Ann. Telecommun. (2009) 64:197–214
DOI 10.1007/s12243-008-0081-y

WComp middleware for ubiquitous computing: Aspects and composite event-based Web services

Jean-Yves Tigli · Stéphane Lavirotte · Gaëtan Rey · Vincent Hourdin · Daniel Cheung-Foo-Wo · Eric Callegari · Michel Riveill

Received: 6 March 2008 / Accepted: 5 December 2008 / Published online: 16 January 2009
© Institut TELECOM and Springer-Verlag France 2009

Abstract After a survey of the specific features of ubiquitous computing applications and corresponding middleware requirements, we list the various paradigms used in the main middlewares for ubiquitous computing in the literature. We underline the lack of works introducing the use of the concept of Aspects in middleware dedicated to ubiquitous computing, in spite of them being used for middleware improvement in other domains. Then, we introduce our WComp middleware model, which federates three main paradigms: *event-based Web services*, *a lightweight component-based approach to design dynamic composite services*, and *an adaptation approach using the original concept called Aspect of Assembly*. These paradigms lead to two ways to dynamically design ubiquitous computing applications. The first implements a classical component-based compositional approach to design higher-level composite Web Services and then allow to increment the graph of cooperating services for the applications. This approach is well suited to design the applications in a known, common, and usual context. The second way uses a compositional approach for adaptation using Aspect of Assembly, particularly well-suited to tune a set of composite services in reaction to a particular variation of the context or changing preferences of the users. Having detailed Aspect of Assembly concept, we finally comment on results indicating the expressiveness and the performance of such an approach, showing empirically that principles of aspects and program integration can be used to facilitate the design of adaptive applications.

Keywords Ubiquitous computing · Web services for devices · Event-based component middleware · Software composition

V. Hourdin is employed by MobileGov since Oct 2008.

J.-Y. Tigli (✉) · S. Lavirotte · G. Rey · V. Hourdin · D. Cheung-Foo-Wo · E. Callegari · M. Riveill
Laboratoire I3S, Université de Nice - Sophia Antipolis / CNRS, Bâtiment Polytech'Sophia - SI 930 route des Colles, B.P. 145, 06903 Sophia-Antipolis Cedex, France
e-mail: tigli@polytech.unice.fr

S. Lavirotte
e-mail: lavirott@polytech.unice.fr

G. Rey
e-mail: rey@polytech.unice.fr

V. Hourdin
e-mail: hourdin@polytech.unice.fr

D. Cheung-Foo-Wo
e-mail: cheung@polytech.unice.fr

E. Callegari
e-mail: callegar@polytech.unice.fr

M. Riveill
e-mail: riveill@polytech.unice.fr

D. Cheung-Foo-Wo
CSTB 290, route des Lucioles, BP209, 06904 Sophia-Antipolis, France

1 Ubiquitous computing

We are standing on the brink of a new computing era, one that will fundamentally transform our computing usages. In September 1991, Mark Weiser in [38] unveiled his vision of ubiquitous computing. He described the future like a world where computing systems are available anywhere but not visible. Already, early forms of ubiquitous computing are obvious in the widespread use of laptops and mobile phones. But how did we get here?

Leaving the mainframe time, the society, motivated by desires of individualism, did migrate to a personal computing model. Supported by lot of technologies' innovations, two majors ways, identified by Lyytinen in [23], appear. Firstly, the mobility integrates the society way of life, and at the same time integrates phones and computers. Secondly, a kind of technophobia or, more precisely, a society rebuttal in front of the growing difficulty to use the new technologies did give birth to concept of integration. The computing systems integration with the physical environment act toward hiding computing systems complexity and diversity for end-users.

However, beyond these criteria of mobility and integration, what are the ubiquitous computing challenges? The principal challenge of ubiquitous computing is to resolve the new computing "multiple-multiplicity." Indeed, now, many users can simultaneously use many applications (fragmented in many pieces of software often called services). These users interact with many devices to communicate with other people located in many different physical places and environments.

In summary, we could identify three concepts concerning entities (users and devices) evolving in the new ubiquitous world. The entity's mobility is the first concept of the new world. It describes motions of users and of their devices. The second concept is the entity's heterogeneity, which outlines the diversity between entity's capabilities and possibilities offered by various unknown functionalities of new smart objects. Finally, the last concept is the environment high dynamicity. It illustrates the ubiquitous world entropy with its appearance and disappearance. As a result, future ubiquitous computing architectures should implement these concepts to solve ubiquitous computing challenges.

The scope of this paper can now be outlined briefly (Section 2). We will first draw a state of the art on middlewares for ubiquitous systems according to the most relevant criteria found in literature (Section 3). We then study paradigms used in the ubiquitous computing research field: services oriented architectures, component-based software engineering, event-driven middlewares, and finally aspect-oriented programming. (Section 4) From what we have learned about existing middlewares and paradigms characteristics, we defined WComp, our lightweight component model, ubiquitous computing ready, using services to abstract devices and context from the environment, and aspects for structural adaptation of dynamic applications (Section 5). We explain more deeply our aspect adaptation approach, called *Aspect of Assembly* (Section 6). We then validate our contributions, studying performances and complexity. (Section 7) Adaptation is either user-driven or context-driven. We explain used mechanisms for adaptation of applications (Section 7). Use cases are finally described, in the ubiquitous computing environment (Section 8). We summarize our approach of middleware for ubiquitous computing and give directions for future works.

2 Approaches for adaptation in ubiquitous computing environments

Many middlewares have appeared in the ubiquitous computing world, and even more in pervasive or sensors networks, dedicated to adapting software architectures to context changes. We start by listing relevant middlewares dedicated to ubiquitous computing and studying the main characteristics of ubiquitous computing systems. We will compare paradigms used in this field in the next section (Section 3) with the same characteristics and requirements.

2.1 Middleware requirements for ubiquitous computing

The requirements and main characteristics of middlewares for ubiquitous computing have been widely studied in lots of papers [6, 15, 25, 27]. These papers try to define basic requirements for such middleware systems. We will only focus on a subset, which represents relevant characteristics of ubiquitous computing middleware and referring to our research work.

Of course, all these middlewares support *adaptation*, but we distinguish two categories [6]: *structure* changes or *behavior* changes. Structural adaptation consists in modifying an assembly of components while preserving its behavioral services. A behavioral service describes a sequence of operations to be executed on a particular component. Thus, a behavioral adaptation may, in some cases, lead to the failure of the black-box abstraction of components or services.

Heterogeneity is the ability to handle different programming languages, operating systems, hardware, or communication protocols. *Extensibility* is the ability to extend or add new functionalities to the system easily. *Scalability* refers to the ability of a system to grow in the future, to extend to higher-load applications or to a wider network. *Security* can be an important concern in some applications since ubiquitous computing may use private data from the user. Some middlewares use authentication and authorization mechanisms to protect user data.

Reactivity is a key feature for pervasive or ubiquitous adaptive systems. If it has to react on context changes, middleware has to handle some kind of event notification, like a publish/subscribe mechanism. *Mobility* is, of course, handled by all ubiquitous middlewares, since they create applications from mobile devices, and a changing context. The *discovery* of those devices is important too; it is better to discover dynamically which device is in the environment than hard-code them beforehand. The last characteristic we will focus on for the state of the art is *updating*, which is the ability to update parts of the middleware, like components or services at run-time.

Mascolo et al. [25] have isolated other characteristics to adaptive systems, like feasibility, which is a mechanism-handling resource unavailability resulting in middleware functionalities that cannot be provided at some time. Since ubiquitous systems are context-dependent, they have to deal with such concerns. Robustness is another characteristic, which should be handled by ubiquitous adaptive systems. Feasibility can be a part of it. Execution environment moving, devices appearing/vanishing, and error rises must not affect middleware stability and its capacity to adapt continuously.

2.2 Existing middlewares

We focus on few middleware approaches for ubiquitous computing, and we summarize in Table 1 how they handle previous characteristics and requirements.

