SLCA, Composite Services for Ubiquitous Computing

Vincent Hourdin
MobileGov et I3S
2000, route des Lucioles
06901 Sophia-Antipolis France
hourdin@polytech.unice.fr

Jean-Yves Tigli
I3S (UNS - CNRS)
930 Route des Colles - BP 145
06903 Sophia-Antipolis France
tigli@polytech.unice.fr

Stéphane Lavirotte
I3S (UNS - CNRS)
930 Route des Colles - BP 145
06903 Sophia-Antipolis France
stephane.lavirotte@unice.fr

Gaëtan Rey
I3S (UNS - CNRS)
930 Route des Colles - BP 145
06903 Sophia-Antipolis France
rey@polytech.unice.fr

Michel Riveill
I3S (UNS - CNRS)
930 Route des Colles - BP 145
06903 Sophia-Antipolis France
riveill@unice.fr

ABSTRACT

Main concepts to handle in ambient computing applications are hard to integrate at the same time. After studying middlewares handling a part of the challenge, and after studying possiblities of main paradigms in name of CBSE and SOA, we present our Service Lightweight Component Architecture (SLCA) model, based on three main paradigms: Web services, enabling entities interoperability, dynamic discovery, and distribution; lightweight component assemblies to create composite Web services, allowing a high dynamicity; and finally events, giving applications reactivity and a maximal decoupling between entities, thus enabling an even higher dynamicity. This leads to conciliate both service oriented and event driven approaches in a new way to manage a graph of cooperating services in ubiquitous systems.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*domain engineering, reuse models*

Keywords

Ubiquitous computing, web services for devices, event driven architecture, software composition, service oriented architecture, component-based software engineering

1. INTRODUCTION

Ubiquitous computing, as described by Marc Weiser [29], relies on devices in the user's environment or the one of applications he's using. Indeed, with the miniaturization of computer hardware, processing units become invisible and integrated in buildings, clothes, vehicles, and so on. Applications must then evolve and offer the required dynamicity to handle context changes due to environment evolutions.

Software frameworks tackling this kind of challenges are commonly called *ubiquitous computing middlewares*. Numerous characteristics have been described by two survey works of ubiquitous computing middlewares [19, 21]. If we put apart crosscutting concerns, like security, main characteristics are: *interoperability* between entities, the environment is made of lots of devices from different hardware architectures, programming languages, or communications protocols or mediums; *reactivity*, required by context-aware applications in order to handle quickly state changes of the environment or applications, as well as interactions with the user; *dynamic adaptability*, allowing applications to handle external changes and adapt to their context of execution at run-time; and finally, ubiquitous computing applications are often made of several distributed entity, which leads to *mobility* of applications.

In this article, we study how existing approaches for ubiquitous computing middlewares are taking into account these characteristics. Then, we study component and service programmation paradigms, and what are their key points for ubiquitous computing, and how they integrate events. We propose the SLCA model to meet ubiquitous computing problems, based on an event-based service infrastructure and lightweight component assemblies creating composite services. Finally, we describe an experimentation, using an implementation of the model.

2. MIDDLEWARES FOR UBIQUITOUS COMPUTING

Ubiquitous computing is an omnipresent computing [16] in the environment, through a large number of objects and new devices in our everyday life (*everyware* [13]). They can be at the same time mobile, and integrated to the physical environment [14]. They increase application fields of computing by a growing quantity and diversity of smart devices in the physical environment of users [30]. The ubiquitous space becomes then more and more interactive and communicating.

It is therefore a computer science evolving in a physical environment made of users, devices, smart objects and which has to adapt permanently to changes of the environment. Lots of research works explore this problem. We will study ways they put forward to take into account issues described in the introduction.

Projects Gaia [23], Oxygen [1] and Aura [26] head towards user's tasks migration in a mobile environment, and thus deal with the problem of *mobility*. Gaia bases itself on *active spaces*, such as smart rooms or living environments. It uses the concept of distributed operating system, in which all inputs, outputs and processing units of a room are considered as a single computer, which allows the migration of processes or activities from an active space to another. Oxygen is more user-centered, aiming directly at computer/man interaction, with voice and vision technologies. More concretely, it defines topologies of dynamic smart networks, and embeds code in mobile devices, which allow the user to continue using services while moving. Aura defines the concept of *personal aura* of information to study the attention of the user to migrate tasks from a computer to another, basing mostly on data from sensors.
These projects can migrate tasks from an execution context to another, but they use centralized approaches to data management and environment discovery. For Oxygen, using objects limits interoperability and needs the middleware to be installed inside the entities of the environment. Adaptations are thus limited to known entities at design-time, as well as in Aura, some situations do not allow the extraction of the expected behavior, the programming is implicit.

