Cascaded Aspects of Assembly for Ubiquitous Computing

Nicolas FERRY\textsuperscript{1,2}, Stéphane LAVIROTTE\textsuperscript{1}, Jean-Yves TIGLI\textsuperscript{1}, Gaëtan REY\textsuperscript{1} and Michel RIVEILL\textsuperscript{1}

\textsuperscript{1}University of Nice – Sophia Antipolis / CNRS, I3S Laboratory, Sophia Antipolis France 
\{ferry,lavirott,tigli,rey,riveill\}@polytech.unice.fr

\textsuperscript{2}CSTB (Centre Scientifique et technique du bâtiment), Sophia Antipolis France

Abstract: Ubiquitous systems are characterized by using devices and objects of everyday life. The software infrastructure of such systems appears dynamically populated by functionalities of those devices. Because environment’s nature is highly variable, even the corresponding software infrastructure, ubiquitous systems have to handle those variations. It implies that a designer cannot predict a priori the environment in which its application will be executed. So, the approaches used should not only address specific cases of context. AOP provides a mean to encapsulate adaptations into aspects. Such an encapsulation improves the reusability of adaptation mechanisms. In this context our previous works called Aspect of Assembly (AA) was based on AOP to propose mechanisms for self-adaptation of ubiquitous applications as an assembly of components. But they present few limitations in terms of reusability and to manage the dynamic variability of the environment. The paper presents Cascaded Aspects of Assembly as an extended model of AA to address these limitations. They consist in chaining AA's weaving as cascades.

Key words: Ubiquitous computing, software composition, self-adaptation, aspect-oriented programming, context awareness

INTRODUCTION

Ubiquitous computing relies on computers present everywhere, at any times and in any things. Indeed with recent years advance in mobile communication technologies and the miniaturization of computer hardware, processing units are becoming invisible and a part of the environment. Ubiquitous systems are characterized by using devices and objects of everyday life, and can take many forms. The technological heterogeneity of those devices and objects is still a big challenge to overcome in order to enable them to interact with the system. Service oriented architecture (SOA) is a technological solution to tackle the heterogeneity of these devices, and especially one of its evolutions: Web Service Oriented Architecture for Device (WSOAD) [BUS 07]. Most of the time, these services, provided by the devices, cannot be modified by the system itself.

In order to add new functionalities or to create new applications, a ubiquitous system has to enable interactions between these services; it must be able to compose them. The software infrastructure of the system appears dynamically, populated by functionalities of those devices. Because environment’s nature is highly variable, even the corresponding software infrastructure, ubiquitous systems have to handle those variations and offer such
dynamicity. These systems have to be able to adapt to their context; they must be context-aware. Because those changes occur frequently, managing these adaptations may lead to an administrative overhead. The system must adapt itself to these changes; it must provide mechanisms for self-adaptation.

Context-awareness, including application adaptation [BOT 07], is a crosscutting concern [DAV 06]. Mechanisms and toolkits for context-awareness must then be proposed by middleware for ubiquitous computing [SAT 96, BOT 07]. Currently most of them [BOT 07, SAC 05] are interested in restricted environments such as smart conferences rooms, smart rooms or smart houses. Thus, one challenge offered to middleware for ubiquitous computing [HEN 01] is to design systems that can scale in large environments such as open or public spaces, hospitals, airports [MEH 08], etc.

For example, here is a piece of scenario involving simple behaviors that we will use as a simple example in the paper. This scenario takes place in the context of a hospital. However, many other behaviors like these may be added to the system for the whole hospital. The hospital, for ecological reasons, decided to implement a policy to reduce its energy consumption. Eve is a nurse at the hospital, when she enters a room the system would enable the switch to open the shutters rather than turning on the lights when the
outside brightness is sufficient. She is entering in the room 500, newly assigned to an old woman who is visually impaired. The old woman’s profile is a priority when entering a room, so in such a case artificial lighting is always used.

This system’s ability to scale implies some major constraints as the system’s reactivity and its overall performance or its ease of design, reusability and the flexibility of mechanisms for context-awareness. Such systems have to manage a huge number of devices or of data. So, their design becomes a real problem [PEN 10]. Indeed, the multitude of data and devices involved in a ubiquitous system and their mobility implies that the designer cannot predict a priori the environment in which its application will be executed. So, the used approaches should be abstract and reusable and should not only address specific cases of context. This unpredictability also implies independence between application and adaptation entities, because these entities may evolve with their surrounding environment.

Aspect Oriented Programming (AOP) is a way to adapt at runtime an application while encapsulating the mechanism of adaptation into aspects [ZAM 04]. Such an encapsulation improves the reusability of adaptation mechanisms. In this context our previous works called Aspect of Assembly (AA) was based on AOP to propose mechanisms for self-adaptation of ubiquitous applications as an assembly of components. But as we shall see in this paper, they present some strong limitations in terms of reusability and to manage the unpredictable dynamic variability of the software infrastructure. The paper presents Cascaded Aspects of Assembly as an extended model of Aspect of Assembly to address these limitations. They consist in chaining AAs weaving as cascades.

The remainder of this paper is organized as follows: section 2 introduces the challenges for ubiquitous computing in existing solutions. Section 3 describes our previous works to address these challenges; we will present in details the AA model, the associated DSL and the weaving process. Finally we present some limitations of the model, especially in terms of reusability and variability. In section 4, we present the Cascaded Aspects of AA principle and how they propose to address these issues. The approach will be applied to the scenario presented in the introduction. Finally, we will see how cascaded AAs maintain good overall system performance.

