Context adaptative systems based on Horizontal architecture for Ubiquitous Computing

Nicolas Ferry
I3S (UNS - CNRS) and CSTB
930 Route des Colles - BP 145
06901 Sophia-Antipolis France
ferry@polytech.unice.fr

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

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

Many adaptative context-aware middleware exist and mostly rely on so-called vertical architectures that offer a functional decomposition for context-awareness. This architecture has a weak point: it leads to data centralization. Our mechanism for adaptation: the Aspects of Assemblies is based on a horizontal architecture. This type of architecture separate the system into behavior and is based on a decentralized approach. However, after having shown some limitations of AAs in the field of context-awareness we will introduce a way to improve them using a multi-cycle weaving approach. Then, using this approach we will be able to build context-adaptative systems that interact directly with their environment. Finally we will evaluate our approach in term of reactivity.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Configuration Management; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Management, Design, Measurement

Keywords

Ubiquitous computing, software composition, service oriented architecture, component-based software engineering, aspect-oriented programming, context-awareness

1. INTRODUCTION

Ubiquitous computing, as described by Mark Weiser [21], relies on a computer present everywhere, at any times in any things. Thus ambient computing systems must consider multi-users, multi-devices environments. Indeed with the recent years advance in mobile technologies and the miniaturization of computer hardware, processing units are becoming invisible and a part of the environment. So pervasive systems have to be aware of many heterogeneous entities that compose their environment. The topologies of this infrastructure are dynamic due to arbitrary node mobility. Because environment's nature is highly variable and entities may be mobile, pervasive systems have to handle those variations and offer such dynamicity. These systems have to be able to **adapt** to their environment (**context** of execution).

1.1 Context-awareness

Many context-aware middlewares exist, they mostly rely on so-called vertical architectures that offer a functional decomposition (see Figure 1) for context-awareness.

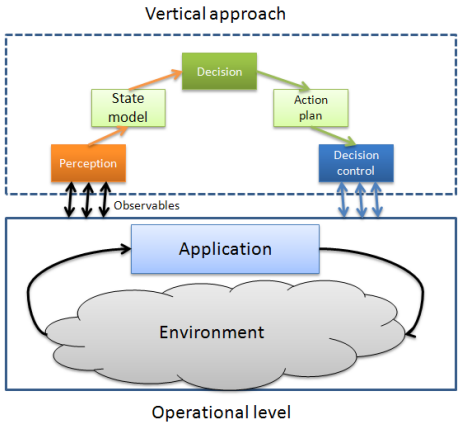

Figure 1: Vertical architecture

Being aware of its environment traditionally relies on different mechanisms [8] (see Figure 1). Most of adaptive and context-aware middleware fulfill all those mechanisms, and are often specialized in one of them. The first stage of sensing is to capture observables on the environment [10, 6, 12]. These observables are then processed in the form of symbolic

observables about the state of the environment, for example using ontologies [11]. Thanks to these data, a decision stage or situation identification stage will provide an action plan to be implemented by the decision control stage. This stage is designed to process an action plan based on changes in the environment. In order to do this, different mechanisms can be establish such as event-condition-action rules [9].

These architectures have as weak point: the data centralization in at least one of the components of the decomposition. Thus, in Gaia [17], a context-aware middleware, observables are stored in a single entity: the "context file system". We found the same problem with SOCAM [11] or CARMEN [3]. For example, CARMEN use a specialized LDAP containing user profiles for the decision stage. This data centralization remains hardly imaginable in ubiquitous computing where we need to manage numerous and heterogeneous entities. On the other hand few middleware consider data decentralization [13, 6] or are based on pure ad-hoc architecture as CoWSAMI [1].But they do not consider all the mechanisms previously described (see Figure 1) including the adaptation stage.

1.2 Adaptation

Context-awareness processes always end with the adaptation stage. In this field, adaptation aims to change system's behavior according to its surrounding. For such systems adaptation can take two forms [16]. On the one hand static adaptations require to shutdown the system to be implemented. On the other hand dynamic adaptations are able to change the system behavior at run-time. Since we consider systems that must continuously evolve in order to be in lines with their environment, we must center on dynamic adaptations. However, static adaptations, as in CAMidO [2], can be considered as an extra .

To do such adaptations there are various approach: altering data that are processed or modifying internal processing [14]. The first is used to configure algorithms, components or strategies, the second aims to exchange algorithms, components or strategies with others. This type of adaptation is implemented in most context-aware middleware. To do this they use various techniques such as reflection [5], code generation [2], proxies to redirect methods' call [19] or aspect oriented programming as in SAFRAN [9].

Many of these mechanisms also rely on centralized architectures and/or do not really address the needs of reactivity. Consequently these approaches for adaptation and more globally vertical architectures do not fulfill all ubiquitous computing requirements.