An example taking charge of *reactivity* is the CORTEX project [8]. It uses sensors to capture the execution context of an application and react adapting applications. CORTEX proposes a novel *sentient* object model to address the emergence of a new class of application that operate independently of human control. Those sentient objects (mostly based on sensors) use the publish/subscribe paradigm to get information from the environment, and provide new ones, in large-scaled networks. The execution framework modifies dynamically the behavior of sentient objects, thus needing introspection capabilities on objects.

With the multitude of different devices types present in an ubiquitous computing environment, *heterogeneity* of discovery and communication protocols for devices become a major problem when designing applications. The Amigo project [28] proposes a middleware addressing the specific problem of *interoperability*. It integrates seamlessly devices and services relative to the application in a domestic system. Devices are dynamically discovered, independently of their type or communication protocol. On the contrary, CORTEX, with objects based on COM, makes it difficult to extend the execution environment to varied devices.
Moreover, we have seen an industrial willingness clearly displayed to extend computing standards to those new devices and objects: standards like UPnP, Jini [6], or OSGI. The first two describe service-based middleware, enabling the dynamic discovery of devices in the distributed environment. OSGi concern is to create local services-based applications, with the ability to create composite services, eventually communicating with UPnP or Jini services. However designing composite services based on a simple service composition requires to know services interfaces and referring to it in the code, which limits dynamicity.

Finally, lots of other works provide a software framework, based on distributed objects [5, 24] or components [7, 9, 31] aiming in making ubiquitous or mobile application design easier. They are dynamic, and provide adaptation capabilities for the applications they execute, but they don't always handle reactivity or interoperability or more concepts studied in the next section. Adaptation mechanisms [12] are however out of the topic of this paper, in which we define an architecture model adapted to the design of dynamic ubiquitous computing applications.

Table 1: Main characteristics of major works in ubiquitous computing

	Amigo	Aura	CORTEX	Gaia	Oxygen
Interoperability	X	x		x	
Reactivity		x	X	x	
Dynamicity,adapt.	x	x	x		x
Mobility	x	X		X	X

The table 1 summarizes the problems aimed by the different projects we have seen, noted 'X'. Some other problems are partly treated, or derive from the main choices, they are noted 'x'. All of these projects have a precise goal, and don't provide an adapted solution to all problems at the same time. Aura approaches, but aims mobile computing, without giving the needed interoperability or extensibility for ubiquitous computing. Software paradigms and technological choices underlying these different approaches are difficult to study because they are not always the main concerns [20]. We will study them more deeply, since they are the basis for meeting the challenges of ubiquitous computing, including the two main: component-based software engineering (CBSE) and service oriented architectures (SOA).

3. PARADIGMS FOR UBIQUITOUS COMPUTING

For over a decade, component and services have evolved to provide new ways of designing dynamic and heterogeneous applications. These two key points of ubiquitous computing applications place them at the base of a large number of works in the field.

3.1 Components

Systems based on components are born from an evolution of the object-oriented programming to address the problems of reuse of code in this paradigm, mostly due to the entanglement of classes. *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"* [2]. Applications are then assemblies of components, made of instances of component types (as well as class instances for objects), and bindings between those instances. Lighter and more dynamic component models (JavaBeans, .NET components) contain no reference to other components at design-time. This principle is called *late* *binding* or also *control inversion*. Components from these models are then more easily reusable, dynamic, and adaptable. Other models (EJB, CCM, Fractal) have dependencies towards other components, through their required interfaces. The file corresponding to the provided interface of a required component must indeed be known at compile-time.

