective [23]. Under this perspective: (i) service providers publish their services using the OWL-S language [37]; (ii) ubiquitous applications are service consumers and (iii) OpenCOPI is a mediator that provides uniform access to services used by ubiquitous applications.

Moreover, OpenCOPI offers automatic service composition, orchestration, execution and adaptation to support these applications. Such composition and orchestration are performed through a goal-oriented *workflow*, which decouples applications from the underlying services that accomplish a given workflow goal and enables automatic service discovering, service selection, composition and orchestration. The composition model adopted by OpenCOPI is based on both services functionality and service metadata, thus enabling a better choice among the available services with similar functionality, e.g. different services that provide the same set of inputs and outputs.

In OpenCOPI, a *semantic workflow* is an abstract representation of a workflow described in terms of *activities*, thus representing the application execution flow. In other words, a workflow defines the sequence in which these activities must be executed. These activities are described in terms of semantic Web services descriptions. In OpenCOPI, each application has its own workflow and each workflow activity is a high-level description of an application task. A workflow is independent of specific concrete services as it separates the abstract activity from the concrete services that execute the activity. This is useful mainly in cases where there are several similar available services[11] offered by different providers. In such cases, the service that best meets the non-functional user requirements (i.e. service quality) can be chosen to be executed based on a given high-level workflow. In order to execute a semantic workflow, it is necessary to create at least one concrete specification for the workflow, which is called *execution plan* and contains a

set of concrete, orchestrated Web services. Execution plans are built through an on-the-fly process of service discovering and composition, according to the semantic enriched interface of the selected services and the semantic workflow specification.

OpenCOPI also provides an adaptive mechanism (Section 4) that deals with service failures or other changes in ubiquitous environments, which are highly dynamic. Whenever a service failure happens, this mechanism automatically replaces the failing service by an equivalent (and available) one. In addition, the adaptation mechanism provided by OpenCOPI can reconfigure the application execution whenever there is a significant degradation in the quality of the services that are being used, emergence of new services with higher quality, or in case of user mobility. In this perspective, OpenCOPI supports the fault-tolerance requirement of ubiquitous environments, thus increasing the availability of such systems.

3.1 Architecture

OpenCOPI architecture encompasses two layers, namely *ServiceLayer* and *UnderlayIntegrationLayer* as depicted in Figure 3. The components of these layers are explained in Sections 3.1.1 and 3.1.2, respectively.

3.1.1 ServiceLayer

ServiceLayer is responsible for managing abstractions of services (OWL-S descriptions) supplied by service providers. The components of *ServiceLayer* use such abstractions to support workflow creation and execution, service selection, service composition and

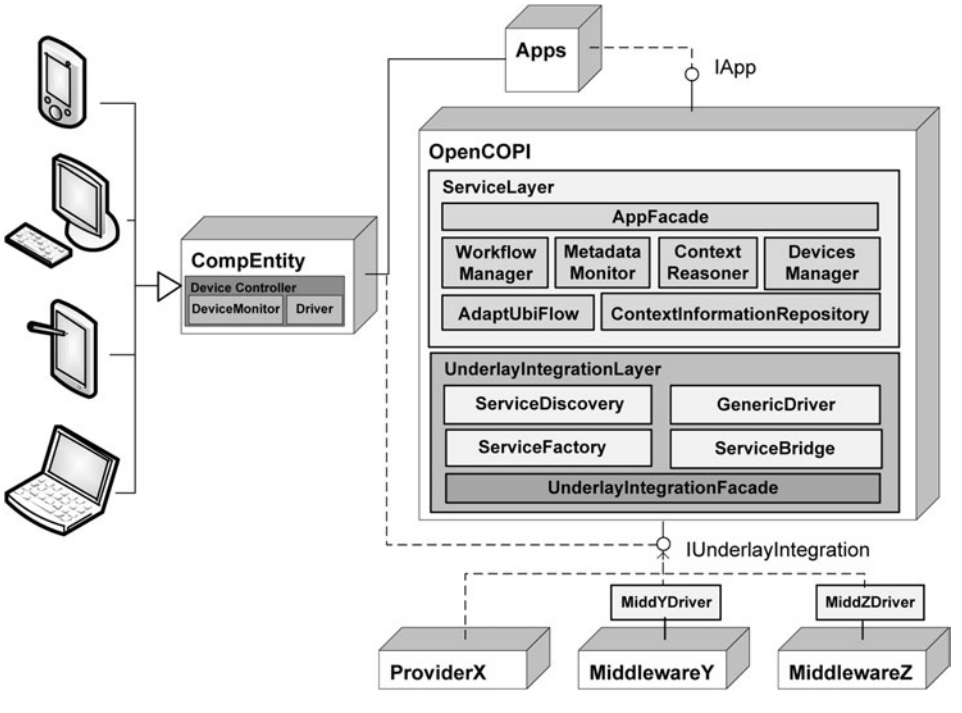

Figure 3. OpenCOPI architecture.

adaptation. In addition, they support context reasoning, context storing, among other functionalities related to ubiquitous applications. These applications should implement the *IApp* interface to communicate with the *AppFacade* facade, which is responsible for receiving requests from the applications and forwarding them to the components of *ServiceLayer*. Figure 4 shows a code snippet of the *IApp* interface with the basic operations provided by OpenCOPI. Such interface enables the user to create, open and execute a workflow and to specify his/her preferences regarding the selection and adaptation processes, as detailed in Sections 3 and 4.

WorkflowManager. This component manages workflows, which are based on activities described from abstractions of services provided by context-provision middleware. The *WorkflowManager* component is composed of the following four (sub)components, which provide support for specifying semantic workflows and generating execution plans. The *ServiceManager* component is responsible for the following: (i) importing OWL-S descriptions from service providers to OpenCOPI and validating such descriptions and (ii) providing capabilities to search for basic concepts in the knowledge base (ontology) using inputs and outputs of the available semantic Web services. *SemanticComposer* is responsible for discovering and composing Web services according to the semantic workflow specifications, i.e. it makes the mapping between workflow goals and Web services. First, it tries to discover the services (among those available at the *SemanticServicesRepository*) that can be used to compose the execution plan, given the goals specified in the application request. Next, it tries to combine the discovered services in order to consume all inputs and pre-conditions and to produce all outputs and effects specified in the request. Then, the combined services are organised by *SemanticComposer* according to the message flow between outputs of a service and inputs of the subsequent service. The *SemanticServicesRepository* component stores both the ontologies that describe the Web services and the execution plans. Finally, the *WorkflowExecutor* component supports the workflow execution. It receives the execution plans generated by the *SemanticComposer* and chooses a plan to be executed taking into account the QoS/QoC of the service providers included in the plan. At run-time, *WorkflowExecutor* executes the selected execution plan by making calls to the services provided by the underlying context-provision middleware. In case of failure in one execution plan, the *WorkflowExecutor* component chooses another execution plan (if any) in an attempt to successfully execute the workflow.

MetadataMonitor. This component is responsible for gathering metadata about services and context provided by context-provision middleware in order to feed the *ContextInformationRepository*. OpenCOPI adopts an *Service-Level Agreements* [19] approach in which service providers publish the quality metadata of their services and

```
public interface IApp
{
   public void createWorkflow(Collection<Object> requiredElements);
   public void openWorkflow(File workflowOWLFile);
   public void executeWorkflow(SemanticWorkflow workflow);
   public void setQualityParameters(File qualityPreferences);
   public void setAdaptationParameters(File adaptationPreferences);
}
```

Figure 4. Code snippet of the *IApp* interface.

these metadata are used to select the services to be provided to the consumers. The quality metadata are obtained from *QoMonitor* [4], a generic metadata monitoring system that supports synchronous and asynchronous requests from OpenCOPI (or other middleware), recovers metadata from several context providers and sends them to OpenCOPI. By using *QoMonitor* system, OpenCOPI can abstract away the burden of dealing with the complexities related to synchronous and asynchronous metadata monitoring. In a nutshell, *QoMonitor* contains the following: (i) a *metadata repository*, which persists all QoS and QoC metadata provided by service providers; (ii) a *request handler*, which receives the requests from OpenCOPI (or other middleware using the *QoMonitor*), gathers the metadata and sends the responses to OpenCOPI and (iii) an *assessment module*, which is responsible for effectively monitoring and assessing QoS/QoC metadata of the services. In such component, there is a specific assessor for each given QoS/QoC parameter. More details about this monitoring system are presented in [4].

ContextReasoner. This component makes inferences about context data (low-level context) gathered through the several context-provision middleware to supply high-level and consistent context information for the applications.

ContextInformationRepository. This component stores context data and context metadata, thus supplying context data to both the *WorkflowManager* and the *ContextReasoner* components.

AdaptUbiFlow. The *AdaptUbiFlow* component [21] is responsible for the *adaptation* process in OpenCOPI. As we previously mentioned, ubiquitous environments are highly susceptible to changes, and several of them are unpredictable. In this context, *AdaptUbiFlow* was specifically designed to deal with adaptation. In *AdaptUbiFlow*, an adaptation means to replace the previously selected execution plan for another execution plan, in which this new plan needs to meet the objectives of applications specified in the workflow. This component works directly with the *MetadataMonitor* and *WorkflowManager* components to identify a fault and automatically change the execution flow in order to use another execution plan.

DevicesManager. The *DevicesManager* component manages the use of computational entities in order to enable the migration of an application from a device to another in case of user mobility or in case of resource limitations of the currently active device (e.g. low level of energy, low level of free memory). These devices can provide services to be consumed by applications, including services to provide context information of the device (e.g. location, battery level, free memory level). When a user wants to start the execution of an application, the *DevicesManager* component considers the device that is currently used by the user as the *active device*, through which all interactions with the user are made. When the user needs to change the current computational device (e.g. due to his/her mobility), a new device is transparently reconfigured as the current active device. After this reconfiguration, all interactions between user and applications are made through this new active device. Each computational device must have a *DeviceController* component, which enables the migration from a device to another and it is composed of two subcomponents, *DeviceMonitor* and *Driver*. The *DeviceMonitor* component monitors the device resources, such as level of energy, level of memory, level of free processor, device location and services provided by the device itself. For instance, consider a situation in which the user is using his/her tablet. When the *DeviceMonitor* detects that such tablet is low-level energy, it notifies the OpenCOPI's *DevicesManager* component, and then the *DevicesManager* can evaluate the need of migrating the active application from the user's tablet to user's smartphone or another user's device. This notification is possible because each *DeviceMonitor* provides a specific service to make notifications about the status of

the monitored devices resources. This service can be consumed by the OpenCOPI's *DevicesManager* through device's *Driver* component. Moreover, the *DevicesManager* component enables the device to provide other (non-monitoring) services, e.g. a location service using an embedded GPS that can be consumed by applications of other OpenCOPI's users. This is an interesting characteristic because it allows sharing of resources, thus enabling devices with low computational resources (cheaper devices, in general) to use resources of more powerful devices in terms of computational resources, which typically are more expensive.

