HAL
open science

WComp, Middleware for Ubiquitous Computing and System Focused Adaptation

Nicolas Ferry, Vincent Hourdin, Stéphane Lavirotte, Gaëtan Rey, Michel Riveill, Jean-Yves Tigli

To cite this version:

Nicolas Ferry, Vincent Hourdin, Stéphane Lavirotte, Gaëtan Rey, Michel Riveill, et al.. WComp, Middleware for Ubiquitous Computing and System Focused Adaptation. Computer Science and Ambient Intelligence, 2012, 978-1-84821-437-8. hal-01330474

HAL Id: hal-01330474

https://hal.science/hal-01330474

Submitted on 23 Jun 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

0

WComp, a Middleware for Ubiquitous Computing

Nicolas Ferry*, Vincent Hourdin, Stéphane Lavirotte, Gaëtan Rey, Michel Riveill and Jean-Yves Tigli
I3S/Université de Nice-Sophia-Antipolis
France

1. Introduction

Ubiquitous computing relies on computers present everywhere, at any times and in any things. Indeed with recent years advance in mobile communication technologies and the miniaturization of computer hardware, processing units are becoming invisible and a part of the environment. Middlewares for ubiquitous computing have to manage three main features specific to their environment: devices' *mobility*, devices' *heterogeneity* and environment's *dynamicity*. The devices' *mobility*, due to motion of users and their associated devices, forbids to assume that entities are known and will always be available. The second concept, entity's *heterogeneity*, outlines the diversity between devices's capabilities and functionalities provided by new smart objects. Finally, the environment high *dynamicity* illustrates the ubiquitous world entropy with the appearance and disappearance of devices. Devices used to create applications are thus unknown before discovering them. Then, ubiquitous computing must deal with such a dynamic software environment (called software infrastructure afterwards). As a result, future ubiquitous computing architectures must take into account those three constraints to solve ubiquitous computing challenges.
Our model of middleware WComp is based on three parts: a software infrastructure, a service composition architecture, and a compositional adaptation mechanism.
To manage the dynamicity and heterogeneity of entities in the software infrastructure, we highlight the use of Web Service Oriented Architecture for Device (WSOAD). This will be discussed in section 2. Ubiquitous applications are then based on a set of Web services for devices that must interact with each other. Consumers can not edit these services. Therefore, in order to add new functionalities to the system, an application has to be a composition of services for devices. Such an application, and thus such a composition, must be modifiable at runtime.
The second part of the WComp middleware enables us to make such applications by *dynamically* composing services from the software infrastructure. To allow reusability of newly created functionnalities, and for scalability purposes, such composition can be encapsulated as a composite service. This part of the system will be presented in Section 3. Moreover, the infrastructure of ubiquitous computing applications evolves dynamically led by appearances and disappearances of objects or devices. The variation of this infrastructure is dynamic due to arbitrary node *mobility*, failures or energy constraints. The service composition must be as relevant as possible according to the underlying software infrastructure. Managing these

*also supported by CSTB

compositions musn't lead to an administrative overhead but must be self-adapted at runtime, in a transparent way.
The third part of the WComp middleware proposes to address this issue using Aspect of Assembly (AA). AA is an approach for composition adaptation particularly well-suited to tune a set of composite services in reaction to a particular variation of the infrastructure or changing preferences of the users. AA will be introduced in Section 4.. Finally in the last section of the chapter we will present some experimental results about performances of our composition and adaptation mechanisms.

2. Web services for device infrastructure

According to (MacKenzie et al. (2006)) "Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.". They were originally used to design distributed applications and to deploy them easily. SOA allows to build dynamic applications using a set of basic entities called services. A service is defined as the way to connect consumers and providers capabilities.
SOA are a way to manage a set of independent software applications and to handle the infrastructure of a set of interrelated services. Each of them being accessible through standardized interfaces and protocols. They define an interaction between software entities as an exchange of messages between service consumers (clients) and service providers (Papazoglou (2003)).
A third entity exists: the registry. Thanks to the registry the system is able to discover available services providers and consumers (FIG. 2) (Champion et al. (2002)). Generally, it stores the service description, and not only a reference to the provider.

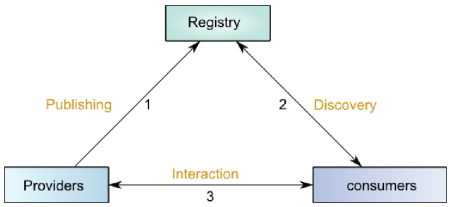

Fig. 1. Service oriented architecture

Here are services main features (Breivold & Larsson (2007); MacKenzie et al. (2006)) and commonalities with ubiquitous computing devices that lead us to the use of SOA as software infrastructure for ubiquitous applications:

- *Encapsulation*: any functionality can be encapsulated in a service and thus be part of available services creating applications. All entities of the system represented by a similar standard of service will be accessible through the same infrastructure. SOA thus provides a logical consistency. Devices also provide a set of functionalities, it is possible to encapsulate them in services at the functional level.
- *Loose coupling and autonomy*: services have no direct dependencies between them. The absence of a service does not prevent others from fulfilling their functions. Services are independent and control their internal logic without any external intervention. This is also a property of devices, for instance a light does not require shutters to operate.

- *Contracts and abstraction*: services describe the functionalities they offer using *contracts*. Services must comply with these contracts. It is the only way for consumers to obtain information about their functionality, as well as their non-functional concerns and meta-data. Services are black boxes, only their contract is known. The way they are implemented is unknown and cannot be edited. This is also true for devices. Reuse of services is facilitated by the fact that service providers are not designed for a specific consumer. At launch time, service providers publish their contracts into the registry to become an available entity of the system.
- *Dynamic discovery*: services are discovered according to some criteria and can be replaced during runtime. Service consumers send a request to the registry to find providers in line with their criteria. They get the provider's contract, and a reference to contact it, generally an URL.
- *Composability*: services can be coordinated and composed to create new composite applications or services. Such a composition is not involved in services, but is organized by an external entity. The content of the composite service may be changed at runtime and the black box abstraction is not fulfilled anymore. Composite services can be seen as gray boxes.

All these properties, common to services and devices, and their adoption in other works (Bottaro et al. (2007); Chakraborty et al. (2005); Vallée et al. (2005)), lead us to consider the use of SOA as infrastructure for ubiquitous computing. In such a case, devices and information systems are represented by services. Applications are compositions of these services. They are continuously observing changes of the infrastructure (appearance or disappearance of services) and react to them if needed FIG. 2.

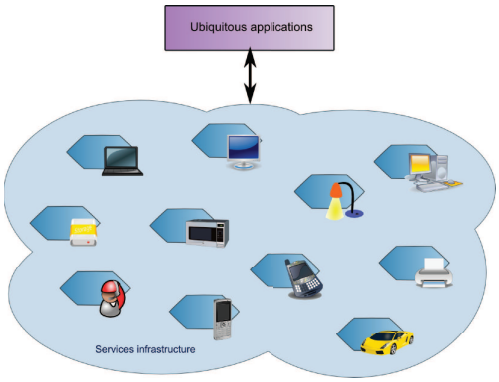

Fig. 2. Using SOA to represent the software infrastructure in ubiquitous computing

To validate the use of services as an infrastructure for ubiquitous computing, we will now investigate if they allow to consider the properties described in the introduction: heterogeneity, reactivity and frequent disconnections due to mobility. Since their creation, SOA have evolved in several directions: Web, informations systems, mobile and ubiquitous computing. Despite these different directions, a new type of services oriented architecture, including all these evolutions is born: Web Services Oriented Architecture for Devices (WSOAD).

2.1 Interoperability

In classical SOA, the choice of a programming language, a data representation, or a communication protocol should be jointly done by designers of consumers and producers jointly in order to be compatible. To partially resolve the problem, distributed applications were usually designed by a same working group. But two kinds of interoperability must be guaranteed. First, the interoperability of platforms, in which all entities that want to provide a service must be based on a virtual machine. This is the case of JINI (Arnold et al. (1999)) which is Java-based. The second type of interoperability is at the level of communication protocols. This is proposed by CORBA (Vinoski & Inc (1997)) or Web services. Using Web services (Champion et al. (2002)), designers of providers do not know how their service will be used. They will probably use some hardware platforms and languages that are different from those of future consumers.

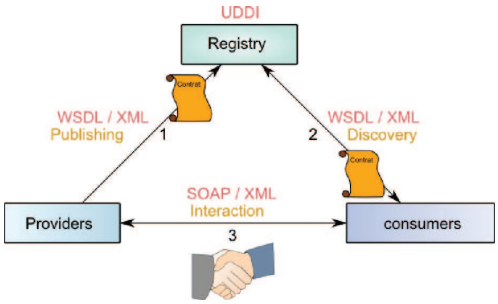

Fig. 3. Web services oriented architecture

From the perspective of the use for devices infrastructures, Web services standards provide a great benefit to handle their heterogeneity. Then, interoperability, as provided by Web services, is an essential feature for ubiquitous computing.