Non-functional properties. Most frequently, component execution frameworks embbed a number of non-functional properties, also called *technical services* of the framework, aiming in making programming of non-functional parts of component easier, like life-cycle management or persistence. This latter makes often loose a considerable execution time, when global message passing systems are implemented, and consequently, limits the use of such component models to richer hardware architectures. Generally, the more technical services a framework offers, the less simple it is to control precisely application execution and to adapt its behaviour. Of course, some projects propose having dynamically configurable technical services, but are implemented as component grafted to components. An approach based on optional components would permit to adapt technical services like the rest of the application.

Hierarchy. Some component models, like Fractal [9] or Darwin [18] allow to define a hierarchy between components, constituing composite components with internal component assemblies. However, interfaces of composite components are not dynamic, and their adaptation needs to generate pieces of code, like in the SCORPIO [7] approach, which modifies the structure of component assemblies to create new composite components.

3.2 Services

Services oriented architectures (SOA) are based on components concepts concerning separation of concerns between different entities, and the use of functionalities through interfaces. Their main asset is the loose coupling between services, which provides a large dynamicity, reusability and autonomy. Here is the definition of the OASIS reference model: [17] : *"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"*. Services are adapted to distributed programming. They provide communications capacities between services, and arrival or vanishing notifications of service providers. We can outline that distribution is an important problem in ubiquitous computing, since devices are distributed in the environment.

Concepts of services and components are well defined but can overlap on certain points. We will study them in the next subsection, then we will focus on characteristics of services for ubiquitous computing.

3.3 Components or Services?

The border between services and components is often blurred. Some complex component models offer functionalities similar to services ones: distributed programming, dynamic discovery, interoperability... They then allow the deployment of an application on several nodes of a network. The important point we emphasize in ubiquitous computing is to create applications able to adapt to environment changes, like devices appearing, more than ordering the changes. Moreover, services have an autonomous existence, and have no information on their use in applications, which seems more appropriate when used for devices. Therefore, we base ourselves on services to communicate with various entities in the environment, devices included, and on components for their adaptation capability.

Locality. Lightweightness of component or services models depends mostly on two criteria: the quantity of non-functional properties included, and the locality. The locality determines the execution space of entities, which can be local or distributed. Gather entities in the same memory addressing space enables better communication performances between them, but also keeping a state of the local application regardless to variations of the environment. OSGi services, JavaBeans and .NET components are examples of entities executing in the same locality.

Black boxes. Their capacity to be distributed, and their greater autonomy require services to use the concept of black box entities. They limit interactions with a service (or component) to the use of their required and provided interfaces, forbidding direct access to implementation, for example for behaviour adaptation purposes. Using black boxes promotes reusability, since a component is chosen for its functionality and not for its implementation. In hierarchical models, composite components are most often white boxes, allowing modifications of their internal structure, but still without sight inside basis components [9]. We will use this approche in our works.

CBSE and SOA are two main programmation paradigms. They provide important characteristics in terms of entity decoupling and thus a great dynamicity. The hierarchy makes even easier program designs, enabling a better structuration, and black boxes enabling a better reusability. Non-functional properties automatically provided can play an important role in the ease of development, but also in applications lightweightness. We will now study evolution of services, which have lead to their diversity of use, specially in ubiquitous computing applications.

3.4 Services Evolution

The use of SOA to create applications based on physical or virtual devices proved its worth for almost ten years, with Jini (1999) and UPnP (1999), then more recently DPWS (2004). Indeed, like services, devices are autonomous, independant, and provide a set of functionalities. In ubiquitous computing, the user is moving in an environment made of numerous mobile devices, with which communicating schemes evolve. Web services for devices [10] include concepts of services and event frameworks, as well as decentralized and dynamic discovery. While they have been existing for years, a convergence is occurring with the outcome of new SOA (Advances SOA), which integrate event-driven architectures (EDA) with SOA. We will now study main characteristics of SOA, which have allowed this evolution towards services for devices.

Interoperability: two types have appeared. The first one, at the middleware level, with CORBA, interoperability of entities of a distributed application was put forward. CORBA defines a service Interface Description Language (IDL), used to generate skeleton code of services in different programming languages. The framework then marshalls messages between objects or services. So, CORBA reaches interoperability between programming languages, and hardware architectures. With the same goal, Web services were later created, using Web technologies like HTTP and XML to describe and communicate with services. These technologies do not depend on an operating system, or on a programming languages, making Web services easily interoperable, as long as TCP/IP is available. This meets perfectly the interoperability problem of ubiquitous computing entities.