3.1.2 UnderlayIntegrationLayer

UnderlayIntegrationLayer is responsible for integrating service providers, mainly (but not only) context-provision middleware, thus performing context conversion whenever it is needed (from middleware context model to the OpenCOPI context model) and communication protocol conversion. This latter type of conversion is required to integrate middleware that are not compliant to Web services protocols and standards, but instead adopt different protocols (e.g. sockets, RMI, CORBA). The *IUnderlayIntegration* interface links service providers and OpenCOPI's *UnderlayIntegrationLayer*. The components of *UnderlayIntegrationLayer* are in charge of integrating service providers:

ServiceDiscovery. This component is responsible for discovering services into environment and registering them in OpenCOPI. When discovered, traditional Web services can be directly added to the *SemanticServiceRepository* since these services do not deal with context information and therefore do not require context model transformations. However, services provided by context-provision middleware need additional operations to be properly integrated. For each context-provision middleware, it is necessary to build a *driver* to implement the context model transformation, from the middleware context model to the OpenCOPI context model. For context-provision middleware that does not provide APIs complaint to the Web services technology, it is necessary to build drivers in order to abstract away from the different APIs and to allow the transparent access to the context data provided by these context-provision middleware (see Figure 3). The driver is also responsible for issuing context queries and subscriptions from OpenCOPI to the underlying context-provision middleware. Each driver should extend the *GenericDriver* component.

GenericDriver. This component implements the OpenCOPI side of the interface and defines operations for context model transformation and communication between a specific context-provision middleware and OpenCOPI. Figure 5(a) shows an excerpt of code in the Java programming language regarding the implementation of the *GenericDriver* component. Figure 5(b) shows part of the driver built to the CT [11] context-provision middleware. Such driver extends the *GenericDriver* class provided by OpenCOPI.

ServiceFactory and *ServiceBridge.* The *ServiceFactory* component is responsible for creating context services that encapsulate the specific middleware APIs; the *ServiceBridge* component inserts these context services into the *SemanticServiceRepository* component (subcomponent of the *WorkflowManager* component). Thus, each service provided through the middleware API is represented by a Web service created by the *ServiceFactory* component. Each one of these Web services uses the driver tailored for the specific underlying middleware. Finally, *UnderlayIntegrationFacade* intermediates the interaction between OpenCOPI and the underlying middleware.

3.2 Context model

The OpenCOPI context model is specified as an ontology [13]. Such context model was inspired in CONON ontology [35], but it includes extensions to allow the execution of our workflow-based strategy. For example, *Task* and *Object* classes were inserted in the ontology because they are used to create the workflow activities, in which each activity is

(a)
```
public abstract class GenericDriver {

    /** Object responsible for the connection with the respective underlying middleware */
    private Object middlewareConnectObject;

    /** Original state of the persisted object. Used in case of compensatory action */
    private Object originalState;

    /** 'true' if this service supports compensatory action and 'false' otherwise */
    private Boolean supportCompAction;

    /** 'true' if this service supports rollback and 'false' otherwise */
    private Boolean supportRollback;

    /** Method responsible for performing compensatory action */
    public abstract void doCompensatoryAction();

    /** Method responsible for performing rollback */
    public abstract void doRollback();

}
```

(b)
```
public class CTDriver extends GenericDriver
{
    ...
    supportRollback = false;
    supportCompAction = false;
    middlewareConnectionObject = new BaseObject();

    private CTListener listener;
    private ClientSideSubscriber cs;

    public CTListener subscribe()
    {
        // Send the subscription to the ContextToolkit widget
        Error errorS = middlewareConnectObject.subscribeTo(listener, remoteWidgetId,
                        remoteWidgetHost, remoteWidgetPort, cs);

        widgetSubId = cs.getSubscriptionId();
        return listener;
    }

    public void unsubscribe()
    { middlewareConnectObject.unsubscribeFrom(widgetSubId); }
}
```

Figure 5. Code of the *GenericDriver* component (a) and part of the driver to CT that extends the *GenericDriver* class (b).

composed by a tuple ⟨*Task*, *Object*⟩. Similarly to the CONON approach, the OpenCOPI ontology is implemented using the OWL language.

Since applications and services are commonly grouped as a collection of sub-domains in ubiquitous environments, the OpenCOPI ontology is organised in two layers, namely *Generic Ontology Layer* and *Domain Ontology Layer*. The *Generic Ontology Layer* aggregates common concepts that can be modelled using a generic context model, which is shared by all of the ubiquitous subdomains. In turn, the *Domain Ontology Layer* groups different and extensible ontologies that describe particular ubiquitous environments, e.g. home domain, office domain and petroleum exploration domain.

Figure 6 shows the representation of the OpenCOPI *Generic Context* ontology. The model is structured around *objects* and *tasks* [14]. *Tasks* represent operations implemented by one or more Web services, while *objects* can be physical or conceptual things, including people, equipments, computational entities, messages and locations, as well as can be used to describe *inputs*, *outputs*, *pre-conditions* and *effects* (IOPEs) related to the *tasks*. Furthermore, it is possible to define associations between *objects* and *tasks*.

Figure 7 presents a partial definition of the *Oil Exploration* domain ontology (used in the running example presented in this paper), which describes specific sub-classes (objects and tasks) related to oil exploration environments. This ontology shows several relationships between objects and between tasks and objects. An example of an *object–object* relationship is the *locatedIn* relationship between the *Equipment* and *Location* objects, which means that an equipment is located in a particular location. An example of a *task–object* relationship is the relationship between the *Send* task and the *Message* object, which means that a message (email or SMS) can be sent (using some communication channel).

3.3 *Managing workflows*

In OpenCOPI, the user needs to specify a workflow that represents the application activities. Based on this specification, OpenCOPI performs a *service composition*, i.e. searches for services that meet each specified activity and creates the possible execution plans. If more than one execution plan is created, OpenCOPI needs to select one of them to be executed. This selection is based on service and context quality

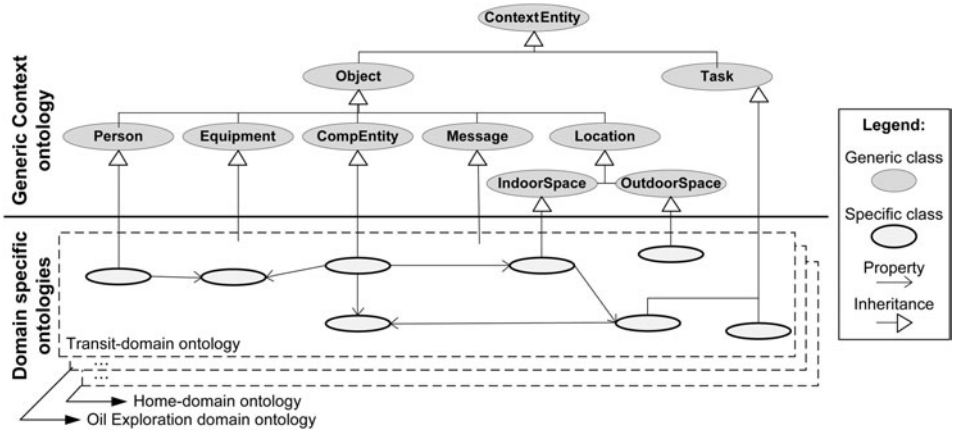

Figure 6. Partial description of the *Generic Context* ontology.

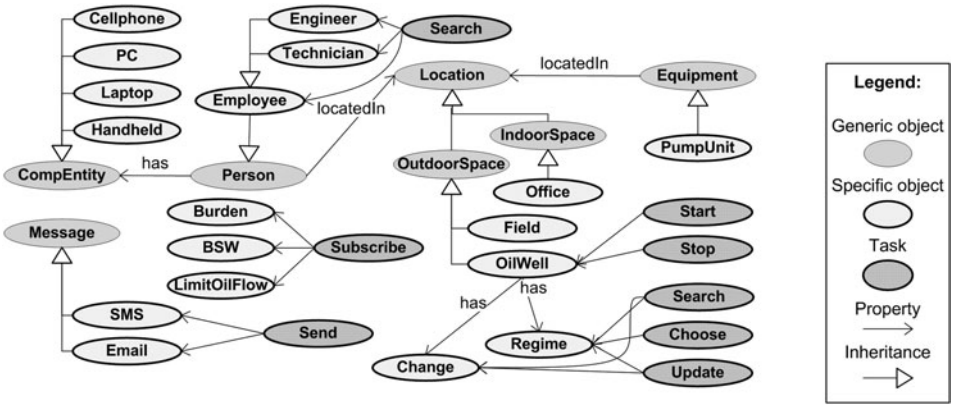

Figure 7. Partial description of the *Oil Exploration* ontology.

(respectively, QoS and QoC). After selecting an execution plan, OpenCOPI can start the execution of the execution plan by invoking each service. However, during the execution of the execution plan, it may be necessary to trigger an adaptation process. This adaptation process can be started due to a fault or quality degradation of a service or if new services (with better quality) emerge. Users can configure the service selection and adaptation processes in an easy way through an XML configuration file. Details about the workflow specification are presented in Section 3.4, while details of the creation and execution of execution plans are presented in Sections 3.5 and 3.6, respectively. Section 4 presents the *AdaptUbiFlow*, which is the component responsible for selecting execution plans and performing the adaptation process. The whole process is depicted in Figure 8.

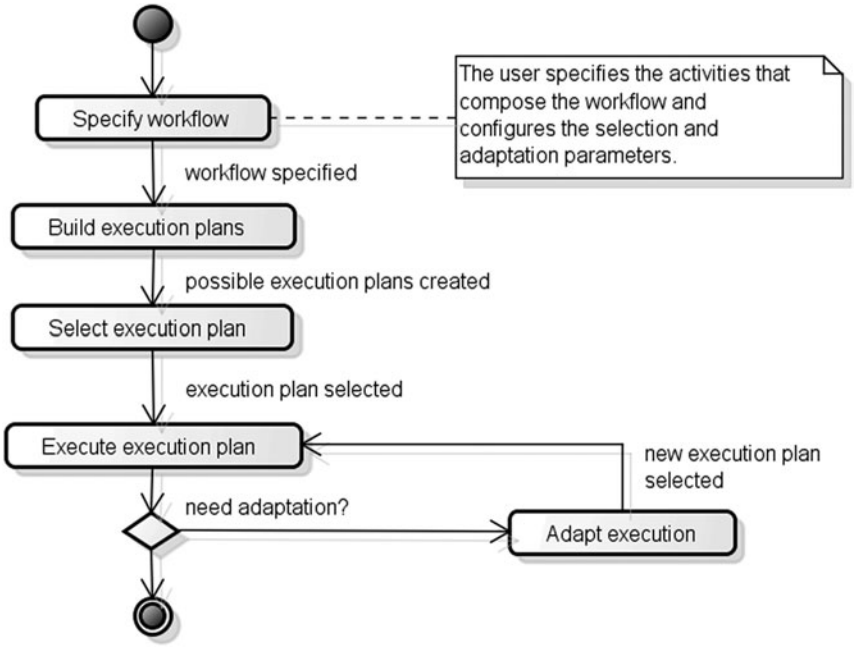

Figure 8. Workflow management process in OpenCOPI.