Web technologies have brought to SOA interoperability at several levels. WSOA allows to create applications based on services executing with heterogeneous programming languages, and on various hardware architectures. This evolution of services was required in order to use SOA as infrastructure for ubiquitous computing, because of the heterogeneity of involved devices or applications.

2.2 Evented communications

Mobile computing applications are reactive. Frequent disconnections must be addressed promptly. Indeed, because of battery limitations, programs have to be effective and use eventing rather than *polling*. Moreover, human-machine interactions must be as fast as possible. Processing capabilities of mobile devices are reduced, in terms of processing time, memory and duration of use, because of the battery power. Processes should be as effective and relevant as possible. Using systems based on queue or complex mechanism of publish/subscribe is hardly suitable.

Services for devices (Hourdin et al. (2006)), which are SOAD providers, are lightweight services using evented communications. Evented communications enable applications based on devices an effective use of hardware interruptions. Moreover, they provide loose coupling between services. As an example, considering a smart switch, when activated, a hardware interruption is raised. Thanks to events, a notification will be immediately sent to consumers. The dynamics and efficiency of the application are then maximal. Without the use of events, the switch should keep a state variable up to date, until consumers request its value. Since

these calls are done periodically, there is a risk of missing a change of state if many occur before the end of the period.
Evented communications add to classical services the dynamicity required for interactions between devices that may be involved into some ubiquitous systems.

2.3 Appearance and disappearance

As we have seen previously, in the field of ubiquitous computing, the infrastructure of an application is highly variable due to node mobility. Users' mobility, and then devices' mobility, must lead to frequent disconnections and network changes. The topology of such infrastructures cannot be known *a priori*. Moreover, in a mobile network, we cannot know in advance the address of service registries, or even assume that one exists. To build applications based on these entities, a middleware must know the entities that are in this infrastructure. SOAD proposes to address this issue.
When a service provider enters or leaves a network, it broadcasts a notification across the network. Those appearances or disappearances by announces are asynchronous, and provide the property of dynamicity to the software infrastructure. If a provider is disconnected abruptly, or undergo a programming error, the announcement of their disappearance will not be sent. To overcome these problems, providers use a lease mechanism which involves periodic notifications of their presence.
Thanks to these mechanisms of announcements, network entities are then able to know the entities in their network dynamically and to see them appear and disappear. This mechanism makes sense when coupled with a decentralized dynamic mechanism of discovery.

2.4 Decentralized discovery

Another evolution of SOA provided by SOAD impacts the discovery mechanism. This evolution aims to enable considering network, consumers and providers unknown in advance. The entities that will be present to create an application are not known and must be discovered dynamically.
When a registry is used, consumers send a request based on some criteria (a service type, a name or some more complex expressions (Hourdin et al. (2006))). Some architectures provide a second type of interaction between registries and consumers. Consumers can subscribe to an entry on the registry in order to be notified when a relevant provider (according to the entry) is registred (Bustamante et al. (2002)). When consumers get a reference to the provider, generally an URL, they perform a query to get the service contract. Then, they will be able to interact with the service at a functional level. This contract describes the technical characteristics of the service. Thanks to the interpretation of this contract the service can be discovered.
Then, mechanisms of decentralized search emerged in industry standards (SLP (Guttman (1999)), Jini (Arnold et al. (1999)), Bluetooth SDP [1], Salutation [2], Bonjour [3]) and in many research projects (Chen et al. (2000); Huang et al. (2002); Preuß (2003); Sedov et al. (2003); Ververidis & Polyzos (2008); Zhu et al. (2005)). They allow consumers to find providers without using a centralized registry. In fact, the search mechanism is introduced into providers and consumers so that they communicate directly with each other. Then, the discovery phase uses the mechanisms of appearance / disappearance previously described. This mode is

[1] Bluetooth Service Discovery Protocol, in *Specification of the Bluetooth System. Core*, version 1.1. 2001.
[2] the *Salutation Consortium* no longer exists.
[3] Bonjour is used in Mac OS X to discover printers and to share data.

centered on consumers. As in architectures using services registries, consumers can process a search request, but the request is broadcasted across the network.
In the figure (FIG. 4), the dotted lines represent communications by broadcasting, while those in solid lines are point to point ones.

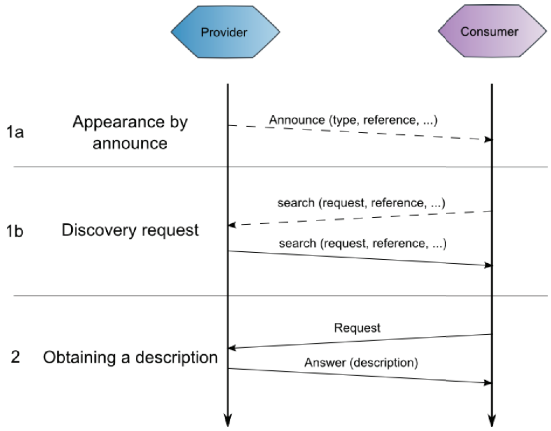

Fig. 4. Dynamic decentralized discovery : discovery and search protocols

However, for some efficiency or safety issues, various protocols (SLP, Jini) use some broadcasting mechanism to discover a service registry. In mobile environments, wireless communications are often expensive in term of energy consumption. A registry then allows consumers to consider a single contact and then send point to point request. They are considered as mandatory to scale to large networks(Ververidis & Polyzos (2008)). Such a registry built its database from announces of appearance and disappearance of service providers.
Discovery is the first step in creating applications from a service infrastructure. In pervasive computing, it is especially the case because the environments in which they are created are not known a priori. *Dynamic discovery coupled with decentralized mechanisms of appearance/disappearance allows to discover services without knowing them* a priori *and without relying on a static entity.*

2.5 Web services for devices oriented architecture (WSOAD)

We have seen the major evolutions of classical SOA:

- *Web Service Oriented Architecture*: the use of web technologies addresses the issue of interoperability between services (2.1).
- *Service Oriented Architecture for Device*: decentralized discovery (2.4) and appearance/disappearance (2.3) mechanisms allow to discover services in mobile environment that are not known *a priori*. Evented communications (2.2) introduce some dynamic interactions between consumers and providers. Moreover, it adds a loose coupling between services.

WSOAD (*Web Service Oriented Architecture for Device*) is born of the combination of these evolutions. They benefit from all the advantages of SOAD, on which is added interoperability from WSOA. Currently there are two implementations of WSOAD: UPnP[4] and DPWS [5].

[4] Universal Plug and Play
[5] Devices Profile for Web Services

WSOAD provides the properties needed for software infrastructures of ubiquitous systems. Since the code of these entities is not editable, to create some new ubiquitous application we must enable interactions between services. We must compose them together. There are two majors types of service composition: orchestration and choreography (Singh & Huhns (2005)). Orchestration is based on a centralized entity. This entity performs all the methods calls on the various services of the application by relaying their messages. On the other hand, choreography consists in a decentralized approach. Indeed, the choreography implies that services are able to organize themselves independently to communicate with each other. It is more complex to implement. In fact it implies that each service knows each other. Moreover, any adaptation mechanisms must then be embedded in each service.
Thus orchestration seems more suited to ubiquitous computing, since the frequent changes in the infrastructure would be complex to manage in each services (Cardoso & Issarny (2007)). The orchestration of services is usually described by defining workflows or abstract processes, such as BPEL (Business Process Execution Language). In the next section, we will present more in detail how to achieve such compositions in the field of ubiquitous computing.

3. Dynamic service composition

Ubiquitous applications are based on a set of Web services for devices that interact with each other. These services are non-editable software entities. Therefore, an application is a composition of services for devices that has to be adapted at runtime because of the dynamicity of the software infrastructure. If services propose to address the issue of interoperability, components offer high dynamicity and reusability. As explained in (Brønsted et al. (2007)): ''The ability to seamless compose services from various devices in a more or less ad-hoc manner is a frequently emphasised feature of ubiquitous computing.''.
They are created thanks to some components factories into containers. Containers provide non-functional properties to components. Components factories define the type of component to be instantiated. They can be instantiated and manipulated easily by a developer. Conversely, services on devices are fixed and sustained. However, services could be deployed on nodes of a controlled network, but they would not provide the same benefits of dynamicity. This is due to interface connections and because services are stateless and cannot be instantiated by the developer. Therefore, we base ourselves on services to communicate with various entities in the environment, devices included, and on components for their adaptation capability.
To address this issue of dynamicity, the proposed architecture is based on two paradigms:

- *Events:* they are taking place in the model as well as at the services level, with Web services for devices for example, than in lightweight assemblies of components. Their advantages are twofold: they promote reactivity of systems, increase the loose coupling between entities, and thus dynamicity of applications.
- *Lightweight components assemblies :* composite Web services are created from a dynamic assembly of black box components, executing in a local container, which doesn't provide mandatory non-functional services. Dynamicity of applications is then provided, and reusability is increased.

