HAL
open science

Aspect of Assembly: From Theory to Performance

Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Nicolas Ferry, Vincent Hourdin, Sana Fathallah Ben Abdenneji, Christophe Vergoni, Michel Riveill

To cite this version:

Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Nicolas Ferry, Vincent Hourdin, et al.. Aspect of Assembly: From Theory to Performance. Transactions on Aspect-Oriented Software Development IX, 7271, , 2012, Transactions on Aspect-Oriented Software Development IX, 10.1007/978-3-642-35551-6_2 . hal-01330258

HAL Id: hal-01330258

https://hal.science/hal-01330258

Submitted on 21 Jun 2016

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

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

Distributed under a Creative Commons Attribution - NonCommercial - ShareAlike 4.0 International License

Aspect of Assembly: from Theory to Performance

Jean-Yves Tigli[1], Stéphane Lavirotte[1], Gaëtan Rey[1], Nicolas Ferry[1,2], Vincent Hourdin[1], Sana Fathallah Ben Abdenneji[1], Christophe Vergoni[1,3], and Michel Riveill[1]

[1] Université de Nice - Sophia Antipolis (Laboratoire I3S - UNS/CNRS) 930 route des Colles - B.P. 145 06903 Sophia-Antipolis Cedex - France
{firstname.lastname}@unice.fr
[2] CSTB (French Scientific and Technical Centre for Building), 290, route des Lucioles, B.P. 209 06904 Sophia-Antipolis Cedex - France
[3] GFI Informatique, Emerald Square - Batiment 2, Avenue Evariste Galois - B.P. 199 06904 Sophia Antipolis Cedex - France

Abstract. Ubiquitous computing systems raise numerous challenges in software engineering. Among these, the dynamic variation of open ubiquitous computing environments requires continuous adaptation of applications. Aspect-Oriented Programming is a well-adapted technique to bring together independence of concerns and extensibility for dynamic adaptation. However, the dynamic adaptation has to occur within a reasonable timeframe, which requires a detailed knowledge of the weaving duration. In this paper, we introduce "Aspect of Assembly", an aspect-oriented approach to develop services- and components-based applications. Then we study the response time of the adaptation process by decomposing the weaving process. The model of the duration of the adaptation process enables us to define *a priori* constraints to meet temporal requirements for real-world applications. Throughout this paper, we illustrate our work with an actual industrial use case to provide service continuity for a hydrant worker in the water industry.

Keywords: AOP, runtime adaptation, ubiquitous computing, CBSE, consistency, performance

1 Introduction

The miniaturization of computer hardware makes the Ubiquitous Computing vision[37] come true, as many objects with computational capabilities are appearing in our daily lives. This set of smart devices can be viewed as a new kind of dynamic software infrastructure on top of which applications can be built. Most devices are not continuously used by applications, either due to their sporadic usefulness, their mobility, or their frequent failures. In order to manage the dynamic variability of this new kind of architecture, applications have to be adapted in reaction to these variations. In such task, the whole set of adaptations that may be applied to an application cannot be anticipated at design-time

[7]. The ability to *extend at runtime the set of adaptations* deals with this issue. However, it can be complex for a designer to modify the set of adaptation rules, because they can be numerous and may require expertise in many domains. For example, in a smart home application, rules can be related to security, energy consumption, presence recognition, and so on. We can take advantage of this natural separation of specific expertise in ubiquitous applications by introducing the *independence of concerns* in the design of the adaptation process. It is very close to separation of concerns addressed by aspect-oriented approaches, as long as it manages interactions [19], interferences [22] or even conflicts [32] that may appear in the weaving process.

All these features make more complex the whole adaptation mechanism. Nevertheless, the time required to adapt the system to variations of the infrastructure must be taken into account in order to enhance the user experience and make applications well suited and consistent with their actual software infrastructure. In this context, **the challenge we address in this paper is the control of the duration of such adaptation process.**

For this purpose, after presenting our aspect-oriented approach to implement adaptations capabilities, we propose a model and an evaluation of the duration of the weaving process. Our approach, called "Aspects of Assembly", targets a model of applications based on service and component assemblies. The weaving process implements adaptations of applications through compositional modifications. AOP approaches aim to modularize crosscutting concerns to modify the behavior of software. Aspect-oriented methodology is now more generic and can be applied on various kinds of targets, even structural, always using three key concepts: joinpoints, pointcuts and advice [6]. Interactions between aspects are managed by a merging mechanism embedded in the weaver ensuring the symmetry property of the weaving operation. In this context, *the contribution of this paper is to present a formal description of Aspects of Assembly and their weaver, including models and experiments on the duration of the various processes involved in the weaver, enabling us to understand and control the duration of the Aspect of Assembly weaving process.*

To detail our approach, we start by presenting a motivating scenario (Section 2) from the field of ubiquitous computing that illustrates the needs for low response time. This use case then serves as support to describe the internal mechanisms brought into play by Aspects of Assembly (Section 3). Then, these mechanisms are formally described with mathematical models and algorithms, and their performance evaluated in the context of the example application (Sections 4 and 5). We then discuss the results and comment on our approach (Section 6), present the related work (Section 7) and finally conclude (Section 8) with the main contribution of this work and its perspectives.

2 Foundations and motivating scenario

Before presenting Aspects of Assembly (AA), we will characterize the applications to be adapted (2.1) and present a case study that will highlight features and benefits of AAs (2.2).

2.1 Designing applications in ubiquitous environments

Composing services of the infrastructure. Software services are often used to encapsulate functionalities provided in ubiquitous environments [29, 36], whether they are device-based or purely software services. Indeed, they provide the required capabilities for these environments such as loose coupling, autonomy, discoverability, and composability [34].

The software infrastructure that we consider is a *dynamic* set of services. Users' mobility, and then devices' mobility, lead to frequent disconnections and network changes of device-based services. Web services for devices [33] such as UPnP (Universal Plug and Play) or DPWS (Device Profile for Web Services) address this issue by providing a dynamic decentralized discovery mechanism. Accordingly, we will consider software services as well as device-based services. Applications should then be created as compositions of the *available* services of the infrastructure.

Component assemblies have proven to be a good solution to dynamic composition of services in ubiquitous computing [33, 17]. In particular, components can be instantiated when services appear and destroyed when services disappear, thanks to service discovery announcements [34]. Obviously, applications have to be based on dynamic component models, providing application's dynamic extensibility thanks to compositional adaptation [25].

The implementation of the compositional adaptation mechanism presented in this paper is the Aspect of Assembly weaver of our WComp[4] framework, which is based on the dynamic and lightweight component model SLCA (Service Lightweight Component Architecture) [34]. The example below is based on this implementation. However, AA are a generic compositional adaptation mechanism which can rely on other implementations of SLCA or on other component models, as long as they make explicit the links between components and they provide an entity that can be used as an external interface for dynamic reconfiguration of the component assembly.