- Gaia [31] aims to provide middleware support for active space environments such as smart rooms and living environments. It essentially provides a distributed operating system where all inputs, outputs, and processing units within a room are considered as a single computer. Gaia uses a component repository and centralized approaches to events and services discovery. Code can be updated, replacing components in the repository, which limits used services to a static list, thus preventing new services from being dynamically added to applications. Gaia handles heterogeneity with encapsulation of active spaces, presenting them as a programmable environment. However, when heterogeneity is handled by abstraction, there are always some capabilities unavailable through the abstraction layer. Moreover, its objects can be distributed on any node, which requires all nodes to be set up with an object container (CORBA).
- ExORB [32] project's main aim is to contribute towards construction of configurable, updatable, and upgradable middleware services. It targets the mobile phone industry; thus, mobility is explicitly addressed. Code updating is possible but requires human intervention to spread the changes. ExORB uses IIOP and XML-RPC, enabling heterogeneity. Its software configuration can change at runtime, implying an adaptability potential. However, applications are designed for local execution, and the discovery of entities is made with a local object broker and does not address reactivity.
- CORTEX [35, 37] proposes a novel sentient object model to address the emergence of a new class of application that operates independently of human control. Infrastructure-based and ad-hoc-based wireless environments are considered to address mobility. The middleware is highly configurable at run-time. It reacts on events by changing the behavior of objects. CORTEX does not use black-boxes for its objects. They are adapted modifying their internal content. It does not address security, while it aims at quality of service in large-scale architectures.
- Aura [18] is a context-aware middleware that can be used to create mobile applications. It represents the user by its aura, like a personal area network, and brings the appropriate resources from the services of the environment to support the user's task. Aura migrates tasks depending on context changes, which are notified by events. It is also interesting to

Table 1 Characteristics of middleware approaches for ubiquitous computing

	Structural adapt.	Behavioral adapt.	Heterogeneity	Extensibility	Scalability	Security	Reactivity	Mobility	Discovery	Updating
Gaia	x			x		x	x	x	x	x
ExORB	x	x		x					x	
CORTEX		x	x				x	x		
Aura		x	x			x	x	x	x	
Oxygen		x		x		x		x	x	x
SATIN	x						x	x	x	x
DoAmI	x		x					x		
SCORPIO	x		x		x			x		

note that it suspends tasks that cannot be processed anymore due to a context change, storing their state for a future resume. User location information is secured by a SPKI/SDSI (Simple Public Key and Simple Distributed Security Infrastructures). Moreover, constraints can be expressed in task descriptions, and the middleware restricts some of its operations in order to stop a violation if they are violated (on context, for example). Heterogeneity in Aura is handled by several mechanisms: services can be dynamically discovered with Jini, which is Java-based only, and interactions with services can be done using different mechanisms like CORBA, COM, or RPC, which are not device-oriented.

- Oxygen [1] addresses human needs using speech and vision technologies that enable the user to communicate with it as if the user were interacting with a person. It enables pervasive human-centered computing. It defines intelligent networks with dynamic topologies according to devices locations, fixed and mobile devices with embedded software. Code can be automatically updated thanks to that. Network rules can be specified to allow sets of users to use particular resources. However, it uses objects, communicating with method invocation, and does not handle reactivity or heterogeneity.
- Self-adaptation targeting integrated networks [39] (SATIN) argues that the application of logical mobility primitives in a component system assists in building self-organizing mobile systems. They define a component-based middleware, dynamically updatable, for example, on context changes. It aims at the reconfiguration of mobile device code. Components communicate together inside the same address space, which is a good point for performances. They use Java for supporting heterogeneity of architectures but not for programming languages. They also use Jini for discovering services.
- Domain-specific ambient intelligence [6] (DoAmI) is a service-oriented middleware architecture. It uses CORBA, which enables language heterogeneity handling capabilities and centralized discovery of services. Depending on found services and the current context, DoAmI interconnects them and sets them into running state. However, it only relies on services, so services must be created with adaptation purposes in mind. Indeed, adaptation is limited to basic service compatible interfaces, and there is no middleware message processing. DoAmI does not yet include event-based communications between services.
- SCORPIO [8] proposes a work about structural adaptation of software components. It restructures components in order to match heterogeneous structures when integrating new components. Moreover, they propose to divide behavioral services into several groups so as to deploy them separately on different systems for load balancing. Structural adaptation can be made adapting interfaces of composite components, but it requires code generation and application relaunch. Reactivity is not addressed.

Table 1 gathers these overviewed approaches. For each, we checked the characteristics from (Section 2.1) of ubiquitous systems programming supported. Those approaches are not able to fulfill all our requirements for the kind of system we deal with. They often base themselves on standard middlewares like CORBA, which handles a lot of the characteristics. However, they are not ubiquitous computing oriented and lack a way to properly manage highly dynamic or mobile applications and adaptation, which is exactly what these projects want to add.

3 Paradigms

Adaptation requires the ability to reconfigure the deployed code, which is considerably simplified when applications are loosely coupled and modular rather than monolithic blocks of codes. Numerous systems have been designed in order to partially respond to ubiquitous applications' problems. We distinguish component-based, service-oriented, event-driven, and aspect-oriented systems. By *component-based*, we refer to dynamic and easily manipulable system engineering; by *service-oriented*, we refer to architectures based on services descriptions and interactions; by *event-driven*, we focus on publish–subscribe-based middlewares, which notify entities by significant changes; and by *aspect-oriented*, we refer to the methodology enabling separation of concerns.

3.1 Component-based software engineering

Components, as they were defined in [2], are an alternative to object-oriented programming in the design and handling of basic entities. Components provide functionalities, exported and used through their interfaces. We focus on black-box components, for which we only know the semantic, and not the implementation, like services. In this case, adaptation cannot be reflection-based, like in object-oriented programming for white-box components. The behavior of components cannot be modified internally; we can only modify the composition between them. The corresponding adaptation is, thus, much more structural than behavioral.

Components are usually more finely grained than services. However, some component models, like EJB or CCM, are more seen as services on this point since, in addition, they can be deployed and distributed on a network. On the other side, OSGi is often considered as a component-based system, due to its relative lightness. We mainly focus on *lightweight* component models, like JavaBeans or .NET components despite their need for a virtual machine, which can be fitted in embedded systems and easily used in pervasive systems. Conversely, *heavy* components include a part of the middleware that makes them become services, i.e., capable of automatic injection of proxy and dynamic construction of glue codes. They can also handle multiple requests at the same time, and robustness is increased, due to several nonfunctional embedded properties. The intersection of a lightweight component and a service constitutes the *core* functional component. [17] suppressed a level of complexity by introducing the self-adaptive component model K-Component, which enables individual components to adapt to changing environments through a complex decentralized coordination, model which simplified the integration of multiple objectives and allowed groups of components to collectively adapt their behavior. Component-based systems bring dynamicity to local application, enabling pieces of softwares and relations between them to be added, removed, or updated at run-time.

3.2 Service-oriented systems

The main features of service-oriented systems are their flexibility in handling dynamicity and their suitability for the integration of new devices. They are also relevant to and very used by distributed computing. Services can appear or vanish on a network, notifying the whole system, and reconsidering the services used for the application that best suits the needs. From a certain point of view, this notification can be considered as being part of an event-driven system, but this is only made by the services repository, and services cannot send applicative events to other services.

Our aim is not that different from the aim of the nondistributed lightweight service architecture OSGi [24]. However, this approach remains Java-dependent, and therefore, the model stays, to some extent, confronted to object-oriented architecture interdependency. We draw from the CORBA standard, which was a precursor of services-oriented architectures used for distributed computing, enabling different languages and different computer architectures to share data and act in the same application. Later, Web services came up, providing this kind of wider software interoperability using Web standards.

A second point is lacking in services-oriented systems when used in pervasive systems: they rely on a centralized architecture, like the CORBA broker, or a UDDI repository for Web services. Mobile systems can appear on several networks, often wireless and not persistent, and need a more flexible approach.

Furthermore, [39] provided logical primitives to transfer codes to reconfigure software systems and enhance robustness. Robinson et al. [30] focused on the configuration and integration of devices in pervasive computing scenarios, which include self-organizing configuration for pervasive computing environments supporting unskilled installation. They coupled a domain-specific language (DSL) and middleware, but with a centralized approach. Service-oriented systems allow robustness, coordinating services in a programmatic decentralized collaboration.

3.3 Event-driven systems

Event-driven architectures have been used for self-adaptive or reconfigurable systems for many years. Their common distinctive feature is the weak coupling of components, meaning that individual components do not know the components realizing their required functionalities at design time. The information is set at runtime either by the component itself or another one. The first case is illustrated by the reflective component model OpenCOM v2, where new types of components can be added and function calls can be altered by modifying a *process vtable* [10]. The second case is known as the principle of *inversion of control* that has been experimented on in a *lightweight container* in [4, 14] as an interactive adaptive system. The third case appears in distributed systems, like service-oriented architectures, and even more nowadays with the outcome of event-driven SOA [26]. Events serve well such distributed black-box systems, highly loose-coupled. Two points make events helpful in ubiquitous computing middlewares: loose coupling and reactivity. When request/response communications are used in an application, when it needs to be adapted, all the irrelevant method invocations need to be reconfigured. Thanks to events and their high decoupling, entities do not have to explicitly manage communications between them. This is also a consequence of inversion of control. Weak-coupling offers a high degree of expandability, but its relatively low level of abstraction does not allow complex software design.

Events also increase reactivity of applications. It was generally used in parts of applications interacting with users, like GUIs. Now it is used for discovering services as soon as they appear on a network, notifying changes in states of pieces of softwares, or get a better reactivity when creating applications from devices of the ubiquitous environment. Event-driven systems are not suitable for very complex designs, but they are adequate for reactivity, dynamicity, and high adaptability.