The second type of interoperability is the framework's one, reached by programming languages executed by a virtual machine, like Java for the OSGi standard [3]. Its goal was to make services providers from different businesses collaborate to create something bigger than only one could do, or to deploy services on domestic gateways of their customers. Services are then portable, but collaborate only in the same memory addressing space, on only one machine, while they can be deployed on several distinct OSGi framework.

Event-based communications: interactions and communications between services have also evolved a lot. Traditionally, they are remote procedure calls, like RPC. The question of control flow passing from a service to another when the invocation is made is rarely defined. When a service invocates a method of a remote service, obviously, it should suspend its execution flow until the return is made, eventually with return values. However, asynchronous method invocations allow the calling service to continue execution, notifying it with a callback when the invocation is done.

Method invocation communications are well fitting to data processing applications or information services. In ubiquitous computing, users (human beings or services) wish to be notified by services when a change have occurred in data or in the environment. Publish/subscribe mechanisms or event notifications are born from this need of *reactivity* in applications. CORTEX uses the anonymous publish/subscribe STEAM middleware, in order to get behaviours which are independent from traditional blocking communication paradigms problems. Furthermore, events provide a great decoupling capacity to entities. Data producers and consummers are not specially designed for a specific use, and do not have information on the number of subscriptions. Moreover, with event models without message queues, like .NET component's one, emitting an event corresponds to direct control flow passing from a component to another, and then allow a better control flow handling in applications.

Decentralized dynamic discovery: older service models relie on centralized architectures, like CORBA broker or UDDI Web service registry. When they become available, service providers register to the registry so consummers can discover and use them. They will then be able to make a search on the registry matching some optional criteria, and find providers which fit to be used for the application. Some service registry propose consummers to be notified of availability changes of services, in order to react fastly possible to infrastructure changes [3, 25]. But this mechanism must not be assimilated to an event communication, since they are only sent by the framework, and have no functional vocation. Gaia and Aura use this mechanisms, via CORBA.

The mechanisms put forward for the discovery of services in an ubiquitous environment propose, generally, not to use a service registry. Lying on a fixed architecture when the application or the user are mobile, and when visible entities fluctuate fastly, is not an optimum solution, since the registry itself can be unreachable. Decentralized search solutions are then implemented, based on message multicast, like SLP used by Amigo and CORTEX.

Dynamicity et adaptation: dynamicity in SOA is intrinsic: service communicate between each other through their provided and required interfaces, but never address implementation directly. This allows several service providers to implement an interface, and have several implementations available at the same time in the environment. The arrival or departure of a service provider being notified, as we have just seen it, the service consumer can switch from one to another during execution. Adaptation of a service-based application depends of mechanisms put in place in the service orchestration or composition platform.

We have seen that services are no longer reserved for massive applications in information systems, but more and more evolve towards all types of computing, specially towards ubiquitous computing. Indeed, they are now providing event-based communications, a great interoperability handling, decentralized dynamic discovery, and still a great dynamicity. But gaps are still present for the creation of ubiquitous computing applications only based on services.

3.5 The Emergence of Multi-Paradigms for Service Composition

As we saw, components fit well to the application adaptation problem, while services fit more to heterogeneity handling and entities distribution. Interoperability on communication protocols, hardware, and programming languages can be handled by Web services. Reactivity needs an eventing mechanism to spread information to services of the environment when it becomes available, moreover favouring decoupling and better adaptability. However, creating applications based only on services can reveal complex and hardly adaptable, since discovered services, or more exactly their interface, have to be known at design-time. Also, an application based only on components can hardly handle execution context variations and communication with devices. Multi-paradigms systems have thus emerged, like Advanced SOA a.k.a. SOA 2.0, which use services and events, or SCA (Service Component Architecture) [4], dedicated to service composition. SCA defines a component and service architecture, using components to manipulate services orchestrations, and create composite services. SCA corresponds to the creation of applications for information services, and is not adapted to light targets nor to the dynamicity of pervasive environments. Indeed, the execution framewok provides a set of technical services, like life-cycle management with lazy instantiation, or a reduced transaction handling, called conversation. Moreover, the structure of a composite service is defined at compilation-time, and is not dynamic.