1. Challenges for ubiquitous computing in existing solutions

We have seen that a ubiquitous application is a composition of services for devices. Such an application, and then such a composition, must be editable at runtime. As explained in [BRO 07]: “The ability to seamless composes services from various devices in a more or less ad-hoc manner is a frequently emphasized feature of ubiquitous computing”. If services propose to address the issue of interoperability, components offer a high dynamicity and reusability. They are created into containers, thanks to some components factories. Containers provide non-functional properties to components. Components factories define the type of component to be instantiated. They can be instantiated and manipulated easily by a developer. Conversely, services on ubiquitous users’ devices are fixed and undergone in the point of view of the ubiquitous system. Only pure software services can be instantiated by developers. Therefore, we base ourselves on services (pure software ones and services linked to devices) to communicate with various entities of the environment. Accordingly, our applications are in the form of components assemblies orchestrating web services for devices. Various components models are based on such an approach as SCA [BLO 05] or SLCA [HOU 08]. The adaptation of a ubiquitous application will consist in adapting a component assembly.

We have seen that AOP allows encapsulating adaptation into aspects [ZAM 04]. Adaptations are then reusable and exchangeable. Dynamic aspects provide a means to add / remove, select / unselect aspects and then adaptations at runtime. These abilities and the modularity offered by aspects are an approach to address the variability of ubiquitous system. They will allow, according to the underlying software infrastructure, to choose or exchange adaptations. The abstraction offered by pointcuts allows the duplication of different instances of the same aspect in different places. This corresponds, for example, to add the same kind of functionality to same kind of devices’ services. However, several challenges are open to the implementation of aspects in the context of ubiquitous computing that are related to challenges defined in [HEN 01]. In this paper we will focus on the following three challenges.

The first challenge, for the adaptation mechanism, is to respect the dynamics of the software infrastructure of an application. An application is based on services for devices located in its infrastructure. So, the application is a service consumer. It must evolve according to appearance / disappearance of these services. Especially an application should not attempt to use a service that no longer exists in its infrastructure. To respect this dynamic, it is necessary to react to these changes in the infrastructure [HIR 06]. This, while offering response times short enough to adapt the application with a frequency as close as possible to that of the infrastructure’s evolutions. We will present in section 5 some results on experiments on the response time of our approach.

A second challenge is the definition of a language of pointcut and advice that provides the right abstractions for ubiquitous systems and to adapt component assemblies. This challenge comes from both the large number of entities involved in the infrastructure and its variability. To make easier the
development of adaptations and systems themselves, we have seen that the adaptation mechanism should provide a sufficient level of abstraction to be highly reusable. Using AOP, this abstraction must be present in the language of advices but also for pointcuts. Because the software infrastructure of a ubiquitous system may involved a same type of devices several times (many brightness sensors for instance), the abstraction provided by pointcuts plays a major role in the reuse and duplication of aspects. But most approaches are based on languages too generic and not enough declarative to adapt component assembly. They remain, despite a high level of abstraction, complex to use. However some approaches exist as FScript [DAV 09] or ISL4WComp [CHE 09].

Another constraint is related to the management of the variability at runtime of ubiquitous systems. This is due to the high variability of applications and of their infrastructure, as well as their dynamicity. A designer cannot predict all the possible configuration of a system because of the unpredictability of its infrastructure. Despite AOP allows to encapsulate adaptation into aspects and then allows to reuse and exchange adaptations, it appears that all configurations of a system cannot be considered by his or her designers. Moreover, it often appears that an aspect for a concern is setting up mechanisms that are also implemented by aspects for other concerns. It is often the case in the field of autonomic and ubiquitous computing. As shown in [FER 10], most of middleware for context-awareness are based on a vertical architecture. It consists in a functional decomposition of the classical approach for context-awareness. This approach can basically be divided into three major steps: (1) perception to gather some contextual information, (2) decision to choose a reaction and (3) the reaction implementation. Most of the concerns encapsulated in aspects for ubiquitous systems are also implementing these three steps. Also, it is common that various concerns use a same mechanism of perception. In our scenario, the two concerns of energy consumption and assistance to the person shall interact on several aspects and to chain the application of aspects for a same concern, for instance using aspects. Some approaches as [CHA 04], provide such a possibility. But, as a designer cannot predict all possible configurations of the system, he cannot either predict all combinations of aspect. So the chaining of aspects must not be defined explicitly. Moreover, for each aspect of each combination the weaver must check if it can be applied according to the underlying infrastructure. It also appears that the order in which aspects are applied must not be important. Indeed, because of the unpredictability of resources of ambient systems and of users need, all the concerns cannot be scheduled in advance. Therefore, the order in which different concerns must be implemented cannot be calculated in advance. Moreover, calculating all possible combinations to choose the most appropriate [MUN 08] would be too expensive in time and harmful to the system reactivity.

Some previous works on Aspect of Assembly proposed to address the challenges of defining a language with the right abstraction level and of respect of the dynamics of the software infrastructure of an application. However they offered limited responses to the problematic of handling the variability of the runtime infrastructure and application.