3.4 Aspect-oriented systems

We consider that the three previous paradigms are made to be used for *composition*, creating the system behavior and initial structure. Aspect-oriented programming can be seen as an orthogonal approach, used for *adaptation*.

Aspect-oriented systems [21] consist of a set of join points, pointcuts, advice, and weaving loops, which operate at runtime or design-time to construct or modify an executable program from cross-cutting concerns. It cannot exist alone, and is most often associated with object-oriented programming. The trend consists now in considering adaptations as cross-cutting components woven as classical AOP aspects. David and Ledoux [16] designed a DSL and expressed adaptation concerns as aspectual components in order to monitor self-adaptive systems. He also proposed to express pointcuts in terms of binding scripts. However, this approach does not provide a collaborative combination and does not avoid semantic conflicts by the bindings declaration. Aspect-oriented systems provide an enhanced modularity as they include separation of concerns, but are not intended to achieve service collaboration.

3.5 Paradigms of existing middlewares

To link these paradigms with real-world approaches, we list what paradigms are used by the middlewares we studied in Section 2.2 (Table 2).

- Gaia relies on services with events and components, but in a centralized approach. This is not convenient for networks that change frequently. Most ubiquitous applications must deal with various networks and intermittent connections. Thus, centralized approach might be impracticable. Because brokers might be unreachable, in centralized approaches, parts of the applications could be ineffective. Thus, basically distributed solutions, with, for example, a dynamic discoverable registry or broker, like in WS-Discovery, seem to be the only efficient way to deal with ubiquitous applications.
- ExORB uses microbuilding blocks, which behave close to a component architecture for the modularity and updatability; it has a component structure but is called ORB like object request broker. However, the communication protocols used look more like services ones. Therefore, it is not very clear what paradigm is used by ExORB.
- CORTEX uses objects as a main entity of its network and events for communications between them.
- Aura finds services in the environment and notifies the middleware using events for context changes. However, it is built with components.
- Oxygen uses a distributed object-oriented database to upgrade software, improve performances, and add features easily.
- SATIN defines a component model but does not rely on any other paradigm.
- DoAmI uses CORBA services only.
- SCORPIO performs adaptations using structural modifications of a component assembly.

We see that approaches used to create middlewares for ubiquitous computing can vary a lot for the paradigm used. But what are the advantages of using one paradigm or another, or even several at a time? The two following subsections will answer this concern.

3.6 Comparison of paradigms characteristics

Table 3 summarizes relative strengths and weaknesses of studied paradigms. We see that we cannot get all ubiquitous computing requirements if we do not use both component, service, and event paradigms. Aspects

Table 2 Paradigms used by middleware approaches

	Events	Services	Components	Objects	Aspects
Gaia	x	x	x		
ExORB			x		
CORTEX	x			x	
Aura	x	x			
Oxygen				x	
SATIN			x		
DoAmI		x			
SCORPIO			x		

Table 3 Comparison of self-adaptive approaches

	Component	Service	Event	Aspect
Adaptation	x			x
Heterogeneity		x		
Extensibility	x			
Scalability	x	x	x	
Security	x	x	x	x
Reactivity			x	
Mobility	x	x		
Discovery		x		
Updating	x			x

are orthogonal to these, and give full strength to component assembly adaptations.

Components are best suited for adaptation, due to their modularity and dynamicity, as a support for aspect to weave on. Heterogeneity or communication protocols, devices, and languages can be reached using Web services. Reactivity needs a publish/subscribe mechanism to broadcast information to several services of the environment at the time it gets available.

3.7 Multi-paradigm systems

Multiparadigm systems are born to take advantages of several paradigms at the same time. Table 3 (Section 3.6) has proved that using only one paradigm cannot achieve full support of ubiquitous computing requirements.

For example, services-oriented systems and event-driven systems have given birth to *services for devices* [12] like JINI [7]. They give services the ability to send events by themselves to any other service which want to receive them. They also break the need for a centralized repository and make fully distributed architectures, using multicast discovery. To enable interoperability and standardization brought by Web services, Web services for devices were created, the two currently existing being UPnP [20] and DPWS [33]. However, creating applications based on Web services for devices only may be quite nonevolutive since discovered services, or, more exactly, their interface, have to be known at code time.

Another example of combination is SCA, which stands for service component architectures [3]. SCA handles *components and services*, using components to manipulate service orchestrations and create higher-level services. However, components used in this model cannot be classified as lightweight components since the framework provides life cycle operations such as lazy instanciation, or a reduced transaction management, called conversations.

Several combinations of *components and aspects* have also appeared. For example, Aokell [34] uses aspects to create component containers (called “membranes” in the Fractal component model). This only adds nonfunctional properties handling using aspects. The same team has created Fractal Aspect Component (FAC) [28], which uses a symmetric solution representing aspects by components. With FAC, aspects can intercept messages between components since they fit in the controller but do not handle conflicts and structural adaptation of the assembly when inserting or removing aspects. Pointcuts can be specified as a method name, component name, or return type, using regular expressions. Aspect-Oriented Component Infrastructure [36] is another approach using aspects to adapt component-oriented middlewares, using EJB, which makes it quite heavy for pervasive computing, but its main drawback is that they use grey-box components and violate the interface access only.

The most promising approach is Self-Adaptative Fractal Components [16], though it is autonomic-computing oriented. It defines a structural–adaptable component platform, using aspects as adapting tools. Aspects advice represent a list of structural modification to be applied on the base assembly. The pointcuts can be defined by two kinds of events: changes from the execution context (e.g., memory or attribute of a resource) and changes from the execution of the target application itself (e.g., reception of a message or creation of a new binding).

Other works on adaptive middlewares are using aspects and services. *Services and aspects* are getting on the front of the scene; lots of works are appearing in pervasive computing or context-sensitive worlds [22, 29]. Service fits well with handling of devices in the environment, or mobile-computing issues needing dynamic discovery, and in these works, aspects bring an adaptation layer to services, respecting cross-cutting concerns between the functional code and adaptation code.

4 Our middleware model: WComp

We propose a middleware approach called WComp, taking into account, at best, all the previously explained principles for ubiquitous computing. For that matter, it federates three main paradigms:

- *Event-based Web services paradigm*: we distinguish two kinds of services: *composite services*, which are services whose implementation calls other services. They are opposed to *basic services*, whose implementations are self-contained and do not invoke any other services. They are generally Web services for devices like UPnP or DPWS (Section 3.7). These services handle heterogeneity, extensibility with the reuse of composite services, scalability, security using Web services security standards like WS-Security, reactivity since they use event-based communications, dynamic decentralized discovery, and mobility. Ubiquitous applications are then a graph of event-based Web services.
- *Lightweight component-based paradigm inside composite Web services*: a composite service is based on an internal lightweight components assembly to manage composition between other event-based Web services and to design the interface of a new higher-level composite service. We call this paradigm service lightweight component architecture (SLCA) [19], which is based on events, and a minimum of extra-functional properties unlike SCA [3]. A composite service is then a WComp container managing a dynamic assembly of lightweight WComp components and providing an event-based Web service interfaces. Components handle the high dynamicity of the model, providing a way to be structurally adapted. They also address reactivity, since they use event-based communications. A composite event-based Web service is dynamically managed using an internal lightweight components assembly.
- *Adaptation paradigm using the original concept called Aspect of Assembly (AA)*: this concept allows us to prepare kinds of *independent and crosscutting* schemes of adaptation dealing with separation of concerns, logically mergeable in case of conflicts and applicable to every composite Web service of the application, not necessarily known (a priori). Aspects provide adaptation to the model, which is structural, since we modify the internal component assembly of composite services without modifying black-box base components.
  Adaptations, as a set of AA, are designed without knowing event-based Web services of the applications. They are applied (weaved for AA) to the set of event-based Web services of the applications at runtime implementing required adaptations.

Thus, our middleware allows us to adopt both ways to dynamically design ubiquitous computing applications. The first implements a classical component-based compositional approach, using SLCA, to design higher-level composite Web services and then increments the graph of cooperating services for the applications. This approach is well suited to design the applications in a known, common, and usual context (Fig. 1). We call such a compositional approach *composition for higher-level services*.

The other way uses a compositional approach for adaptation using AA, particularly well-suited to tune a set of event-based Web services in reaction to a particular variation of the context or even new preferences of the users. We call such a compositional approach *composition for adaptation*.