LCA (*Lightweight Component Architecture*) (Tigli et al. (2009a)) thus defines a compositional architecture model based on events, to design lightweight components assemblies, and increment the cooperation graph of services and applications. The environment consists of mobile users interacting with the world or other users with weared or mobile devices. We see

them as some services momentarily available in the infrastructure. Components assemblies are a suitable way to orchestrate them.

3.1 Web service for device composition : LCA

The component model LCA is a model derived from Beans (Englander (1997)), adapted to other programming languages, with concepts of input, output ports and properties.
These components are called ''light'' for several reasons. The first is that they execute in the same memory addressing space, and in the same process (Clarke et al. (2001)), so their interactions are reduced to the simplest and the more efficient, the function call. The second reason, which stems from the first, is that they don't embed non-functional code for middleware or other non-mandatory technical service in this local environment. Their memory footprint is then reduced and they can be instantiated and destroyed quickly (Tigli et al. (2009b)). A container is not limited to one type of components but may contain all components of an application. To finish, they don't contain any reference between them at design-time, and respect black box and late-binding concepts. The dynamicity of the model is thus maximal, since they use events to communicate between them, components are fully decoupled, and highly reactive.

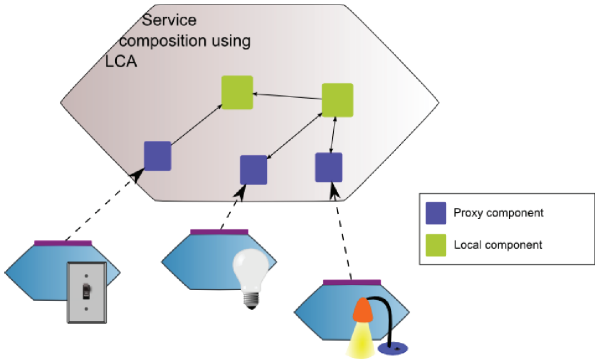

Fig. 5. Composing Web services for devices using lightweight components assemblies based on evented communications

The only non-functional code present in the components is event management and properties accessing. Higher level programming languages define these operations; component code is then a simple object, like JavaBeans or .NET components, not overloaded with code injection for any purpose. The container does not provide technical services easing the programmer work, but consequently allows the creation of components with various requirements, like components needing to access hardware and thus low-level functions. Adding non-functional properties, like security, journaling, or persistence of messages can be made by adding components in the assembly, guaranteeing scalability of the model.
As described in the LCA model (Fig. 3), components have an interface, defined by the component's type. This interface is a set of input ports (methods), and output ports (events), each one being typed by its parameters, and having a unique identifier. Interactions between components are bindings. They link an output port of a component to one or more input port of components. Ports being explicit, no code has to be generated, nor studied by introspection to know what to modify in components to change the target of a binding at run-time. When an event is emitted, the control flow is passed to recipients in an undefined order, but this

can be fixed adding sequence components. When limiting to unique bindings, and using sequence components, control flow managing of the application is fully deterministic. Not having indirections, due to technical services of the framework, gives a full control on control flow, and eases their debugging.

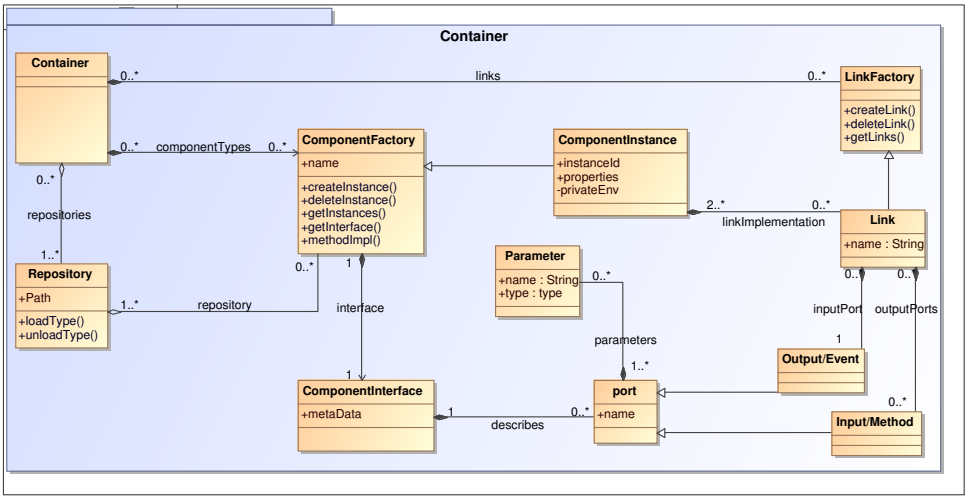

Fig. 6. LCA meta-model: lightweight components

When a service is discovered, a specific component, called *proxy* component, is generated from the description (contract) of the services. Then, the component type will be loaded and instantiated in order to be a part of the component assembly. *Proxy* components will enable communication with services of the environment.

Composing Web services for devices using LCA, the lightweight components model, allows to dynamically create new applications from services that are available in a software infrastructure. However, newly created functionalities are only available locally creating a specific application. It is then necessary to add reusability to the model, and to export the new functionalities created by a component assembly as a new service for device in the infrastructure.

3.2 Distributed composition: SLCA

Multi-paradigms systems have emerged, like new SOA 2.0, which use services and events, or SCA (Service Component Architecture) (Chappell (2007)), dedicated to service composition or iPOJO (Escoffier & Hall (2007)).
SCA defines a component and service architecture, using components to manipulate services orchestrations, and create composite services.
SLCA (*Service Lightweight Component Architecture*) (Hourdin et al. (2008)), is a model of architecture for service composition based on an assembly of lightweight components, inspired by SCA. The SLCA model relies on a software and hardware execution environment evolving dynamically. We define this environment as a set of resources, which are all software/hardware entities that undergo appearing or vanishing from the infrastructure. It's not the application that drives this process. SLCA is based on a Web service for device infrastructure using events, and dynamically discoverable in a distributed way. They

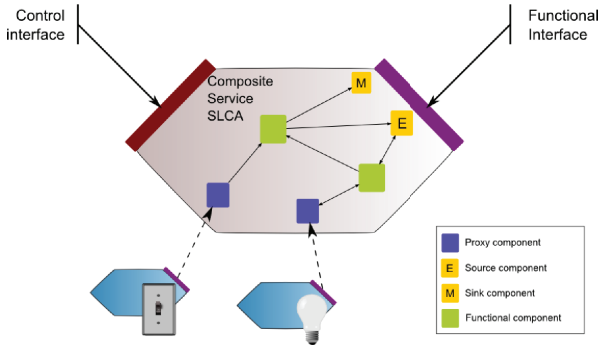

Fig. 7. Composite web service with evented communications

represent devices used in ambient computing applications, as well as composite services created by SLCA.

Applications are designed by service composition mashup, assembling components. A composite service then contains an LCA assembly of components, in a container encapsulated into a service for device. Proxy components to other Web services are thus instantiated in the container of a composite service, and create applications from services available in the environment. Composite service can then create an application communicating with another composite service. It can be seen as a gray box, since it is possible to alter the assembly and access to some functionalities of the components assembly.

A *composite service* provides two service interfaces (FIG. 7). The first one, the dynamic functional interface, allows publishing and accessing functionalities provided by the composite Web service; the second one, the control interface, allows dynamic modifications of the internal component assembly which provides these new functionalities.

The *functional interface* allows to export events and methods of the internal component assembly to the service infrastructure. Then composite services will be part of the graph of the software infrastructure (FIG. 8). The interface is dynamic and exports events and methods of the internal component assembly using *probe components*. Adding or removing a probe component dynamically modifies the functional interface and its description in the corresponding composite service. Adaptation to environment variations can be made by modifying the interface of a composite service, without stopping its execution.

Two types of probe components exist (FIG. 9): *sink*, which adds a method to the composite service interface, and which, in the internal component assembly, has only an output port. The invocation of the method from the service interface thus emits an event in the component assembly. The second type of probe is the *source*, which adds an event to the composite service interface, and has only an input port. The invocation of the method from the component interface thus emits a Web service event.

The *control interface* addresses dynamic modifications of the internal component assembly. It provides methods for adding or removing component instances, types, or bindings, and also to get information about the assembly. Therefore, another client, which can be a composite service using a proxy component for this service, can act on the structure of a composite service. The structural adaptation of composite services and applications is thus possible in the model, by its own entities.