2. Aspect of Assembly

Aspect of Assembly (AA) is the result of our previous works [TIG 09]. It’s a model based on AOP for adaptation schemas and of a weaving process with logical merging. They allow structural reconfiguration of components assemblies at runtime, keeping black-box property of components. Modifications include adding components and bindings between them. In traditional AOP [KIC 97], aspects are composed of pointcuts and advices. Pointcuts point out “where” to inject the code to weave while advices describe the code to be injected. In the context of AA these concepts are still valid but with some deviations. Here an advice describes a structural reconfiguration of a components’ assembly, while a pointcut identifies components’ ports on which changes will take place. Thus, joinpoints are all entities of the assembly that structurally represent the application, on which changes will take place: components and their ports. The result of the weaving of AA is a set of basic instructions such as adding a link, removing a component ... Thus, our approach can be applied to several types of dynamic components platforms as SCA [BLO 05] or SLCA [HOU 08], for example.

Pointcuts are defined as sets of filters on joinpoints meta data (port ID or name). Those filters construct lists of parameters satisfying the list of variables of the associated advice. They are the set of components ports or components on which the advice will be woven. For each generated list (including a joinpoint for each variable), the advice is duplicated, and the variables are syntactically replaced in the advice to match the base assembly joinpoints. Thanks to pointcuts, AA are applied on components assemblies which are not necessarily known a priori. For our experiments, we choose for convenience to express filters using some simple pattern matching as regular expressions on components or ports names.

Advice is not a piece of code which will be woven into components. Because an AA modifies the structure of components assemblies, advice consists in a set of basic structural changes: adding a component or a connection between ports to the base assembly. The removal of components or connections is performed if an AA is withdrawn. Advises are specified in a DSL called ISL4WComp using interaction specification defined in [TIG 09].

2.1. ISL4WComp

ISL4Wcomp is based on the ISL Interaction
Specification Language that describes patterns of interactions between independent objects [BER 01]. ISL4Wcomp adapts these specifications to consider interactions based on messages or events between components. This language can be composed: multiple instances of advice can be composed and merged into a single assembly combining their respective behavior. The merging mechanism, embedded into the weaver, ensures the property of symmetry (associativity, commutativity and idempotency) [CHE 09] of the AA weaving operation. This means that, the order in which aspects are woven is not important.

Advices written using ISL4Wcomp are based on three types of rules: (1) the addition of black-box components, (2) rewriting links between components of the assembly and (3) the creation of new connections. A rule cannot remove a component. Rewriting involves components ports, it consists in: forwarding an input port or redirecting a message (output port). These rules are identified thanks to two key words: “&” for black-box components instantiation and “->” for rewriting and creating links. An advice describes a set of adaptation rules to be applied on variable components defined in pointcut. Thanks to some language operators as call and delegate, a designer can control how the composition of instances of advice will be done. These keywords, associated to sequence and parallelism operators are similar to classical AOP keywords: before, around and after. However, they can all be dynamically merged, this merging is symmetrical. Table 1 presents ISL4Wcomp keywords and operators.

<table>
<thead>
<tr>
<th>Keywords / Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port types</td>
<td>comp.port “.” is to separate the name of an instance of component from the name of a port. It describes a provided port. comp.^port “^” at the beginning of a port name describes a required port.</td>
</tr>
<tr>
<td>Rules for structural adaptations</td>
<td>comp : type To create a black-box component</td>
</tr>
<tr>
<td></td>
<td>comp : type(prop=val) To create a black-box component and to initialize properties</td>
</tr>
<tr>
<td></td>
<td>required_port -&gt; (required_port) To create a link between two ports. The keyword -&gt; separates the right part of the rule from its left part</td>
</tr>
<tr>
<td></td>
<td>provided_port -&gt; (required_port) To rewrite an existing link by changing the destination port</td>
</tr>
<tr>
<td>Operators (symmetry)</td>
<td>… ; … Describes the sequence</td>
</tr>
</tbody>
</table>

Figure 1 presents an example of advice based on ISL4WComp. First we define an independent adaptation schema for a demotic application. It aims to link a switch to any kind of light in order to control the light using the switch. Both light and switch proxy components are generated into the components assembly. The advice presented below proposes to adapt this behavior by adding an energy saving concern as described in the scenario. To be applied it requires a brightness sensor, so that the user can turn on the light only when the brightness is under a defined threshold. Moreover, the new assembly sends a message to give a feedback to the user when it tries to switch on the light while the brightness is too high.

We will now study the code of the advice. It is called brightness_light. The three variables light, brightness, switch associated to the name of the advice describe the joinpoints, identified thanks to the pointcut matching, that will be used in the advice. This AA highlights the three types of rules previously defined. At lines 3, 4 and 5 some black-box components are added. The threshold component is instantiated with the property threshold up to 10. A property is a public variable from a component available through its interface. Lines 7,8,9 define an input port rewriting rule. All links connected to the input port (method) SetState will be rewritten. This rule involves the operator if, this mean that a if component will be instantiated. The condition to be evaluated by this component comes from a call on the method IsReached from the threshold black-box component. If the condition is true, then the feedback message is sent, else the rewritten link is done. Rules defined at lines 10 and 11, 12 allow to define two new connections. As an example, the first rule links the output EmitStringValue from the black-box component Emitter to the input method set text from the black-box component t1.