2. HORIZONTAL APPROACH

Accordingly other approaches from reactive systems or robotics propose to address these issues. For this purpose they evoke horizontal architectures (see Figure 2)[4].

The aim of this kind of approach is to specialize a generic minimal core of the application with some specific modules [22] called horizontal layers. These layers are independent of each other. As part of a context-aware system, each layer includes the previously described mechanisms and is connected to the world via a set of sensors. This crosscuting

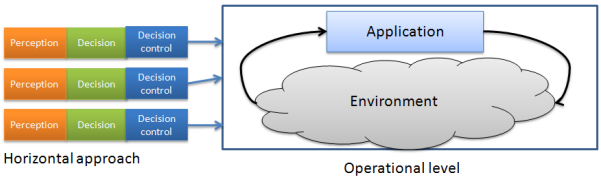

Figure 2: Horizontal architecture

approach of mechanism for context-awaress induces decentralization.

So a contribution of this architecture is to see the system not as a sequential process. Indeed, the decomposition separates a system into behaviors. So more complex behaviors are achieved with simple ones like for divide and conquer strategies. Therefore this type of architecture corresponds to the needs of situated systems (systems that interact directly with their environment):

- A layer sees only what is relevant for him and not "the world".
- Each layer captures what is relevant in the environment directly, sensing the surrounding to a degree sufficient to achieve the necessary.
- No representation of the environment, a decentralized approach.
- Many small behavior of low complexity for the best possible reactivity.

Horizontal architectures need a coordination mechanism to combine the output data of each layer in order to obtain a rational and coherent global behavior. Horizontal decomposition is often associated to aspect oriented programming (AOP) and in such case the coordination mechanism is the aspects weaver. Otherwise in Brooks' subsumption architecture [4] it is a priority mechanism. A simpler mechanism induces more reactivity.

So we propose a mechanism respecting this horizontal approach and then based on a decentralized architecture in order to offer a maximum reactivity.

3. HORIZONTAL AUTO-ADAPTATION USING ASPECTS OF ASSEMBLIES

As we said before, ubiquitous computing systems must manage numerous and heterogeneous entities and try to be reusable. To do this they are often built using component based software engineering (CBSE). In component based systems, assemblies govern application's behavior and modifying these assemblies will affect their behavior. Thus, using this notion of component assembly, we can define software architecture as a graph of components. In such a graph, nodes are instances of components types and edges are connections between those components. So application and assemblies can be written:

$$Ass = App = (C, L)$$

Let C be a set of components and L a set of connections. We can write $C = \{\emptyset, c_0, ..., c_i\}$, where c_i is a unique instance of component. In the same way, we can express L as $L = \{\emptyset, l_{00,01}, ..., l_{n-1z,nx}\}$, where $l_{n-1z,nx}$ is a connection between components n-1 and n on ports number z and x.

Aspects of Assemblies consist of an extended model of AOP for **adaptation** schemas and of a weaving process with logical merging. **AAs are a way to build systems based on a horizontal architecture.They allow structural reconfiguration of components assemblies at runtime, which is a kind of dynamic reactive composition.** AOP is an approach that enables separation of crosscutting concerns. In traditional AOP [15] aspects are composed of pointcuts and advices. Pointcuts point out "where" inject the code to weave while advices describe this code. The scheduling of these aspects occurs at joinpoints, which are anchors where aspects have to be weaved. In the context of AAs these concepts are still valid but with some shadings. Here an advice describes a structural reconfiguration of an assembly of components, while a pointcut match the components' ports from the assembly on which will take place those changes. Accordingly, joinpoints are ports rather than execution's points. Moreover, as long as it is to reconfigure an assembly of components (which can be grouped either in a component, either in a service) we lose the property of encapsulation for components (or composite components) assemblies. In contrast, they preserve the component's black box property. We define therefore an aspect of assembly as follows:

$$AA_i = (pointcut_i, advice_i)$$

The pointcut section ($pointcut_i$) is a set of filters on ports of the assembly while the advice section ($advice_i$) consists of a set of rules describing changes in the initial assembly. AAs are applied on an initial assembly Ass_0 which is by default empty: $Ass_0(\emptyset, \emptyset)$. So weaver's inputs are: a set of AAs A_n and Ass_0 , output is: a new assembly Ass_{n+1}.

$$T(Ass_0, A_n) = Ass_{n+1}$$

In the manner of automatons cycles consisting of a phase of acquisition (storage of inputs), then execution and finally writing outputs, we're talking about weaving cycle. Where inputs are AAs and an assembly, execution is the weaving process and output is a new assembly, a new application (see Figure 3). This output of a weaving cycle will not be the base for the next round of weaving. Each round is always based on the initial assembly to secure the property of symmetry. **This property must be preserved because it simplifies the development of adaptations, ensuring a similar result regardless of when adaptations are incorporated.** Then we allow permutations between adaptations without effect on the resulting application.