Temporal properties of adaptations. As we have seen, adaptation allows us to cope with the changing software infrastructure. However, this adaptation process can be time consuming, especially when it must comply with constraints of the field of ubiquitous computing such as independence of concerns or dynamic modification of the set of adaptation rules. In particular, our adaptation mechanism, the AA weaver, will have to manage, automatically and at runtime, interferences arising between adaptation rules. We will present in Sections 4 and

[4] http://www.wcomp.fr/

5 response times provided by AA. Below, we will explain why providing low response time is important illustrating the problems that may arise with high response times.

Adaptive applications are always in one of the three states depicted in Figure 1. State (1) is the normal execution state of the application, where it is consistent with its environment. State (2) is reached when a change happens in the infrastructure, whilst the application is still in the same state as before the infrastructure change. The time spent in this state is the *latency* to trigger an adaptation. During state (3) the application is in its adaptation phase and unavailable for other adaptations, and may be only partially available. Applications are considered in an inconsistent state in states (2) and (3), because the application is not in line with its environment.

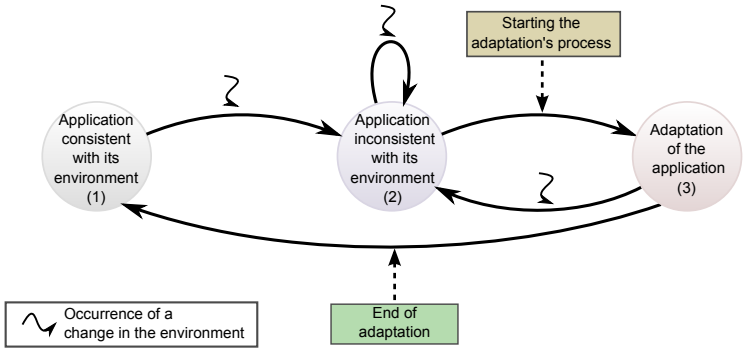


Fig. 1: States of adaptive applications [15]

The time spent in state (2) and (3) is the response time of the adaptation. It must be low in order to adapt application's functionalities consistently with available entities (services or devices) that may appear or disappear. The latency (2) must not be too high, in order to prevent the use of outdated or unavailable services. Moreover, if the system is not available for too long while adapting (3), the application can become unusable. And if new variations occur in the software infrastructure during an adaptation, the application would be continuously re-adapting and never reach state (1). Such phenomena is denoted in Figure 1 by the cycle that appears between states (2) and (3). Introducing such lag in applications could divert the user from using the system [24]. These temporal properties will motivate our scenario.

2.2 Example application

We will now present the case study that will be used throughout the paper to present our approach. We first explain the scenario, then we describe some adaptation rules and how they integrate in the application.

The Scenario. The scenario is taken from the CONTINUUM project[5] from the French national research agency. It takes place in the context of a hydrant worker whose task is to close various valves in a water pipeline network, for the purposes of maintenance operations on the network. When undertaking the action of closing valves, our mobile worker is surrounded by a constellation of devices to help him in his task. For example, GPS and compass information from his helmet are used to guide him towards the location of the valve he needs to operate. In case several valves are in proximity, he can use his smartphone or tablet to get information about which one is to be operated. When he does not find a valve, he has an augmented-reality helmet providing him location information. When he operates a valve, sensors located on the pipe or the valve allow the hydrant worker to know if there is a pressure problem and if he needs to stop before a burst happens. His car is also fully equipped to interact with the information system of the company and creates a local area network for all the devices. In case of emergency, a hydrant worker can be helped by a better equipped working truck embedding new devices and adaptations.

This scenario illustrates the properties emphasized in the introduction:

- ***Response time*** requirements appear as the hydrant worker moves in the physical environment. He will discover or activate new devices that he wants to use readily. He does not want to wait for the system to detect that he has activated or deactivated some functionality of his wearable computer. The system must comply with the user, not the opposite. Moreover, if the hydrant worker starts to operate valves and the system has not yet taken into account the valves' sensors, alert systems will not be on-line, possibly not letting the worker know about potentially dangerous situations. Those sensors are powering up when the valve wrench is near or touching the valve to save energy, making response time particularly important in this scenario.
- *Dynamicity* of the software infrastructure is the major concern in the CONTINUUM project. The hydrant worker moves from a situation to another, making use of different sets of devices. In his car, he will not use the same devices than when he's walking to find a valve, and the same is true when he's operating a valve. The main goal of the project is in fact to provide service continuity while devices change, meaning that the user should be offered appropriate functionalities whatever the devices surrounding him. Thus adaptations have to be triggered according to software infrastructure changes.
- *Dynamic modifications of the set of adaptations* arises first when new or updated devices are given to the workers. In that case, the company also delivers them new AAs without necessarily verifying their integration with already existing ones. Then, at runtime, the ability to modify the set of adaptations is also required when a special emergency and a better equipped working truck arrives. It embeds devices unknown by the worker's computer system, because they are developed by another branch of the company, and

[5] http://continuum.unice.fr/

are mostly used by workers other than hydrant men. For example, they provide large screen displays enabling hand-free visualization of alerts or of the state of the water network at the current location.

Example of Aspects of Assembly. To explain AA mechanisms and evaluate their response time, we focus in this paper on a small though relevant part of the application, which makes use of interference management between adaptation rules. This will allow us to evaluate, subsequently, the impact in terms of response time of this mechanism. First, an AA is written to provide a connecting logic from the water pipe pressure sensors to an alert system (Fig. 2).

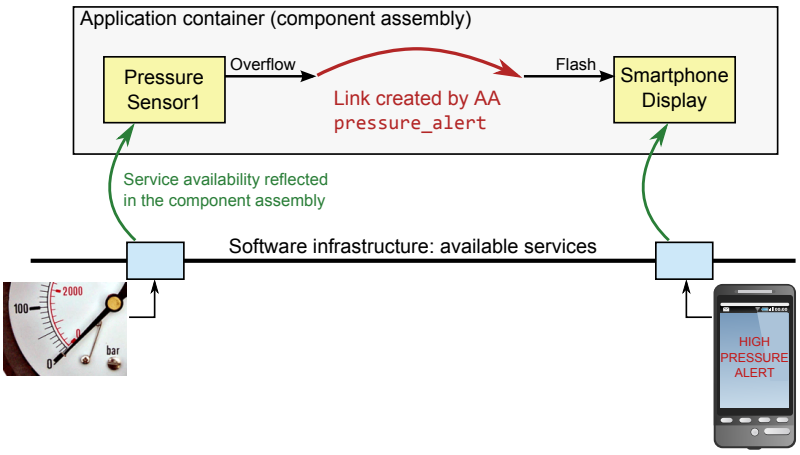


Fig. 2: Illustration of the resulting application from AA `pressure_alert` weaving and the underlying software infrastructure

Another AA describes that the pressure alert should be audible instead of visual when the brightness is too high (Fig. 3). A similar example in the scenario is the selection of the display device based on the battery level of the handheld device.