We thus propose the SLCA model, which has the goal to define a dynamic architecture for service composition, taking into account all problems of ubiquitous computing, taking advatage of various paradigms presented.

4. THE SLCA MODEL

SLCA (*Service Lightweight Component Architecture*) is a model of architecture for service composition based on an assembly of lightweight components. The SLCA model relies on a software and hardware execution environment evolving dynamically. We define this environment as a set of resources, which are as much software/hardware entities for which the application does not drive but adapt to the appearance and disappearance of entities.

Following the reasons mentioned in previous chapters, we propose an architecture taking into account three main paradigms:
– **Web service oriented architecture.** Ubiquitous computing applications are then a graph of Web services and composite Web services. Interoperability, distribution, and discoverability are then assured.
– **Lightweight assembly of component.** Composite Web services are created from a dynamic assembly of black box components, executing in a local container, which doesn't provides mandatory technical services. Dynamicity of applications is then provided, and reusability is increased.
– **Events.** They are taking place in the model at the services level, with Web services for devices for example, as well as in lightweight assemblies of components. Their advantages are twofold: they promote reactivity of systems, and increase decoupling between entities, and thus dynamicity of applications.

SLCA thus defines a compositional architecture model based on events, to design composite Web services, and increment the cooperation graph of services and applications. The environment consists of mobile users interacting with the world or other users with worn or mobile devices. We see them as services momentarily available in the infrastructure, in which event communications between everyday objects replace classical workflows between services.

4.1 Composition for New Services

SLCA is based on a service infrastructure using events, and dynamically discoverable in a decentralized way. They represent devices used in ubiquitous computing applications, as well as composite services created by SLCA. Interoperability is maximal, thanks to the use of Web services.

The architecture is completely dynamic. Services appear and vanish on the network reflecting the presence of devices, without knowing beforehand any service registry. It is possible to take into account these changes in applications without knowing what devices shall be met at design-time. Indeed, from the XML description of Web services, it is possible to generate automatically *proxy components* which will communicate with services of the environment.

The service infrastructure of a SLCA architecture is thus used for the discovery and the communication with devices and composite services distributed in the environment. Applications are designed by service composition *mashup*, assembling lightweight components. A composite service then contains an assembly of components, in a container. Proxy components to other Web services are thus instantiated in the container of a composite service, and create applications from services present in the environment. Moreover, since the container is inside a composite service, it also has a service interface, and functionalities created by the component assembly can be seen by other composite services. Then, a composite service can create an application communicating with another composite service. The concept of hierarchy is then introduced, through the service layer.

A composite service (container) provides two service interfaces (Fig. 1). 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.

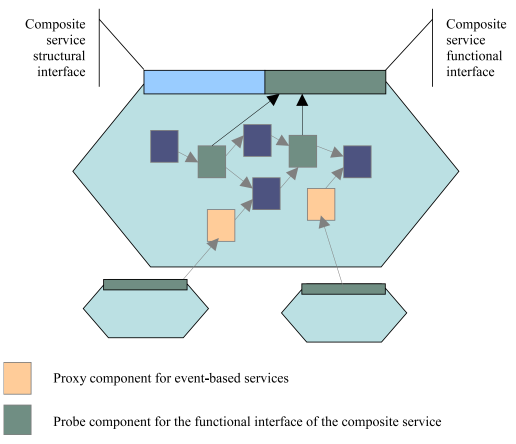
Composite service structural interface
Composite service functional interface
Proxy component for event-based services
Probe component for the functional interface of the composite service

Figure 1: Composite event-based Web service

The dynamic functional interface 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. 2): *sinks*, which add 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 structural 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, a client, which can be another 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.