In details we can present the cycle as in Figure 4. The weaving process can be split into 3 steps. First, pointcut matching is a function that has a set of components from the initial assembly and pointcuts as inputs. Its goal is to find the components on which advices will be woven. Thanks to

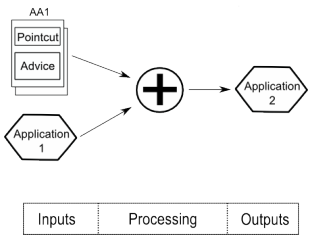

Figure 3: Weaving cycle

pointcut matching an advice can be weaved many times in the same weaving process. This function produces joinpoints P_i.

$$pointcutMatching(C, pointcut_i) = P_i$$

The second step uses those joinpoints. It consists in generating instance of advices. This advice factory has for input a set of joinpoints p_i and advices ($advice_i$). It produces an instance of advice describing some changes to apply on the initial assembly.

$$adviceFactory(p_{ij}, advice_i) = A_{ij}$$

Finally the composition and merging mechanism merges all these instances of advices in order to generate a unique instance of advice which is the final assembly.

$$\varphi_t(A_{00}, ..., A_{ij}) = A_{i+1j}$$

The whole weaving cycle respects some properties: commutativity, associativity, idempotence [7] on a cycle. This means that *there is no order on the application of AAs* and therefore no potentially quickly complex historic of applied AAs.

Thus we can build applications by composing several aspects of assemblies at the same time.

4. REACTIVITY AND OPPORTUNISM

In order to be more reactive these cycles can be triggered in several ways, we call these triggers disruptions. During consideration of disruptions, system does not tolerate other disruptions.

4.1 User-triggered

The first type of disruption are variations in the set of AAs given as input to the weaver. This is the selection / deselection or adding / removing aspects of assemblies. This kind of disruption can be noted:

$$App_{n+1} = T(A_n, App_n)$$
$$App_{n+2} = T(A_n + 1, App_n)$$

So, whenever a change occurs in the set of selected aspects, a new cycle is triggered on the same initial assembly.

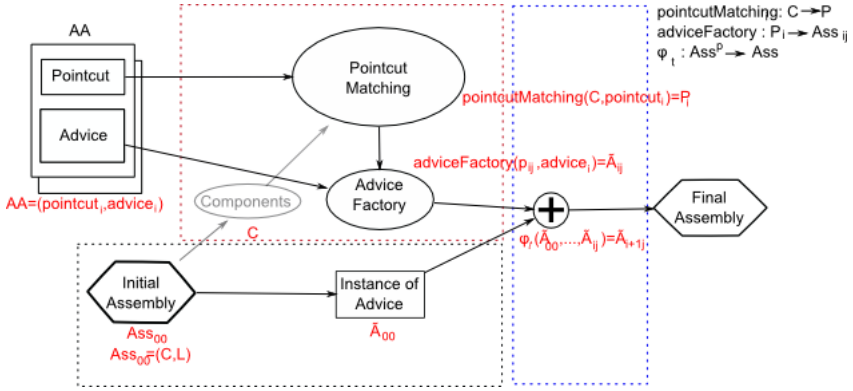

Figure 4: Detailed weaving cycle

4.2 Context-triggered

In the same way at any time a new device can appear / disappear in the environment and be dynamically inserted / removed in the initial assembly. These variations in the initial assembly are also the source of a new weaving cycle . In such a case, the pace of the interaction is determined by the environment, so **thanks to AAs we can build situated systems**. This kind of disruption can be noted (Δapp_n describes a variation in the application software infrastructure):

$$App'_{n+1} = F(App_n, \Delta app_n)$$
$$App'_{n+1} = App_n \text{ if } \Delta app_n = null$$

In traditional context-aware approaches the software infrastructure is know *a priori* and the relevant infrastructure is fixed *a priori* by the developer. For each entities from the infrastructure the developer decides if it is relevant to this specific application. The inclusion of this type of disruption allows us to build applications with a "bottom-up" approach. Thanks to this approach, we can build an application opportunistically in terms of some infrastructure services, devices. So we can speak of application in line with its infrastructure. **Opportunism is an important feature since we no longer consider context-aware systems for specific scenarios or type of contexts.** Both types of disruptions bring into play a cycle of weaving that can be represented as in Figure 5:

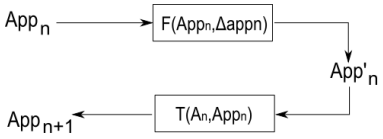

Figure 5: Formal weaving cycle

$$App_{n+1} = T(A_n, F(App_n, \Delta app_n)) = T_n(F_n(App_n))$$
So that if $A_{n+1} = A_n$ et $\Delta app_n = null$ then
$$App_{n+1} = App_n$$

It is therefore to answer opportunistically to unpredictable changes. This unpredictability implies the loss of commutativity between several weavings cycles.