3.4 Workflow specification

In OpenCOPI, a workflow is represented by a direct acyclic graph in which each intermediary node represents a specific service and each directed edge represents the two services of the execution sequence. Each graph begins with an *initial node* and ends with a *final node*. *Initial* and *final nodes* do not represent any service, but they are used to indicate the beginning and the ending of the graph. Each complete path between the *initial node* and the *final node* is an *execution path*, which represents a possible execution plan in the workflow. Thus, the graph represents the workflow with all possible execution plans. In Figure 9, a graph with some possible execution paths are shown, for instance (i) S1 → S2 → S3 → S4 → S5, (ii) S1 → S2′ → S3 → S4″ → S5, (iii) S1 → S6 → S4″ → S5 and so on.

The workflow specification consists of describing the activities of a business process through the combination between a *task* and an *object* (classes in the context model). OpenCOPI provides an assistant that enables users to select these activities without the need of creating each of them since they are already pre-defined according to the set of *tasks* and *objects* specified in the context ontologies. These activities are, abstract meaning, that the binding to the concrete service that will perform the activity is done at run-time. The assistant shows a *Task* list and an *Object* list, in which the user can select a ⟨*Task, Object*⟩ tuple (e.g. ⟨*Subscribe, BloodPressure*⟩ or ⟨*Send, Message*⟩) to describe each activity of a semantic workflow. The assistant also shows the list of possible IOPEs related to each specified activity, enabling the user to select which IOPEs should be added to the activity definition. Thus, the user describes only the abstract activities of the workflow, so that when executing the workflow, OpenCOPI performs inferences on the ontologies to discover which services perform each workflow activity and composes the possible execution plans with these discovered services.

Figure 10 shows the process of creating workflow activities. The user requests, via OpenCOPI's GUI, the creation of a new activity. The request is received by the *WorkflowManager* component, which is responsible for creating and editing the workflow. This component asks the *ServiceManager* component for the list of possible *tasks*. Then, *ServiceManager* recovers the concepts described in the domain ontology stored in the *SemanticServiceRepository* component, loading all concepts in memory, and returning the list of tasks and respective objects (*objects* related with each *task*) to the *WorkflowManager* component. Next, the *ServiceManager* component shows these concepts for the user, which selects the desired *task* and the *object* (e.g. the *Subscribe* task

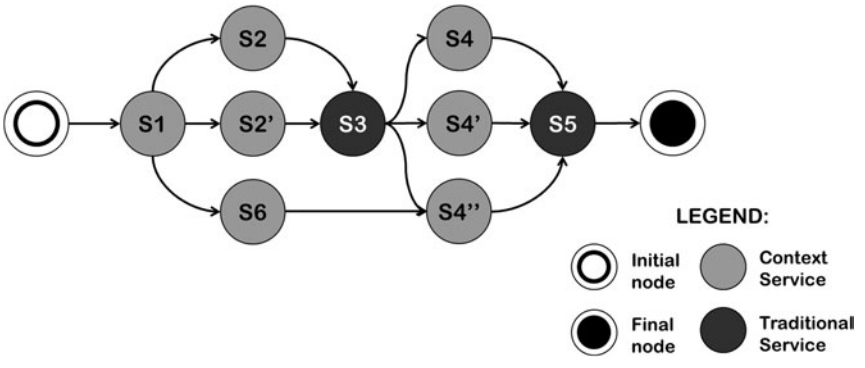

Figure 9. Example of graph representation for a workflow.

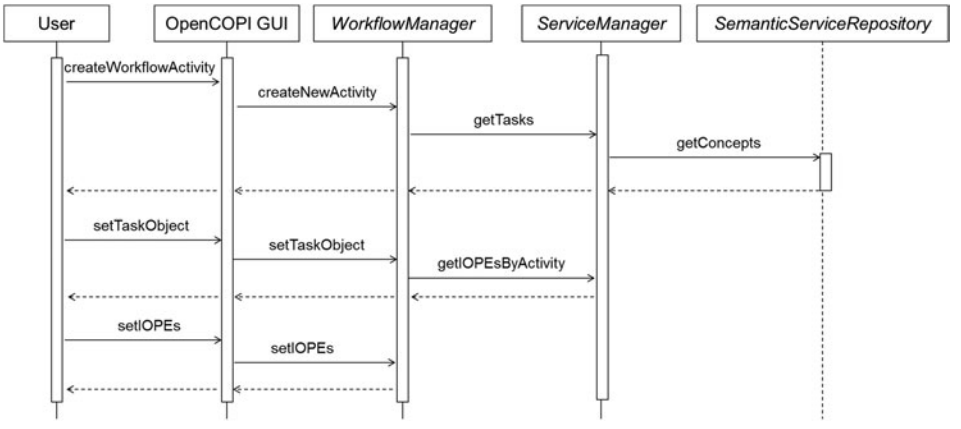

Figure 10. Process of creating a workflow activity.

and the *Burden* object), so that this tuple is sent to *WorkflowManager*. Then, *WorkflowManager* asks the *ServiceManager* component for the list of respective IOPEs of this activity. With the concepts in memory, the *ServiceManager* returns the list of possible IOPEs for the *WorkflowManager*, which forwards this list to the user. Then, the user selects which IOPEs should be considered for this activity, thus finishing the process of including an activity. This process is repeated for each activity of the workflow.

Once the user finishes specifying a workflow using the provided assistant, OpenCOPI generates a representation of such workflow in the OWL ontology language, as illustrated in Figure 11. Figure 11(a) depicts part of the OWL description that represents the workflow activities. In this figure, it is possible to identify the activity arbitrarily named *Activity_1*, which is composed of the *Subscribe task* and the *Burden object*, thus resulting in the ⟨*Subscribe*, *Burden*⟩ tuple, which describes such activity, as well as the associated inputs (*PumpUnit* and *OilWell*, respectively, represented by the elements *Input_4* and *Input_5*) and outputs (*Regime*, represented by the element *Output_6*). Figure 11(b) shows another part of the OWL workflow description that presents the configuration of the execution process of the activities. In this specific case, we have a list of activities in which the first activity to be executed is represented by the *Activity_1* index; next, the activity represented by the *Activity_7* index is executed and so on. Such specification is used by the OpenCOPI's semantic reasoner for making inferences over the ontologies of the available Web services and then discovering the services that are suitable to perform the activities that compose the semantic workflow that specifies the business process regarding the ubiquitous application.

The OpenCOPI's assistant also allows using flow control connectors (condition, repetition, etc.) at the semantic workflow specification. Figure 12 presents the assistant screenshot in which three activities have already been added to the workflow and the activity *Choose_RegimeOption* composed of the *Choose* task and the *RegimeOption* object is being added.

Figure 13 presents a screenshot showing the second step of the addition of an activity. This step consists of selecting the IOPEs, thus showing the inputs (*RegimeList* and *ChangeList*) and an output (*Regime*) selected for this activity. When the user presses the *Finish* button, the new activity is added to the workflow.

(a)

```
<process:AtomicProcess rdf:ID="Activity_1">
    <process:hasLocal>
        <process:Local rdf:ID="Task_2">
            <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://localhost:8080/OpenCOPI/kb/concepts.owl#Subscribe
            </process:parameterType>
        </process:Local>
    </process:hasLocal>
    <process:hasLocal>
        <process:Local rdf:ID="Object_3">
            <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://localhost:8080/OpenCOPI/kb/concepts.owl#Burden
            </process:parameterType>
        </process:Local>
    </process:hasLocal>
    <process:hasInput>
        <process:Input rdf:ID="Input_4">
            <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://localhost:8080/OpenCOPI/kb/concepts.owl#PumpUnit
            </process:parameterType>
        </process:Input>
    </process:hasInput>
    <process:hasInput>
        <process:Input rdf:ID="Input_5">
            <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://localhost:8080/OpenCOPI/kb/concepts.owl#OilWell
            </process:parameterType>
        </process:Input>
    </process:hasInput>
    <process:hasOutput>
        <process:Output rdf:ID="Output_6">
            <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
                http://localhost:8080/OpenCOPI/kb/concepts.owl#Regime
            </process:parameterType>
        </process:Output>
    </process:hasOutput>
</process:AtomicProcess>
```

(b)

```
<process:CompositeProcess rdf:about="#SemanticWorkflowProcess">
    <process:composedOf>
        <process:Sequence>
            <process:components>
                <process:ControlConstructList>
                    <list:first>
                        <process:Perform>
                            <process:process rdf:resource="#Activity_1"/>
                        </process:Perform>
                    </list:first>
                    <list:rest>
                        <process:ControlConstructList>
                            <list:first>
                                <process:Perform>
                                    <process:process rdf:resource="#Activity_7"/>
                                </process:Perform>
                            </list:first>
                            <list:rest rdf:resource="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#nil"/>
                        </process:ControlConstructList>
                    </list:rest>
                </process:ControlConstructList>
            </process:components>
        </process:Sequence>
    </process:composedOf>
    <service:describes rdf:resource="#SemanticWorkflowService"/>
</process:CompositeProcess>
```

Figure 11. Partial OWL description of a workflow.

3.5 Creating execution plans

Once the workflow is completed, the next step is the creation of the execution plans. Such creation is done by the *SemanticComposer* component in three steps: (i) discovering which registered services can be used to satisfy the activity's IOPEs; (ii) matching the selected services in order to consume inputs, produce outputs and meet pre-conditions and effects of each workflow activity and (iii) ordering the services of each created execution plan, following the sequence of activities defined in the workflow specification (service orchestration). OpenCOPI uses the matching algorithm presented

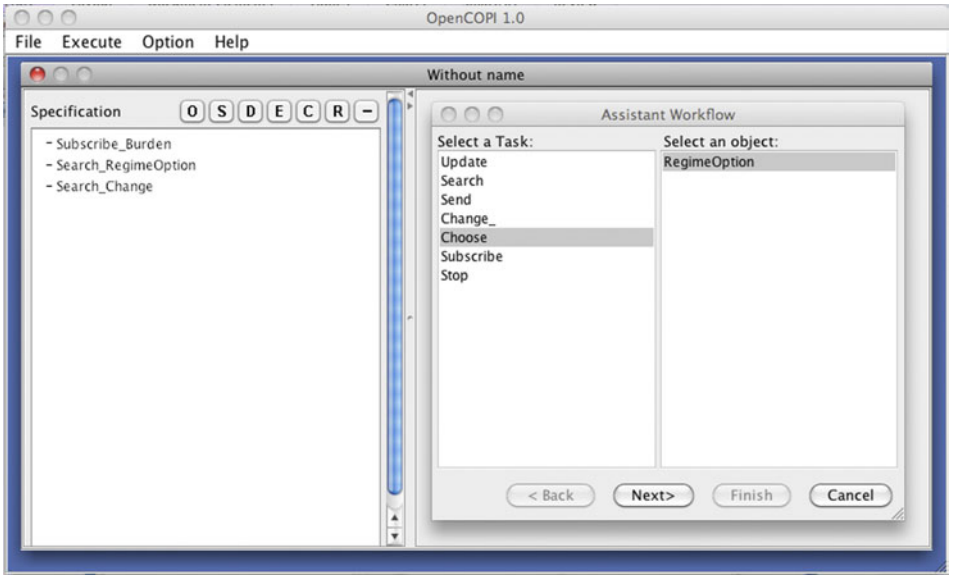

Figure 12. Assistant for creating a workflow (part 1 of 2).