This interface also provides events in order to notify subscribers when structural changes occur into the composite service. Thanks to this mechanism we can adapt composite services

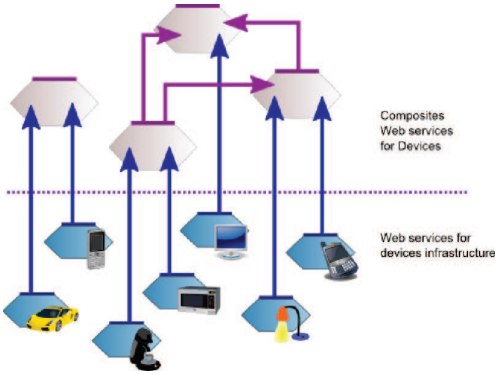

Fig. 8. Graph of Web services with evented communications

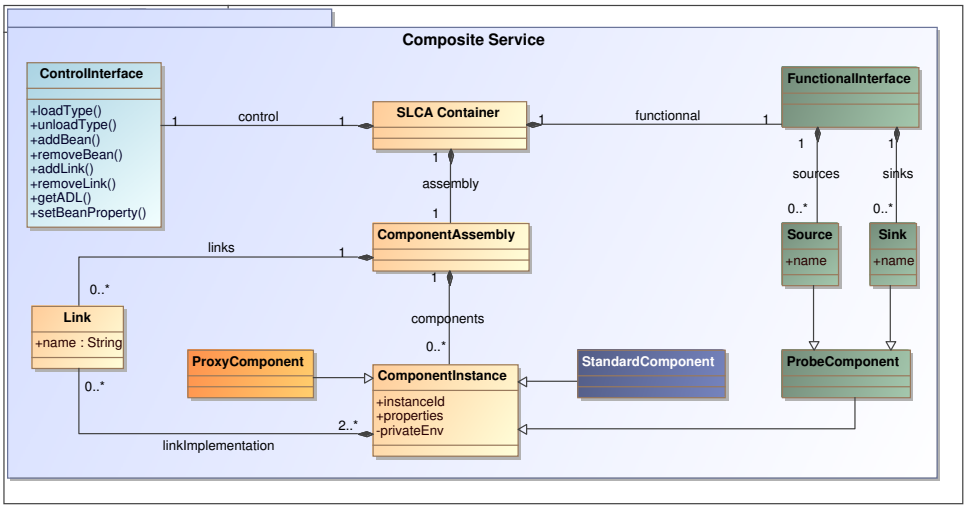

Fig. 9. SLCA Metamodel : composite services interfaces

in a reactive way. Then, we can imagine many sorts of mechanisms of adaptations, and tools to create applications. They are called *designers*. Designers are service consumers for which the control interface of composite services is a required interface. They enable the viewing or modification of a component assembly of composite services using various formalisms and representations. Among these designers, we can mention three that are most often used in the design of ubiquitous computing applications: a designer to visualize an assembly, a designer to dynamically generate proxy components for Web services for devices of the infrastructure, and the Aspect of Assembly designer.

The Aspect of Assembly designer aims to adapt a composite service. It is based on the composite service's control interface to manage a set of adaptation rules. They are triggered when a change occurs in the assembly of the composite service. We will study this mechanism in the next section.

3.3 Synthesis

SLCA composite services are a mean to create fully dynamic applications from a services infrastructure, particularly web services for devices based on communication events.
This dynamicity of the model is available in both (1) setting up applications thanks to functionalities exported in the software infrastructure and (2) the control of composite services. Various tools exist or can be created to consider various concerns of application adaption, such as availability of infrastructure entities, or users needs.

4. Dynamic application adaptation to the infrastructure evolution

We have seen that ubiquitous systems must be able to consider changes occurring in their environment. And since these changes occur continuously, this must be done at runtime. Reflection is a mechanism that offers such a possibility. Reflection is itself based on two mechanisms: (1) intercession which is the ability for a system to modify itself and (2) introspection which is the ability for a system to observe itself.
Thus reflection through the use of object-oriented programming allows to change the code of objects in their interpretation. It introduces some modularity through the representation of data and their relationships (inheritance ...). At runtime, objects are reified into editable meta-objects (a representation of the said object) and meta-object protocols (MOP) (Kiczales et al. (1999)) ensure the consistency between object and meta-object. The possibilities of intercession and introspection offered by this approach are maximal. But MOPs use languages that are too generic for ubiquitous systems.
Aspect-Oriented Programming (AOP) is a way to provide some specific abstractions for some crosscutting concerns. Dynamic aspects allow to adapt an application at runtime while encapsulating the adaptation into aspects(Zambrano et al. (2004)). Thanks to this encapsulation, adaptation mechanisms can be more easily reused. Aspects applied over components can be seen in two ways: (1) aspects can be used to adapt components code, (2) aspect can be used to adapt structurally components assemblies. Because Web services for device from the software infrastructure cannot be modified by the system as a consumer; components have to be seen as blackboxes.

4.1 Aspect oriented programming principles

AOP has been proposed by Kiczales *et al* in 1997 (Kiczales et al. (1997)). It allows to define software abstractions that will be woven (applied) on a base application. Originally, AOP appeared to tackle the following problem: *despite all efforts to achieve it, there is still a strong coupling between functional concerns and crosscutting concerns (security, monitoring . . .)*. The idea of AOP is to separate, into aspects, the representation of crosscutting concerns. Then, the code described in an aspect is injected in the base application thanks to the aspect weaver.
Aspects are composed of *pointcuts* and *advices*. Pointcuts point out "where" to inject the code to adapt the application while advices describe the code to be injected, "what" functionality will be added.
Pointcut genericity allow an aspect to be woven in many parts of the application. AOP allows to minimize code dispersion, grouping it into reusable entities. Joinpoints represent all hooks of applications where advices can be woven. Classically, the aspect language provides mechanisms for adding behavior to pointcuts thanks to operators *after*, *before* and *around*. Thus an advice whose pointcut specifies the *before* keyword will be executed before the execution of the jointpoint matching the pointcut; and inversely with *after* advices. An advice *around* allows to replace or to execute some code before and after the pointcut. The genericity offered

by pointcuts provides an abstraction that reduces the complexity of use of reflection allowing a high reusability of aspects and a good separation of concerns. According to the paradigm on which is based AOP, the nature of joinpoints can change (objects, components, code ...) but AOP still offers a good separation of crosscutting concerns (Charfi & Mezini (2004)).

```
public aspect Aspect_Name {
pointcut Method_Name() : // code

//// Advice
before():Method_Name() {
        // code
} }
```

Fig. 10. Aspect model in AspectJ

The weaver is the mechanism that takes as input a set of aspects and an application in order to produce an augmented application. Initially, weavers were static and were involved at compile-time as in AspectJ (Kiczales et al. (2001)). The code described into advices is woven into the application code to generate a new source file (eg *.java* or bytecode with AspectJ). So that, the separation of concerns introduced by aspects is no longer relevant at runtime. Aspects are not always independent of each other, some interactions may occur between them. In classical approaches, there is no support offered to resolve these interactions, this must be done by developers. Therefore, weavers have evolved in order to adress these issues. As an example EAOP (Event-based AOP) (Douence & Sudholt (2002)) proposes a dynamic weaving triggered according to some events related to the execution of the base application. Moreover, these works propose mechanisms to resolve interactions between aspects. One approach is to explicitly compose advices relying on a same joinpoint. The second approach is to encapsulate aspects into aspects. In the latter, the weaver intends to evaluate events and, according to this evaluation, execute the corresponding advice.
Finally some works were interested in the implementation of aspects on components, such as SAFRAN (David & Ledoux (2006)), CAM/DAOP (Pinto et al. (2005)) or on services such as AO4BPEL (Charfi & Mezini (2004)). These approaches provide the required modularity for adaptation of applications based on components, with respect to the component's blackbox property.
SAFRAN introduces a new type of triggering mechanism of aspects. Aspects can be triggered on the occurance of endogenous events (events from the system) or on the occurance of exogenous events (events from outside); adaptation composition is external. CAM/DAOP proposes some mechanisms for concurrency checking of aspects, their composition is external. In CAM/DAOP aspects are components; to improve aspects reusability their pointcuts are separated from advices. AO4BPEL proposes to use aspects to extend BPEL (Business Process Execution Language) in order to increase its modularity and to enable its dynamic adaptation. Accordingly, this approach does not consider changes occuring in the software infrastructure of the application.
In the following sections we will present an example of persistent structural adaptation mechanism based on aspects triggered on changes occuring in the software infrastructure of the application. Those aspects will be merged in case of aspects interactions. This approach allows, in a modular and declarative way, system self-adaption as well as creating applications from scratch.