5. SYNTHESIS

Finally aspects of assemblies allow us to build auto-adaptative applications where adaptation is a crosscutting concern. AAs are reusable entities that can be exchange in a decentralize approach.

5.1 Aspects of assemblies, a decentralized approach for reactive adaptations

AAs are a way to build opportunistically application. This can be done by composing various AAs. Thanks to a mechanism of conflict resolution [7] they always provide consistent results. Furthermore preserving the property of symmetry(commutativity, associativity, idempotence) allows us to weave AAs without order. A developer should only be concerned with AAs that need to be written. Moreover AAs are reusable entities; they provide a transversal modularity and therefore a good reusability. This allows us to offer minimum code dispersion through pointcuts. Thanks to pointcuts we can apply that same advice in several places.

So AAs are a way to create situated systems using a horizontal architecture where adaptation is a dynamic reactive composition.

5.2 But still with some limitations

AAs are a way to build systems based on a horizontal architecture. But with limited skills in context-awareness since it only consider variation in the software application. Schilit [18] classified context informations into three domains. The user domain focus on notion such as users' profiles, the system domain relies on the software infrastructure and thus on resources available at runtime and finally the environmental domain. Using AAs, we are only able to work on the system domain and a little on the user-domain (user-triggering).

Consider for example a building equipped with various locations systems and light sensors in each rooms. We wish to use those equipments to enable the room's energy system to automatically turn on/off lights and heaters. It may depend on the presence (or not) of someone in the room and its brightness. In this context we need to use multiple modules in AAs building such systems. Modules similar to those previously presented for context-awareness in horizontal architectures. It means a module for perception aiming to tell whether a room is occupied and the brightness of the room

with a maximal reliability. Therefore it will place filters and components to transform observables into symbolic observables for presence detection (occupied or not). The second module of decision shall, depending on the brightness and on the room occupancy, choose an energy-saving policy. The last one, the reaction module, will connect lights and heaters to the decision-making component.

Thanks to this decomposition, if one wishes to install a new tracking system, changing the observation module will be sufficient. The same goes for other modules. However, it is not possible to build such modules with the weaving mechanism presented above. Indeed, with the latter, **aspects of assemblies can not use components instantiated by other aspects of assemblies**.

This can be proofed using the formalism previously defined: As we said before, the properties of commutativity, associativity and idempotent on a cycle have been proofed in [7]. So we can write:

$$\varphi_t(A_{00}, A_{10}, A_{20}) = \varphi_t(\varphi_t(A_{00}, A_{10}), \varphi_t(A_{00}, A_{20})) = \varphi_t(\varphi_t(A_{00}, A_{20}), \varphi_t(A_{00}, A_{10}))$$

AA1 and AA2 are two Aspects of Assembly, let:

1. the initial assembly be empty : $Ass_0 = (\emptyset, \emptyset)$
2. AA2 be applied only if the component c_2 is in the assembly : $\forall j \, \exists \widetilde{A}_{2j} \Rightarrow pointcut_i(C) = P_i \,|\, p_{ij} \in P_i \wedge c_1 \in p_{kj}$

Consider that AA1 add the component c_1 to the initial assembly then AA2 add the component c_2:

$$\varphi_t(A_{00}, A_{10}) = A_{10} = Ass_{10} = (C_0 \cup C_n, L_0 \cup L_n) | c_2 \in C_n$$
$$\varphi_t(A_{10}, A_{20}) = A_{20} = Ass_{20} = (C_0 \cup C_n, L_0 \cup L_n) | c_1, c_2 \in C_n$$

So we can rewrite our first equation as:

$$\varphi_t(A_{00}, A_{10}, A_{20}) = \varphi_t(\varphi_t(A_{00}, A_{10}), A_{00}) = \varphi_t(A_{00}, A_{10}) = (C_0 \cup C_n, L_0 \cup L_n) | c_1 \in C_n$$

Component c_2 cannot be instantiated since we want to respect the property of commutativity. So joinpoint are only components from the initial assembly. To conclude we can say that an AA cannot use components instantiated by others AAs in a same weaving cycle.