2.3 Reconfiguration description language

The reconfiguration descriptions used in AA are based on three types of rules corresponding to structural modifications: component creation, link creation and link rewrite. In this paper we will use the ISL4WComp [9] language, but other domain specific languages can be created by applications designers if needed. As an example, we have developed another language based on graphs transformation [13]. ISL4WComp is based on the Interaction Specification Language (ISL) [3] that describes patterns of interactions between independent objects, adapted to consider interactions based on messages or events between components. This

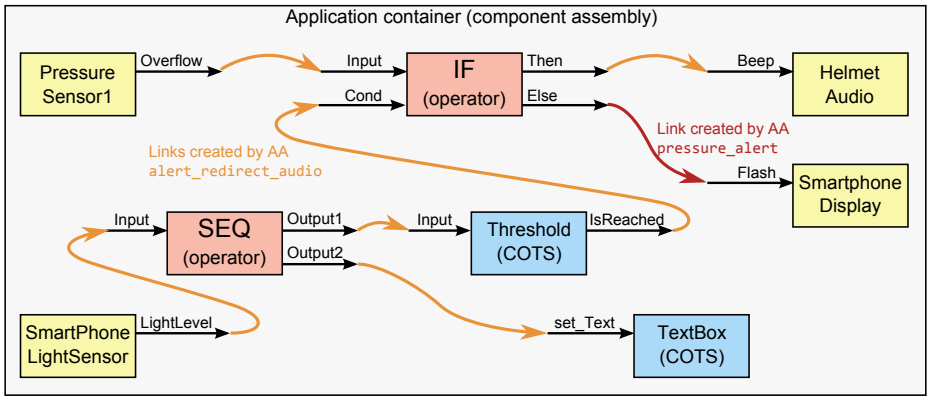


Fig. 3: Illustration of the resulting application from AAs `pressure_alert` and `alert_redirect_audible` weaving, with interference managed

language has been used in several national projects, including the RNTL Faros[6] project and the ANR Continuum project.

The keywords and tokens of the language are presented in Table 1. Black-box component creation rules are identified by the "`:`" token; link-based rules are identified by the "`->`" token. As in the reconfiguration to link the pressure sensor to the visual alert system , a simple link is created with the port `PressureSensor1.^Overflow` as source.

```
PressureSensor1.^Overflow -> (SmartphoneDisplay.Flash)
```

other reconfiguration, to redirect the alert on an audio device when the brightness is too high, can be written as in Figure 4. Lines 1 and 2 are component creation rules, with optional parameters, here the threshold for the light level considered too bright for visual interaction. Lines 4 and 6 are respectively a link creation and a link rewrite rule, distinguished by the fact that the rule at line 6 contains the `call` operator that allows rewriting existing links. The PressureSensor1.^Overflow port is thereby rewritten in the `else` branch of the `if` operator. The resulting application does not integrate the direct link specified by the first AA because of the `if` operator, as shown in Figure 3. Using language operators, such as `call` or `delegate`, a designer can control the manner in which rules are composed. The sequence operator, denoted by the "`;`" token, is also used in Figure 4, in order to create two links from the same output port.

In the next section, we will present the main features of Aspects of Assembly. The weaving process will be described to better understand how applications are adapted according to existing AAs.

[6] http://www.lifl.fr/faros – retrieved Sept. 2011

```
threshold : 'WComp.BasicBeans.Threshold' (threshold=500)
textbox : 'System.Windows.Forms.TextBox'

LightSensor1.^LightLevel -> (threshold.Input ; textbox.set_Text)

PressureSensor1.^Overflow -> (if (threshold.IsReached)
                                 { AudioHelmet1.Beep }
                                 else { call })
```

Fig. 4: Reconfiguration describing the adaptation logic replacing the visual alert of the pressure monitoring by the audible alert when the brightness is too high.

Table 1: ISL4WComp simplified grammar, with keywords and operators

	Advice rules / Operators	**Description**
Port types	*comp.port*	'.' separates the name of a component instance from the name of a port. It describes a provided port.
	comp.^port	'^' at the beginning of a port name describes a required port.
Rules for structural adaptations	*comp* : *type*	To create a black-box component
	comp : *type* (*prop* = *val*, ...)	To create a black-box component and to initialize its properties.
	provided_port → (required_port)	To create a link between two ports. The keyword → separates the right part of the rule from the left part.
	required_port → (required_port)	To rewrite an existing link by changing the destination port.
Operators (symmetry property, interference resolution)	... ; ...	To describe a sequence between two links.
	... || ...	To describe that there is no order (parallelism).
	`if (condition)` {...} `else` {...}	The `condition` is evaluated by a black-box component
	`nop`	The link is discarded, take no action.
	`call`	To allow the left part of a rule to be reused in a rewriting rule.
	`delegate`	To specify that a link must be unique in case of shared joinpoint interference.

3 Aspect of Assembly Overview

Aspects of Assembly [33] consist of a model based on AOP [23] for adaptation schemes, and a weaving process with logical merging. They are described using the concepts of *pointcut* and *advice* from AOP with some deviations. Advices describe the structural reconfiguration of a component assembly, whereas pointcuts

match joinpoints from the assembly on which changes will take place. Joinpoints are all entities of the assembly that structurally represent the application: components and their ports. We define AAs as follows:

$$AA_i = (pointcut_i, advice_i)$$

They permit the structural reconfiguration of component assemblies at runtime, whilst keeping the "black-box" property of existing components and of off-the-shelf components (COTS) that they may instantiate. The component assembly is thus not a black-box, as well as composite components since their structure is modified.

Figures 5 and 6 are the two AAs presented in our scenario: `pressure_Alert` (Fig. 2) and `alert_redirect_audible` (Fig. 3). We can see in the definition the keyword `advice` separating the pointcut definition from the advice definition.

```
pressure = Press*.^Overflow
flash = *.Flash

advice pressure_alert(pressure, flash) :

pressure -> (flash)
```

Fig. 5: AA `pressure_alert` describes the simple adaptation logic connecting the pressure device to the flashing alert device.

```
brightness = *.^LightLevel
pressure = Press*.^Overflow
sound_alert = *.Beep*

advice alert_redirect_audible(brightness, pressure, sound_alert) :

threshold : 'WComp.BasicBeans.Threshold' (threshold=500)
textbox : 'System.Windows.Forms.TextBox'

brightness -> (threshold.Input ; textbox.set_Text)

pressure -> (if (threshold.IsReached) { sound_alert } else { call })
```

Fig. 6: AA `alert_redirect_audible` describes the adaptation logic replacing the visual alert of the pressure monitoring by the audible alert when the brightness is too high.

In the next sections we will present the main features of AAs: the pointcut (3.1) and advice (3.2) parts of their definition and what they allow in terms of adaptation; state-diagram management of AAs in the weaver by selecting them or not for adaptation (3.3); the symmetry property of advice rule operators enabling independency between AAs (3.4); and the weaver's main algorithm (3.5).