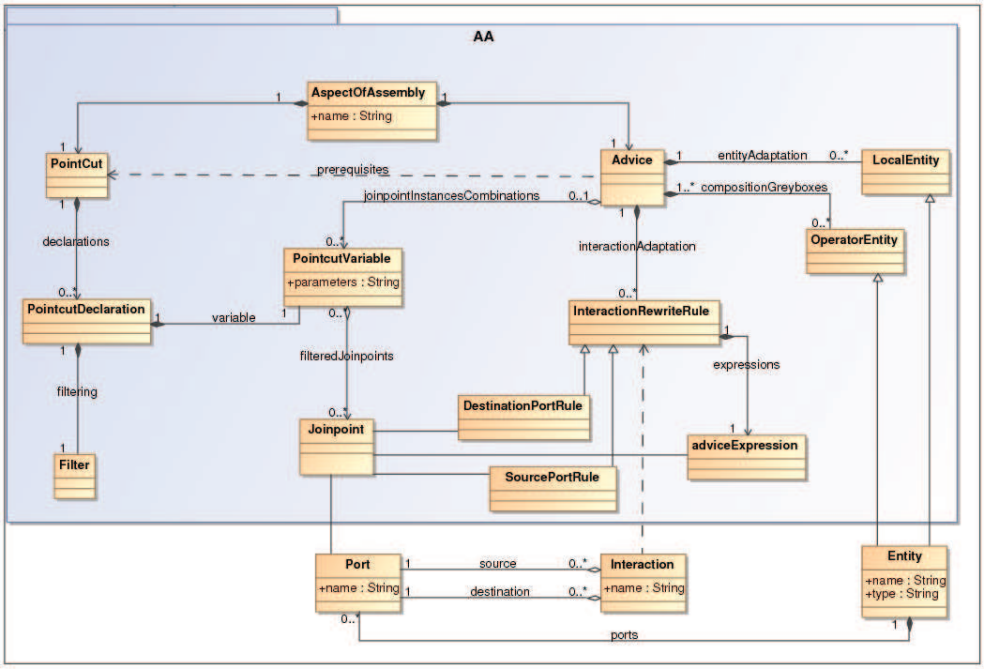

Fig. 11. Aspect of Assembly meta-model

4.2 Crosscutting adaptation

Aspects of Assembly (AA) are an original concept, based on aspect oriented programming. They define some structural reconfigurations of an application that are triggered in response to events from the software infrastructure. Events inform of the appearance / disappearance of devices in the software infrastructure. These rules are woven and composed according to a well-defined logic in case of conflict. They are applied on components assemblies which are not necessarily known *a priori*. Various languages and composition rules can be defined according to the type of applications on which they will be applied. So, Aspect of Assembly can be used to adapt applications based on a component model using evented communications (Cheung-Foo-Wo (2009)). AA are based on a non-invasive model and respect the blackbox property of components.

In the following section we will present the main concepts of Aspects of Assembly as modelized in Figure 11: the joinpoint model(4.2.1), the pointcut model (4.2.2), the advice model (4.2.3), and the weaver (4.2.4).

4.2.1 Joinpoint

Joinpoints are all entities of the assembly that structurally represent the application, on which changes will take place: components and their ports. This allows to consider messages related to ports.

4.2.2 Pointcut

Pointcuts are defined as a set of filters on joinpoints. In fact, they are filters on joinpoint's meta-datas (port name, types ...). Those filters produce some combination of *instanciated joinpoints* (FIG. 12). AA may use various strategies to perform the pointcut matching, the most common one is based on a pattern matching language. Instead of identifying elements of code, since AA describe structural changes, they identify components and ports thanks to their meta-datas like their name. Pointcuts allow to introduce into an application some crosscutting concerns described in their associated advice without knowing the assembly on which they will be applied. At runtime, they interface a real assembly with some abstract advices to produce some real configurations to be incorporated in the application. These configuration are called instances of advice. An advice produces many instances of advice when many combinations of instanciated joinpoint are matched by pointcuts.

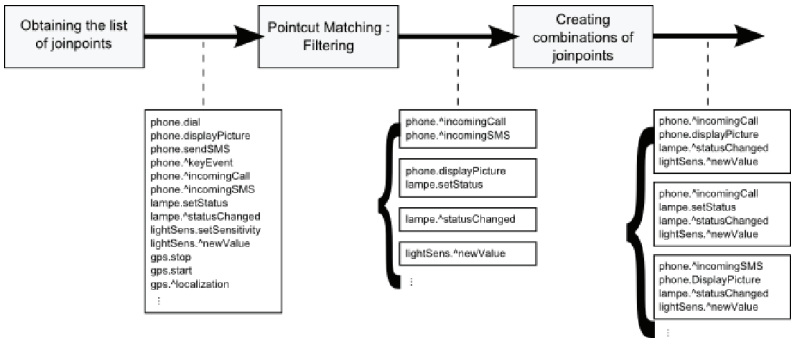

Fig. 12. Example of pointcut matching process

Some others mechanisms for pointcut matching than pattern matching can be studied. For instance, if an AA defines an adaptation related to a switch, rather than processing a pattern matching as `switch*`, a semantic matching would be more appropriated. Indeed, one of the ideas that we stand for in ubiquitous computing is the separation of features offered by devices into more basic features. As an example a phone is designed to make call or send messages, but it could act as a display, a switch, etc... The french national project Continuum[6] is working on such an extension of AA.

Pointcut are evaluated when an AA is selected or when a change occurs in the software infrastructure (appearance/ disappearance of devices) of the application to be adapted (FIG. 18). When at least one joinpoint is statisfaying each pointcut rule, some combinations of joinpoints are generated and the AA is now in the state : *applicable*. Then, it can be woven on the application assembly.

Therefore, pointcuts define the prerequisites to weave AA; even if they are selected by users to adapt an application, AA cannot be woven unless the application assembly is in a state compatible with its poincut (i.e. it contains the entities and ports required for adaptation).

4.2.3 Advice

Because AA modify the structure of components assemblies, adaptations consist in a set of basic structural changes: adding a component or a connection between ports to the

[6] ANR CONTINUUM — ANR-08-VERS-005. http://continuum.unice.fr/

base assembly. The removal of components or interactions is performed too if an AA is withdrawed.
Links additions are usually done by rewriting the existing ones, for adaptations that fit with an existing application, rather than redefining the application. Any change can be seen as a transformation from an assembly to a new assembly (FIG. 16).
A specific language is also used to express advices. In traditional aspects, advices are often code written in the same language as the language of the targetted application. In AA, entities are added and their code is not dependent of the advice's expression; an advice must describe the integration of these new entities in the existing assembly. The specific language of AA's advices is not fixed by the model and designers can define their own language according to the type of adaptation espected.
The various keywords, or operators of a language allow a designer to define how will be composed advices. In the following section we will study the ISL4WComp language, that is used in our works in the field of ubiquitous computing. Thanks to this language, we can describe advices based on events streams.

4.2.3.1 Components used in advices

As explained previously, advices describe changes that must be done in a component assembly. Those changes consist in adding components or bindings between components.
The added entities must be defined and available in the system to manage the component assembly during AA weaving.
Among components that can be instanciated by AA, we can distinguish two types of components: blackbox components and greybox components.
Blackbox components encapsulate functionalities that are only accessible through their ports (Szyperski et al. (1999)). Only the way they interact can be managed. The entities explicitly added to provide a new functionality in the adaptation described by an advice are blackbox components. When several advices involving blackbox components are composed, we talk of external weaving. As a consequence the weaver cannot process some internal merging between those components since only their interface is known. Entities explicitly added by advices are also called local entities. They are represented as `LocalEntity` in the AA meta-model presented in (FIG. 11).

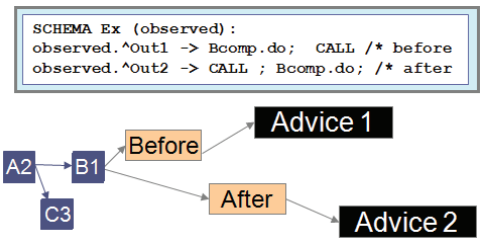

Fig. 13. Blackbox components

As an example, considering and adaptation whose aims which filters one of two messages between two components. This adaptation will add a component to do this. Such a component is a blackbox, only its interface is known by the weaver. It will be added in place of the former interaction. Two new interactions will be created to link the ports of the primary interaction.
Greybox components, conversely, partially explain their semantics, either using an interface description or reflectivity, either it is at least partly known by the weaver. From this

knowledge, it is possible to work on a way to compose them manually or automatically. Such a process is called *composition* or *merging* process (Cheung-Foo-Wo (2009)) of greybox advices. Then, thanks to this mechanism the system is able to manage interactions between various instances of advice from various AA. Greybox components are instanciated by the weaver when some advice language operators are used or when the weaver has to manage some interactions between instances of advice.