4.1 Composition for higher-level services with WComp

WComp is a lightweight component-based approach to design composite Web services (Fig. 2). *A composite service encapsulates a WComp container*, managing a dynamic assembly of lightweight WComp components. The WComp component model is a slightly modified JavaBeans model adapted to other programming languages with the concepts of input and output ports, properties, and hierarchy. Still an instance of a component type, but not necessarily serializable, a component has a unique name and an interface composed of two sets of events and methods (event's names are prefixed by '^'). Types of components define their interfaces. We consider C the set of component instances, E the set of

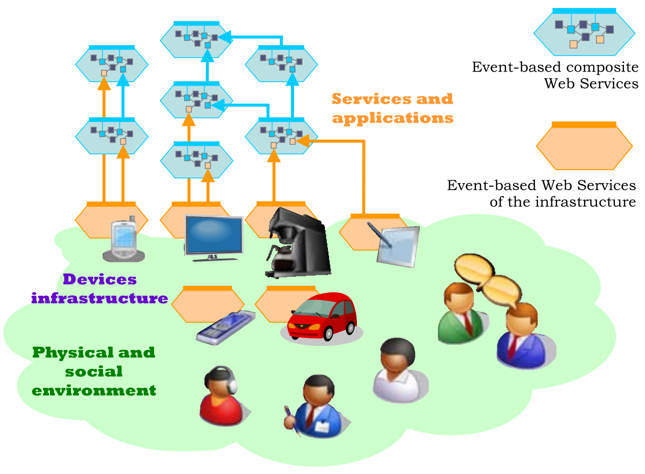

Event-based composite Web Services
Services and applications
Event-based Web Services of the infrastructure
Devices infrastructure
Physical and social environment

Fig. 1 Graph of event-based Web service

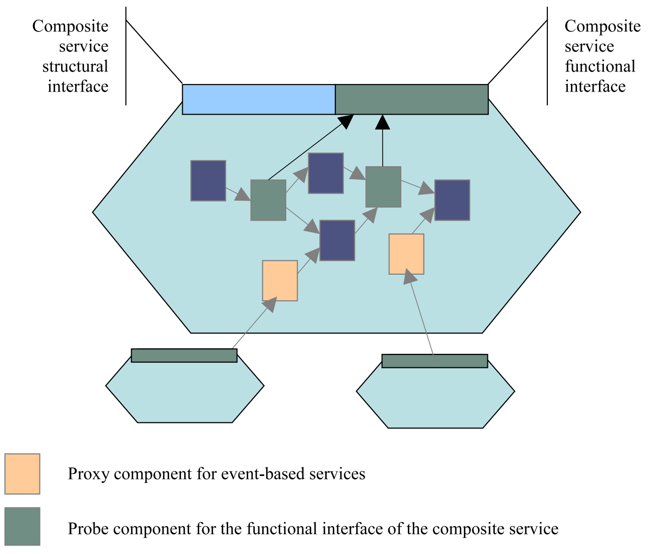

Composite service structural interface
Composite service functional interface
Proxy component for event-based services
Probe component for the functional interface of the composite service

Fig. 2 Composite event-based Web service

events characterized by their unique name, and M the set of methods. We gather the declaration of events and methods in the term "port." We consider a set of links L, which are lists composed of an instance event and a type method. Then, an assembly consists of a subset of C and L. The *container component* implements an API to dynamically control this assembly and, consequently, the addition and removal of elements in C and L. Roughly speaking, we use events—also known as late-bindings, "*push*" *mechanism*, or *inversion of control*—in lightweight containers for communications between components, what we can call connectors in our model. This is now a shared characteristic of adaptive component models [9].

A composite service provides both service interfaces. A first interface concerns the new functionalities provided by the composite Web service (called functional interface) and the second one allows us to dynamically modify the internal assembly of WComp components (called structural interface).

The Functional interface exports events and methods of the internal assembly using *probe components*. The insertion of a new probe component dynamically modifies the functional interface and the description of the corresponding composite service.

The structural interface allows an external client service to dynamically modify the internal assembly of the service by adding or removing links and components. It is possible for the external client service to be another composite service using a proxy component on this service.

Hierarchical composition of services Remote event-based Web services can be handled in a WComp lightweight assembly by using a lightweight *proxy component*. This is how a hierarchical relationship can be constructed in the WComp model.

4.2 Composition for adaptation with WComp

In our first model of WComp [14] (a middleware for ubiquitous computing), we have introduced an adaptation mechanism based on sets of AAs. An AA can either be selected by the user or triggered by context changes in a self-adaptive process. Multiple selected AAs are then composed by a weaver according to logical merging rules in a high-level specification (Section 5). The result of the weaver is projected in terms of pure elementary modifications (PEMs)—add, remove components, link, unlink ports. Components are then involved in different interaction patterns. We detail the AAs principles in the next section.

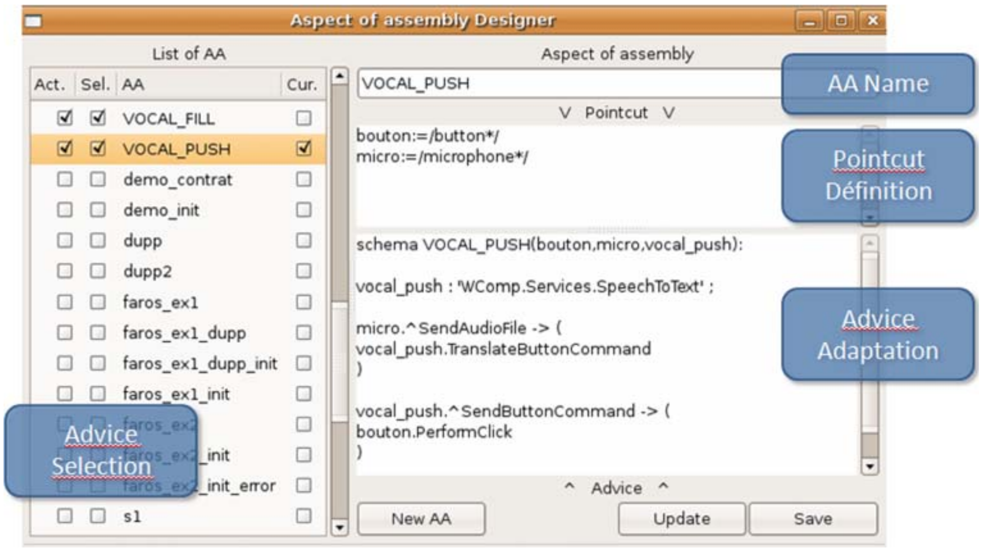
5 Aspects of assembly

We propose a component-oriented integration that takes into account the adaptation characteristics in (Section 2.1). Our architecture is twofold: it consists of an extended model of AOP for adaptation advices and of a weaving process with logical merging. We implemented a toolkit (Fig. 3) that includes AAs as the central concepts. We introduce here concepts used in the rest of the paper:

Base assembly: an assembly of components.
Join point: components and ports of the base assembly.
Pointcut: a description of a set of join points for a particular adaptation advice.
Adaptation advice: adaptation advice describing architectural reconfigurations.
Weaver: mechanism integrating advice according to specified pointcuts selecting join points from a base assembly. It is also responsible for the merging of conflicting advice.

An AA is structured as an aspect with a pointcut and advice (adaptation advice), which is specified in a DSL using interaction specification firstly defined in [11]. This DSL has then been enhanced in [14] to integrate event-driven declarations. With this present approach, self-adaptive pervasive software developers can reason, plan, and validate AA-based assemblies at all stages of the development phase. Using logical predefined validation rules, logical configurations' incompatibilities can be detected at runtime.

Fig. 3 AA definition



List of AA

Act.	Sel.	AA	Cur.
☑	☑	VOCAL_FILL	☐
☑	☑	VOCAL_PUSH	☑
☐	☐	demo_contrat	☐
☐	☐	demo_init	☐
☐	☐	dupp	☐
☐	☐	dupp2	☐
☐	☐	faros_ex1	☐
☐	☐	faros_ex1_dupp	☐
☐	☐	faros_ex1_dupp_init	☐
☐	☐	faros_ex1_init	☐
☐	☐	faros_ex2	☐
☐	☐	faros_ex2_init	☐
☐	☐	faros_ex2_init_error	☐
☐	☐	s1	☐

Aspect of assembly

VOCAL_PUSH

V Pointcut V

```
bouton:=/button*/
micro:=/microphone*/
```

```
schema VOCAL_PUSH(bouton,micro,vocal_push):

vocal_push : 'WComp.Services.SpeechToText' ;

micro.^SendAudioFile -> (
vocal_push.TranslateButtonCommand
)

vocal_push.^SendButtonCommand -> (
bouton.PerformClick
)
```

^ Advice ^

Advice In AAs, the advice is not a piece of code that will be weaved into components. It defines a set of component instances (C) and links (L) that will be weaved inside the targeted assembly of components.