3.1 Pointcuts

The pointcut part of an AA i, $pointcut_i$, is a set of filtering rules on the joinpoint's meta-data (port name, types, ...), each associated with a variable that identifies them and receives their result. The *pointcut matching* process uses these filters to identify and then to fill each variable with a list of joinpoints in the component assembly. The occurences of the variable in the advice of the AA will be replaced by these actual joinpoints. Such approach allows us to specialize the generic adaptation they describe to an adaptation targeting an actual application. It allows AAs to be reused, and apply to real applications despite their configuration being unknown at design-time. The AA weaver (3.5), which is the mechanism responsible of the adaptation evaluation and enforcement, has to maintain a list of available joinpoints in the application to allow such matchings. A pointcut is defined as follows:

$$pointcut_i = \{Rule_1, \ldots, Rule_j\}$$

Pointcut rules can take various forms, such as syntactic matching using regular expressions on component or port names or other information like the ports' types. In the two example AAs above, filters are defined as simple syntactic expressions containing wildcards, and they filter component and port names, separated by a dot. The "^" token is used at the beginning of a port name to denote that it is a required port, generally an event. Variables, the left part of a rule, are associated with each of these filters, for example `pressure` and `flash` in the first AA (Fig. 5). Simple wild-cards are used in this example but more complex pattern matching algorithms based on full regular expressions can be used. Component instances' names and component ports' names can both be wildcarded in the same rule, like in line 3 of the second AA (Fig. 6). For example, line 6 of AA `pressure_alert`:

```
pressure -> (flash)
```

will be replaced at runtime by real component ports such as (see also Figure 2 representing the application's component assembly):

```
PressureSensor1.^Overflow -> (SmartphoneDisplay.Flash)
```

Moreover, an AA can be applied several times on the same component assembly when a pointcut rule identifies several joinpoints. In the case of our scenario, AA `pressure_alert` will be duplicated if there are several pressure

sensors or flashing displays available. For the pointcut rule at line 3 of the example, the result would typically be: `sound_alert = {HelmetAudio.Beep, SmartPhone1.BeepWarning}`.

Beyond allowing joinpoints from the actual application to be used in the advice's adaptation rules, pointcuts are used as a prerequisite for weaving an aspect. In our example, if no component with audio capabilities is found, the AA will not be woven at all.

3.2 Advices

The advice part of an AA i, $advice_i$, is not a piece of code to be woven into the base application code like in traditional AOP. It can be considered as component assembly factory. To do this, an advice is composed of a set of rules. They define which components or bindings between components have to be instantiated. The advice's rules can use the variables defined in the pointcut part of the AA, so that the generated assemblies are based on joinpoints from the pointcut. Advices are based on three types of rules: 1) instantiation of black-box components, 2) rewriting of existing links between components of the assembly, and 3) creation of new links. Rewriting a link involves specifying a destination component port, and all existing links connected to that port are forwarded to the new input port described in the rule. Another type of link rewrite is what we saw in the example, using the `call` operator in a link creation rule. An advice is defined as follows:

$$advice_i = \{ARule_1, \ldots, ARule_w\}$$

An advice rule cannot *remove* a component, because if this component is used by another aspect, changing the order of application of the two aspects would result in different adapted applications and in the loss of the symmetry property. Removal of components or bindings can thus happen only if an AA is withdrawn. An AA is withdrawn when a component (corresponding to a service that disappears in the software infrastructure for example) required for its weaving disappears or when it is removed from the weaver. In our two AA from the scenario, advices are written using ISL4WComp. The next section presents details on the various states in which an AA may be.

3.3 AA selection for adaptation and state diagram

Inputs of a weaving process are 1) the assembly of the original application, called the *base assembly*, and 2) a set of AAs.

The base assembly is required in order to be synchronized with the weaver's own model of it on which all adaptations will be computed and interferences managed before being projected on the running assembly. The set of aspects is not static and can be dynamically extended. The single result of the weaving process is the final assembly corresponding to the adapted application. The final assembly is projected in terms of elementary modifications - add, remove

components, bind, unbind ports. The underlying adaptive execution platform is in charge of achieving these modifications. Each weaving operation is processed on the base assembly, free of any previous AA adaptation. We will describe the various processes involved in a weaving operation in the next section (3.5), but we can still explain how it is triggered:

- Selection or deselection of AAs given as inputs. An AA registered in the weaver can be activated or not, depending on its *selection* status. If a new AA is selected or if an already applied AA is deselected, a new weaving process is triggered. Selection can be done by users if they know what functionalities they want to use or by external applications, generally based on the users' preferences or context.
- Appearance or disappearance of joinpoints in the base assembly. This is typically caused by a new service appearing or disappearing in the software infrastructure of the application. Only AAs that can be applied to this new assembly (i.e. which satisfy pointcut matching) will be woven.

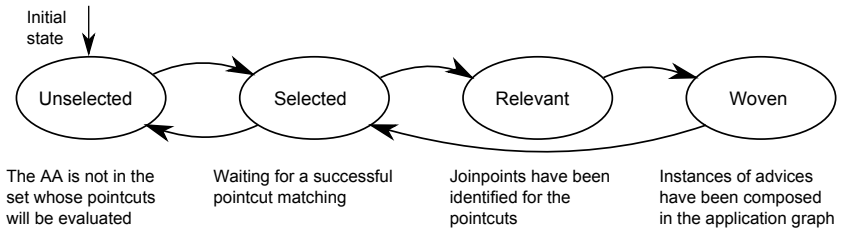


Fig. 7: AA state diagram

Therefore, the state diagram of an AA goes through several states (Figure 7). Originally, when the weaver loads it, an AA is in an *unselected* state. In that case, its pointcuts are not even evaluated in weaving processes. When it is *selected*, its pointcuts are evaluated in weaving processes and a process is started. As long as it stays *selected*, it will be evaluated on each following process. If some joinpoints satisfy all the pointcut rules of the AA, it becomes *relevant* before being *woven*. AAs that were initially not *relevant* can thus become *relevant* only when a new component appears in the application assembly. Similarly, those that have been *woven* can become disapplicated when a joinpoint matched by their pointcut disappears, in which case they will still be *selected*, waiting for their relevance to change.

3.4 Symmetry property

More than a simple mechanism for compositional adaptation of assemblies, the AA weaver allows managing interferences that may occur when several AAs are applied simultaneously to the same assembly. Interference appears when two

(or more) adaptations target a shared port, hence its name, *shared joinpoint interference*. Moreover, during the application of several AAs, the weaver guarantees the property of symmetry of the weaving operation, namely idempotency, commutativity, and associativity [9].

Classically, aspect languages provide mechanisms for adding behavior to pointcuts, by means of the operators *after*, *before* and *around*. These mechanisms represent an externalized way of managing interferences between aspects. With AAs, this is no longer necessary. As an example, the merging mechanism using ISL4WComp [9] is based on language operators with a well-known semantics (Table 1). Their composition ensures a consistent result, as well as the property of symmetry between AA's weaving operation. Section 5.2 will present more details on how the merging operation of adaptation rules works. Interferences between AAs are then managed internally. Thanks to the merging mechanism, all AAs are provided at the same time, without giving any order to the weaver, and are *superimposed* (section 5.1).