Table 1 ISL4WComp operators and keywords

<table>
<thead>
<tr>
<th>property, conflicts resolution</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>If(condition)</td>
<td>If(condition) is evaluated by a blackbox component</td>
<td></td>
</tr>
<tr>
<td>[...]else[...]</td>
<td>Condition is unique in case of conflict</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>Nothing to do</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>Allow to reuse the left part of a rule in a rewriting rule</td>
<td></td>
</tr>
<tr>
<td>delegate</td>
<td>Allow to specify that an interaction is unique in case of conflict</td>
<td></td>
</tr>
</tbody>
</table>

1 schema brightness_light ( light, brightness, switch ) :
2 2
3 3 Emitter : 'BasicBeans.PrimitiveValueEmitter'
4 threshold : 'BasicBeans.Threshold' ( threshold = 10 )
5 5 f1,f2 : 'System.Windows.Forms.TextBox'
6 6
7 7 light.SetState -> {
8 8 if (threshold.IsReached) (Emitter.FireValueEvent)
2.2. The weaving process

The weaving process can be decomposed into three steps (Figure 2). First, pointcut matching is a function that has a set of components from the base assembly, and pointcuts, from a set of selected AA, as input. Its goal is to find the joinpoints on which advice will be woven. Based on pointcut matching results, an advice can be woven several times during the same weaving process. The second step is called the advice factory. It generates instances of advices, replacing variable components in advices of selected aspects by joinpoints obtained during the first step. Instances of advices describe modifications to be woven in the base assembly of components. Finally, the composition engine merges all instances of advices with the initial assembly. It generates a single instance of advice that will be woven as the final assembly. This merging mechanism is able to resolve conflicts between various instances of advices [TIG 09]. Two instances of advice are conflicting (interacting) when they involve at least a same joinpoint. The merging mechanism considers the rules from both advices and merges them in pairs. Some operators in the advice language can define how conflicts should be managed. By default, when both interactions are conflicting, the parallelism operator (keyword par) is introduced between the two interactions. The rules and operators merging matrix is presented in [CHE 09]. Since these operators are symmetric, the order in which the rules are merged is not important and therefore the order in which instances AAs are woven either. Thus we can build applications by composing several aspects of assembly at the same time, at runtime.

2.3. Adaptations Triggering

In order to be more reactive, the weaver can be triggered in two ways. The first is user-driven: user changes the set of AA given as input to the weaver, by selecting/unselecting or adding/removing aspects of assembly at runtime. When the set of AAs is modified, the weaver is triggered; leading to adaptation if an added AA can be applied or if an AA has been removed. The second way of triggering adaptation is driven by infrastructure changes. When a new device appears or disappears in the environment, a new component to interact with the device is dynamically instantiated in or removed from the assembly. The adaptation process is triggered and only AAs that can be woven according to newly available components are applied. Weaving is not triggered depending on the control flow of the application, but according to independent flows pursuing their own dynamics, those of the infrastructure and of users.

2.4. Limitations

We have seen that the AA model benefits, in term of reusability, of AOP encapsulation. However, this approach still suffers some limitations in term of reusability and variability management.

Let illustrate this thanks to a simple example. In the field of ubiquitous computing, context perception mechanism, which aim is to gather some observables on the application’s environment, often involves a large number of sensors. Raw data obtained from sensors often needs to be aggregated or refined in order to be interpreted. So often, several sensors of the same type interact with one entity, one component to filter the raw data from the various sensors. These sensors can provide different interfaces. But both sensors are not necessarily present in the infrastructure. This example cannot be implemented easily using AAs. Indeed, since the weaving operation is symmetrical, an AA cannot reuse the components instantiated by others AA. Otherwise the order in which AA are woven becomes important and according to this order the result of a weaving cycle is no longer necessarily the same (loss of the commutativity property). This implies that, in this type of configuration, pointcuts are less generic. We must write an AA for each configuration; this means that pointcuts must precisely identify the components on which the AA will be applied. Otherwise, the average black-box component will be instantiated as many times as there are duplications of AA. And thus, we must write several pointcuts as \( brightness1=brightness1; brightness2= brightness2 \) and then several AA instead of a pointcut as \( brightness=brightness\). So the number of AA to write is potentially large when one wishes to permit a large number of configurations for the system in order to improve its capacity of self-adaptation. Moreover, it implies that the designer must select the right AA for the right infrastructure to avoid the duplication of the same part of the system. This limits the way AA can
manage the dynamic variability of the system and the unpredictability of these variations.

3. Increasing the number of reconfiguration using less AA: Cascaded AAs

We propose a mechanism to reduce the number of AA to write, enabling to distinguish functional production of AA and to compose them dynamically at runtime. The aim is to increase the combinatory of adaptable configurations starting from a same set of AA and to combine AA dynamically to manage the variability of the system.

3.1. Functional decomposition

In the field of ubiquitous computing as in autonomic computing, systems often rely on architectures based on a functional decomposition. This decomposition is usually based on key functionalities [COU 05]. The first functionality of sensing is to gather contextual information also called observables. These observables are then transformed as symbolic observables describing the state of the environment, for example using ontologies. Hence, as a result of contextual information collection, data will be used during a decision stage, also called situation identification stage. It will produce an action plan which will be implemented by the adaptation mechanism also called the control mechanism.