6. A MULTI-CYCLE APPROACH

To address these limitations we propose a multi-cycle weaving approach. We assume that the assembly resulting from a weaving cycle can be used as the initial assembly in the next cycle (see Figure 6). It is on this assembly that pointcut matching and composition/merging will be done. Thus the components instantiated in the first cycle may be used by the following one. But this simple implementation entails the loss of the property of commutativity and hence the symmetry between aspects of assemblies.

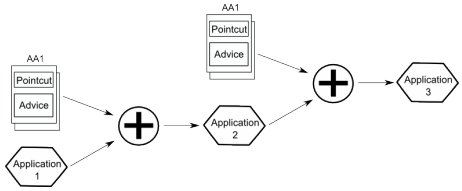

Figure 6: Multi-cycle weaving

6.1 Improving reusability while preserving the symmetry

To maintain properties of associativity, commutativity, idempotence, we consider aspects such as ordered sets of AAs. We can call aspects with cardinality greater than one cascade aspects. AAs include in a set are called modules, each module is indexed according to the weaving cycle in which it must be weaved.

$$AA_j = \{AA_{0j}, AA_{1j}, ..., AA_{ij}\}$$

Keeping in mind the need for modularity and reusability of modules, they must be traditional aspects of assemblies with pointcut and advice. Thanks to pointcut matching, AAs can be applied if some components are in the assembly regardless of the modules previously woven. These sets are not necessarily continuous (i.e. there is not necessarily an aspect for each weaving cycle):

$$AA_z = \{AA_{0z}, \emptyset, AA_{2z}, ..., AA_{iz}\}$$

So there is an order in the merger(see Figure 7):

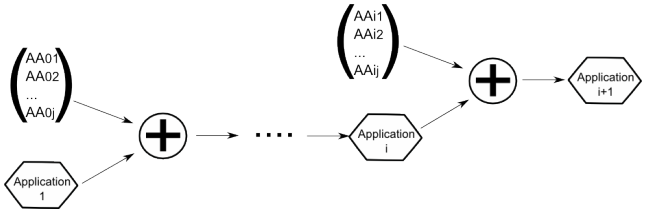

Figure 7: Cascade AA as a set of modules

$$\varphi_t(AA_{ij}, AA_0) = \varphi_t(\varphi_{t-1}(\varphi_{t-2}(...\varphi_{t-i}(AA_{0j}, A_{00})..., AA_{ij})$$

Finally all aspects given to the weaver as input are seen as sets of aspects but with various cardinalities (since we want to weave old fashioned aspects with cascade aspects). The number of weaving cycles is equal to the cardinality of the cascade aspect AAz consisting of greater cardinality. This means for the weaver of aspects to consider in the same cycle i all aspects AA_{ij} whatever j. The number of aspects woven in a cycle is not necessarily equal to the number of cascade aspect. Thus each element consisting of these cascade AAs are woven in the order defined. Between each cycle the assembly to consider is always the same; there is no order between modules in the same cycle. For example, consider the following cascade aspects:

$$AA_1 = \{AA_{01}, AA_{11}, AA_{21}\}$$

$$AA_2 = \{AA_{02}, AA_{12}\}$$
$$AA_3 = \{AA_{03}\}$$

Aspects AA_{01}, AA_{02} and AA_{03} will be woven in a same first cycle. The resulting assembly is then taken as the initial assembly for the following weaving cycle planned: AA_{11} and AA_{12}. In the same way it is on the assembly thus obtained that will be woven the aspect AA_{21} (see Figure 8).

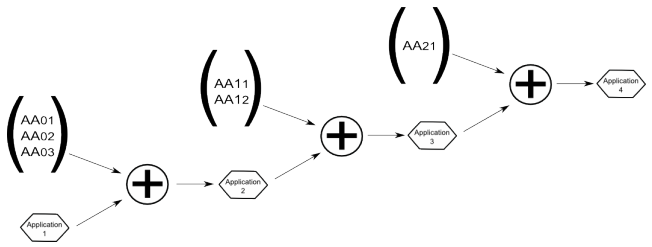

Figure 8: Multi-cycle weaving process

Here we can proof that this approach allow us to preserve symmetry between cascade aspects of assemblies.

$$\forall i, n\ \varphi_{t+max(i,n)}(AA_{ij}, AA_{nm}) =$$
$$\varphi_{t+max(i,n)}(\varphi_{t+(i-1)}(...(\varphi_t(AA_{0j}, AA_{00}), ..., AA_{ij}),$$
$$\varphi_{t+(n-1)}(...(\varphi_t(AA_{0m}, AA_{00}), ..., AA_{nm}))$$

So at instant t we can write :

$$\varphi_t(AA_{0j}, AA_{0m}) = \varphi_t(AA_{0m}, AA_{0j})$$
$$So\ \varphi_t(AA_{0m}, AA_{00}) = AA'_{00} = \varphi_t(AA_{0j}, AA_{00})$$