We present an example of advice that is used in a practical situation for raising an alarm when someone has not been visible for a time or is out of reach. The advice, called "Ex," redefines an input and an output port and is applied to a set of components symbolically represented by the *observed* and *alarm* variables:

```
ADVICE Ex (observed, alarm):
    observed.^Out ->
            (  IF ( alarm.Check ) CALL )
    alarm.Check ->
            (  alarm.Start ; CALL      )
```

Description Firstly, it redefines the *ˆOut* output of the *observed* component, which specifies that actions possibly defined in the base assembly are executed only if the *alarm* component authorizes it. Secondly, it redefines the *Check* input of *alarm*, which specifies that, before the execution of the input possibly required by other components, *alarm* must be started, i.e., the *Start* input must be executed.

We defend a minimalistic approach in order to be able to cope with scalability. For this reason, those specifications are translated into a set of PEM. Any modification can be regarded as an assembly-to-assembly transformation. Thus, the AA designer depicted in the bottom window in Fig. 3 communicates its PEM to a *container* (Section 4.1).

Pointcut We define pointcut descriptions as sets of filters on base assembly metadata—component ID, their types, etc. Those filters construct a list of parameters satisfying the list of variables of an advice for the latter to be integrated in the base assembly. If only one list is constructed, the advice is integrated only once in the base assembly and the symbolic variables are syntactically replaced in the advice to match the base assembly join points. If several lists are constructed, the advice is duplicated and each set of variables, with one occurrence of each join point, are respectively replaced. For our experiments, we choose for convenience to express filters in the AWK language [5] and define a simple grammar to make AWK responses correspond to advice variables: "<variable>:=<AWK filters>;..." For example:

```
1 observed := /user*/ ;
2 alarm :=
3     /err*/ { a[substr($1,3)]=$1 }
4     END  {for(i=1;i<=NR;i++){print a[i]}};
```

Description The *observed* variable is matched against component ID starting with "*user*" and *alarm*, against those starting with "*err*." The second filter (lines 3–4) is an AWK program that, more than matching the beginning of component IDs, actually sorts them by alphanumeric order. Line 3 stores the IDs in a table, depending on their suffix. At the end of the matching test, the program displays stored IDs sorted.

The order of the components is not specified and can be random when a specific program in AWK to sort them is not written. In this example, the first pointcut is unordered and the second is ordered. We consider a base assembly of five components: *err*1, *err*2, *err*3, *user*1, and *user*2. The advice is duplicated into two applicable advices (Ex1, Ex2). The global result is a two-dimensional table where duplicated advices' parameters are represented in the columns:

user2	user1		← this line is not sorted
err1	err2	err3	← this line is sorted

Consequently, in the two duplicated advices Ex1 and Ex2, the parameters of Ex1 and Ex2 are not associated with the parameter with respectively the same ID: user2 is, rather, associated with err1 and user1 is associated with err2.

```
1 ADVICE Ex1(user2,
    err1):
2  user2.^Out ->
3   (IF(err1.Check)
        CALL)
4  err1.Check ->
5   (err1.Start ;
        CALL)
```

```
1 ADVICE Ex2(user1,
    err2):
2  user1.^Out ->
3   (IF(err2.Check)
        CALL)
4  err2.Check ->
5   (err2.Start ;
        CALL)
```

The decision to integrate adaptation advice according to specified pointcut follows the following rules: (1) only the first complete columns of the table become parameters of the duplicated advices (in this example, only the two first columns become parameters). (2) The order of the ID in the first line $\{user2, user1\}$ can change. Therefore, to apply an advice deterministically, lines must be sorted.

Weaver with logical merging The logical integration rules are represented by a matrix representing the two-by-two merging of operators. Indeed, advices are described using a set of operators and connectors names. We created a set of seven operators (inspired by previous works [11]), fitting most needs for adaptation of the behavior of the overall application. Future works will focus on other specific domains for composition, like HMI, for example.

We give only a few examples of logical rules in Fig. 4, but all merge algorithms between operators are well defined. We explain the weaving of two advices called "Ex" and "AA0" (lines 1 and 5 below). Hypothesis: two pointcuts, respectively specifying the "*observed*" variable and the "*worker*" variable, are in conflict (produce the same join points):

```
1  ADVICE Ex (observed,alarm):
2    observed.^Out ->
3        (  IF ( alarm.Check ) CALL  )
4    alarm.Check -> ( alarm.Start ; CALL )

5  ADVICE AA0 (producer,worker,consumer):
6    producer.^Out -> ( worker.In )
7    worker.^Out -> ( consumer.In )
```

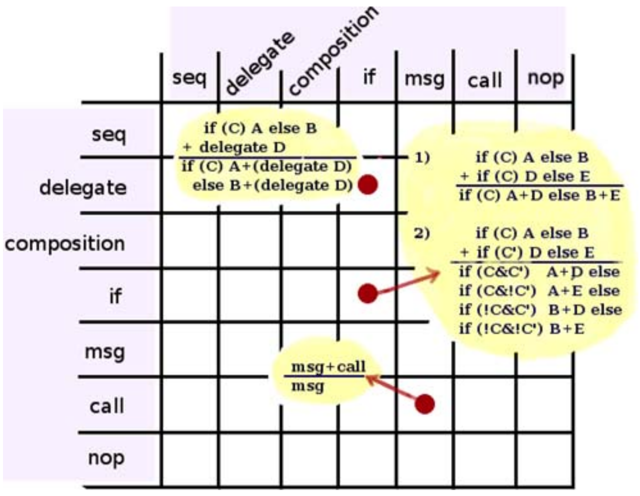


Fig. 4 Operator merging matrix

Merging example The specification rules (SRs) at lines 4 and 6 are not conflicting. Thus, they are copied in the resulting advice (lines 5 and 6 below). However, the SRs at lines 2 and 7 are conflicting because they redefine the same output *ˆOut* of the confounded *observed*/*worker* component. Therefore, their respective specification programs are logically merged and the resulting "AA0+Ex" advice is calculated using the merging matrix (Fig. 4). The "+" operator corresponds to the unordered couple of operations to execute. The merging process replaces *CALL* at line 3 of "Ex" by *CALL + consumer.In* in "AA0+Ex." The resulting AA is then translated into a set of PEMs. For instance, *IF* operator is interpreted as the addition of a generic component of type IF.

```
1 ADVICE Ex+AA0 (observed,alarm,
2                producer,consumer):
3  observed.^Out ->
4    (IF(alarm.Check) {CALL + consumer.In})
5  alarm.Check -> ( alarm.Start ; CALL )
6  producer.^Out -> ( observed.In )
```

In more complicated merging cases, the merging process can be explained as follows: we parse an advice into a semantic tree in which nodes are operators and leaves are components ports. When weaving two advices, operators are merged two by two according to the operator merging matrix, propagating the merge down to the leaves. This matrix verifies associativity, commutativity, and idempotence properties in the overall merging process. Thus, the order in which more than two advices are merged is not important at this level, and we can write this equivalence: $Ad1 \otimes Ad2 \otimes \ldots \otimes Adn = (((Ad1 \otimes Ad2) \otimes \ldots) \otimes Adn$ [13]. Multiple advices merging results, finally, in a single merged advice semantic tree, which will be translated into a set of PEMs.

We saw the AA-specific design process as well as one cycle of the adaptive pervasive application. In the next section, we present the process cycles used to perform self-adaptation.

6 Validation

We comment on the results of a few experiments on sets of randomly generated assemblies. The purpose is to show the advantages of AAs while evaluating additional costs concerning adaptation time of composite Web services in ubiquitous applications. Moreover, these experiments will allow us to identify parameters of the model that will be explained for the duration of the pointcut matching process and validate the adaptation process.

In this section, we present the first part of a step-by-step model of the weaving process using AAs. We draw some experimental results in order to verify and identify parameters of a simple performance model we propose to predict pointcut matching performance under certain conditions.

6.1 Step-by-step model of the weaving process

The *assembly size* is the number of components and links. The weaving process is separated in four steps (Fig. 5): *selection* of AA (1), *pointcut matching* (2), *composition and merging* of AAs (3), and *translation and modifications* from an AA to elementary modifications (4) for the *container* of the corresponding composite service.

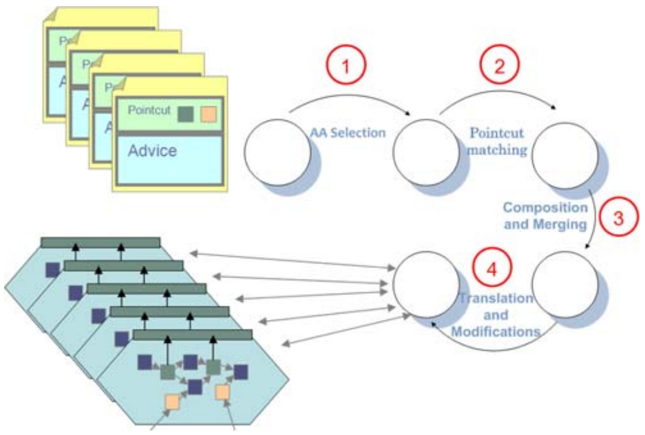


Fig. 5 Adaptation cycle with AA

First of all, the composition process (3)—logical merging of AAs—depends on the advice of the AAs. Indeed, even if we can assign a measure to an AA in terms of the sum of the number of links and components that are necessary for its description, we remain unable to predict the rules that would be processed in order to compose and possibly merge the AAs together. As an example, terminal rules, such as the "msg+call" rule, cost less than recursive rules, such as the "if(...)+if(...)" rule in (Fig. 4), and rule selection depends on AA advice specification. It is difficult to predict the content of an AA, which depends on many factors (application domain, scenario complexity, etc.). Therefore, we cannot provide a model of the composition time process yet. Finally, modification and transformation model (4)—AA to PEMs and interpretation by the container—exploits composition results. This is the reason why we cannot provide a model for it yet.