So when a designer wishes to create a context-aware application or to add a context-awareness mechanism in an application using AA, these three functionalities will be implemented in the application. Introducing this decomposition provides facilities for the reuse of parts of the mechanism. It also improves its evolving facilities. This means that it will be easier to identify which part of the system remove or swap according to the context. For example, in our scenario, according to the rooms visited by the nurse, the mechanisms to monitor the brightness can change; it may be a sensor into the room or a weather service of the hospital for not equipped rooms. To make such changes, we must clearly identify the functional production of AA in order to know what AA need to be exchanged and not to group all these productions in a single AA. However, as it has been mentioned in the limitations of AA, they cannot reuse the components instantiated by others AA. Thus, the result of an AA adding a mechanism for context perception cannot be reused by an AA adding a mechanism for decision. Those two mechanisms must be grouped in a single AA. So to change the perception part, we should rewrite a new AA and not only the part of the system that has evolved.

Therefore, we propose to group the AA according to the functionality they intend to weave and to dedicate a functional group to a weaving cycle. Classically, for a ubiquitous application, we will create three groups and therefore three cycles of weaving: a cycle for a group of AA that produces the perception mechanism, a cycle for a group of AA that produces the decision mechanism and finally a cycle for a group of AA that produces the action mechanism. The cycles are ordered in such a way that the result of a weaving cycle will be the base assembly for the next cycle of weaving. Thus, a component instantiated in a weaving cycle can be reused by AA from next weaving cycles through their pointcut and thus in their advices. Then, AA may be triggered in a cascaded way, i.e. the application of AA for a feature from a concern in a cycle \( n-1 \) may be the origin of the weaving of an AA in a cycle \( n \). Thus, the cycle number 0 is always woven on an initial assembly blank of any AA. A weaving cycle \( n \) is always woven on the result of the weaving cycle \( n-1 \). The cascaded weaving of AA proceeds as follows: AA for the first cycle are woven, on the resulting assembly, AA for the second are woven and so on until the last cycle. Then, the whole process will be restarted, beginning with the cycle number 0.

Thus a reconfiguration for a particular concern that was previously implemented using an AA can be decomposed into several AA according to the produced functionalities. Each AA for a feature is woven with other AA for the same functionality. So between several AA for a same functionality (i.e. a same weaving cycle) the symmetry property of the weaving operation is preserved and conflicts are resolved. So that the order in which AA from a same group are woven is not important. Because the order in which groups of AA are woven is fixed, and because the order in which AA are posted in a group is not important, for a given base assembly and a set of AA, the result is unique and calculable. As for classical AA, this offers a significant advantage in terms of performance of the approach since it is not necessary to calculate many configurations according to the order in which they are woven and then to choose the more suitable.

In the weaver, changes consist in grouping AA according to their functional productions and to chain weaving cycles according to these groups. Finally, when decomposed, an AA is defined as an ordered set of group of AA that is deployed in various repositories according to their functional production (a repository for each weaving cycle). Another change consists in the maintenance of what is the initial assembly for the various cycles. This information is especially important during the unwrapping / weaving process in order to modify only what has changed in the assembly at each cycle.

3.1.1. Cascaded AA applied to the scenario

For example, in our scenario, we can identify two behaviors: (1) assistance to the person and (2) energy saving. The different AA, which we will present in this section are distributed as shown in Figure 3 in the various weaving cycles. The component framework used is WComp, which is an implementation of the SLCA model [HOU 08]. This platform allows the dynamic orchestration of services using components assemblies. WComp allows the dynamic
instance/proxy acting as a web service in their environment. These appearances/disappearances may trigger AA weaving.

![Decision module (AADecision)](image)

Figure 3: Cascaded AA

At first, we will describe the behavior of assistance to the person which has priority. This behavior will involve three rounds of weaving. Initially, we will write a first AA (Fig. 4) for a first weaving cycle. This is the decision-making part of the system. It will be the link between the perception part and the action part of the system. Therefore, it will be heavily reused by other parts of the behavior. We could have deployed the AA for the perception mechanism first and AA for decision in the second cycle so that the decision part would be deployed according to the perception mechanism. But, for this scenario, it would have meant rewriting many times the pointcuts part of the AADecision aspect according to the perception mechanisms required for its application. AADecision aims to instantiate a timer and a component (decision) whose role is to indicate whether to turn the light on or to open the shutters according to an identifier and a time given as input.

![Decision module (AADecision)](image)

Figure 4: Decision module (AADecision)

In two AA for a second weaving cycle, we describe the mechanism of perception that will be implemented in the application. These two AAs (Fig. 5) aim to connect the RFID reader and the switch to the decision component. According to its pointcut, the first AA, to be woven, requires the presence of a component with a name beginning by RFID and a component with a name beginning by decision and ending by a number. The second AA requires the presence of a component with a name beginning by switch and a component with a name beginning by decision. So when a badge is read by the reader or when the switch changes its state, the decision-making module will be informed of it.

Finally, we must add some AA (Fig. 6) to bind the decision part to lights and shutters. These AA are destined to a third round of weaving. We design two AA to ensure that the system is still running even in the absence of one of those two actuators.

![Perception modules for RFID and switch (AARFID & AASwitch)](image)

Figure 5: Perception modules for RFID and switch (AARFID & AASwitch)

We will now consider the behavior of energy consumption. Similarly this behavior can be decomposed. AA for perception and AA for decision from the other concern are reused. Finally we create a module (Fig. 7) for the third weaving cycle to add a filter on a call to open the shutter and to redirect those calls to the lamp according to the brightness outside.