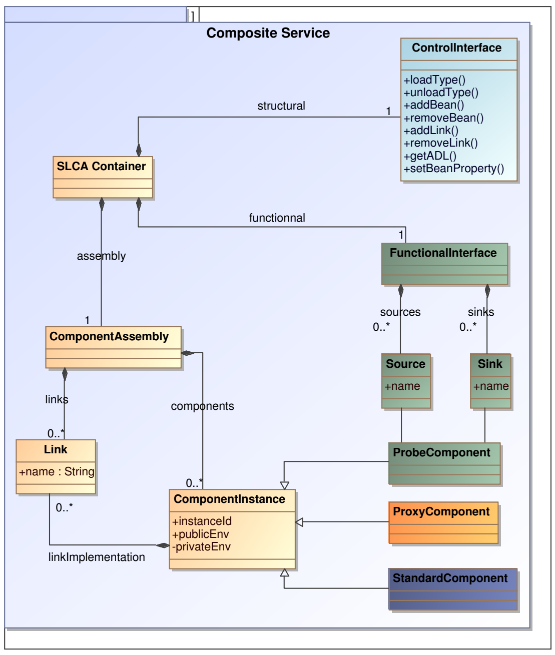

Figure 2: SLCA Meta-model: interfaces of composite services

Colors of the UML diagram of Fig. 2 match those of Fig. 1 to make the reading easier. Proxy components allow services of the environment to be used in the composite service, while probe components allow new services to be added to the environment, which can eventually be used by other composite services.

4.2 Lightweight Component Assemblies

SLCA uses lightweight components to design composite Web services. As we saw, a composite service encapsulates the SLCA container which contains a dynamic lightweight component assembly. The component model LCA (*Lightweight Component Architecture*) is a model derived from Beans [27], adapted to other programming languages, with concepts of input and 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 processus, 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 irrelevant technical service in this local environment. Their memory footprint is then reduced and they are instantiable and destructible quickly. 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.

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 done by adding components in the assembly, guaranteeing dynamicity 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 an 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.

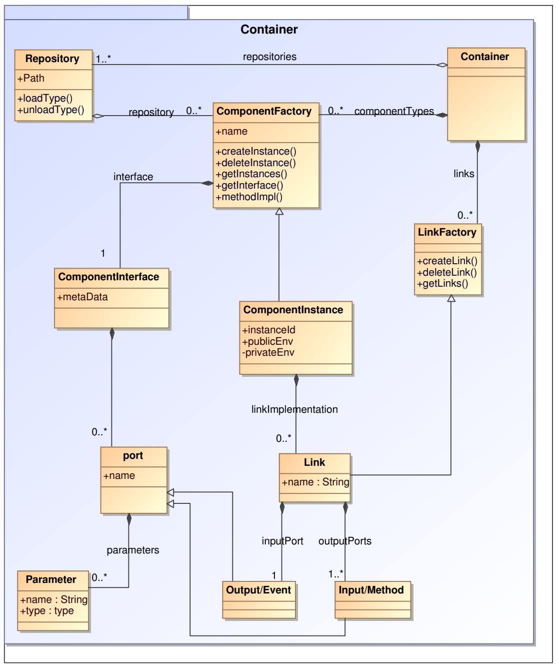

Figure 3: LCA meta-model: lightweight components

Component types which can be instantiated in a container depend on the list describing and implementing them in a repository. This list is also modifiable at run-time. When a service is discovered, it can be immediately loaded and instantiated in the component assembly to contribute to functionalities of the composite service.

Component assemblies inside composite services can create applications or new functionalities from services of the environment (Fig. 4). Unlike the service infrastructure, they are executed locally, and their logic is not disturbed by appearing or vanishing of services. When a service used by a composite service becomes unavailable, there are two possible reactions: either the state of the assembly is unmodified until a replacing compatible service is found, either its proxy component can be removed and the composite service can be adapted. In the first case, the locality of the assembly of component makes it able to save its state. Of course, adaptation mechanisms [12] should be applied to take into account new requests to the composite service, which may or may not be able to completely satisfy a request.

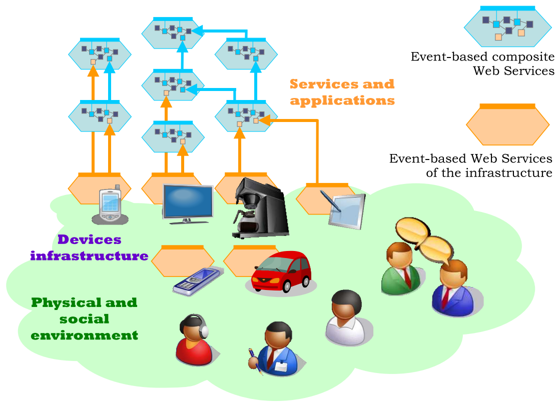
Event-based composite Web Services
Services and applications
Event-based Web Services of the infrastructure
Devices infrastructure
Physical and social environment

Figure 4: Graph of event-based Web services

5. EXPERIMENTATION

As we have seen, SLCA can be used to design composite services graphs. We can then imagine ubiquitous or mobile computing applications, using services of the environment to create new services and adding new functionalities. We will now study an example which adds a basic authentication functionality to control access to an interaction service executing on a tablet PC. Composite service then created will allow an identified user to access new functionalities, like the device localization.