However, pointcut matching (2), together with selection (1) (which gives join points and specifies duplications of AAs), are processes that perform computation on sets of components and AAs. Thus, their duration varies according to our initially defined parameters (program size and AA size). We may thus design a simple model.

Let D_p be the duration of the pointcut matching process. Let $a_1, \ldots, a_2$ be the model parameters (specific to a particular type of hardware) and c the number of components in the base assembly.

6.1.1 Pointcut matching

Opposed to standard adaptation mechanisms, AAs allow more generic modifications by matching the points at which adaptations occur inside any composite service. Thus, this matching process is what has to be explicitly modelized.

Let A_{init} be the number of initial AAs (those in the repository). We note d_i the number of duplications of the AA number i. We have A_{init} AAs, and each of them is associated with a pointcut specification. Hence, each pointcut gives the number d_i of duplications. Each duplication is processed in order to calculate the new advice of the AA.

The calculation consists in an AWK processing. We propose here a very simple model of the AWK processor, saying that it behaves like c^2, where c stands for the number of components. We have the following model for pointcut matching:

$$D_p = a_1 . \sum_{i=1}^{A_{init}} (d_i + 1) * c^2 + a_2$$

The quantity d_i is not easy to determine because it might depend on usage or characteristics of the application. Typically, this quantity depends on how components appear in the system. We have experimented on a probabilistic model and we have estimated a value for d_i.

This model, however, might not be sufficient for relatively small values for c because it is rather simple. However, we obtain a quite good approximation, as shown in the following sections.

6.2 Experimental results

We have measured the pointcut matching duration and confronted it with our simple model. In this section, we describe firstly the experimental conditions under which we have performed the experiment. Then, from the results, we propose an identification of the parameters of the model.

6.2.1 Pointcut matching example

The experiment was performed on an Intel T2300 1.66 GHz processor. During this experiment, components appeared randomly according to a binomial law. We can already infer that the number of duplications noted d_i in the previous section is dependent on this law.

On this system, we programmed an application as follows. Every tenth second (for about 20 s), a new component among two categories l_x and s_y is randomly added to the base assembly and their indices x and y are incremented each time from 0 to 200. Every time a new component is added, the pointcut matching process is executed and a set of AAs is selected and applied. To keep the example simple, we defined only one AA, called aa_0, but able to duplicate when specified.

We explored two cases: the first case consists in disallowing aa_0 to duplicate; the second case allows it. We confront our model (in green in the following figures) with the real-world measures (in red).

1) **Without duplication** The first case we analyze is without duplication, and we obtain the graphs in (Fig. 6). This means that, for every aspect i, the number of duplications d_i is zero, and we only use one AA. So, the formula is then simplified as follows:

$$D_p = a_1.c^2 + a_2$$

We draw in (Fig. 6) the experimental result with the predicted model. Although simple, we can see that the model fits the experimental data quite well. This

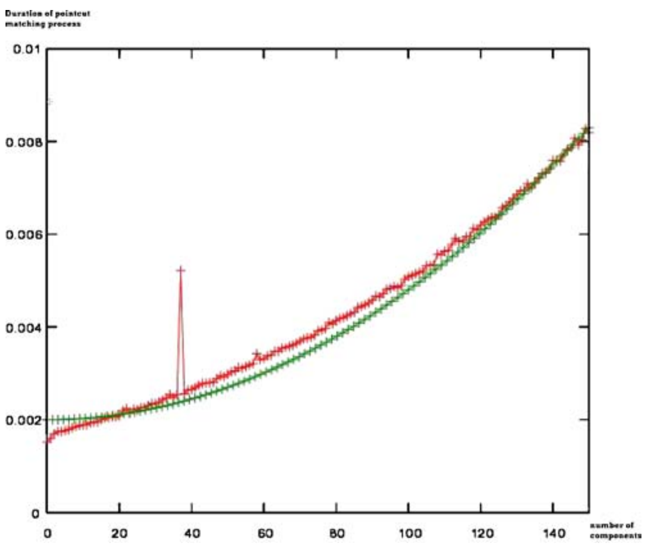


Fig. 6 Experimental results without duplications

first experiment permits us to determine the model parameters. We found $a_1 = 280.10^{-9}$ and $a_2 = 2.10^{-3}$.

2) **With duplication** The second case consists in systematically duplicating the AA. However, in such a case, we need to know how d_i behaves. As a matter of fact, we know that a component appears randomly, chosen among l_x and s_y. Therefore, the AA has a probability of being duplicated of $\frac{1}{2}$. This is why we take $d_i = \frac{1}{2}$, which is the probability of getting the right parameter in order for the AA to be duplicated. After simplifications, we obtain the following formula for the performance model and reuse the parameters we determined in the previous experiment:

$$D'_p = a_1.\frac{3}{2}.c^2 + a_2$$

Figure 7 shows that the parameters are correct. Those parameters are characteristics of the hardware system. We can see irregularities in the experiment. This is due to memory collection.

6.2.2 Parameters identification example

We have identified the parameters of our model for a T2300-based computer. By bringing the results of the experiment and the model face to face, such as in (Fig. 6), we obtain an approximation of the model parameters (a_1 and a_2). We obtain approximately for the two cases the values for a_1 and a_2 that remain the same for the two experiences and, thus, characteristics of the hardware system:

$a_1 = 280.10^{-9}$ and $a_2 = 2.10^{-3}$

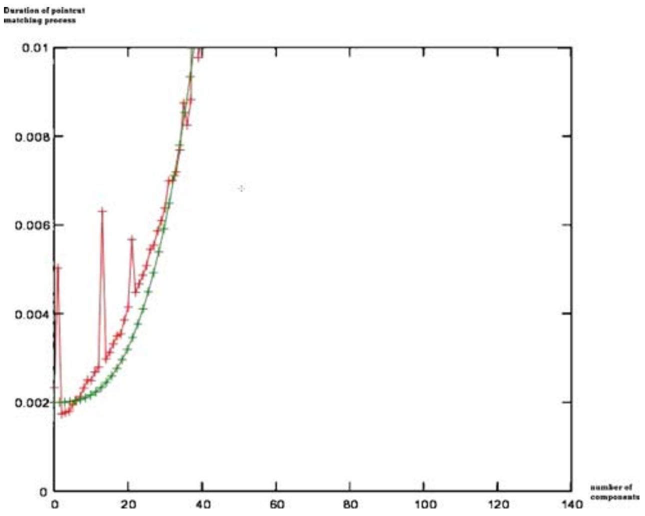


Fig. 7 Experimental results with duplications

7 Demonstration of self-adaptation cycles

Self-adaptation consists in reacting to modifications operated by the user or the environment. Self-configuration is processed by the decoupled AA designer. We describe the user-driven approach and the process, which permits us to adapt the application to its environment (Fig. 8).

The *user-driven adaptation* consists in (de-)selecting AA in order to integrate or erase some behaviors and functionalities in the system. The user can also intervene on the base assembly and operate directly on the assembly. Concerning the area of end-user programming, we distinguish expert and end users. Expert users can design new AAs for new situations, whereas end-users do not have to create AAs, but only select predefined AAs. In that case, the interaction with the user is simplified.

The *context-driven* adaptation consists in scanning the underlying infrastructure periodically in order to verify if devices are still present in the environment. New devices can asynchronously inform the system of their presence by broadcasting a notification. Therefore, when a device is removed from the system's environment, the software component representing the device is unlinked and removed from the base assembly. Conversely, when a new device appears, a new software component representing this new device is added to the assembly. Consequently, the self-adaptation process consists in detecting those structural changes in the base assembly and each cycle of the process checks if either new AA are applicable, or is applied AA are not valid anymore. This depends on whether required components to an AA are present or not. If a notable change occurred, it recalculates PEMs to be applied on the base assembly.