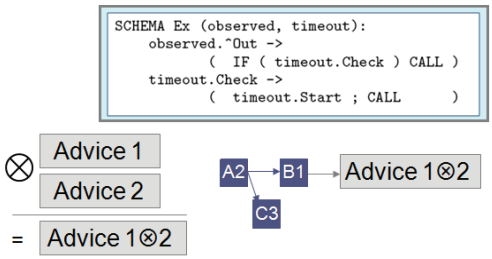

Fig. 14. Greybox components

As an example, an advice rule can specify that when two interactions from a same component are created from various instances of advice, a *sequence* type greybox component will be introduced to establish a priority between those two interactions.

4.2.3.2 ISL4WComp : A language for advices

ISL4WComp is based on the ISL Interaction Specification Language that describes interactions patterns between independent objects (Berger (2001)). ISL4WComp adapts these specifications to consider interactions based on messages or events between components. This language is composable: multiple instances of advice can be composed and merged into a single assembly combining their respective behavior. The merging mechanism, embedded into the weaver, ensures the property of symmetry (associativity, commutativity and idempotency) (Cheung-Foo-Wo, Blay-Fornarino, Tigli, Lavirotte & Riveill (2006)) in the weaving operation of various AA. This means that, the order in which aspects are woven is not important. It implies that the result of a weaving cycle will not be the base assembly given as input to the next cycle (Ferry et al. (2009)). Hence, there is no history of AA appliance. This property is ensured for AA's weaving operation because the operators of the language are symmetrical. This means that the order in which rules are merged is not important.
Advices written using ISL4WComp are based on three types of rules: (1) the addition of blackbox components, (2) rewriting links between components of the assembly and (3) the creation of new links. Rewriting involves components ports, it consists in : forwarding an input port or redirecting a message (output port). These rules are identified thanks to two key words, ':' for blackbox components instanciation and '→' for rewriting and creating links.
An advice describes a set of adaptation rules to be applied on variable components defined in pointcut. Some specific language operators as `call` and `delegate` allow to control how the composition of instances of advice will be done. These keywords, associated to sequence and parallelism operators are similar to classical AOP keywords : *before*, *around* and *after*.

4.2.3.3 A sample of ISL4WComp based advice

To illustrate the concepts previously presented, we will now study an example of advice based on ISL4WComp. First we define an independent adaptation schema for a domotic application. It aims to link a switch to any kind of light in order to control the light using the switch.

	Keywords / Operators	Description
port types	*comp.port*	'.' is to separate the name of an instance of component from the name of a port. It describes a provided port.
	comp.^ port	'^ ' at the beginning of a port name describes a required port.
Rules for structural adaptations	*comp : type*	To create a blackbox component
	comp : type (prop = val, ...)	To create a blackbox component and to initialize properties
	provided_port → (required_port)	To create a link between two ports. The keyword → separate the right part of the rule from its left part
	provided_port → (provided_port)	To rewrite an existing link by changing the destination port
Operators (symmetry property, conflicts resolution)	... ; ...	Describe the sequence
	... || ...	To describe that there is no order (parallelism)
	`if (condition) {...} else {...}`	`condition` is evaluated by a blackbox component
	`nop`	Nothing to do
	`call`	Allow to reuse the left part of a rule in a rewriting rule
	`delegate`	Allow to specify that an interaction is unique in case of conflict

Table 1. ISL4WComp operators and keywords

Both light and switch proxy components are generated into the components assembly. The advice presented below proposes to adapt this behavior by adding an energy saving concern. To be applied, it takes into account a brightness sensor, therefore allowing to switch on the light when the brightness is under a defined threshold. Moreover, the new assembly sends a message to give a feedback the user when it tries to switch on the light while the brightness is too high.

```
1 advice brightness_light ( light, brightness, switch ) :
2
3 Emitter : 'BasicBeans.PrimitiveValueEmitter'
4 threshold : 'BasicBeans.Threshold' ( threshold = 10 )
5 t1,t2 : 'System.Windows.Forms.TextBox'
6
7 light.SetState -> (
8        if (threshold.IsReached) {Emitter.FireValueEvent}
9        else {call})
10 Emitter.^EmitStringValue -> (
11       t1.set_Text)
12 brightness.^Value_Evented_NewValue -> (
13       threshold.set_Value ; t2.set_Text)
```

Fig. 15. Sample of ISL4WComp advice

The advice is called `brightness_light`. The three variables `light`, `brightness` and `switch`, defined at the first line, describe the joinpoints (eg components) identified by the pointcut matching that will be used in the advice. They will be replaced by the instanciated joinpoint identified at weaving time. This AA highlights the three types of rules previously presented. At line 3,4 and 5, some blackbox components are added. The *threshold* component is instanciated with the property threshold up to 10. A property is a public variable from a component accessible through its interface. Line 7 defines a rewritting rule for input ports. All links connected to the input port (method) 'SetState' will be rewritten. Line 10 and 12 describe a creation rule for interactions from output ports (event).
Among all the operators found in this example, the more complex is `call` (line 9). The `if` block describes that if the brightness threshold is reached the system must sent an error message. Otherwise, the `call` operator is replaced with the left part of the rule, by the original method call that is being rewritten. This means that when this adaptation is woven into the base application, if a link to the input port `lumiere.SetState` was previously defined, the interaction already defined will still be implemented.
ISL4WComp is well-defined to describe reactive adaptations to create ubiquitous applications. Its operators can define more complex behavior than a structural reconfiguration. Moreover, it guarantees that the result of the weaving of several behaviors is idempotent, associative and commutative.

4.2.4 The weaver

The weaver is the program responsible for aspects weaving. It builds a unique component assembly from a base assembly and a set of Aspect of Assembly. The base assembly of an application is the assembly without any AA applied. The weaver manages all the processes that are required to weave some adaptations (FIG. 16). The weaving process can be decomposed into three steps. First, the pointcut matching is a function that has a set of components, from the initial assembly, and pointcuts, from a set of selected AA, as input. Its goal is to find the joinpoints on which advices will be woven. The second step is called the advice factory. It then generates instances of advices, replacing variable components in advices of selected aspects by joinpoints obtained during the first step. Instances of advices describe modifications to be woven in the actual base assembly of components. Finally, the composition engine merges all instances of advices with the initial assembly. It generates a single instance of advice that will be woven as the final assembly. In the next section, we will present more in details those processes.

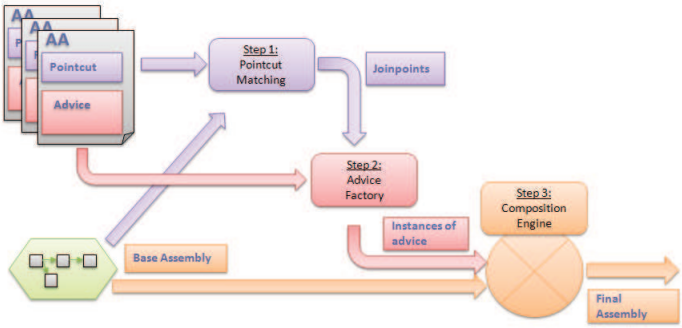

Fig. 16. The weaving process

4.2.4.1 Pointcut matching and advice factory

The weaver has a list of aspects of assembly that have been selected by users for the adaptation of an application. Their weaving in the target assembly depends on the evaluation of pointcuts. When a joinpoint is identified for each pointcut rules of an AA, it becomes *relevant*. (FIG. 18). Then the advice factory generates instances of advices, replacing variable components in advices of selected aspects by joinpoints obtained during the weaving. Several strategies are possible when several joinpoints are identified for a same rules (FIG. 12). An example can help to understand the different options: let us consider an application composed of a pair of switch and a pair of lights. A first strategy would create two combinations: the first containing the first switch and the first lamp, and the second consisting of the second switch and the second lamp. An ''all combination" strategy would create the combinations of all pairs of switches and lights. The choice of a strategy is up to the designer.

4.2.4.2 Conflict identification

Two instances of advice are conflicting (interacting) when they have at least one joinpoint in common. So, when several instances of advice and the base assembly are composed, some conflicts may occur. They have to be detected and considered by the weaver. Some operators in the advice language can define how conflicts should be managed. As an example, using ISL4WComp, the `call` and `delegate` operators are replaced by conflicting rules, but do not define any order when there are more than two rules conflicting.