This non-ordered weaving and symmetric merging addresses the issue of the combinatorial explosion of generated assemblies introduced in [27]. Without these properties, the adaptation system would need to compute all possible combinations between AAs based on the weaving order of each AA, and then choose the best combination according to a given situation. With these properties, for a given set of AAs and an assembly given as input to the weaver, the result will be the same (deterministic) and there is no history of AA application. This is particularly important in ubiquitous computing, since we cannot know the order in which components will appear or disappear from the assembly, and thus the order of application of adaptation rules. Thus, the major point is that when this property is combined with the merging mechanism, it improves aspect's independence since there is no order and no relations between aspects.

Now that we have presented AAs principles, we will present the various processes used in a weaving operation, and for each of them we will study their response time in the following sections.

3.5 Weaving process: Weaver's Internal Operations

The weaver and a weaving process are described in Figure 8. The first step of a weaving process is the pointcut matching ① for selected AAs. It takes as input the pointcut section of an AA ($pointcut_i$) and the list of joinpoints ($Jpoint = \{port_{00}, \ldots, port_{nz}\}$ where z is the identifier of the port from component n) from the base assembly. It produces a list of results for pointcuts $LJPoint = \{l_1, \ldots, l_j\}$, and each result l_k is a set of joinpoints satisfying the rule $Rule_k$. This list of joinpoints can then be filtered ②, for instance to prohibit weaving on some ports. In our example, matching ports with `Beep*` could lead to `BeepNbMSec` being matched; however we would filter it out because it takes a beep length argument that we are unable to provide here. The result of this filter is a new list of joinpoint lists with a cardinality of internal lists smaller or equal to that of the original lists.

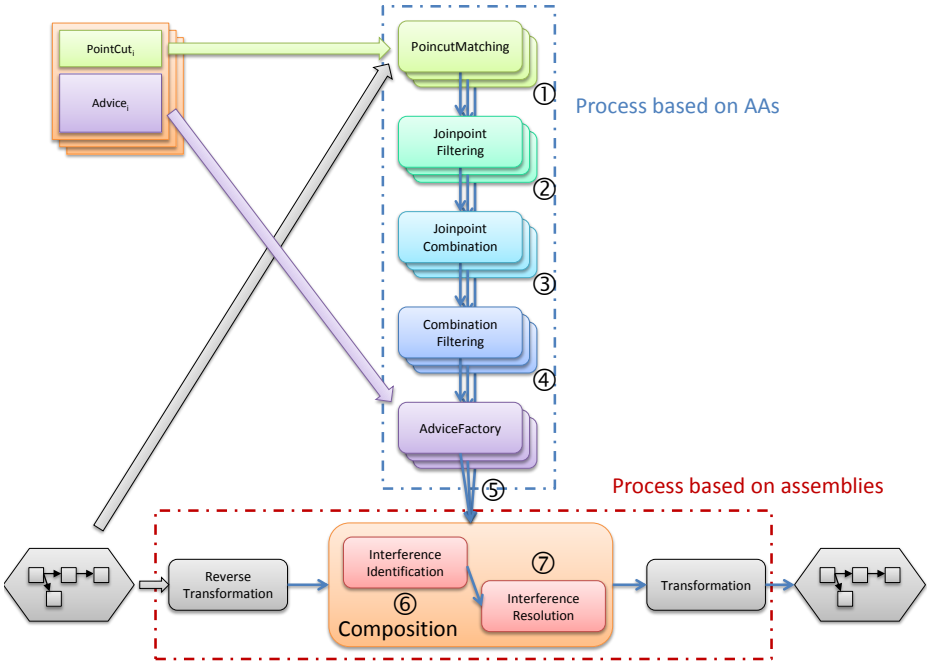


Fig. 8: AA Weaver description

To be instantiated, each advice requires at least one matched joinpoint for each pointcut rule. If many joinpoints satisfy the same rule, the advice can be applied to all combinations of joinpoints from the various lists. These combinations, that will impact the scope of the aspect, are calculated by the joinpoint combination mechanism ③. This mechanism can be implemented using various algorithms, as we will see in Section 4.2. From the list of joinpoints obtained during the pointcut matching, each algorithm must produce a new set of lists $JPointComb$. The cardinality of a combination is equal to the number of pointcut rules. Any combination with different cardinality cannot be applied. To illustrate this, the first AA of our case study cannot be woven unless both pointcut rules *pressure* and *flash* are matched. With the application depicted in Figure 2, one combination will be returned by the joinpoint combination mechanism: {{`PressureSensor1.^Overflow, SmartphoneDisplay.Flash`}}.

The set of joinpoint combinations, $JPointComb$, can also be filtered ④ to remove some joinpoint combinations. This is useful when an AA should be duplicated for all combinations but one. We don't use it in the hydrant man scenario though. The resulting list is the input of the advice factory.

The advice factory ⑤ generates instances of advices, from advice definitions and joinpoint combinations. Instantiation is done by replacing the pointcut variables used in an advice by actual joinpoints from a combination. There are thus as many instances of advices as joinpoint combinations. Instantiation transforms

an abstract description of an adaptation into a concrete one. In fact, it will later be shown that there is an equivalence between an instance of advice and a component assembly (Section 5).

These first five operations from pointcut matching to the advice factory, referred to as *AA processing flow*, are duplicated for each AA given as input to the weaver. This ensures a high degree of modularity and allows any step of the flow to be changed, according to the AA being processed. For example, by modifying step ③, it is possible to modify the scope of an AA, by modifying step ①, the pointcut definition language can be modified. AA processing flows can be seen as a single composite operation, taking AA definitions and a base assembly, and returning a list of adaptation rules. The response time of the five operations of the processing flows will be evaluated in Section 4.

Subsequently, all results of these processing flows are superimposed. In parallel, the base assembly is transformed into an instance of advice G_0 (a model of the base assembly) to be composed with those created by the advice factories. We call this process the *reverse transformation*. The composition mechanism takes the set of instances of advice and produces a single instance of advice G_F representing the adapted assembly. There are two steps in the composition engine: the first step is used to superimpose instances of advice and to identify potential interferences ⑥, and the second step, the merging engine ⑦, is used to resolve these interferences. These two operations' response time will be evaluated in Section 5.

Each operation will be described both algorithmically and in terms of complexity. Since response time is our main concern, we will compare the mathematical models derived from the algorithms with values from experimentation. These experiments were conducted on a standard laptop computer (Athlon X2 1.6 GHz, 512MB RAM). The weaver has been implemented using the WComp adaptive execution platform as a component assembly, one component for each functionality presented above. The execution platform is in charge of implementing the elementary modifications provided by the weaver at the end of the weaving process.