The control interface of the composite service allows it to be managed by several remote tools, like the UPnP wizard controller. It is a UPnP control point receiving advertisements from services of the environment, which generates proxy components and instantiate them into a bound container. The service (T) of the device to which we want to add functionalities, the authentification service (A) for users, and the localization service (L) are thus detected and instantiated in our composite service (S).

Then, the user has to create, either by hand with a visual composition of components, or automatically, with a scripting controller for example, the service interface of the proxy component, using probe components. This interface shall be the same than the interface of T, to which we add new localization functionalities and an authentication method. For each provided interface of T (each method), a sink probe component is added, and for each required interface (each event), a source probe is added in the internal component assembly of our composite service S.

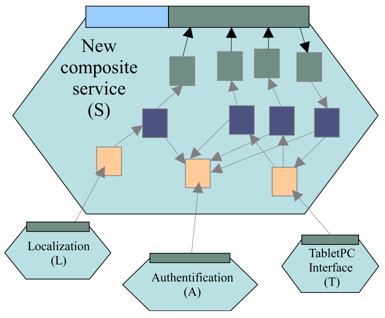

Figure 5: Graph of WComp assembly

The last step consists in creating links between probe components and the proxy components. A sub-assembly of components will actually be added to create the new functionality that we add to T. In this example, we add a simple identification method with a login/password couple, which is sent to A for checking. If the user is authentified, a session identifier will be generated and returned, and have to be used for all future invocations. The localization functionality of the device will be available to users authorized, through invocations to service L. Thus, the access to the tablet PC, and to its localization, will be protected by an authentification mechanism, and a new composite service is provided, which can be used in other applications. However, if no network-based solution to hide T is implemented, it will still be possible to reach it, and use it without localization. Solutions like a VPN or executing the composite service on the same computer are possible.

This example is implemented with a projection of the SLCA model called SharpWComp 2.0, which is a copyrighted software in France, used and developped in two programs of the French National Research Agency (ANR). The first explored auto-adaptation of software applications to assist people with disabilities. It is creating interaction devices, so they are adapted to profiles of reduced mobility people, and self-adapting to variations of the profile in time. The second project adds contracts inside composite services, like bounding the exectution time of a service, or catching execution points of an application to add some actions.

6. CONCLUSION

We have presented our SLCA model for the design of ubiquitous computing applications. It is based on an event-based Web service composition, providing discoverability and interoperability of devices in the environment, made with internal lightweight component assembly, for their dynamicity. Such an architecture allows designing dynamic applications, according to the principle of classic modularity of reuse of functionalities of a service or a component by another, especially when loosely-coupled bindings such as events are implemented.

Other works focus on the need of a crosscutting modularity, easing incremental evolution of applications, and implementing replicable modification schemes on a large number of services. If we consider the experimentation, we understand that managing the availability of services of the environment can be a difficult and transversal task. The Aspect paradigm well known in the object oriented programming field [15], is now widely applied to other architectural paradigms (AO4BPEL for orchestrations [11], FAC for components [22], and so on). Our SLCA model takes the same direction, evolving towards an approach using the concept of Aspect of Assembly (AA). It allows a dynamic evolution and adaptation of the graph of services for ubiquitous computing systems.

7. REFERENCES

[1] MIT Oxygen project. http://oxygen.lcs.mit.edu/.

[2] WCOP'96: Summary of the WCOP'96 workshop in ECOOP'96, 1996.

[3] OSGi Alliance. http://www.osgi.org/, 2002.

[4] Service Component Architecture specification. http://www.osoa.org/, 2006.

[5] M. Anastasopoulos, H. Klus, J. Koch, D. Niebuhr, and E. Werkman. 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)*, Sept. 2006.

[6] K. Arnold, editor. *The JINI Specifications, Second Edition*. Addison-Wesley Professional, 2000.