However, two cases should be considered when an adaptation calculation occurs. The base assembly can be empty (at least, with no links between components ports). In such a case, the application—more precisely, the interactions between components—is constructed by iterations of the application of AAs. Conversely, the base assembly can be composed of interconnected components. In that case, before adapting the assembly by iterations of application of AA, the base assembly (under the form of ADL) is translated into an AA, which is always selected to be composed so that the composition of PEMs takes into consideration this initial state. For example, the advice "AA0" explained in Section 5 is the AA result of the transformation of a base assembly.

Finally, the adaptation process is projected on a set of services and composite services as defined in Section 4.1 and is considered as a distributed system.

To illustrate the principles of described mechanisms, let us take an example of an ubiquitous application. The example we will describe is based on a multidevices application to send text messages through a network. There are three kinds of communication modes: Wifi or GSM connection and, when we lack any infrastructure, the application can store messages and send them to a cache system.

The described application is developed with basic components or services and the adaptation is made using AAs to reorganize the connections between entities, instanciate new software components, or interact

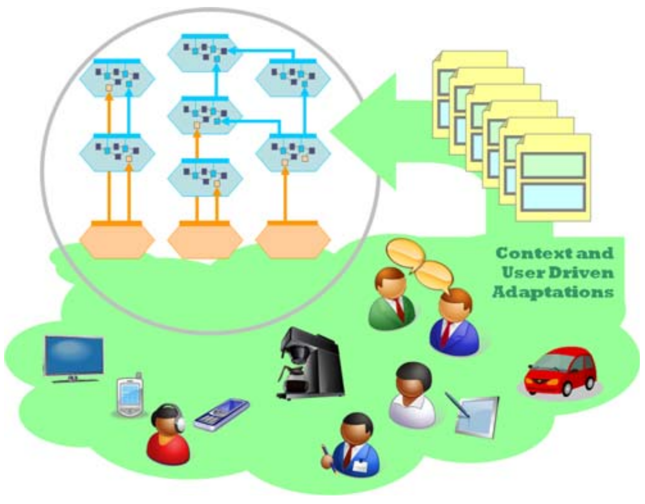
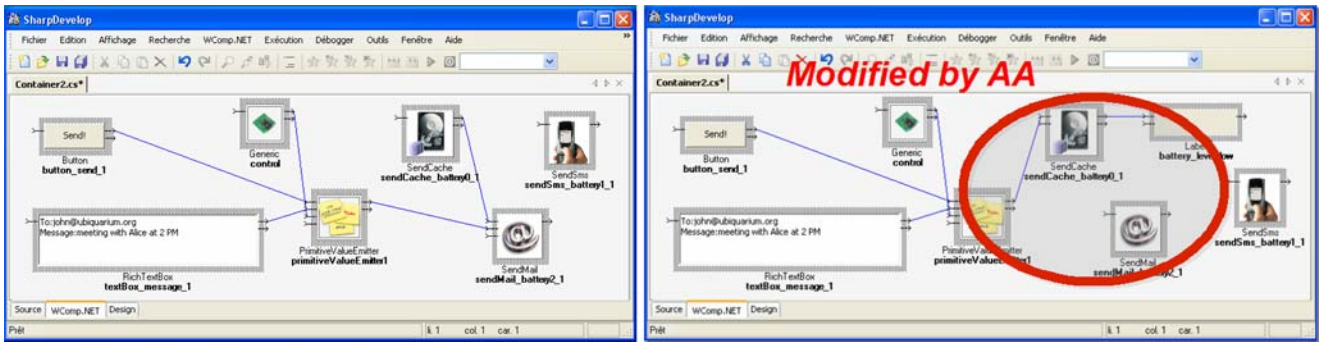


Fig. 8 Context- and user-driven self-adaptation using AAs



To:john@ubiquarium.org
Message:meeting with Alice at 2 PM

(a) Before Adaptation (b) After Adaptation

Fig. 9 Adaptation based on Wifi activation (**a**, **b**)

with new services. We have grouped aspects of assemblies into three categories:

- *Basic functionalities*: These AAs are used to build the application upon the basic available services and components. Each AA specifies the method to weave to add a new functionality (Wifi or GSM communication or caching system) to the existing application.
- *Energy policy*: We have defined three AAs based on power consumption policy:
  - Minimal power consumption: The Wifi and GSM devices will be disconnected from the application and all the communication will be routed to the cache system to store messages.
  - Standard power consumption: Messages are sent by SMS over the GSM network.
  - No limitation to energy consumption: Wifi is used and all messages are routed to this device.
- *Adding new functionality*: The functionality we want to add to our communication application is to be able to use it hands-free. To achieve this, we will define two kinds of aspects:
  - Voice control: This module is dedicated to bind a voice control system to activate part of the user interface.
  - Voice input: This module is used to achieve speech to text recognition to allow input text messages to the system.

All these aspects of assemblies are applied to the initial defined application to dynamically build the connections between components or services to give the right behavior. “Basic functionalities” are selected by the context exclusively (a user cannot decide to use a Wifi communication in the infrastructure if not present in the environment), but all other sets of AAs can be activated by user or by context of the application.

In the first example presented by Fig. 9, the application sends all messages via the Wifi connection (Fig. 9a). When the user wants to minimize power consumption to maximize the autonomy (*user-driven adaptation*) or when the battery is low (*context-driven adaptation*), the AA defined in the “Energy Policy” section modifies the application to send all data to the cache component (Fig. 9b).

The second example presented by (Fig. 10) shows the application adaptation provided by AAs dedicated to “New Voice functionality.” The adaptation consist in adding new software components and modifying the connections between components and services to add the new functionality. All modifications result in structural modifications on the assembly. Defined AAs for this example can be found in Section Annexe.

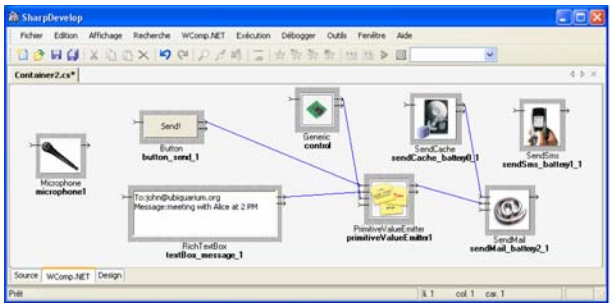
The described example has been implemented using WComp[1] and AAs paradigms. This application is also included in a framework we have developed for the study of mobile computer appliances in simulated environments, called Computer Ubiquarium.[2] The Ubiquarium[3] comprises various devices and services, which can be discovered and composed at runtime. Those devices can either be virtual devices (3D scene objects in which the user is immersed), or physical devices worn by the user or present in his/her immediate environment. All devices of the Ubiquarium, physical or virtual, are based on Web services for Devices that provide a uniform type of interface.

We can also add that the WComp middleware was created and used in French national research projects,

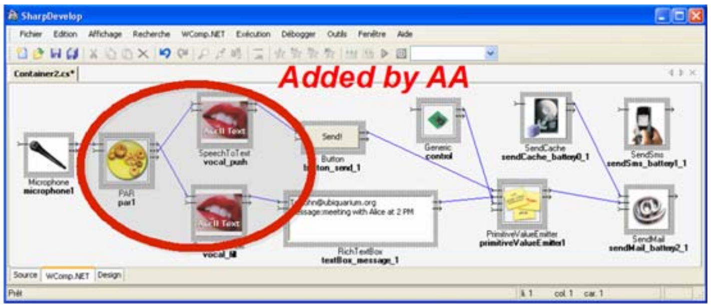
[1]http://rainbow.i3s.unice.fr/wcomp/.

[2]From Latin *Ubique*, everywhere, with the suffix *rium*, meaning location and structure. Hence, Ubiquarium means: “a location or a structure in which the computer is everywhere and in everything”.

[3]http://rainbow.polytech.unice.fr/ubiquarium/.



(a) Before Adaptation



(b) After Adaptation

Fig. 10 Adaptation to pilot application with voice (**a**, **b**)

RNTS Ergodyn, *RNTL Faros*, and *RNTL Continuum*, to provide experimentation prototypes for real case examples. The first aims at the creation of user-centered adaptation for people with disabilities. The second aims at the application of contracts in service-based applications. Finally, the third aims at user-centered service continuity in ubiquitous computing environments. This demonstrates the ability of the middleware to be used and extended to diverse domains. Indeed, other tools, like the AA designer, can be created in composite services and can be used to provide new methods of adaptation.

8 Conclusion

We first studied features and characteristics of ubiquitous computing middlewares and paradigms, which gave us ideas of what was mandatory and lacking in ubiquitous computing. As a result, we introduced the WComp middleware approach, which federates three main paradigms: an event-based Web services approach, a lightweight component-based approach to design composite Web services, and an adaptation approach using the original concept called AA. Then, we introduce both ways to dynamically design ubiquitous computing applications. The first implements a classical component-based compositional approach, using SLCA, to design higher-level composite Web services, and then increments the graph of cooperating services for the applications. This approach is well suited to design the applications in a known, common, and usual context. The second way uses a compositional approach for adaptation using AA, particularly well-suited to tune a set of event-based Web services in reaction to a particular variation of the context, or even new preferences of the users. We call such compositional approach *composition for adaptation*. In such a process, AAs are either selected by the user or triggered on context changes and composed by a weaver with logical merging of high-level specifications. The result is then projected in terms of PEMs of component assemblies. We finally commented on results indicating the expressiveness and the performance of such an approach, showing empirically that principles of aspects and program integration can be used to facilitate the design of adaptive application.