![Action modules for Store and Light (AARollerShutter & AALight)](image)

Figure 6: Action modules for Store and Light (AARollerShutter & AALight)

The appliance of this AA involves the merging mechanism of the weave. Indeed, AA from the assistance to the person behavior for the same weaving cycle also introduces interactions on the same port. Moreover, the appearance in the assembly of several proxies communicating with a brightness sensor may introduce a conflicting interaction between each proxy and the average.AValue port. To resolve these conflicting interactions, a parallel grey-box component will be introduced by the weave between the proxies’ .NewValue port and the average.AValue port.

3.1.2. A sample of sequence of cascaded
weaving

Consider an initial assembly consisting of a proxy component to a web service for device switch, a proxy to a device light and finally a proxy to a device shutter. For now all AA are selected, except AADecision. Since the component decision is not present in the assembly, none of the selected AA can be applied. When the AA decision is selected, it is woven in the first weaving cycle and instantiates a black-box component decision. Because this component is now in the base application, the AA called AASwitch from the second cycle of weaving can be woven. It adds an interaction between the switch and the component decision. The other AA to be woven during this cycle cannot be applied since there is no RFID component in the base assembly. As for the second cycle, thanks to the component decision, two AA for the third cycle can be woven: AALight and AAShutter. They add some interactions between the light and the shutter and the decision component. The resulting assembly is described in Figure 8.

Finally a new brightness level sensor and a new RFID reader appears in the software infrastructure of the application, some new proxy components are generated and instantiated dynamically in the assembly. These appearances trigger the cascade of weaving cycle; only two AA can be woven: AARFID and AALightLevel. The resulting assembly is described in Figure 9. The first one, from the second cycle, links the RFID to the component decision. The second one, from the third cycle, is precisely described in section 2.1.

It appears in this scenario that many concerns may involve, and then reuse, the same AA for one of their functionalities. For example, AA for the action functionality to open or close the shutter or the lamp are reused in both concerns. This decomposition reduces the number of AA and rules to write compared to conventional AA in order to design a same number of possible configurations.

3.2. Opportunistic composition of reconfiguration at runtime

Since the application of these cascades of AA is done at runtime, the reconfigurations of the system are also done at runtime according to the underlying software infrastructure. AA from one group that are applied can collaborate, be composed, with AA from others groups to be woven next cycles dynamically. This composition is not explicit, meaning that an AA cannot embed a rule to trigger another AA. Such compositions can be defined as opportunistic, since an AA from a group for a feature is applied whenever it can. Since each module is independent, each of them will be evaluated and implemented according to the underlying software infrastructure as classical AA. Thus every module of each cycle can be applied independently. The possible configurations of the systems are then numerous and performed at runtime as the combination of AA.

For example, in the scenario a possible combination of applied AA could be: \{AADecision, AARFID, AALight\}, this could allow a user to switch on the light thanks to its RFIDTag. Some other combinations could be: \{AADecision, AASwitch, AALight\} or \{AADecision, AARFID, AASwitch, AALight, AAShutter\}, the latter allows fully implementing in the system the concern of assistance to the person as defined in the scenario.

First, the weaver evaluates pointcuts from each AA for the first weaving cycle. Each of them can be applied or not independently according to the base application given as input to the weaver. Once applied, the new application is used to evaluate pointcuts from AA for the second cycle. So that, if all AAs from each cycles are independent of each other (this means that they do not require a component from an AA previously woven to be applied), the number of achievable configurations for a set of cascaded AA can be calculated. For a set of AA to be woven, a fixed number of weaving cycle, and if each AA is associated to one of them, the number of possible combination between these AA is described in Figure 10.
Using conventional AA, we should have written an AA for each combination. For example, to describe as many configurations as in Figure 3 we should have defined $2^2 \times 2^3 = 128$ AA. But when AA from various cycles require in their pointcut, in order to be applied, some components from AA previously woven, this number of configurations is reduced. In our scenario the aspect $AA_{Decision}$ have to be applied in order to weave the other AA from cycle 2 and 3. In fact, such AA can be considered as a single one, meaning that $AA_{Decision}$ and $AA_{Light}$ can be considered as a single AA. Then the number of configuration in such a case is described in Figure 11.

$$C = \prod_{i=0}^{K} 2^{M(i)}$$

$i$: Cycle identifier

$K$: Number of cycles

$M(i)$: Number of AA involved in the cycle number $i$

$C$: Number of combinations

Fig. 10 Number of configuration that can be generated including related AA.

In the scenario, action and perception features of the system, to be applied, require the decision part. So that, the number of configurations that can be achieve thanks to these cascade of AA is $2^2 \times 2^3 = 32$.

This ability to combine various AA at runtime, more than increasing the number of reconfigurations that can be achieve using a minimal number of AA, also serves to increase the adaptability of applications to their infrastructure for greater continuity of service. Indeed, the various functionalities associated to the various weaving cycles can be implemented in various ways, according to AA that can be applied. Indeed, the pointcut matching of each module deployed is checked at each weaving cycle. During an appearance or disappearance of a device in the software infrastructure of the application, the AA that can be applied are woven in an opportunistic way. The concern to be set up in a weaving cycle is then always implemented with the maximum AA applied depending on the underlying infrastructure. In this way, the loss in the infrastructure of a device, used for a feature, does not lead necessarily to the loss of the feature in the application. Only parts of the feature that cannot be woven are no longer implemented. Similarly it becomes possible to provide alternative mechanisms for these functionalities. That is, if a device is available and can do the same as the one that just disappear, it can be used to replace it at runtime. It adds variability and self-adaptation facilities to the specific concerns addressed by a group of AA.