4.2.4.3 Instances of advice composition

As we have seen, several AA can be applied on the same application. In the ISL4WComp language (4. 2.3.2), operators are based on a set of logical rules that ensure the property of *symmetry* to the instance of advice composition. This property is composed of three subproperties: *associativity*, *commutativity* and *idempotency*. These logical rules are grouped in a matrix of composition operators ensuring the three subproperties. AA composition is an implementation of formal works on the composition of logical rules (Cheung-Foo-Wo, Blay-Fornarino, Tigli, Déry, Emsellem & Riveill (2006)). According to those logical rules, the weaver is able to resolve AA's rules conflicts while ensuring the symmetry property of the weaving operation. So that the order in which rules are merged is not important and neither is the order in which instances of AA are woven.

4.2.4.4 Adaptations Triggering

In the manner of automatons cycles, consisting of a phase of acquisition (storage of inputs), then processing and finally writing outputs, we're talking about weaving cycle (Figure 3). Inputs are AA and an assembly, processing is the weaving process and output is a new assembly, a new application. In order to be reactive, the weaver can be triggered in two ways:

User-driven by changing the set of AAs given as input to the weaver. This can be done by selecting/deselecting or adding/removing aspects of assembly at runtime. When the set of AA is modified, the weaver is triggered, leading to adaptation if an added AA can be applied or if an AA has been removed.

Infrastructure-driven when a new device appears or disappears in the environment, a new component communicating with the device, is dynamically instantiated in or removed from the assembly. The adaptation process is triggered and only AAs that can be woven according to newly available components are applied. AA over SLCA benefits from the dynamicity of such an architecture. They can be triggered when a proxy appears into a

container since it sends a notification trough its structural interface when a new component is instanciated.

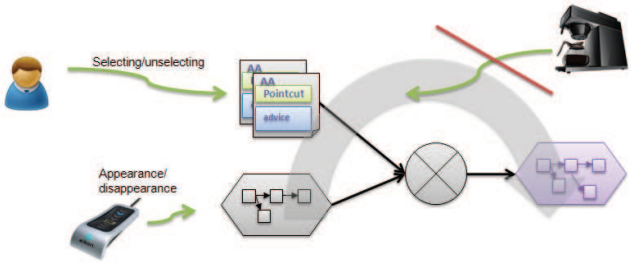

Fig. 17. Triggering mechanisms

An important point for the reactivity of such a mechanism is that the system doesn't require any information about the state of the software infrastructure. With a dynamic of its own, the infrastructure imposes its pace. However, during a weaving cycle, the system does not tolerate other disruptions (FIG 17).
Therefore, th life-cycle of an AA passes through various states (FIG 18). Originally an AA is in an *unselected* state. This means that the user does not want to apply it. In such a case, its pointcuts are not even evaluated. When an AA is selected, its pointcuts are evaluated. They will be evaluated for each modification of the assembly on which will be applied the AA. If some joinpoints are satisfying all the pointcut rules of an AA, it becomes *relevant* before being woven. AA that were not relevant may become, unless a new component appears in the application assembly. Similarly, those that have been woven can become disapplicated and still *selected* if an entity identified by their pointcut disappear.

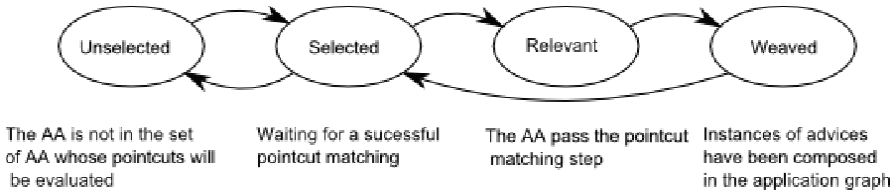

Fig. 18. AA life cycle

4.3 Synthesis

Aspect of Assembly are a model of compositional adaptation mechanism triggered by events from the software infrastructure. The model is generic enough to allow its use in various fields and for various concerns. According to the language used to express AA, various adaptations policies and composition policies can be defined. The ISL4WComp language helps building safe adaptations thanks to the property of symmetry and the respect of components blackbox properties.

5. Experiments

To validate our works in term of performances some experiments on service composition and adaptations have been made. First, we will present some results on the creation time of basic

components in a composite service. Second, we will describe some results on the major step of the adaptation process and for the overall process.

5.1 Experiments on service composition

The SLCA model has been projected into an implementation called SharpWComp 2.0, which was deposed as copyrighted software in France, used and developed in three programs of the French National Research Agency (ANR). Service composition in pervasive computing needs to be reactive to take into account changes of the infrastructure quickly and to adapt to users' needs. We measured time of creation and destruction of components in a composite service in SharpWComp 2.0 (FIG. 21).

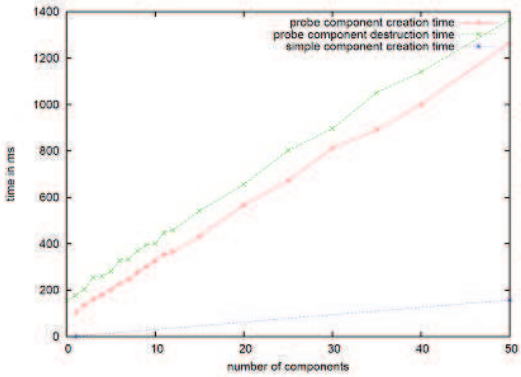

Fig. 19. Component creation and destruction time measures.

The creation time of basic components, as well as proxy components, is constant, around 3ms. Therefore, to create n components, $3 \times n$ ms are needed. The removal of such components couldn't have been measured, because we are in a managed memory environment. This is equivalent to dereference the instance of the component, and remove it from the container's list, which was too fast to be measured. Link creation and destruction time are also too simple operations and could not be measured. These measures correspond to the Lightweight Component Architecture (LCA). For probe components, that rely, in SharpWComp 2.0, on Intel's C# UPnP stack, the creation and destruction time are more important. This is due to the fact that when changing the service interface of a composite service, service advertisements are sent to inform that the previous interface is no longer valid, and then they are reissued with the new interface. With UPnP, an advertisement has to be made for each existing service, so if we consider that a probe component creates a service, every new probe will correspond to sending one more message each time. This is why adding the fortieth probe will take nearly one second

The generation time for proxy component is an important factor in our model. We measured it for a standard light device, containing ten methods divided into two services and two evented variables: the average value is 140.6ms. Thus, the time elapsed from the appearance of a service on the infrastructure to the adaptation of a composite service can be calculated. It will be a sum of the proxy component generation time (140.6ms), the component instantiation time (3ms), the adaptation of the composite service time, depending on how many new components are created, especially probe components and their number in the former assembly.

5.2 Experiments on assembly adaptation

We validate our approach in term of reactivity with some experiments on components assemblies randomly generated. Weaving cycles can be divided into three categories, each with its own cost in time.

1. Selection of AAs and pointcut matching
2. The advice factory
3. Composition and potentially merging of advice instances.

Those experiments were conducted on a standard personal computer (Athlon x2 1,8GHz processor). For this purpose various types of components have been instantiated randomly.
The advice factory step is a low cost process in term of duration. Experiments have shown that for an assembly including about 300 joinpoints and 2 AA, the duration of the process is between 2 or 3 ms.
Some experiments have been made on pointcut matching duration. They have involved a pointcut consisting of three rules, and a set of joinpoints ranging from 0-300. Several experiments have been made, the curve presented in Figure 20 is an average of these series and the standard derivation between the values obtained. We can conclude that the pointcut matching process is not time consuming.
The curve presented in Figure 20 shows the experimental results of the merging mechanism with a conflict probability about 0.5 for the red curve and about 0.33 for the blue curve. These evaluations highlight the high cost of the merging mechanism which is about 85 percents of the total cost of the weaving process. Then the probability of conflict between several instances of advice also plays a major role in the duration of the conflict resolution mechanism.

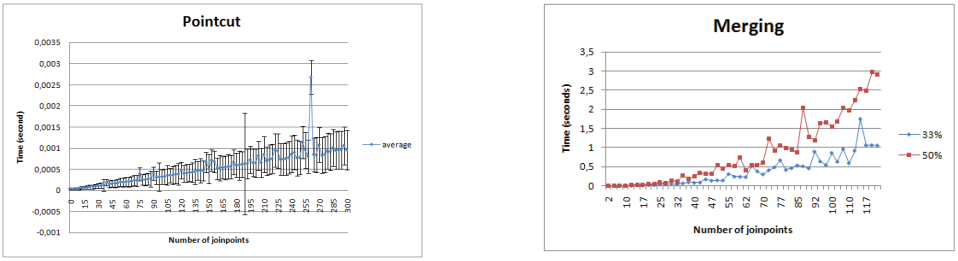

Fig. 20. Pointcut matching and merging time measures.