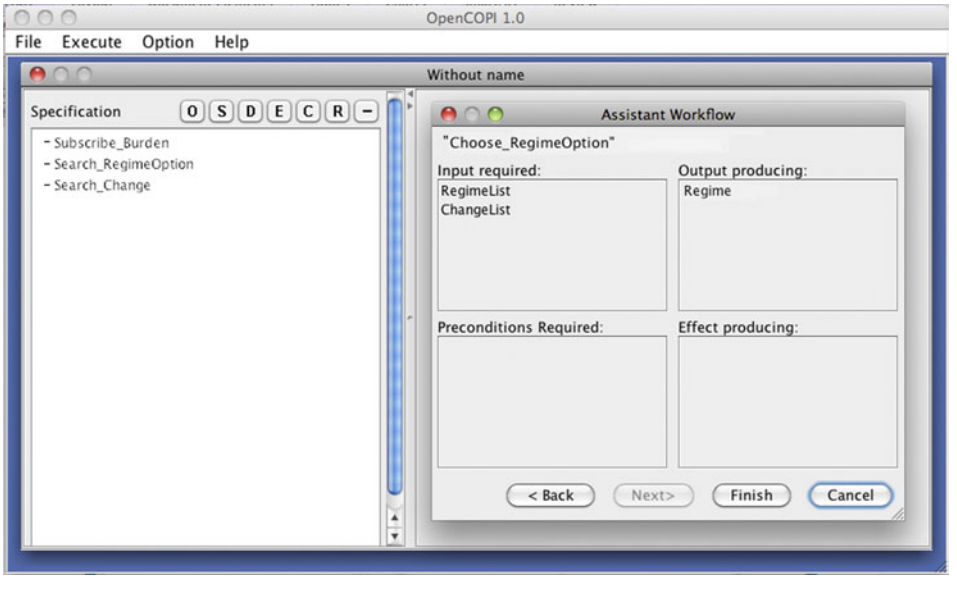

Figure 13. Assistant for creating a workflow (part 2 of 2).

by Mendes et al. [24] that composes Web services from a semantic workflow specification taking into account the required inputs and pre-conditions and the produced outputs and effects regarding each activity. Similar to workflows, the produced execution plans are also described in the OWL ontology language, as illustrated in Figure 14. This figure shows part of the description of an execution plan in which, for example, the service described by the *SendSMStoEmployees.owl* service ontology is

```
<process:Perform rdf:about="http://localhost:8080/OpenCOPI/ExecutionPlan.owl#Perform_21">
    <process:process
            rdf:resource="http://www.example.org/owls/SendSMStoEmployees.owl#SendSMStoEmployeesProcess" />
    <process:hasDataFrom>
        <process:InputBinding>
            <process:toParam
                    rdf:resource="http://www.example.org/owls/SendSMStoEmployees.owl#SendSMStoEmployeeServiceList" />
            <process:valueSource>
                <process:valueOf>
                    <process:fromProcess>
                        <process:Perform rdf:about="http://localhost:8080/OpenCOPI/ExecutionPlan.owl#Perform_18" />
                    </process:fromProcess>
                    <process:theVar
                            rdf:resource="http://www.example.org/owls/SearchClosestTechniciansGPS.owl#SearchClosestTechniciansGPSServiceReturn" />
                    </process:theVar>
                </process:valueOf>
            </process:valueSource>
        </process:InputBinding>
    </process:hasDataFrom>
```

Figure 14. Partial OWL description of an execution plan.

immediately executed after the service described by the *SearchClosestTechniciansGPS.owl* service ontology.

3.6 *Executing a workflow*

*Fig*ure 15 shows the sequence of messages started after a user requesting the execution of a workflow. Message *1* is sent to OpenCOPI through *AppFacade*, the facade responsible for receiving requests of all applications. Then, the request is sent to *WorkflowManager* (message *2*), which generates the possible execution plans and asks *AdaptUbiFlow* (message *3*), to select one of the generated execution plans. Next, *AdaptUbiFlow* queries *ContextInformationRepository* about the service metadata (message *4*), receives the query response (message *5*) and calculates which execution plan has the best quality (this process will be presented in Section 4). After this step, *AdaptUbiFlow* sends the selected execution

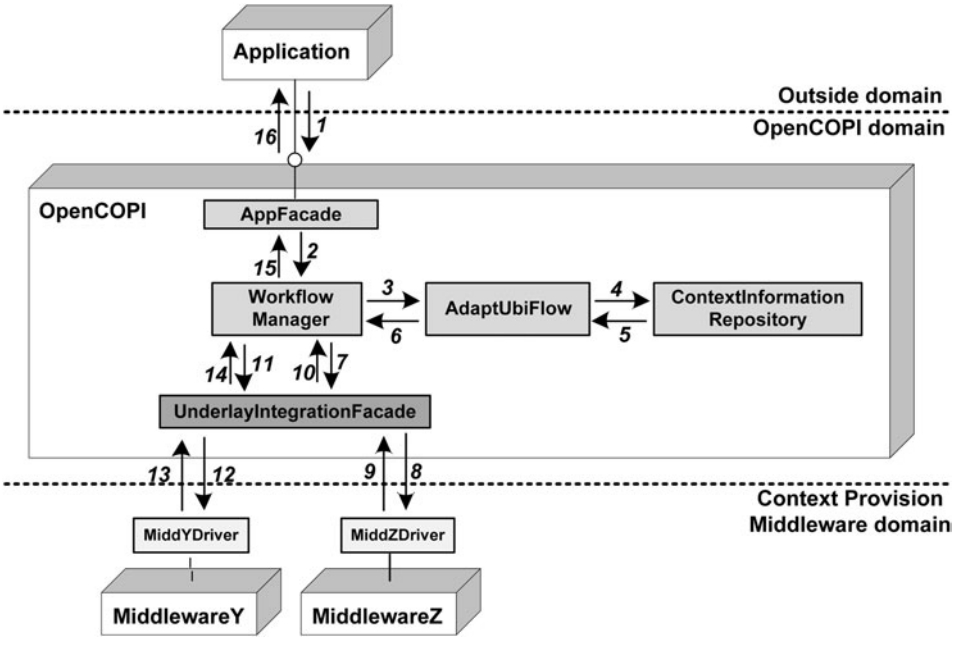

Figure 15. Execution of a workflow in OpenCOPI.

plan to *WorkflowManager* (message *6*). In the next step, *WorkflowManager* starts the execution of the selected execution plan through service invocations (messages *7* and *11*). These invocations are performed through *UnderlayIntegrationFacade* and forwarded to the providers of the corresponding services. These invocations are represented by messages *8* and *12*. The responses of those requests are represented by messages *9*, *10*, *13* and *14*.

4. *AdaptUbiFlow*

AdaptUbiFlow is the component responsible for the selection and adaptation processes in OpenCOPI. For each workflow, after creating the execution plans, *AdaptUbiFlow* selects the best execution plan (i.e. the plan with the best quality) to be executed. In addition, as we have already mentioned, ubiquitous environments are highly susceptible to changes and many of them are unpredictable. Thereby, another responsibility of *AdaptUbiFlow* is to support adaptation in the workflow execution. This component directly interacts with the *MetadataMonitor* and *WorkflowManager* components in order to identify faults. In case of a fault, it is necessary to automatically change the execution flow to another execution plan. In addition, *AdaptUbiFlow* can perform adaptation in other cases, such as: (i) when the user leaves the service's operation area due to his/her mobility; (ii) when the quality of a service provider decreases to a compromising level or (iii) when new services with higher quality become available. This section presents the execution plan selection and the adaptation processes.

4.1 Service metadata

OpenCOPI selects execution plans according to the quality metadata of services that compose each execution plan. These parameters are categorised (for didactical purposes) in two types: QoC and QoS. Although QoC describes the quality of the contextual information provided by a service, QoS describes the quality of the service. The meaning of each parameter is shown below, separated by category.

OpenCOPI considers the following QoC parameters, as proposed by Buchholz et al. [9]:

(1) *Precision*, which denotes the level of accuracy of the information provided by a given technology/technique for context provision. For example, a GPS receiver or an RFID reader can provide the location of a person with precision of few centimetres, while a triangulation algorithm based in 802.11 signal strength can provide a precision of few metres.
(2) *Correctness*, which denotes the probability that a piece of context information is correct. For example, consider a temperature sensor in a room. Internal problems in such sensor can generate wrong temperature values (e.g. measuring 30°C while the correct temperature would be 20°C). Thus, this parameter estimates how often a context information provided by a source will be unintentionally wrong due to internal problems.
(3) *Resolution*, which describes the granularity of the context information. For example, a service announces the room temperature as 25°C, but the room temperature can vary in different room's regions. If there are few thermometers in the room, the service is not able to supply the information with a proper granularity level.
(4) *Freshness*, which represents the time elapsed between the generation of the context data and the retrieval of this data by applications. Thus, a particular context data may be prioritised whether it is newer than other similar data.

The QoS parameters considered in our work are as follows:

(1) *Response time*, which represents the time between sending a service request to a service provider and receiving the response.
(2) *Availability*, which represents the probability that the service provider is up and the service is running, i.e. the service is ready for immediate use.
(3) *Performance*, which describes the number of service requests fulfilled by the service provider at a given period of time.

4.2 Service selection

As in general there are more than one execution plan for each workflow and the number of these execution plans depends on the amount of available services with the same functionality in the environment (at a given moment), it is necessary a *service selection* algorithm to choose which execution plan should be executed. This section presents the service selection algorithm supported by *AdaptUbiFlow*.

4.2.1 Aggregation functions

The process of selecting an execution plan begins with the calculation of the quality of each plan. The quality of an execution plan is determined by the quality parameter (QoS and QoC) values of all services contained in the execution plan. Before computing the execution plan quality, it is necessary to compute the *global quality* of each quality parameter. Global quality of a parameter means the quality parameter value for the whole execution plan, i.e. the value that represents the parameter of all services of this execution plan. The global quality of each parameter can be computed by aggregating the corresponding values for such parameter of all services in the respective execution plans. Different aggregation functions [2,39] are necessary to compute the global value of each parameter. Typical quality parameter aggregation functions are addition, multiplication, minimum and average relation (see Table 1). For instance, *Response time* is the QoS parameter used to measure the response time to execute each service. Thus, the value of the *Response time* parameter for an execution plan EP ($q_{\mathrm{R}}(\mathrm{EP})$) is the sum of the *Response time* values of all services s ($q_{\mathrm{R}}(s)$) that compose EP. The *Availability* QoS parameter can be aggregated through a multiplication function of the availability value of each services of the execution plan EP ($q_{\mathrm{A}}(\mathrm{EP})$). The *Performance* QoS parameter describes the number of service requests served by the service provider at a given period of time. Thus, the performance of an execution plan EP ($q_{\mathrm{P}}(\mathrm{EP})$) is limited to the service with the smaller value for *Performance* attribute. *Freshness* QoC parameter describes the life span of

Table 1. Aggregation functions.