[7] G. Bastide, A. Seriai, and M. Oussalah. Adapting software components by structure fragmentation. In *Proc. of ACM Symp. on Applied Computing*, 2006.

[8] G. Biegel and V. Cahill. A framework for developing mobile, context-aware applications. In *Second IEEE Int. Conf. on Pervasive Computing and Communications (PerCom'04)*, page 361, 2004.

[9] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Workshop on Component-Oriented Programming (WCOP) at ECOOP'02*, June 2002.

[10] N. Bussière, D. Cheung-Foo-Wo, V. Hourdin, S. Lavirotte, M. Riveill, and J.-Y. Tigli. Optimized contextual discovery of web services for devices. In *IEEE Int. Workshop on Context Modeling and Management for Smart Environments*, Oct. 2007.

[11] A. Charfi, B. Schmeling, A. Heizenreder, and M. Mezini. Reliable, Secure, and Transacted Web Service Compositions with AO4BPEL. In *Proc. of the 4th IEEE Euro. Conf. on Web Services*, Dec. 2006.

[12] D. Cheung-Foo-Wo, J.-Y. Tigli, S. Lavirotte, and M. Riveill. 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, USA, Nov. 2007.

[13] A. Greenfield. Everyware: the dawning age of ubiquitous computing. *New Riders*, pages 12–12, 2006.

[14] ITU Internet Reports 2005. The internet of things.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. European Conf. on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[16] K. Lyytinen and Y. Yoo. Issues and challenges in ubiquitous computing. *Communications of the ACM*, 45(12):62–65, 2002.

[17] M. MacKenzie, K. Laskey, F. McCabe, P. Brown, and R. Metz. Reference model for service oriented architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS, Feb. 2006.

[18] J. Magee, N. Dulay, S. Eisenbach, , and J. Kramer. Specifying distributed software architectures. In *5th European Software Engineering Conf. (ESEC'95)*, pages 137–153, Sept. 1995.

[19] C. Mascolo, S. Hailes, L. Lymberopoulos, G. P. Picco, P. Costa, G. Blair, P. Okanda, T. Sivaharan, W. Fritsche, M. Karl, M. A. Rnai, K. Fodor, and A. Boulis. Survey of middleware for networked embedded systems. Technical Report D5.1, 2005.

[20] A. Messer, H. Song, D. Cheng, and S. Gibbs. A classification of pervasive system software. In *Common Models and Patterns for Pervasive Computing Workshop, at the 5th Int. Conf. on Pervasive Computing (Pervasive 2007)*, May 2007.

[21] E. Niemela and J. Latvakoski. Survey of requirements and solutions for ubiquitous software. In *MUM '04: Proc. of the 3rd Int. Conf. on Mobile and ubiquitous multimedia*, pages 71–78, New York, USA, 2004. ACM.

[22] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A model for developing component-based and aspect-oriented systems. In Springer, editor, *5th Int. Symp. on Software Composition*, volume 4089 of *LNCS*, pages 259–274, Mar. 2006.

[23] M. Roman, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. In *IEEE Pervasive Computing*, pages 74–83, Dec. 2002.

[24] M. Roman and N. Islam, editors. *Dynamically Programmable and Reconfigurable Middleware Services*, volume 3231 of *LNCS*. Springer, 2004.

[25] J. Schlimmer and J. Thelin. Devices Profile for Web Services. schemas.xmlsoap.org/ws/2006/02/devprof, Feb. 2006.

[26] J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. *3rd Working IEEE/IFIP Conf. on Software Architecture*, 2002.

[27] SUN Microsystems. JavaBeans 1.01 specification. http://java.sun.com/products/javabeans/, 1997.

[28] M. Vallée, F. Ramparany, and L. Vercouter. Flexible composition of smart device services. In *The 2005 Int. Conf. on Pervasive Systems and Computing(PSC-05)*, June 2005.

[29] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991.

[30] Wireless World Research Forum. Book of visions. http://www.wireless-world-research.org/.

[31] S. Zachariadis, C. Mascolo, and W. Emmerich. The SATIN component system - a meta model for engineering adaptable mobile systems. *IEEE Trans. on Softw. Eng.*, 32(11):910–927, Nov. 2006.