And we can rewrite our equation as:

$$\forall i, n\ \varphi_{t+max(i,n)}(AA_{ij}, AA_{nm}) =$$
$$\varphi_{t+max(i,n)}(\varphi_{t+(i-1)}(...(\varphi_t(AA_{1j}, AA'_{00}), ..., AA_{ij}),$$
$$\varphi_{t+(n-1)}(...(\varphi_t(AA_{1m}, AA'_{00}), ..., AA_{nm}))$$

Continuing the same reasoning:

if i>n
$$\forall i, n\ \varphi_{t+max(i,n)}(AA_{ij}, AA_{nm}) =$$
$$\varphi_{t+max(i,n)}(\varphi_{t+(i-n)}(AA'_{(i-n)}, AA_{nj}))$$
if n>i
$$\forall i, n\ \varphi_{t+max(i,n)}(AA_{ij}, AA_{nm}) =$$
$$\varphi_{t+max(i,n)}(\varphi_{t+(n-i)}(AA'_{(n-i)}, AA_{im}))$$
if n=m
$$\forall i, n\ \varphi_{t+max(i,n)}(AA_{ij}, AA_{nm}) =$$
$$\varphi_{t+max(i,n)}(\varphi_{t-1}(AA'_{(i)}, AA'_{(n)}))$$

In all cases we respect the propriety of symmetry while we preserve it in a same weaving cycle. **Since the multi-cycle approach allow AAs to use components instantiated by others AAs; considering AAs as an ordered sets of AAs allow us to preserve the properties of symmetry between cascade aspects of assemblies. Moreover it improve the modularity and reusability of aspects of assemblies. Because we are still working on reusable entities, now organize as sets, we still preserve a decentralized approach for context-awareness.**

6.2 A new kind of disruptions

If the two ways to trigger a weaving cycle described in section 3 are still valid, the multi-cycle approach introduces a new kind of disruptions. This is to automate changes in the application's software infrastructure (i.e. to automate changes in the initial assembly). It is no more changes caused by variations in the environment of the application but by the system itself. This kind of disruption can be noted:

$$App_{n+1} = T(A_{in}, App_n)$$
$$App_{n+2} = T(A_{i+1n}, App_{n+1})$$

All cycles are directly woven one after the other, so once the weaving process is started, the system is no longer open to others disruption (see Figure 9). Thus the set of the aspects selected for the first weaving A_{0J} will be retained until the last cycle.

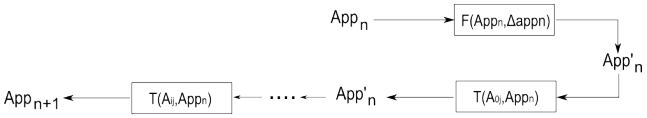

Figure 9: Formal Multi-cycle weaving

6.3 For context-awareness

Using the mono-cycle weaving approach we only were able to work on the system domain, due to the impossibility for AAs to use components instantiated by others AAs. **Thank to our multi-cycle weaving we are able to work on the system domain as well as on the environmental domain**. **More than situated systems now we can build decentralized context-adaptive systems** that continuously adapt their behavior according to infrastructure evolution [14]. The multi-cycle approach enables us from now on to direct ourselves toward context-awareness in a similar way as described previously in section 1. We will consider some cascade aspects as contextual aspects: $AA - contextual = \{AA - audit, AA - decision, AA - reaction\}$

Such decomposition allows us to reuse modules that compose contextual AAs. Aspects of audit will aim to instantiate components on the shelf (COTS) observers. But also to verify the presence, among the candidates for the observation of the context which are in the application's software infrastructure, of those necessary to weave AA_i (i.e. necessary to the horizontal layer). Aspects of decision will aim to implement an evaluation policy for the collected observables. The composition implementation may rely on different mechanisms of decision (fuzzy logic, event-condition-action rules ...). Finally aspects of reaction will have to adapt the application.

As we can see, there is a direct matching between the horizontal architecture proposed by Brooks and our decentralized approach. But since our coordination mechanism between all horizontal layers is the aspects weaver there is some

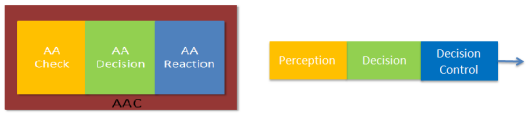

Figure 10: Contextual Aspects of Assemblies

advantage. Indeed our approach allows us to share information from the various layers during the weaving. Furthermore thanks to the merging mechanism we can assume that after each cycle there are no conflicts in the assembly. So back to the example given in section 5.1, we can be context-aware using the following aspects:

```
POINTCUT:
locationSensor:=/ubisense[[:digit:]]/
sensor:=/luminosite[[:digit:]]/

ADVICE:
schema location_light_audit(locationSensor, sensor):
 occupation:'WComp.BasicBeans.OccupationFilter' ;
 filter:'WComp.BasicBeans.LightValueFiltering' ;
 locationSensor.^NewPosition_EventedValue->
 (occupation.isOccupied)
 sensor.^Value_Evented_NewValue->(filter.filter)
```