Our perspectives get organized around four ways. Firstly, we plan to separate the DSL from the AA concept in order to specify advices by means of assemblies of components making up “good practice” advices. Secondly, we want to explore a new and generalized AA-merging algorithm allowing the expert user to define his/her own merging strategies. At the same time, we will ripen the cost model of the composition step. Finally, we will press on our works on the AA to introduce a trigger mechanism to the AA selection mechanism.

Annexe: AAs

Adding new functionalities: vocal interface

Vocal control AA

```
POINTCUT
   button:=/button*/
   micro:=/microphone*/

ADVICE Vocal_Control(button, micro, vocal_push):
  vocal_push : 'WComp.Services.SpeechCommand';

   micro.^SendAudioFile -> (
       vocal_push.TranslateButtonCommand
   )

   vocal_push.^SendButtonCommand -> (
       button.PerformClick
   )
```

Vocal input

```
POINTCUT
   box:=/textBox*/
   micro:=/microphone*/

ADVICE Vocal_Input(box, micro, vocal_input):
   vocal_input : 'WComp.Services.SpeechToText';

   micro.^SendAudioFile -> (
       vocal_fill.TranslateText
   )

   vocal_input.^SendText -> (
       box.set_Text
   )
```

References

1. Computer Science and Artificial Intelligence Laboratory (2004) Mit oxygen project. http://oxygen.lcs.mit.edu/
2. WCOP'96 (1996) Summary of the WCOP'96 workshop in ECOOP'96, Linz, July
3. Open SOA (2006) Service Component Architecture spec. http://www.osoa.org/
4. Ahmed M, Ghanea-Hercock R, Hailes S (2006) MACE: adaptive component management middleware for ubiquitous systems. In: Proc. of the 4th int. workshop on Middleware for perv. and ad-hoc comp. ACM, New York, p 3
5. Aho AV, Kernighan BW, Weinberger PJ (1988) The AWK programming lang. Addison-Wesley, Reading
6. Anastasopoulos M, Klus H, Koch J, Niebuhr D, Werkman E (2006) DoAmI—a middleware platform facilitating re-configuration in ubiquitous systems. In: System support for ubiquitous computing workshop. at the 8th annual conf. on ubiquitous computing (Ubicomp 2006), Irvine, September
7. Arnold K (ed) (2000) The JINI specifications, 2nd edn. Addison-Wesley Professional, Reading
8. Bastide G, Seriai A, Oussalah M (2006) Adapting software components by structure fragmentation. In: Proc. of ACM symposium on applied computing
9. Bencomo N, Blair G, Grace P (2006) Models, reflective mechanisms and family-based systems to support dynamic configuration. In: Proc. of the 1st workshop on MOdel driven development for Middleware. ACM, New York, pp 1–6
10. Blair G, Coulson G, Ueyama J, Lee K, Joolia A (2004) OpenCOM v2: a component model for building systems software. In: IASTED software engineering and applications
11. Blay-Fornarino M, Charfi A, Emsellem D, Pinna-Dery A-M, Riveill M (2004) Software interactions. J Object Technol 3(10):161–180
12. Bussière N, Cheung-Foo-Wo D, Hourdin V, Lavirotte S, Riveill M, Tigli J-Y (2007) Optimized contextual discovery of web services for devices. In: IEEE int. workshop on context modeling and management for smart environments, October
13. Cheung D (2008) Dynamic adaptation weaving aspects. Ph.D. thesis
14. Cheung-Foo-Wo D, Tigli J-Y, Lavirotte S, Riveill M (2006) Wcomp: a multi-design approach for prototyping applications using heterogeneous resources. In: 17th IEEE int. workshop on rapid syst. prototyping, Crete, pp 119–125
15. Cheung-Foo-Wo D, Tigli J-Y, Lavirotte S, Riveill M (2007) Self-adaptation of event-driven component-oriented Middleware using aspects of assembly. In: 5th int. workshop on Middleware for pervasive and ad-hoc computing (MPAC), California, November
16. David P-C, Ledoux T. (2006) An aspect-oriented approach for developing self-adaptive Fractal components. In: Softw Comp, pp 82–97
17. Dowling J, Cahill V (2004) Self-managed decentralised systems using K-Components and collaborative reinforcement learning. In: Proc. of the 1st ACM SIGSOFT workshop on self-managed systems. ACM, New York, pp 39–43
18. Garlan D, Siewiorek D, Smailagic A, Steenkiste P (2002) Aura: Toward distraction-free pervasive computing. In: IEEE pervasive computing
19. Hourdin V, Tigli J-Y, Lavirotte S, Rey G, Riveill M (2008) Slca, composite services for ubiquitous computing. In: ACM t.b.p. (ed) Mobility'08: the 5th int. conf. on mobile technology, applications & systems, September 2008
20. Jeronimo M, Weast J (2003) UPnP design by example. Intel Press, May
21. Kiczales G, Lamping J, Menhdhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J (1997) Aspect-oriented programming. In: Proc. European conf. on object-oriented programming, vol 1241. Springer, Berlin Heidelberg New York, pp 220–242
22. Lagaisse B, Joosen W (2006) True and transparent distributed composition of aspect-components. In: Middleware 2006. LNCS, vol 4290. Springer, Berlin Heidelberg New York, pp 41–61, November
23. Lyytinen K, Yoo Y (2002) Introduction. Commun ACM 45(12):62–65
24. Marples D, Kriens P (2001) The open service gateway initiative: an introductory overview. In: IEEE Commun Mag. pp 110–114, December
25. Mascolo C, Hailes S, Lymberopoulos L, Picco GP, Costa P, Blair G, Okanda P, Sivaharan T, Fritsche W, Karl M, Rnai MA, Fodor K, Boulis A (2005) Survey of middleware for networked embedded systems. Technical Report D5.1
26. Michelson BM (2006) Event-driven architecture overview. Event-driven SOA is just part of the eda story. Technical report. Feb
27. Niemela E, Latvakoski J (2004) Survey of requirements and solutions for ubiquitous software. In: MUM '04: Proc. of the 3rd international conference on mobile and ubiquitous multimedia. ACM, New York, pp 71–78
28. Pessemier N, Seinturier L, Duchien L, Coupaye T (2006) A model for developing component-based and aspect-oriented systems. In: Springer (ed) 5th int. symposium on software composition. LNCS, vol 4089. pp 259–274, March
29. Rho T, Kniesel G (2004) Uniform genericity for aspect languages. Technical Report IAI-TR-2004-4. Computer Science Department III, University of Bonn, December
30. Robinson J, Wakeman I, Chalmers D (2007) Composing software services in the pervasive computing environment: Languages or APIs? J Perv Mobile Comput Apr
31. Roman M, Hess CK, Cerqueira R, Ranganathan A, Campbell RH, Nahrstedt K (2002) Gaia: a middleware infrastructure to enable active spaces. In: IEEE Pervasive Computing, pp 74–83, December
32. Roman M, Islam N (eds) (2004) Dynamically programmable and reconfigurable middleware services. LNCS, vol 3231. Springer, Berlin Heidelberg New York
33. Schlimmer J, Thelin J (2006) Devices profile for web services. schemas.xmlsoap.org/ws/2006/02/devprof, Feb
34. Seinturier L, Pessemier N, Duchien L, Coupaye T (2006) A component model engineered with components and aspects. In: CBSE. LNCS, vol 4063. Springer, Berlin Heidelberg New York, pp 139–153

35. Sivaharan T, Blair G, Friday A, Wu M, Duran-Limon H, Odanka P, Sorensen C (2004) Cooperating sentient vehicles for next generation automobiles. In: ACM MobiSys 2004 workshop on applications of mobile embedded systems (WAMES 2004), June
36. Söldner G, Kapitza R. (2007) AOCI: an aspect-oriented component infrastructure. In: WCOP 2007, twelfth int. workshop on component-oriented programming, at ECOOP 2007, July
37. Verissimo P, Cahill V, Casimiro A, Cheverst K, Friday A, Kaiser J (2002) Cortex: towards supporting autonomous and cooperating sentient entities. In: Proc. of European wireless 2002
38. Weiser M (1991) The computer for the twenty-first century. Sci Am 265(3):94–104
39. Zachariadis S, Mascolo C, Emmerich W (2006) The SATIN component system—a meta model for engineering adaptable mobile systems. IEEE Trans Softw Eng 32(11):910–927