Figure 21 presents the duration of a weaving cycle according to the number of joinpoints in the base assembly. We consider that all these joinpoints are satisfying the pointcut matching and all combinations between all those joinpoints are generated.
In the field of human computer interactions, it is considered that the user latency at most is about 100ms. Then, Bérard in (Crowley et al. (2000)) propose that the latency for highly tied interactive systems must be twice lower than user latency : 50ms. Under this bound, we are able to compose about 30 components together. On the other hand, ubiquitous computing does not necessarily require such a response time. In the field of domotics, a bearable latency is about 1 second. Under this bound we are able to compose about 100 joinpoints.

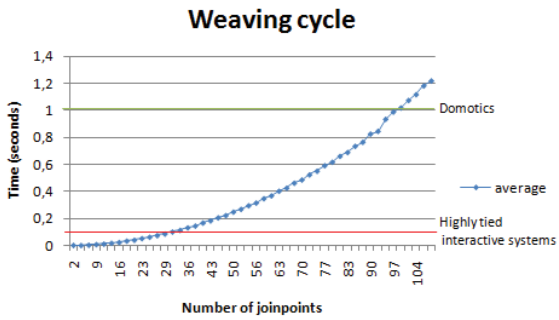

Fig. 21. Weaving time measures.

6. References

Arnold, K., Scheifler, R., Waldo, J., O'Sullivan, B. & Wollrath, A. (1999). *Jini Specification*, Addison-Wesley Longman Publishing Co., Inc.

Berger, L. (2001). *Mise en Œuvre des Interactions en Environnements Distribués, Compilés et Fortement Typés : le Modèle MICADO*, Thèse de doctorat, Université de Nice-Sophia Antipolis - Faculté des sciences et techniques, École doctorale STIC - Informatique.

Bottaro, A., Gérodolle, A. & Lalanda, P. (2007). Pervasive service composition in the home network, *Advanced Information Networking and Applications, 2007. AINA'07. 21st International Conference on*, pp. 596–603.

Breivold, H. & Larsson, M. (2007). Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles, *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pp. 13–20.

Brønsted, J., Hansen, K. & Ingstrup, M. (2007). A survey of service composition mechanisms in ubiquitous computing, *Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI) at Ubicomp*.

Bustamante, F., Widener, P. & Schwan, K. (2002). Scalable directory services using proactivity, *Proceedings of Supercomputing 2002*.

Cardoso, R. S. & Issarny, V. (2007). Architecting Pervasive Computing Systems for Privacy: A Survey, *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, IEEE Computer Society, p. 26.

Chakraborty, D., Joshi, A., Finin, T. & Yesha, Y. (2005). Service Composition for Mobile Environments, *Mobile Networks and Applications* 10(4): 435–451.

Champion, M., Ferris, C., Newcomer, E. & Orchard, D. (2002). Web services architecture, *W3C working draft* .

Chappell, D. (2007). Introducing SCA, *David Chappell and Associates* .

Charfi, A. & Mezini, M. (2004). Aspect-oriented web service composition with AO4BPEL, *Lecture Notes in Computer Science* pp. 168–182.

Chen, H., Chakraborty, D., Xu, L., Joshi, A. & Finin, T. (2000). Service discovery in the future electronic market, *Proc. Workshop on Knowledge Based Electronic Markets, AAAI2000, Austin*.

Cheung-Foo-Wo, D. (2009). *Adaptation Dynamique par Tissage d'Aspects d'Assemblage*, PhD thesis, Université de Nice - Sophia Antipolis.

Cheung-Foo-Wo, D., Blay-Fornarino, M., Tigli, J., Lavirotte, S. & Riveill, M. (2006). Adaptation dynamique d'assemblages de dispositifs dirigée par des modèles, *2ème journées sur l'Ingénieurie Dirigée par les Modèles (IDM)*.

Cheung-Foo-Wo, D., Blay-Fornarino, M., Tigli, J.-Y., Déry, A.-M., Emsellem, D. & Riveill, M. (2006). Langage d'aspect pour la composition dynamique de composants embarqués, *RTSI - L'Objet* 12(2-3): 89–111.

Clarke, M., Blair, G., Coulson, G. & Parlavantzas, N. (2001). An efficient component model for the construction of adaptive middleware, *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Springer-Verlag, pp. 160–178.

Crowley, J., Coutaz, J. & Bérard, F. (2000). Perceptual user interfaces: things that see, *Communications of the ACM* 43(3).

David, P. & Ledoux, T. (2006). An aspect-oriented approach for developing self-adaptive fractal components, *Lecture Notes in Computer Science* 4089: 82.

Douence, R. & Sudholt, M. (2002). A model and a tool for Event-based Aspect-Oriented Programming (EAOP), *LMO'03* .

Englander, R. (1997). *Developing Java Beans*, O'Reilly & Associates, Inc.

Escoffier, C. & Hall, R. (2007). Dynamically adaptable applications with iPOJO service components, *Lecture Notes in Computer Science* 4829: 113.

Ferry, N., Lavirotte, S., Tigli, J.-Y., Rey, G. & Riveill, M. (2009). Context Adaptative Systems based on Horizontal Architecture for Ubiquitous Computing, *International Conference on Mobile Technology, Applications and Systems (Mobility)*.

Guttman, E. (1999). Service Location Protocol: Automatic Discovery of IP Network Services, *IEEE Internet Computing* 3: 71–80.

Hourdin, V., Lavirotte, S. & Tigli, J.-Y. (2006). Comparaison des systèmes de services pour dispositifs, *Technical Report I3S/RR-2006-25-FR*, Laboratoire I3S, Sophia Antipolis, France.

Hourdin, V., Tigli, J.-Y., Lavirotte, S., Rey, G. & Riveill, M. (2008). SLCA, composite services for ubiquitous computing, *Proceedings of the 5th International Conference on Mobile Technology, Applications and Systems(Mobility)*, p. 8.

Huang, P., Lenders, V., Minnig, P. & Widmer, M. (2002). Mini: A minimal platform comparable to Jini for ubiquitous computing, *International Symposium on Distributed Objects and Applications (DOA), Irvine*.

Kiczales, G., Bobrow, D. & des Rivieres, J. (1999). *The art of the metaobject protocol*, MIT press.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. (2001). An overview of AspectJ, *ECOOP 2001 - Object-Oriented Programming* pp. 327–354.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. & Irwin, J. (1997). Aspect-oriented programming, *ECOOP*, SpringerVerlag.

MacKenzie, M., Laskey, K., McCabe, F., Brown, P. & Metz, R. (2006). Reference model for service oriented architecture 1.0, *Technical Report wd-soa-rm-cd1*, OASIS.
URL: *http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm*

Papazoglou, M. (2003). Service-oriented computing: Concepts, characteristics and directions, *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pp. 3–12.

Pinto, M., Fuentes, L. & Troya, J. (2005). A dynamic component and aspect-oriented platform, *The Computer Journal* 48(4): 401.

Preuß, S. (2003). JESA Service Discovery Protocol: Efficient Service discovery in ad-hoc networks, *Lecture notes in computer science* pp. 1196–1201.

Sedov, I., Preuss, S., Cap, C., Haase, M. & Timmermann, D. (2003). Time and energy efficient service discovery in Bluetooth, *Vehicular Technology Conference, 2003. VTC 2003-Spring. The 57th IEEE Semiannual*, Vol. 1.

Singh, M. & Huhns, M. (2005). *Service-oriented computing: semantics, processes, agents*, John Wiley & Sons Inc.

Szyperski, C., Bosch, J. & Weck, W. (1999). Component Oriented Programming, *Lecture Notes in Computer Science* 1743: 184–184.

Tigli, J.-Y., Lavirotte, S., Rey, G., Hourdin, V. & Riveill, M. (2009a). Lightweight Service Oriented Architecture for Pervasive Computing, *International Journal of Computer Science Issues (IJCSI)* 4: 1–9.

Tigli, J.-Y., Lavirotte, S., Rey, G., Hourdin, V. & Riveill, M. (2009b). Lightweight Service Oriented Architecture for Pervasive Computing, *International Journal of Computer Science Issues (IJCSI)* 4.

Vallée, M., Ramparany, F. & Vercouter, L. (2005). Flexible composition of smart device services, *The 2005 International Conference on Pervasive Systems and Computing(PSC-05)*.

Ververidis, C. & Polyzos, G. (2008). Service Discovery for Mobile Ad Hoc Networks: A Survey of Issues and Techniques, *IEEE Communications Surveys and Tutorials* .

Vinoski, S. & Inc, I. (1997). CORBA: integrating diverse applications within distributed heterogeneous environments, *IEEE Communications Magazine* 35(2): 46–55.

Zambrano, A., Gordillo, S. & Jaureguiberry, I. (2004). Aspect-based adaptation for ubiquitous software, *Mobile and Ubiquitous Information Access* pp. 136–140.

Zhu, F., Mutka, M. & Ni, L. (2005). Service discovery in pervasive computing environments, *IEEE Pervasive Computing* 4(4): 81–90.