4 Aspect processing

In the weaver, the processes ranging from pointcut matching to the advice factory are executed independently, for each AA. They form what we call *AA processing flows*. Their purpose is to produce instances of advices from the AAs definitions, which will then be seen as components assemblies in the composition process.

4.1 Pointcut Matching

Pointcut matching can be seen as a function parameterized by the pointcut rules defined in the AA, and produces lists of joinpoints that satisfy each rule (Figure 9) from the list of joinpoints of the current application. If one of these lists is empty, it ends the AA processing flow for this AA.

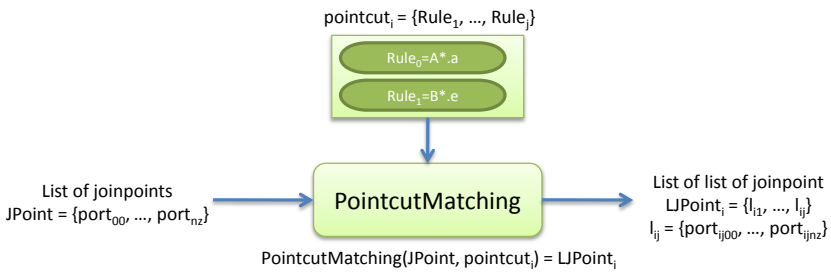


Fig. 9: Pointcut Matching

Model. In a first step, the weaver retrieves all ports of the base assembly and builds the set of joinpoints, called $JPoint$. It then retrieves all the pointcut sections of the AAs. For each AA, all the rules are evaluated on all ports of the assembly, in order to build a list of joinpoints *component.port* satisfying each rule. Figure 10 presents an example of pointcut matching based on our scenario. This operation is described more formally, as written in Algorithm 1, whose complexity is given by:

$$O(j \ \times \ card(JPoint) \times \textit{cost of the satisfy function})$$

The evaluation of a port with respect to a rule depends on how the rules are defined. An implementation example is the definition of pointcut rules using regular expressions (3.1). In that case, the complexity of a rule evaluation is $O(ml)$, where m is the size of the expression and l the size of the word. The complexity of the Pointcut Matching function will thus be:

$$O((j \times card(JPoint) \times O(ml)) = O(j \times card(JPoint) \times m \times l)$$

From this complexity, we can deduce the following mathematical model (Figure 11).

Experiments. Our experiments involved a pointcut consisting of three rules, and a set of joinpoints ranging from 0-300 elements. All joinpoints are matched by one of the three rules. Several experiments were made, and the curve of Figure 12a shows the average of these series, and the standard deviation of the values obtained. Figure 12b provides a comparison between the mathematical model and the experimental values. We identify from these experiments the following values to the model: $a1 = 10^{-6}$ and $a2 = 1,4 \times 10^{-6}$. We can conclude that the pointcut matching process is not significantly time consuming since it generally takes less than a millisecond to compute.

4.2 Joinpoint Combination

Joinpoint combination combines joinpoints that satisfy pointcut matching rules according to various policies, in order to define *how* and *where* the AA will

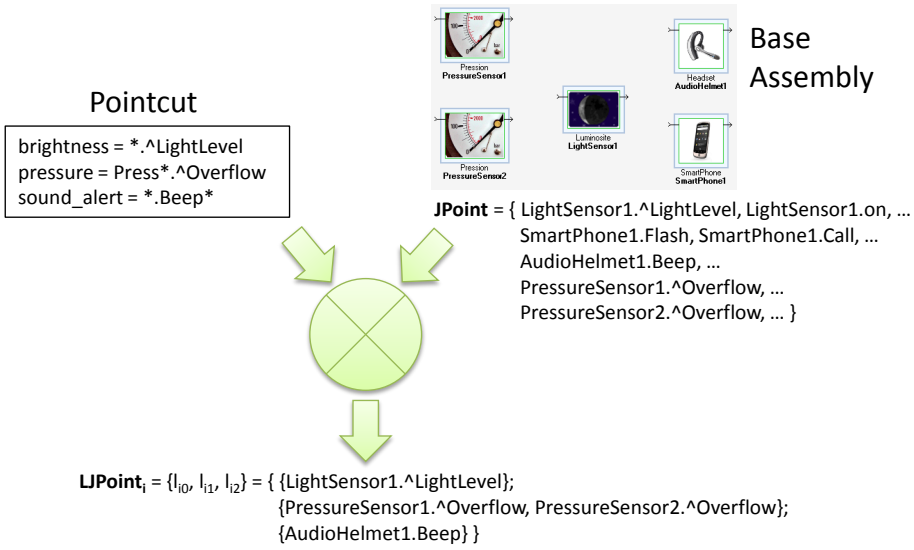


Fig. 10: Pointcut Matching Example

D : duration of the Pointcut Matching process
$a1$; $a2$: model parameters
c : number of joinpoints
i : number of AAs
j : number of rules in the pointcut section of an AA

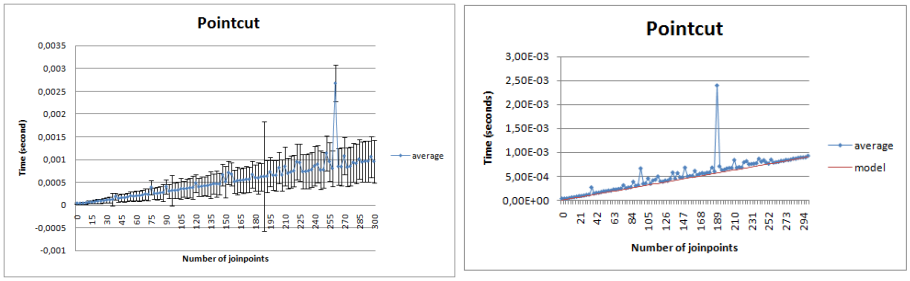
$$D = a1 \times \sum_{k=1}^{i} (j_k.c) + a2$$

Fig. 11: Duration of the Pointcut Matching process

Algorithm 1 $PointcutMatching(JPoint, PointCut_i)$

l_{ij} : a list of ports (joinpoints) where $l_{ij} = port_{ij00}, \ldots, port_{ijnz}$ and j is the number of lists, which is equal to the number of rules in $pointcut_i$
$LJPoint_i$: a set of joinpoint lists where $LJPoint_i = \{l_{io}, \ldots, l_{ij}\}$
$JPoint$: the set of ports from the base assembly $port_{00}, \ldots, port_{nz}$ y

create $LJPoint_i$
for $s = 0$ to j **do**
  Add a new list l_{is} to $LJPoint_i$
  for $t = 0$ to card(JPoint) **do**
    if JPoint[t] satisfy the rule $Rule_{is}$ **then**
      Add JPoint[t] to the list l_{is}
    end if
  end for
end for



(a) Average response time and deviation (b) Average response time and model

Fig. 12: Pointcut Matching process response time.

be duplicated. Indeed, an advice requires all variables defined in pointcuts to be associated to a joinpoint in order to be woven into the application. Combinations containing a value for each pointcut variable are thus created. An AA can be applied as many times as there are combinations of joinpoints computed from the joinpoint lists. The JPCombination mechanism can be seen as a function which builds a new set of lists $JPointComb = \{Combi_1, \ldots, Combi_j\}$ from the joinpoint list $LJPoint$ (Figure 13).