Pointcut are regular expression in AWK and advices are written in ISL4WComp [20]. The **AA-audit** module, when a tracking system and a light sensor system are discovered, composes the tracking system with a component to interpret localizations and the light sensor with a filter to check data's validity. Filters and interpreters are instantiated by this module.

```
POINTCUT:
occupation:=/OccupationFilter[[:digit:]]/
filter:=/LightValueFiltering[[:digit:]]/

ADVICE:
schema occupation_light_decision(occupation, filter):
 decision:'WComp.BasicBeans.LightDecisionModule' ;
 filter.^NewEventedValue->(Decision.setLight)
 occupation.^StateChanged->(decision.setOccupation)
```

The **AA-decision** module, when a filter for brightness and an interpret for localization are discovered (it uses components instantiated by other modules, AAs), composes them with a COTS to evaluate contextual conditions.

```
POINTCUT:
Decision:=/LightDecisionModule[[:digit:]]/
sender:=/switch[[:digit:]]/
receiver1:=/light[[:digit:]]/
receiver2:=/heater[[:digit:]]/

ADVICE:
schema decision_light_reaction(decision, sender,
receiver1, receiver2):
 decision.^ReactionLight ->(switch.setState)
 decision.^ReactionHeater->( receiver2.setState)
 sender.^Status_Evented_NewValue ->( receiver1.SetStatus)
```

The **AA-reaction** module, when lights, heaters, switches and a decision making component are found, composes the last one with the devices mentioned above. Thanks to this contextual cascade aspect we are able to do contextual auto-adaptations and so the build system can manage lights and heaters from a room.

This cascade aspect can be merged with others cascade aspects. For example with an other cascade AA building an infrastructure to manage the lights (just connecting the switch to the light) : $AA_1 = \{\emptyset, \emptyset, AA_{13}\}$

The assembly resulting from the merging of AA_1 and $AA - contextual$ is an application able to manage lights and heaters and to allow users to manage the lights of the room he is in.

7. EVALUATION

We evaluate our approach with some experiments on the cost in time of a weaving cycle over components assemblies randomly generated. They were conducted on a standard personal computer. For this purpose different types of components have been instantiated randomly, in order to activate randomly two types of aspects of assemblies. The aspect involving the merging mechanism is weaved with a probability of 1/3 depending on components instantiated.

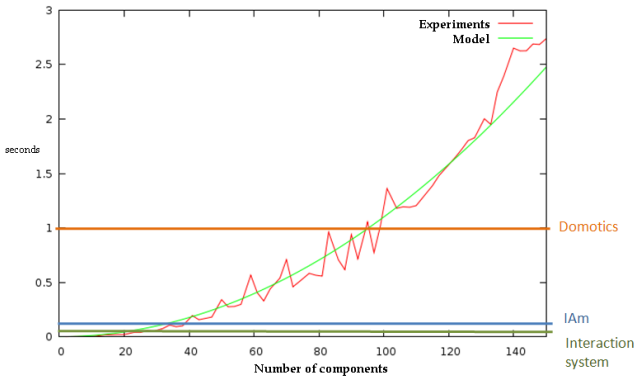

Figure 11: Weaving cycle duration

The inference machine for composition and merging is written in Prolog, so peaks are due to garbage collection mechanism (see Figure 11). Weaving cycles can be divided into three categories each with its own cost in time: (1) Selection of AAs and pointcut matching, (2) Composition and potentially merging of advice instances and (3) Translation of instances of advice into elementary instructions. In the multi-cycle weaving approach each categories are redone every cycle. So let C be the cost in time of a weaving cycle, for a cascade aspect of assemblies it will be about :

$$D = card(AA_{ij}).C$$

Let D, K and T being respectively the cost in time of pointcut matching, composition/merging and translations mechanisms; C can be noted : $C = D + K + T$

Pointcut matching cost in time depends on the number of component c involves in the process. It also belong on A_{init} the number of initial AAs. We have A_{init} AAs and each of them is associated with a pointcut specification. Hence, each pointcut gives the number δ_i of duplications. And each duplication is processed in order to calculate a duplicated AA. So D can be noted:

$$D = a_i . \sum_{i=1}^{A_{init}} (\delta_i + 1).c^2 + a_2$$

The composition and merging mechanism depends on the number of instances of advices generated N and thus on the number of components. Furthermore some rules, as terminal rules, cost less than others, as recursive rules. So merging cost depends on the cost of its merging rules C and there number n.

$$K = b. \sum_{i=1}^{N} n^i (1 + p_i.C(g_0, g_i))$$

Translation depends on merged and generated instances of advice complexity. So we have the model:

$$T = b.((\sum_{i=1}^{c} (p_i.c)) + l)$$

Thus the cardinality of cascade AAs must be taken into account since during a multi-cycle weaving the system does not tolerate other disruptions.