Type	Parameter	Function
Addition	*Response time*	$q_{\mathrm{R}}(\mathrm{EP}) = \sum_{s=1}^{m} q_{\mathrm{R}}(s)$
Multiplication	*Availability*	$q_{\mathrm{A}}(\mathrm{EP}) = \prod_{s=1}^{m} q_{\mathrm{A}}(s)$
Minimum	*Performance*	$q_{\mathrm{P}}(\mathrm{EP}) = \min_{s=1}^{m} q_{\mathrm{P}}(s)$
Average	*Freshness*	$q_{\mathrm{F}}(\mathrm{EP}) = \frac{1}{m}\left(\sum_{s=1}^{m} q_{\mathrm{F}}(s)\right)$
Multiplication	*Correctness*	$q_{\mathrm{C}}(\mathrm{EP}) = \prod_{s=1}^{m} q_{\mathrm{C}}(s)$
Multiplication+	*Resolution*	$q_{\mathrm{RS}}(\mathrm{EP}) = \prod_{s=1}^{m} \frac{q_{\mathrm{RS}}(s)}{q_{\mathrm{RS\,max}}}$
Multiplication+	*Precision*	$q_{\mathrm{PR}}(\mathrm{EP}) = \prod_{s=1}^{m} \frac{q_{\mathrm{PR}}(s)}{q_{\mathrm{PR\,max}}}$

context information, i.e. how long time ago the context information was created. Thus, the value of this QoC parameter for an execution plan EP (q_F(EP)) is the average of context life span of all services of EP. Similar to the *Availability* parameter, the aggregation function for the *Correctness* parameter is also a multiplication; however, it refers to the correctness level of context information provided by the execution plan's services. Finally, the *Resolution* QoC parameter can be aggregated (q_{RS}(EP)) through a special multiplication, in which the operators are the ratio between the resolution of each service and the highest resolution of equivalent services. For instance, being *A*, *B* and *C* equivalent services and *C* the service with biggest resolution among these three services, the operator value for service *A* is ($q_{RS}(A)/q_{RS}(C)$). This is necessary because each service type has its own unit for the resolution parameter. For example, luminosity sensors gather luminosity intensity in *lux* unit and location sensors gather user location in *metres* unit. This approach allows that service resolution of services that compose an execution be aggregated, resulting in the resolution of the execution plan. The same approach is used for *Precision* parameter.

4.2.2 Normalisation of quality parameters

Once the values of all global (or aggregated) quality parameters were calculated, and considering that different quality parameters have different units and ranges, it is necessary to *normalise* these attributes into the same range to allow a unified and uniform measurement of the quality of the execution plans. Some quality parameters could be *positive*, i.e. a parameter in which the quality is better if the value is greater (e.g. *Correctness* parameter). Other parameters are *negative*, i.e. the quality is better if the quality value is smaller (e.g. the *Response time* parameter).[12] This normalisation process of quality parameters is used by several authors [2,39,40].

Equations (1) and (2) present the formulae used to normalise positive and negative parameters, respectively. In these equations, q_{Ni} is the normalised value of the parameter *i*; q_i is the global value of this parameter for the current execution plan; q_{max} and q_{min} are, respectively, the biggest and the smallest global values of this parameter for all considered execution plans (if $q_{max} = q_{min}$, then $q_{Ni} = 1$). In this process, each normalised value results in a value between 0 and 1:

$$q_{Ni} = \frac{q_i - q_{min}}{q_{max} - q_{min}}, \quad q_{max} \neq q_{min}, \tag{1}$$

$$q_{Ni} = \frac{q_{max} - q_i}{q_{max} - q_{min}}, \quad q_{max} \neq q_{min}. \tag{2}$$

4.2.3 Utility of execution plans

The normalisation process is followed by a weighting process to consider the user priority and preferences. Thus, users can prioritise some quality parameters and minimise the importance of other quality parameters according to their needs. To do this, the user assigns a *weight* w_i to each quality parameter *i*, which must be between 0 and 1 and the sum of all of these weights must be 1. Equation (3) presents the function to calculate the execution plan quality according to a set of QoS and QoC parameters. In this formula, q_{EP} is the quality of the execution plan, and it is calculated through a weighted sum between

these normalised values of the m parameters and their respective weights:

$$q_{EP} = \sum_{i=1}^{m} \left(q_{Ni} * w_i\right). \tag{3}$$

Figure 16 shows an example of XML configuration file in which the *disponibility* QoS parameter has weight 0.3 (30%); the *correctness* QoC parameter has weight 0.2 (20%); the *precision*, *performance*, *responsing* and *freshness* quality parameters have weights 0.1 (10%) each one and the *resolution* QoC parameter has weight 0.0 (0%), thus being disregarded from the selection process.

At the selection phase, the *utility* of each execution plan is just the quality of the respective execution plan. The execution with biggest quality is selected. In case of the adaptation phase, the *utility* is represented for both: execution plan quality and adaptation cost for the respective execution plan (see Section 4.3).

```
<?xml version="1.0" encoding="UTF-8" ?>
<parameters>
    <parameter>
        <name>disponibility</name>
        <weight>0.3</weight>
    </parameter>
    <parameter>
        <name>correctness</name>
        <weight>0.2</weight>
    </parameter>
    <parameter>
        <name>precision</name>
        <weight>0.1</weight>
    </parameter>
    <parameter>
        <name>performance</name>
        <weight>0.1</weight>
    </parameter>
    <parameter>
        <name>responsing</name>
        <weight>0.1</weight>
    </parameter>
    <parameter>
        <name>freshness</name>
        <weight>0.1</weight>
    </parameter>
    <parameter>
        <name>resolution</name>
        <weight>0.0</weight>
    </parameter>
</parameters>
```

Figure 16. XML configuration file with the weights assigned to the quality parameters.

4.3 Adaptation

Changes occurred at run-time may affect the application execution and performance. When a change happens, some actions may be necessary to ensure that the application continues running. If an adaptation is required, *AdaptUbiFlow* analyses the best strategy for this adaptation with minimal user awareness (thus promoting the autonomy of the application). This section presents the types of changes that can trigger an adaptation process and the techniques used by OpenCOPI to perform this adaptation.

In the OpenCOPI architecture, a service is considered *in fault* when there is a problem that prevents it to meet/reply to a received request. Examples of service failures are as follows: (i) a service provider that loses the connection with OpenCOPI and therefore cannot reply to the service requests; (ii) a service that crashes due to internal errors in the service provider; (iii) a sensor device that has its energy depleted and (iv) a service that comes out of reach of the user due to user mobility.

Services failures are hard to handle, thus requiring the replacement of faulty services by other equivalent services. Besides failures, the services and service providers are subjected to quality degradation. In highly dynamic environments, the service quality can significantly degenerate due to fluctuations in the network bandwidth, extensive use of a service, etc. This is a less severe problem since such degradation does not necessarily mean a fault; it means that some quality parameters (QoS and/or QoC) may have deteriorated at run-time. In addition, the emergence of new available services also needs to be taken into account since these new services can have better quality than services previously selected. Finally, although mobility can make some services unreachable, other services with better quality may become reachable. When the quality degradation of a service is detected, new services emerge or services become reachable due to user mobility, it is necessary to assess the need of reconfiguring the application execution.

4.3.1 Factors that affect adaptation

The adaptation process performed by *AdaptUbiFlow* chooses an execution plan for replacing the current one in case of workflow adaptation. Section 4.2 presented the process for computing the quality of an execution plan. It was mentioned that the execution plan with the best quality is selected to be executed. In the adaptation process, an alternative (not the first choice) execution plan needs to be selected to replace the running execution plan. Thus, the selection of the new execution plan in the adaptation process is based not only on the quality of this execution plan but also on the cost of the adaptation process with the purpose of reducing the adaptation overhead, i.e. in order to improve the efficiency of adaptation. Our adaptation approach tries to reuse the services already executed before the need to adapt. The adaptation cost of the substitute execution plans is variable and consists of the number of services to be performed after adaptation (including services to be executed after the change of the execution plan), services that require rollback and services that require compensatory actions. Thus, we have defined a relationship between the quality of these substitute execution plans and the cost to replace the current execution plan by them.

Quality of execution plan. The quality (QoS/QoC) of an execution plan is used in this replacement process. Although it is a very important factor, it is not enough to ensure an efficient adaptation. In cases where some services have already been executed in the application workflow, a choice of an execution plan that is very similar to the current one may be a better option than an execution plan with the best quality. Therefore, a similar execution plan can reuse the output of the services executed before the beginning of the

adaptation process without violating services dependencies and performing rollbacks, thus decreasing the adaptation cost.

Adaptation cost for each execution plan. The factors which influence the computation of the adaptation cost are as follows: (i) *reuse of service execution* – some services can be used in two or more execution plans, so that it may be advantageous to give priority to execution plans that reuse the result of services already executed by the current execution plan, in case of adaptation; (ii) *service dependencies* – in case of a service fault, all execution plans that use this service and/or its dependent services cannot be chosen to replace the current plan; (iii) *rollback* – in case of replacement of an execution plan, some services that have already been executed may require a rollback to return to the previous execution state (these services are named rollbackable[13] services) and (iv) *compensatory action* – in case of replacement of an execution plan, if a service needs to return to a previous execution state, but this service does not support rollback, then a compensatory action can be provided by the driver that handles the communication between OpenCOPI and the respective service provider since drivers can store the original state (that can be recovered, if necessary) of a service before the service execution.

To calculate the adaptation cost regarding an execution plan, it is necessary to take into account its *absolute adaptation cost*, as defined in Equation (4). This absolute adaptation cost c_{EPabs} is defined as the sum between the number of services to be executed after changing the execution plan (e) (i.e. defined by the reuse of services and the dependencies between services), the number of services that require rollbacks (r) and the number of services that require compensatory actions (c):

$$c_{\text{EPabs}} = e + r + c. \tag{4}$$

In turn, Equation (5) presents the formula used to calculate the adaptation cost c_{EP} to be minimised since a smaller c_{EP} value means a better adaptation quality. In Equation (5), c_{EPmax} is the biggest absolute adaptation cost value among the available execution plans:

$$c_{\text{EP}} = 1 - \frac{c_{\text{EPabs}}}{c_{\text{EPmax}}}. \tag{5}$$

These factors can have different degrees of importance in the adaptation process; such importance depends on the application configuration that is made by the user. Section 4.3.3 presents how the user can influence the adaptation process so that the efficiency (cost) of the adaptation can be trade by the final quality of service delivered to the user.