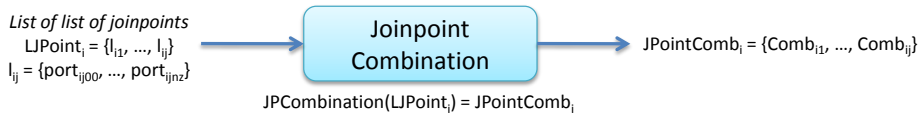


Fig. 13: Joinpoint Combination

The joinpoint combination function computes all the places in the assembly where AA_i's advice can be applied. This mechanism and pointcut matching allow the AAs to take benefits from AOP by providing high reusability, and minimizing the dispersion of code.

An important point is that various combination algorithms can be designed. It would make no sense to build all possible combinations for some adaptations, for example we may want to duplicate our second AA (Figure 6) for all existing pressure sensors, but not for all existing light sensors. Another example combination algorithm is to combine joinpoints according to their names, by firstly sorting each list of joinpoints, and then combining a joinpoint from each list according to their index in the list. Moreover, since AA processing flows are duplicated for each AA in the weaver, they can use a different algorithm. Thus, a designer can manage part of the scope of an AA by associating it to a specific combination algorithm.

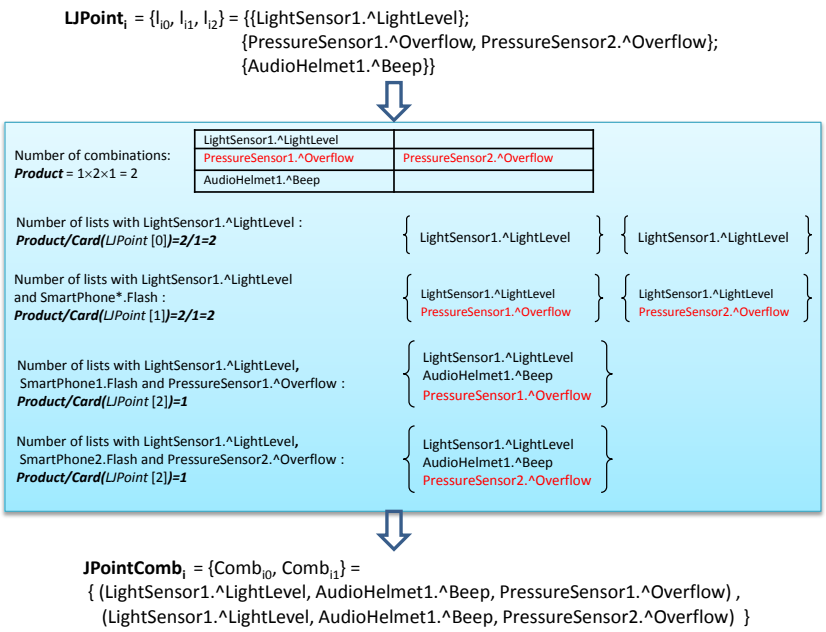


Fig. 14: Example of joinpoint combination

Figure 14 presents combination processing for our case study, based on the algorithm providing all the possible combinations. The input data of the combination function in this example are those obtained in the previous example (Section 4.1). In that case, two combinations are created, since two joinpoints are associated with the variable *flash*.

Model. This algorithm of joinpoint combination calculating all possible combinations is presented as Algorithm 2. The complexity of this algorithm is:

$$O(card(JPoint)^j).$$

The computational cost of this function depends on the algorithm used. It may be lower, since calculating all possible combinations is the worst case. From this complexity, we can deduce the mathematical model described in Figure 15.

Experiments. Our experiments involved the algorithm providing all combinations, a pointcut consisting of three rules, and a set of joinpoints ranging from 0-300 elements. All joinpoints are matched by one of the three rules. Several experiments were made, and the curve of Figure 16a shows the average of these series, and the standard deviation of the values obtained. Figure 16b provides a comparison between the mathematical model and the experimental values. We identify from these experiments the following values to the model:

C : duration of the joinpoint combination process
$a1$; $a2$: model parameters
$JPoint$: set of joinpoints
i : number of AAs
j : number of rules in the pointcut section of an AA

$$C = a1 \times \sum_{k=1}^{i} (card(JPoint)^{j_k}) + a2$$

Fig. 15: Duration of the joinpoint combination process

Algorithm 2 JPCombination($LJPoint$)

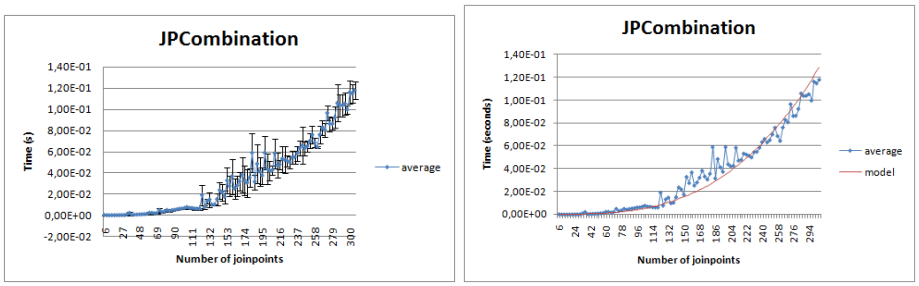
```
ACombination : list of joinpoint
Product : Integer : number of possible combination
mult : Integer : number of combination using the joinpoint
lcomb : list of combination

mult=1;
create JPointComb
for i = 0 to card(LJPoint) do
   Create lcomb
   ACombination.Clean
   product = product/(card(LJPoint[i]) − 1)
   for j = 1 to card(LJPoint[i]) do
      for k = 0 to product do
         ACombination.Add(LJPoint[i][j])
      end for
   end for
   for j = 1 to mult do
      lcomb.Add(ACombination)
   end for
   JPointComb[i]= lcomb
   mult = mult × (card(LJPoint[i]) − 1)
end for
return JPointComb
```

$a1 = 0.45 \times 10^{-6}$ and $a2 = 1 \times 10^{-6}$. We can conclude that the joinpoint combination process can be time consuming when it involves the generation of all possible combinations, with computing times up to 120ms for 300 joinpoints.