8. CONCLUSION

We presented in this paper an improvement for our approach for adaptation in the field of ubiquitous computing. Our approach is based on the concept of AAs as a means of achieving situated systems based on a horizontal architecture. AAs allow us to adapt a component based system to its environment where adaptation is a dynamic reactive composition. But there were some lacks in the field of context-awareness. With this approach we can now work on the system domain, due to the impossibility for AAs to use components instantiated by others AAs. So we proposed a multi-cycle weaving approach. Thanks to it we can now work on the system domain as well as on the environmental domain. The introduced cascade AAs are a way to build **decentralized context-adaptive systems** based on a horizontal architecture.

Acknowledgments

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

9. REFERENCES

[1] D. Athanasopoulos, A. Zarras, V. Issarny, E. Pitoura, and P. Vassiliadis. CoWSAMI: Interface-aware context gathering in ambient intelligence environments. *Pervasive and Mobile Computing*, 4(3):360–389, 2008.

[2] N. Belhanafi, C. Taconet, and G. Bernard. CAMidO, A Context-Aware Middleware based on Ontology meta-model. *Workshop on Context Awareness for Proactive Systems*, pages 93–103, 2005.

[3] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Context-aware middleware for resource management in the wireless Internet. *Software Engineering, IEEE Transactions on*, 29(12):1086–1099, 2003.

[4] R. Brooks. Elephants Don't Play Chess. *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 3–15, 1991.

[5] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, 2003.

[6] G. Chen, M. Li, and D. Kotz. Data-centric middleware for context-aware pervasive computing. *Pervasive Mob. Comput.*, 4(2):216–253, 2008.

[7] D. Cheung-Foo-Wo. *Adaptation dynamique par tissage d'aspects*. PhD thesis, UNSA, 2009.

[8] J. Coutaz, J. Crowley, S. Dobson, and D. Garlan. Context is key. *Communications of the ACM*, 48:49–53, 2005.

[9] P. David and T. Ledoux. Towards a framework for self-adaptive component-based applications. *Lecture Notes in Computer Science*, pages 1–14, 2003.

[10] A. Dey, D. Salber, M. Futakawa, and G. Abowd. An architecture to support context-aware applications. *submitted to UIST*, 99.

[11] T. Gu, H. Pung, and D. Zhang. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28:1–18, 2005.

[12] J. Hong. The context fabric: an infrastructure for context-aware computing. In *CHI*, 2002.

[13] W. X. P. P. P. J. Z. W. L. N. C. W. T. Hung Keng Pung, Tao Gu and N. H. Chung. Context-aware middleware for pervasive elderly homecare. *J-SAC*, 27(4):510:524, 2009.

[14] V. Issarny, M. Caporuscio, and N. Georgantas. A perspective on the future of middleware-based software engineering. In *International Conference on Software Engineering*, pages 244–258. IEEE Computer Society Washington, DC, USA, 2007.

[15] G. Kiczales, J. Lamping, C. Lopes, J. Hugunin, E. Hilsdale, and C. Boyapati. Aspect-oriented programming. *Lecture notes in computer science*, pages 220–242.

[16] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng. A Taxonomy of Compositional Adaptation. *MSU-CSE-04-17*, 2004.

[17] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Nahrstedt. Gaia: a middleware platform for active spaces. *ACM SIGMOBILE Mobile Computing and Communications Review*, 2002.

[18] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994.*, pages 85–90, 1994.

[19] J. Sousa and D. Garlan. Aura: An Architectural Framework for User Mobifity in Ubiquitous Computing Environments. *Software Architecture: System Design, Development and Maintenance*, 2002.

[20] J.-Y. Tigli, S. Lavirotte, G. Rey, V. Hourdin, D. Cheung-Foo-Wo, E. Callegari, and M. Riveill. WComp Middleware for Ubiquitous Computing: Aspects and Composite Event-based Web Services. *Annals of Telecommunications (AoT)*, 64, Apr 2009.

[21] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, 265(3):94–104, 1991.

[22] C. Zhang and H. Jacobsen. Resolving feature convolution in middleware systems. *ACM SIGPLAN Notices*, 2004.