Moreover, it provides a mechanism to manage the unpredictability of ubiquitous systems.

As an example to change or add new sensors for location and identification, only some AA, similar to those previously described (Fig. 5), need to be added. Several AA can be deployed simultaneously based on various identification devices and can be applied indiscriminately. Thus, the system will work with available sensors, even if they are more or less; and this without to have to worry about it, because it is done at runtime, once the AA are deployed. Then, it allows ensuring a system running as long as possible for maximum durability of available features and a better continuity of service. Without cascaded AA, it would have required many AA, containing the decision, perception and action parts, for each possible configuration with each location or identification sensors.

3.3. Replication management

We can see in this scenario that there are several way to how to replicate an AA. For instance, in order to link several location sensors to a component to aggregate their data, the joinpoints combination policy required is of type $1:n$. This means that for each identified joinpoint, we must calculate all possible combinations of joinpoints with other joinpoints satisfying the pointcut matching. This allows to apply the advices as many times as possible (i.e. to all possible combinations of joinpoints). Conversely, when we use the RFID sensor, we will rather select a $1:1$ policy. We only choose a combination for each instantiated joinpoint. In the example there is only one decision component, so only one RFID sensor will be linked to the decision component. In our model, the weaver, process for each AA, independently, the following steps: (1) the pointcut matching, (2) the joinpoint combination and (3) the advice factory. Thus, at each AA can be associated a combination policy. Using cascaded AA this mechanism can be managed more smoothly, as AA are decomposed. Figure 12 presents an example of various policies.

3.4. Symmetry & interactions

All combinations and therefore all possible
configurations of the system can be defined as follows: A combination is an ordered set of non-ordered sets of AA:

$$ C = \{ \{AA_{00}, ..., AA_{0j}\}, ..., \{AA_{i0}, ..., AA_{ij}\} \} $$

A combination can be decomposed as a set of combinations. The range of a set of AA in a combination defines the weaving cycle for which the set is designated. A combination does not necessarily contain a set of AA for each cycle. Designing a concern will often consist in writing a combination.

Various combinations of AAs can also be composed. It consists in the union of sets of AA of the same range (i.e. to be woven in the same cycle). That is to say, AAs from various combinations to adapt a same functionality are all deployed in a same set.

$$ C_a \cup C_b = \{ \{AA_{00}, ..., AA_{0j}\}_a \cup \{AA_{i0}, ..., AA_{ij}\}_b, ..., \{AA_{00}, ..., AA_{0j}\}_a \cup \{AA_{i0}, ..., AA_{ij}\}_b \} $$

The union operator is symmetric, so the order in which combination are composed is not important. So the weaving operation of various combinations is symmetric.

Since these combinations can be composed, another property appears from this approach. An AA for a weaving cycle can have a side effect. An AA for a concern may trigger an AA from a next cycle for another concern. The reverse is not possible. An aspect cannot remove a component that was required to weave another aspect. Indeed an AA cannot withdraw a component unless it is unwoven. The side effect previously described may be the cause of an adverse side effect on the reconfiguration of the system. This kind of interaction can be managed using contracts. A contract will allow or prevent interactions between several AAs for various concerns. Using this approach, interactions are managed explicitly. However, the use of contract is not required in two cases. First, if we want to allow interactions between AAs from various cycles. Then each AA is woven in an opportunistic way. Second, if the combination was designed to be independent of other combinations. In such a case a unique identifier is generated and associated to black box components so that they cannot be reused by AAs from others combinations.

A synthesis comparing AAs and Cascaded AAs presenting improvements and advantages of Cascaded AAs is presented in Table 2.

### Table 2. Synthesis, AAs vs Cascades of AAs

<table>
<thead>
<tr>
<th>Ability for an AA to reuse components from another AA</th>
<th>X</th>
<th>✓</th>
</tr>
</thead>
</table>

4. Experiments

We evaluated our approach in term of performance with some experiments on the duration of a weaving cycle over components assemblies randomly generated. They were conducted on a standard personal computer (Athlon X2, 1.6 GHz, 512Mo RAM). For this purpose various types of components have been instantiated randomly at runtime, in order to activate randomly two types of AA. The first AA links a switch to the light and the second one implement the energy saving concern as defined in 4.1.1. The weaving of these two AA involves the merging mechanism. Components are instantiated so that the AA for the energy saving concern is applied with a probability of 33%. This means that the merging mechanism is involved with a probability of 33%. These results are presented in Figure 13. In term of duration, a weaving cycle can be divided into three major steps: (1) pointcut matching, (2) merging and (3) translation of the resulting instance of advice into elementary instructions. 85% of the overall process is spent during step 2.

![Figure 13: Duration of the weaving process](image)

Thanks to the divide and conquer approach provided by cascades of AAs, the number of rules to be merged during each cycle is lower than the number of rule to be merged in the AA approach. Thus, Cascades of AAs’ adaptation duration is bounded by AA adaptation duration. So the number of cycles is not a limiting factor with regard to the response time of the adaptation process. We are able, in our current implementation of cascaded AA, to compose: in 1s 100 components and 40 components in 100ms. Cascaded AAs allow combining various modules at runtime, increasing the number of reconfigurations that can be achieved using a minimal number of modules, in a timely fashion.