4.3.2 Adaptation process

The adaptation process is composed of two phases. The first phase consists of selecting a substitute execution plan to replace the current one. The second phase consists of restarting the execution.

Selection of a substitute execution plan. The choice of a new execution plan to replace the current one uses two categories of parameters: the *quality* of the execution plan and the *adaptation cost*. The first one is the *quality value* (as shown in Section 4.2), for which it is desired a *high* value of quality. The second parameter stands for the necessary actions to be performed in case of adaptation, for which it is desired a *low* value of adaptation cost. Users can prioritise these selection parameters according to their needs. To do so, *AdaptUbiFlow* adopts an approach based on assigning weights to each parameter. Thus, users can choose different weights for quality and adaptation cost in the decision about

Table 2. Adaptation profiles based on weights assigned to the quality of execution plans and the adaptation cost.

Adaptation profile	Description	Weights to the parameters	
		Quality of execution plans (w_{SQ})	Adaptation cost (w_{AC})
Full service quality	Full priority to quality of execution plans	1.00	0.00
Service quality	Priority to quality of execution plans, but considering the adaptation cost	0.75	0.25
Balanced	Default configuration, with equal weights to both parameters	0.50	0.50
Low adaptation cost	Priority to adaptation cost, but considering the quality of execution plans	0.25	0.75
Lowest adaptation cost	Full priority to the adaptation cost	0.00	1.00

which execution plan will replace the current one, as explained below. This characteristic enables to prioritise quality over cost and vice versa, thus tailoring the decision process to the user's needs.

Unlike the selection phase, in the adaptation phase, the *utility* of an execution plan is also influenced by the adaptation cost. This utility is defined by a weighted sum between the quality and adaptation cost parameters regarding to the substitute execution plans. There are five possible configurations for the weights to be assigned to the parameters, thus producing what we call an *adaptation profile*, as presented in Table 2.

Equation (6) shows the function to calculate the execution utility (μ) of each execution plan. This function consists of a weighted sum between quality of the execution plan EP q_{EP} (computed in Section 4.2) and its adaptation cost c_{EP} and the respective weights w_{SQ} and w_{AC} assigned to them. Thus, the execution plan with the maximum utility is chosen to replace the current one:

$$\mu = \left(q_{EP} * w_{SQ}\right) + \left(c_{EP} * w_{AC}\right). \tag{6}$$

Execution restarting. Once the process of selecting a new execution plan is finished, the process responsible for changing to the new execution plan is started. This process consists of making all necessary actions (rollbacks, compensatory actions and restart execution) in a seamless way for the user.

5. Evaluation

The evaluation of OpenCOPI aimed to (i) assess the performance of service selection, composition, adaptation and workflow execution; (ii) validate OpenCOPI strategies for service selection and adaptation and (iii) analyse the implementation effort required for an application to consume services provided by two distinct context-provision middleware.

All the experiments reported in this section were carried out on Mac OS X operating system, using a computer with processor Intel® Core™ 2 Duo 2.4 GHz and

4 GB of RAM. The experiments were performed considering the case study presented at Section 2.

5.1 Performance assessment of service selection, composition, adaptation and workflow execution

This section presents the performance assessment of four important OpenCOPI features, namely service selection, composition, adaptation and workflow execution. Especially for the performance evaluation, we have created replicas of some services in order to have a high number of execution plans. Thus, considering these new replicas, the case study has the following: (i) six services that perform the six first activities of the workflow (*SubscribeBurden*, *SearchRegimeOptions*, *SearchPreviousChanges*, *ChooseRegi-me-Option*, *ChangeRegime* and *UpdateChange*, one service for each of these activities); (ii) four services that perform the *SearchTechnicians* activity (four replicas representing different technologies for user location) and (iii) two services that perform the *SendMsgToEmployee* activity. Each service replica has different values for quality metadata. This configuration resulted in up to 32 distinct execution plans (different replicas combination) with different qualities.

5.1.1 Execution time for composition, selection and adaptation features

Considering the aforementioned configuration, we executed the workflow with a different number of possible executions plans: 2, 4, 6, 8, 16 and 32. Figure 17 shows the average execution time (in milliseconds) for composition (Figure 17(a)), selection and adaptation (Figure17(b)), each one calculated with a confidence interval of 95%.

Service composition is the process responsible for discovering services to perform each workflow activity and for creating possible execution plans. For the simplest configuration (two possible execution plans), the semantic composition time was 1014 ms on average to build both execution plans, and for the more complex configuration (32 possible execution plans) the composition time was 1657 ms on average. The time to find the possible execution plans is proportional to the number of available services. This is the most expensive process since it requires analysing the ontologies of each

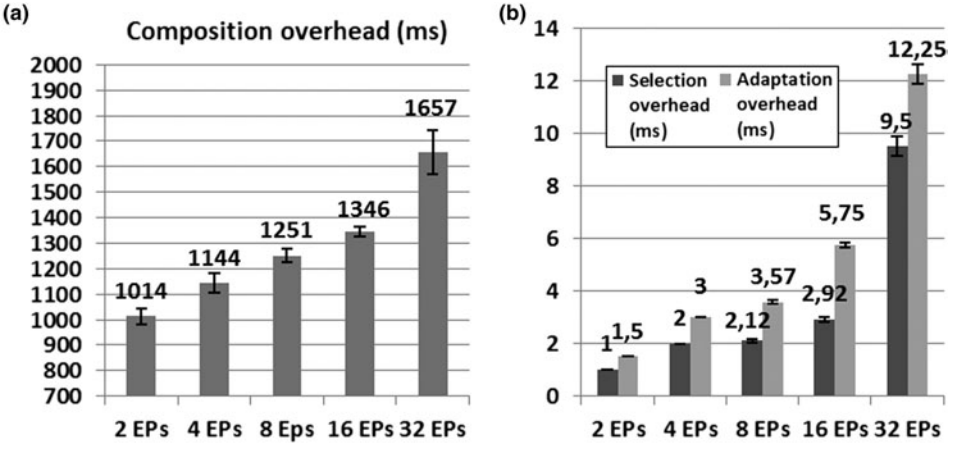

Figure 17. Average execution time regarding composition, selection and adaptation.

available service. This process includes the time spent to parse XML documents (which is mandatory due to the use of Web services and Semantic Web technologies). The goal of this process is to transform an abstract workflow specification into concrete service compositions, enabling late service selection and decoupling the application from used services.

The process of selecting an execution plan is the cheapest considering composition and selection features. For the simplest configuration, the time spent for selection was 1 ms on average, and for the most complex configuration it was about 9 ms on average. The adaptation process (see Section 4) consists of choosing a substitute execution plan and the adaptation preparation (execution restarting). For the simplest configuration, the adaptation time was 1.5 ms on average, and for the most complex configuration it was about 12 ms on average. Nah [27] states that delays in the order of 2 s are acceptable for Web applications. Therefore, we can observe that the time intervals spent for the service composition and for the execution plan selection and adaptation processes are acceptable.

5.1.2 Increase in adaptation execution time

The goal of this evaluation is to analyse the impact in the execution time caused by the adaptation process (presented in Section 4.3.2). Thus, we executed the following four distinct versions of the adaptation algorithm, always considering a workflow with eight possible execution plans: (i) random choice, (ii) considering only the adaptation cost, (iii) considering only the quality of the execution plans and (iv) considering both quality of the execution plans and the adaptation cost.

The average execution times found in this evaluation were, respectively, 3.19, 3.31, 3.42 and 3.57 ms. Considering these results, we can conclude that the adaptation process performed by the OpenCOPI is computationally cheap even considering the quality of the execution plans and the adaptation cost. Moreover, we can observe that the difference among the execution time of all versions of the adaptation algorithm is insignificant. For instance, the difference between the cheapest and most expensive is only 0.38 ms.

5.1.3 Application execution time

Another aspect analysed in the evaluation was the application execution time. Since OpenCOPI represents an additional layer between ubiquitous applications and services provided by many platforms, it is expected that the use of OpenCOPI increases the overall processing time and therefore the response time for users. Thus, it is important to measure the impact of OpenCOPI in the application execution time and if this impact is either significant or negligible from the point of view of the user's experience. For this purpose, two versions of the case study were built. The first version considered the specification and execution of a workflow using OpenCOPI. The second version directly invokes the same services through Java source code in the same sequence followed by the workflow executed by OpenCOPI. The average execution time of the application without OpenCOPI was 1.2 s to call all services involved in the case study. In the application that uses OpenCOPI, the execution time increased by 1.9 s. Again according to Nah [27], the difference (about 0.7 s) between these two versions was not significant for this type of application. Moreover, the second version does not take advantage on the benefits provided by OpenCOPI. For instance, to build the application without OpenCOPI, 139 lines of code (LOC) were necessary only to call all services specified in the workflow. However, the process to build the workflow using OpenCOPI is simpler since it was not

necessary to implement source code but only to build the workflow by defining the applications activities, combining *tasks* and *objects* to satisfy the application goal. Moreover, without OpenCOPI, it is essential to know the services available in the environment and their interfaces. Consequently, the development is harder, reuse is laborious and it is difficult to dynamically select services and also to support adaptation.

5.2 Validation of the service selection and adaptation strategies

In this section, we have firstly validated whether the prioritisation of specific quality parameters really selects the execution plan as expected, i.e. if the selected execution plan is the best available one regarding the specific parameter. Then, we assessed the effect of using different adaptation profiles, changing the weights for the quality of the execution plan and the adaptation cost in order to evaluate if the prioritisation of one of them properly selects the substitute execution plan. To achieve this goal, some services used in the case study presented in Section 2 were forced to fail so that the adaptation process is triggered.

Especially for these validations, we modified the services replicas to generate multiple execution plans (presented in Section 5.1). The services replicated for this validation were four services that perform the *SearchTechnicians* activity (four replicas representing different technologies for user location) and the *SendMsgToEmployee* activity (two replicas). Each service replica has different values for quality metadata. This configuration resulted in eight distinct execution plans (different combinations of replicas) with different qualities. We named each execution plan EP1, EP2, . . . , EP8 to facilitate the explanation of the evaluation.

Considering that there are eight possible executions plans, we executed the selection process 100 times for two approaches (random versus OpenCOPI). The best execution plan was selected in 13.33% of cases for the random approach. Using OpenCOPI, the best execution plan was selected for all (100%) execution rounds. We also evaluated whether the prioritisation of some quality parameter selects the execution as expected. Table 3 presents the utility (or quality) of each execution plan and the selected plan for each different prioritisation tested. For example, when the maximum priority (weight 1) was given to the *Availability* parameter, the execution plan EP4 was selected; when the *Response time* parameter was prioritised, the EP5 execution plan was selected; the equal prioritisation of the *Availability* (weight 0.5) and *Response time* (weight 0.5) parameters resulted in selecting the execution plan EP7 ($q_{EP7} = 0.664$). When the same priority was assigned for all parameters (weight 0.2 for each one), the execution plan EP1 was selected ($q_{EP1} = 0.785$). We found that, for all parameters prioritisation, the selected substitute plan was the expected plan for that configuration.