5. Related work

Some middleware combining components or services and aspects with a weaving at runtime use aspects in an invasive way as AspectJ2EE[COH 04] or PROSE [POP 01]. This means that they use aspects to adapt the code of components that are seen as white
boxes. This is not well-suited for ubiquitous computing where part of the software infrastructure is non editable by the system itself. But the MIDAS [POP 03] middleware using PROSE proposes to use aspects to implement constraints in a community of node. The constraints are defined as aspects and nodes propose services. To do this, they use the concept of spontaneous container; it includes a node to manage the appearance / disappearance of other nodes. PROSE does not define a new language but uses Java. This approach does not provide a language and a level of abstraction appropriate to constraints of ubiquitous computing. However it proposes to address an important challenge of ubiquitous and mobile computing, it considers changes occurring in the software infrastructure of an application. Some middleware, as CAM/DAOP[FUE 07], also combine aspects over components in a non-invasive way. The classical approach to improve aspect reusability which is also proposed by this middleware is to separate the declaration of pointcuts from the definition of advices. Most of the time pointcuts are defined using some declarative language as XML. This greatly facilitates the reuse of aspects but as AAs compared to cascaded AAs it can be improved by weaving aspects over aspects at runtime or combining aspects at runtime.

The plugin architecture proposed in [CHA 09] is based on AO4BPEL [CHA 04] which is an aspect oriented workflow language. The latter allows dynamic adaptation of services composition. In this work, the problem of management of interactions between aspects is not addressed dynamically. This management is implemented using the standard operators: after, before... Since this work is applied to workflows, it does not consider the dynamic evolution of the software infrastructure. In the proposed architecture there are two types of aspects: monitoring aspects that are able to activate or deactivate adaptation aspects at runtime. Aspects can be added / removed or sometimes generated at runtime. In our approach an AA and then combinations of AA may also be added / removed / combined at runtime, but no mechanism allows to generate them. An AA may also be the origin of the triggering of another AA. But in the case of AA this is not necessarily defined explicitly (an AA does not describe in one rule that another AA may be triggered) for better reusability and to manage the unpredictable variability of the system. These interactions can be achieved if an AA instantiates a component that selects AA or by adding meta-data to AA.

JAsCo [VAN 05] is a dynamic AOP middleware. Aspects are encapsulated into components and connectors can deploy them by specifying their interactions. The aspects are woven according to a sequence of events represented as a finite state automaton. Advices can then be associated to the various transitions of this automaton. In this sense aspect weaving can be chained. As for the plugin architecture presented previously, advices of the chain are well-defined and aspects are statefull which is not necessarily the case with the cascaded AA. On the other side this approach allows to weave aspects according to the history of previously checked pointcuts.

In EAOP [DOU 02], the authors propose mechanisms to define aspects of aspects. This mechanism allows to apply aspects on others aspects including a mechanism to manage recursive calls. This is done using a monitor that sequentializes application of aspects. The monitor observes events from the execution of the application and spread them to all aspects. The architecture is sequential, when the base application generates an event and involves the monitor it is stopped. This is not the case with AA and cascades AAs. Moreover, AA’s pointcuts do not concern the execution flow of the application but the structure of the component assembly to be applied. In this approach aspects are processed sequentially.

Many works have identified the interest of aspects for ubiquitous or mobile computing, because of the encapsulation of adaptations into aspects [ZAM 04, HIR 06]. In dynamic service adaptation, aspects are used to integrate services or to correct services mobile communication; they are not used to make structural reconfiguration of services workflow. In SAFRAN [DAV 06] adaptation is identified as a crosscutting concern. SAFRAN is an extension of Fractal in order to facilitate the design of adaptive applications. To do this, they use adaptation aspects that can be added / removed at runtime. SAFRAN’s joinpoint model is not restricted to the execution flow of the application. So that the adaptations can be triggered thanks to some events related to the context of the application. Those events are called exogenous events. The architecture of SAFRAN comprises two parts: (1) an adaptation language FScript to reconfigure a component assembly where the ACID properties for dynamic reconfiguration are guaranteed; and (2) a toolkit to observe the context called WildCAT. An adaptation controller is integrated to the membrane to link, thanks to rules, these two parts. Thus, this approach respects the classical decomposition of ubiquitous systems. The implementation of these mechanisms has less evolving facilities than cascaded AA. Moreover, unwise cascaded AA this architecture does not allow to adapt all the mechanisms of its own decomposition. This is allowed in our approach since the implementation of these functionalities is the result of the weaving. Furthermore, the weaver is itself based on a component assembly that can be adapted by another weaver.

6. Conclusion
In this paper we have presented an approach to combine AA by chaining weaving cycles. This approach allows us to decompose an adaptation for a specific concern in various adaptations grouped according to the functionality they adapt. Adaptation are more easily identifiable and their encapsulation into aspects allows adding / removing / exchanging at runtime adaptations depending on the context. Since cycles are woven the one after the other, an aspect
may trigger other aspects from next cycles and this, not necessarily, in an explicit way. This while preserving the symmetry of the operation of weaving. So that the orders in which concerns are woven is not important. This approach allows, as well as standard AA, to respect the dynamics of the software infrastructure and relies on a language dedicated to the adaptation of components assemblies. But it adds to AA mechanisms to manage the high variability of ubiquitous applications. Mechanisms that make it easier for a designer to manage the unpredictable variability of its system.

ACKNOWLEDGMENT
This work is part of the Continuum Project (French National Research Agency) ANR-08-VERS-005.

REFERENCES