For the adaptation process, we validated if each possible adaptation profile (weights for the quality of execution plans and adaptation cost) selects the best candidate for the substitute execution plan. Note that distinct adaptation profiles can select distinct substitute execution plans. Considering that the execution plan EP1 was selected to execute at the selection phase by assigning equal weights to the parameters (Table 3), we have forced the failure of a service encompassed in EP1 to trigger the adaptation process. Table 4 presents the selected substitute plan for each different adaptation profile tested. For example, for the *full service quality*, *service quality* and *balanced* adaptation profiles, the execution plan EP4 was selected. In turn, for the *low adaptation cost* and *lowest adaptation cost* adaptation profiles, the execution plan EP2 was selected. We found that, for all assessed adaptation profiles, the selected substitute plan was the expected plan for that configuration.

Table 3. Quality of the execution plans for each configuration.

Execution plans	Equal priority ($w = 0.2$)	Availability ($w = 1.0$)	Response time ($w = 1.0$)	Availability/response time ($w = 0.5$)
EP1	**0.785**	0.528	0.400	0.464
EP2	0.772	0.660	0.000	0.530
EP3	0.766	0.830	0.400	0.415
EP4	0.760	**1.000**	0.000	0.500
EP5	0.515	0.528	0.800	**0.664**
EP6	0.496	0.830	0.400	0.615
EP7	0.480	0.000	**1.000**	0.500
EP8	0.430	0.150	0.600	0.375

5.3 Implementation effort required to build drivers to different context-provision middleware

OpenCOPI increases the abstraction level of ubiquitous application specification since they can be built from abstract activities. Thus, applications are independent of specific services provided by context-provision middleware. However, without OpenCOPI, applications need to know how to directly consume services provided by these context-provision middleware. As mentioned in Section 3.1, part of this abstraction is provided by OpenCOPI *drivers*, which are components responsible for abstracting away context-provision middleware APIs, thus enabling OpenCOPI to integrate the underlying context-provision middleware. This section assesses the complexity of implementing drivers through the number of LOC necessary to build them. We have implemented drivers for two distinct context-provision middleware, namely *CT* [11] and *JCAF* (*Java context awareness framework*) [3].

CT is a context-provision middleware based on key/value tuples and RMI communication. *JCAF* context-provision middleware also uses RMI technology for communication, but it adopts an object-oriented context model. The JCAF's context model is more intuitive than the CT's model because in JCAF, for instance, the *Employee* class represents a context entity while in CT the context attributes are represented only through primitive data types, thus making impossible the direct relationship between the context attributes.

The development of the CT driver has required 110 LOC, while the JCAF driver has required 82 LOC. Part of this difference is due to the object-oriented context model adopted by JCAF. Although these drivers seem to be simple to implement, this task is hard and requires a large effort from the application developer since he/she needs to know implementation details of each context-provision middleware. If the application developer uses OpenCOPI, all these details of context-provision middleware implementations are abstracted away.

Table 4. Choice of an execution plan based on different configurations of weights in the adaptation phase.

Adaptation profile	Selected execution plan
Full service quality, service quality, balanced	EP4
Low adaptation cost, lowest adaptation cost	EP2

6. Related work

This section compares OpenCOPI with two different middleware categories. The first category consists in context-provision middleware platforms. The focus of OpenCOPI is not to compete with existing context-provision middleware but to work cooperatively with them to provide as many services as possible for the applications. However, OpenCOPI and these middleware share some requirements, e.g. service composition, selection and adaptation, thus it is important to discuss our work in such context. The second category consists of integration platforms for context-provision middleware. There are few platforms for integration of context-provision middleware and to the best of our knowledge, none of them covers the wide range of features provided by OpenCOPI. Therefore, the following comparisons justify and motivate the usage of OpenCOPI.

6.1 Comparison with context-provision middleware

Several workflow-based middleware platforms have emerged over the last years to assist the development of ubiquitous applications. However, most of them do not meet the wide range of requirements demanded by highly dynamic and heterogeneous ubiquitous environments. In general, these proposals do not allow dynamic service composition and adaptation based in quality metadata. Even few middleware platforms that enable adaptation do not consider factors that allow an efficient adaptation, such as dependence between services, rollbacks, service re-execution and so on.

Ranganathan and McFaddin [29] present a workflow approach for modelling and managing the user interaction with the ubiquitous environment. In such approach, users can determine their overall goal and preferences and the system generates a customised workflow describing how various services should interact with one another. Montagut and Molva [25] present an architecture that supports the distributed execution of workflows in pervasive environments based on decentralised control. Both proposals lack mechanisms to allow dynamic service composition and workflow adaptation.

Tang et al. [32] present a context-adaptive workflow management algorithm, which can dynamically adjust workflow execution policies in terms of current context information and supports service selection based in bandwidth and user location. In such work, context information is limited to bandwidth and location, but user configuration and workflow adaptation are not supported.

The mechanisms presented in [26] support workflow adaptation but just in case of service failure. The adaptation process is modelled before workflow execution and it does not consider QoS to service selection and workflow adaptation.

Marconi et al. [22] presents an interesting set of tools and principles to support context-aware run-time deviations and changes in the workflow execution, thus allowing workflow adaptation in case of service failure but it does not enable the user to configure the adaptation preferences in case of quality degradation of the services. Moreover, unlike OpenCOPI, it does not consider the cost of adaptation to select the new flow in case of adaptation.

Differently from all the previously mentioned proposals, this paper investigates how to automatically manage workflows by selecting the best option of execution plan and automatic adaptation decisions at run-time according to user preferences. In OpenCOPI, users can configure the service selection and adaptation process in an easy way through an XML configuration file.

6.2 Comparison with integration platforms

The platforms presented in this section are focused on context-provision middleware interoperability. In general, they use bridges or drivers to communicate with many middleware, thus enabling applications to consume services provided by different middleware.

AWARENESS [6] is a platform which provides bridges between different context-provision middleware. It enables applications to acquire context information provided by many middleware even communicating only with one middleware. These bridges are responsible for discovering services and mapping communication protocols and context models. The AWARENESS platform does not standardise communication protocols and context models once each bridge is implemented to provide unidirectional interoperability between only a pair of middleware. Thus, a bridge implements context model and communication protocol of both integrated middleware.

Ubicomp Integration Framework [24] is a framework that is aimed to providing interoperability between context-provision middleware through middleware-specific adapters. This platform exposes services provided by many middleware through Web services and converts the context model of each middleware to its own OWL-based context model.

Stavropoulos et al. [31] presents aWESoME, a middleware infrastructure for Ambient Intelligence environments based on Web Services. Although aWESoME does not provide integration of context-provision middleware, it provides an abstraction layer based on drivers for integrating mobile devices infrastructures (e.g. ZigBee). Such devices expose their functionalities and data over WSDL and SOAP to ensure uniform, standardised, remote and platform independent access to them. aWESoME has drivers for ZigBee and Z-Wave platforms but drivers for other technologies can be added whenever it is necessary.

All three aforementioned platforms do not support service composition based on service quality (QoS and QoC) 'and' adaptation. Similarly to OpenCOPI, Ubicomp Integration Framework and aWESoME adopt standardised communication protocols and propose an abstraction layer between context-provision middleware and applications. However, AWARENESS uses an approach in which a context-provision middleware interacts directly with others, thus requiring a bridge between each pair of middleware platform. This is a limitation because in order to integrate a large amount of context-provision middleware a large number of bridges are necessary. In addition, Ubicomp Integration Framework, aWESoME and AWARENESS do not use the Semantic Web technology, thus constraining the degree of automation in service discovery and composition. Moreover, such platforms do not allow that applications are built from abstract and high-level activities. Using OpenCOPI, the user is able to build its own application through abstract activities, allowing the users to create their own applications.

ubiSOAP [10] middleware defines a two-layer architecture to provide network-agnostic connectivity and Web service-oriented communication in ubiquitous environments. The aim of ubiSOAP is to explore diverse network technologies in order to create an integrated multi-radio networking environment and select the best network to connect clients to legacy services. Thus, services can be reached by applications independent of the underlying mechanism to provide network connectivity. The ubiSOAP's network selection is based on QoS quantitative and qualitative attributes. As OpenCOPI, ubiSOAP supports legacy Web services, thus transparently bringing the value-added Ubiquitous Computing environments. However, while ubiSOAP middleware

focuses on providing low-level integration based on a multi-network overlay, OpenCOPI abstracts network selections, focusing on higher level issues as service selection, composition and execution adaptation. These distinct focuses can make ubiSOAP and OpenCOPI complementary platforms.

7. Final remarks

Recent technological advances have made the Ubiquitous Computing a reality in our daily lives. However, in order to reach the full potential of this new scenario, it is crucial to reduce the efforts of developing ubiquitous applications. Applications for ubiquitous environments have a set of requirements that pose new important challenges to developers. Two crucial requirements are context-awareness and adaptation. There was a growing emergence of context-provision middleware in the last years, addressing such requirements and handling different types of contextual information. Such middleware platforms often do not interact with each other, bringing the need for an additional layer promoting integration and interoperability among those middleware platforms in order to build complex ubiquitous applications. In this context, we introduced OpenCOPI, a context middleware platform that provides a unified ontology-based context services for the development of ubiquitous applications and integrates services provided by distinct sources. OpenCOPI adopts a SOA-based approach, decoupling applications from the underlying context-provision middleware. Moreover, OpenCOPI is built on semantic Web services technology, providing value-added functionalities of discovering, selecting and composing services that fulfil the application needs. It also relies on the concept of semantic workflow to provide the coordination and autonomy required by ubiquitous applications. This paper described OpenCOPI and its evaluation under different aspects. According to the evaluation, OpenCOPI has the potential of fulfilling the requirements of ubiquitous applications and leverages the potential benefits achiever from the seamless integration of computational and communication resources in our daily lives. OpenCOPI can be downloaded at the following URL: http://consiste.dimap.ufrn.br/projects/opencopi/.

OpenCOPI has some limitations: (i) in its current version, the integration of new context-provision middleware to OpenCOPI requires considerable programmer's effort to manually develop the required driver and (ii) since the DeviceController and the DeviceManager components are not fully implemented yet, OpenCOPI capability of managing several user's devices is currently limited.

The main future directions of this work are directly related to the above-mentioned limitations: (i) we intend to implement a strategy to develop drivers in an semi-automatic way, minimising the programmer effort and (ii) we will finalise the implementation of the DeviceController and DevicesManager components since those two components will allow the management of multiple devices from a given user, thus enabling to consume their services and send the results of application's operation for distinct devices. Moreover, we plan to implement a Web-based user interface to make easier the workflow creation and execution in large-scale environments.

Acknowledgement

This work was partially supported by brazilian funds through ANP – Agência Nacional do Petróleo, Gás Natural e Biocombustíveis (PRH-22 Program), CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico (grants 310661/2012-9, 485935/2011-2, 311363/2011-3 and 470586/2011-7) and FAPERJ – Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro. This work also was partially supported by portuguese funds through FCT

– Fundação para a Ciência e a Tecnologia, underprojects PTDC/EIA-EIA/HYPERLINK "tel:113993%2F2009"113993/2009 and PEst-OE/EEI/LA0021/2011.

Notes

1. Email: fred.lopes@gmail.com
2. Email: fdelicato@gmail.com
3. Email: thaisbatista@gmail.com
4. Email: evertonranielly@gmail.com
5. Email: thiago.inf@gmail.com
6. Email: paulo.f.pires@gmail.com
7. Email: paulo.ferreira@inesc-id.pt
8. Email: jrsmjr@gmail.com
9. QoC stands for any information that describes the quality of information that is used as context, such as precision, probability of correctness, resolution, up to dateness [9].
10. In some cases, an activity may require a collection of services to satisfy the activity's functional requirements.
11. Similar services mean two or more services that meet the same functional requirements but have distinct quality.
12. It is important to note that we *are not* saying that the values are negative.
13. This information about whether a service is rollbackable or not is obtained from the semantic description of the service when it is registered in OpenCOPI. Service providers are responsible for supporting rollbacks; this is not a responsibility of OpenCOPI.

References

[1] A. Abbasi and Z. Shaikh, *A conceptual framework for smart workflow management*, International Conference on Information Management and Engineering (ICIME'09), Kuala Lumpur, Malaysia, 2009, pp. 574–578.
[2] M. Alrifai, D. Skoutas, and T. Risse, *Selecting skyline services for QoS-based Web service composition*, 19th International Conference on World Wide Web (WWW'10), Hong Kong, 2010, pp. 11–20.
[3] J. Bardram, *The Java context awareness framework (JCAF) – A service infrastructure and programming framework for context-aware applications*, *Pervasive Computing,* in *Lecture Notes in Computer Science*, Vol. 3468, H.W. Gellersen, R. Want, and A. Schmidt, eds., Springer, Berlin/Heidelberg, Germany, 2005, pp. 11–20.
[4] C. Batista, G. Alves, E. Cavalcante, F. Lopes, T. Batista, F.C. Delicato, and P.F. Pires, *A metadata monitoring system for Ubiquitous Computing*, 6th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2012), Barcelona, Spain, 2012, pp. 60–66.
[5] T. Berners-Lee, J. Hendler, and O. Lassila, *The semantic web*, Sci. Am. Mag. 284(5) (2001), pp. 34–43.
[6] M. Blackstock, R. Lea, and C. Kraisic, *Managing an integrated Ubicomp environment using ontologies and reasoning*, 5th IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07), White Plains, NY, USA, 2007, pp. 45–52.
[7] A. Bottaro and A. Gérodolle, *Home SOA – Facing protocol heterogeneity in pervasive applications*, 5th International Conference on Pervasive Services (ICPS'08), Sorrento, Italy, 2008, pp. 73–80.
[8] M. Brito, L. Vale, P. Carvalho, and J. Henriques, *A sensor middleware for integration of heterogeneous medical devices*, 2010 Annual International Conference of the IEEE Engineering in Medicine and Biology Society (IEMBS 2010), Buenos Aires, Argentina, 2010, pp. 5189–5192.
[9] T. Buchholz, A. Küpper, and M. Schiffers, *Quality of Context: What it is and why we need it*, 10th Workshop of the HP OpenView University Association, Geneva, Switzerland, 2003.
[10] M. Caporuscio, P.G. Raverdy, and V. Issarny, *ubiSOAP: A service-oriented middleware for ubiquitous networking*, IEEE Trans. Serv. Comput. 5(1) (2012), pp. 86–98.
[11] G. Dey, Abowd, and D. Salber, *A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications*, Hum. Comput. Interact. 16(2) (2001), pp. 97–166.
[12] T. Erl, *SOA Principles of Service Design*, Prentice Hall, Upper Saddle River, NJ, 2007.

[13] T. Gruber, *A translation approach to portable ontology specifications*, J. Knowl. Acquis. 5(2) (1993), pp. 199–220.
[14] N. Guarino, *Formal ontology and information systems*, International Conference on Formal Ontology and Information Systems (FOIS'98), Trento, Italy, 1998, pp. 3–15.
[15] T. Guo, H.K. Pung, and D.Q. Zhang, *A service-oriented middleware for building context-aware services*, J. Netw. Comput. Appl. 28(1) (2005), pp. 1–18.
[16] T. Hasiotis, G. Alyfants, V. Tsetsos, O. Sekkas, and S. Hadjiefthymiades, *Sensation: A middleware integration platform for pervasive applications in wireless sensor networks*, 2nd European Workshop on Wireless Sensor Networks (EWSN 2005), Istanbul, Turkey, 2005, pp. 366–377.
[17] V. Issarny, D. Sacchetti, F. Tartanoglu, F. Sailhan, R. Chibout, N. Levy, and A. Talamona, *Developing ambient intelligence systems: A solution based on Web services*, Automat. Softw. Eng. 12 (2005), p. 1.
[18] G. Judd and P. Steenkiste, *Providing contextual information to pervasive computing applications*, First IEEE International Conference on Pervasive Computing and Communications (PERCOM'03), Fort Worth, TX, USA, 2003, pp. 133–142.
[19] A. Keller and H. Ludwig, *The WSLA framework: Specifying and monitoring service-level agreements for web services*, J. Netw. Syst. Manag. 11(1) (2003), pp. 57–81.
[20] J. Li, Y. Bu, S. Chen, X. Tao, and J. Lu, *FollowMe: On research of pluggable infrastructure for context-awareness*, 20th International Conference on Advanced Information Networking and Applications (AINA 2006), Vienna, Austria, 2006, pp. 199–204.
[21] F. Lopes, T. Pereira, E. Cavalcante, T. Batista, F. Delicato, P. Pires, and P. Ferreira, *AdaptUbiFlow: Selection and adaptation in workflows for Ubiquitous Computing*, 9th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2011), Melbourne, Australia, 2011, pp. 63–71.
[22] A. Marconi, M. Pistore, A. Sirbu, H. Eberle, F. Leymann, and T. Unger, *Enabling adaptation of pervasive flows: Built-in contextual adaptation*, 7th International Joint Conference on Service-Oriented Computing (ICSOC-ServiceWave'09), Stockholm, Sweden, in *Lecture Notes in Computer Science*, Vol. 5900, L. Baresi, C.H. Chi, and J. Suzuki, eds., Springer, Berlin/Heidelberg, Germany, 2009, pp. 445–454.
[23] D. Martin, M. Paolucci, S. Mcilraith, M. Burstein, D. Mcdermott, D. Mcguinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, *Bringing semantics to Web services: The OWL-S approach*, First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), in *Lecture Notes in Computer Science*, Vol. 3387, J. Cardoso and A. Sheth, eds., Springer, Berlin/Heidelberg, Germany, 2005, pp. 26–42.
[24] R. Mendes, P. Pires, F. Delicato, T. Batista, J. Taheri, and A. Zomaya, *Using semantic Web to build and execute ad-hoc process*, 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2011), Sharm El-Sheikh, Egypt, 2011, pp. 233–240.
[25] F. Montagut and R. Molva, *Enabling pervasive execution of workflows*, 2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2005), San Jose, CA, USA, 2005.
[26] L. Mostarda, S. Marinovic, and N. Dulay, *Distributed orchestration of pervasive services*, 24th IEEE International Conference on Advanced Information Networking and Applications (AINA'10), Perth, Australia, 2010, pp. 166–173.
[27] F. Nah, *A study on tolerable waiting time: How long are Web users willing to wait?* Behav. Inf. Technol. 23(3) (2004), pp. 153–163.
[28] J. Nakazawa, H. Tokuda, W. Edwards, and U. Ramachandran, *A bridging framework for universal interoperability in pervasive systems*, 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06), Lisbon, Portugal, 2006, pp. 3–12.
[29] A. Ranganathan and S. McFaddin, *Using workflows to coordinate Web services in Pervasive Computing environments*, IEEE International Conference on Web Services (ICWS'04), San Diego, CA, USA, 2004, pp. 288–295.
[30] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz, *MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments*, Software Engineering for Self-Adaptive Systems, in *Lecture Notes in Computer Science*, Vol. 5525, B. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, eds., Springer, Berlin/Heidelberg, Germany, 2009, pp. 164–182.

[31] T.G. Stavropoulos, K. Gottis, D. Vrakas, and I. Vlahavas, *aWESoME: A web service middleware for ambient intelligence*, Expert Syst. Appl. 40(11) (2013), pp. 4380–4392.
[32] F. Tang, M. Guo, M. Dong, M. Li, and H. Guan, *Towards context-aware workflow management for Ubiquitous Computing*, 2008 International Conference on Embedded Software and Systems (ICESS'08), Chengdu, China, 2008, pp. 221–228.
[33] H.L. Truong, L. Juszczyk, A. Manzoor, and S. Dudstar, *ESCAPE: An adaptive framework for managing and providing context information in emergency situations*, 2nd European Conference on Smart Sensing and Context (EuroSSC'07), Kendal, UK, in *Lecture Notes in Computer Science*, Vol. 4793, G. Kortuem, J. Finney, R. Lea, and V. Sundramoorthy, eds., Springer, Berlin/Heidelberg, Germany, 2007, pp. 207–222.
[34] H.L. Truong, R. Samborski, and T. Fahringer, *Towards a framework for monitoring and analyzing QoS metrics of grid services*, Second IEEE International Conference on e-Science and Grid Technologies (e-Science 2006), Amsterdam, The Netherlands, 2006.
[35] X. Wang, D. Zhang, T. Gu, and H. Pung, *Ontology based context modeling and reasoning using OWL*, Second IEEE Annual Conference on Pervasive Computing and Communication Workshops (PERCOMW'04), Orlando, FL, USA, 2004, pp. 18–22.
[36] World Wide Web Consortium (W3C). *Web Ontology Language (OWL)*, (2003), Available at http://www.w3.org/2004/OWL/
[37] World Wide Web Consortium (W3C). *OWL-S: Semantic markup for web services*, (2004), Available at http://www.w3.org/Submission/OWL-S/.
[38] H.I. Yang, R. Bose, A. Helal, J. Xia, and C. Chang, *Fault-resilient pervasive service composition*, in *Advanced Intelligent Environments*, A. Kameas, V. Callagan, H. Hagras, M. Weber, and W. Minker, eds., Springer, New York, NY, 2009, pp. 195–223.
[39] Z. Yanwei, N. Hong, D. Haojiang, and L. Lei, *A dynamic Web services selection based on decomposition of global QoS constraints*, 2010 IEEE Youth Conference on Information Computing and Telecommunications (YC-ICT 2010), Beijing, China, 2010, pp. 77–80.
[40] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, *QoS-aware middleware for Web services composition*, IEEE Trans. Softw. Eng. 30(5) (2004), pp. 311–327.