4.3 Filters

Before instantiating advices using joinpoint combinations, some of the combinations may be removed from the list, because they don't fit user's preferences, because they are not relevant to the application, or even because they are known to be semantically conflicting. Another filter can be placed on the list of joinpoints matched by the pointcut matching operation, in order to prevent some



(a) Average response time and deviation (b) Average response time and model

Fig. 16: JoinPoint combination processing response time

joinpoint from being part of adaptations at all. This process is described more formally, according to algorithm 3. The complexity of this algorithm is:

$$O(j \times card(JPoint) \times cost\ of\ the\ filter\ function).$$

Algorithm 3 Filter

```
j : number of combinations
bool filter(joinpoint) : the filtering function

for s = 0 to j do
  for t = 0 to card(LJPoint_i[j]) do
    if filter(l_is[t]) then
      l_is.remove(t)
    end if
  end for
end for
```

The cost of the *filter* function can be constant if it matches a simple joinpoint name, thus making the filtering operation $O(j \times card(JPoint))$. If *filter* matches the current joinpoint with a list of joinpoint names, the complexity is multiplied by the length of this list. Because filters are not mandatory in weaver operations, these components will not be further investigated in this paper.

4.4 Advice Factory

The Advice Factory builds instances of advice from the list of joinpoint combinations (Figure 17). It links an advice, which is an abstract representation of an assembly, to the base assembly. It thus creates as many instances of advice as possible, according to the list of combinations. It consists in replacing variables from advice rules, with the joinpoints from each of the combinations.

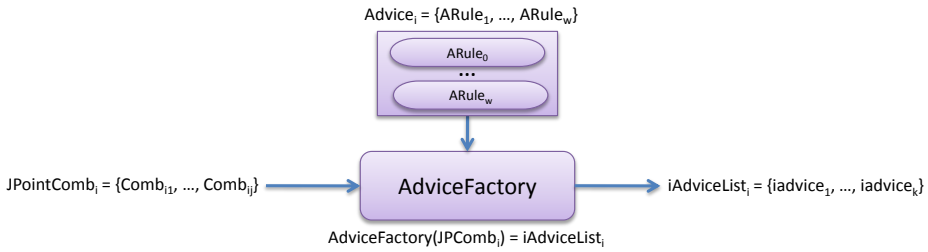


Fig. 17: Advice Factory Example

Model. The Advice Factory produces a list of instances $iAdviceList_i$ of advice $advice_i$. This process is described more formally in Algorithm 4. The complexity of the replace function is constant after grammar parsing of the rules, making the complexity of the instantiation operation as follows:

$$O(k \times w)$$

Algorithm 4 AdviceFactory($JPointComb_i$)

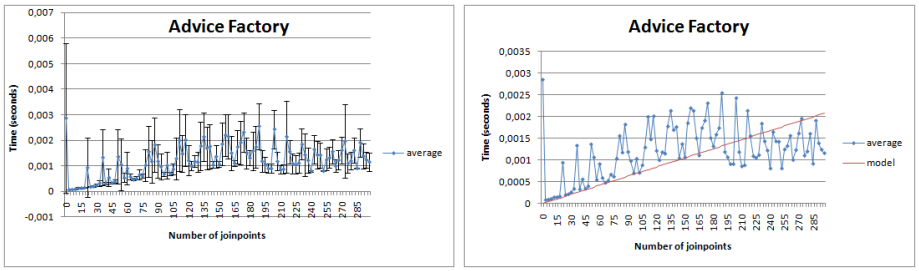
```
k : number of combinations
w : number of advice rules

for s = 0 to k do
  for t = 0 to w do
    Replace variable from ARule[t] using JPointComb[s]
  end for
end for
```

From this complexity, we deduce the mathematical model described in Figure 18.

$$A : \text{duration of instance of advice generation}$$
$$k : \text{number of combinations}$$
$$w : \text{number of advice rules}$$
$$i : \text{number of AAs}$$
$$a1; a2 : \text{model parameters}$$
$$A = a1 \times \sum_{p=1}^{i}(kw_p) + a2$$

Fig. 18: Duration of instance of advice generation



(a) Average response time and deviation (b) Average response time and model

Fig. 19: Advice Factory processing response time

Experiments. Our experiments involved a number of combinations of three joinpoints ranging between 0 and 100. Several experiments were made, and the curve in Figure 16a shows the average of these series, and the standard deviation of the values obtained. Figure 16b provides a comparison between the mathematical model and the experimental values. From these experiments, we identified the following values for the model: $a1 = 3 \times 10^{-6}$ and $a2 = 1 \times 10^{-6}$. We can conclude that the AdviceFactory process does not have a strong impact on the response time, since it instantiates advices in less than 2ms in these inflated conditions.

Overall, among the processes performed on AAs, the production of joinpoint combinations may be the most expensive. However, according to the combination algorithm associated with an AA, considerable improvements can be achieved. As an example, in the field of ubiquitous computing, we generally use an algorithm to combine joinpoints according to their meta-data (for instance to combine devices that are in the same room), whose complexity is lower. We thus conclude that the AA processing flows inside the weaver have a small impact on the adaptation's response time. We will now study the composition of the instances of advices with the base assembly and the result of the adaptation.

5 Assembly processing

Once the instances of advices have been generated, the AAs are no longer used in the weaving process. The weaver then works on instances of advice which are regarded as component assemblies to be composed with the base application. Indeed, an instance of advice $iAdvice_i$ is composed of a set of components C and a set of bindings L. A binding connects an input and an output port together. Since all the joinpoints and the ports from instantiated components are references to actually existing ones, an instance of advice is a component assembly without open connections $iAdvice_i = (C, L)$. It can be considered as a graph instead of a language [14].

The composition mechanism considers a set of instances of advices, in order to generate a new instance of advice that will be applied to the application to adapt it. This process consists of superimposing them one after the other. The single advice G_T is built, containing all the modifications to be applied to the base assembly (5.1). This operation may make interferences appear between the rules included in the final *iAdvice*. A second operation is thus required to resolve such interferences, using a merging mechanism, and return the new assembly G_F (5.2).

5.1 Superimposition

The superimposition is an operation that builds a unique assembly from several intermediate component assemblies (instances of advice). The inputs of the Superimpose function are a set of instances of advice (*iAdviceList*) generated by the Advice Factory of each AA, and the instance representing the base assembly G_0. It produces the instance called $G_T = \{GRule_1, \ldots, GRule_a\}$, which is an aggregation of all of the rules from all of the instances of advice. Starting from this point, interferences may appear in the G_T assembly. Figure 20 gives an example of superimposition of four instances of advice instantiated from the AAs from our case study, with combinations from Figure 14. The first five rules of the new instance of advice represent the base assembly G_0 before being adapted. The following rules are added by superimposition. Note that the variables present in the advice part of each AA have been correctly replaced in their bodies by joinpoints.

Model. This process is described more formally in Algorithm 5. The complexity of this algorithm is: $O(y \times max(Card(iAdvise_i)) \times card(G_0))$. It iterates on all rules from all instances of advice and check if they are not already contained in G_0. From this complexity, we can deduce the mathematical model described in Figure 21.

Algorithm 5 Superimpose(iAdviceList)

```
y : number of instances of advice

G_T = G_0
for d = 0 to y do
   for t = 0 to card(iAdvise_d) do
      if iAdvise_d[t] ∉ G_T then
         Add iAdvise_d[t] to G_T
      end if
   end for
end for
```

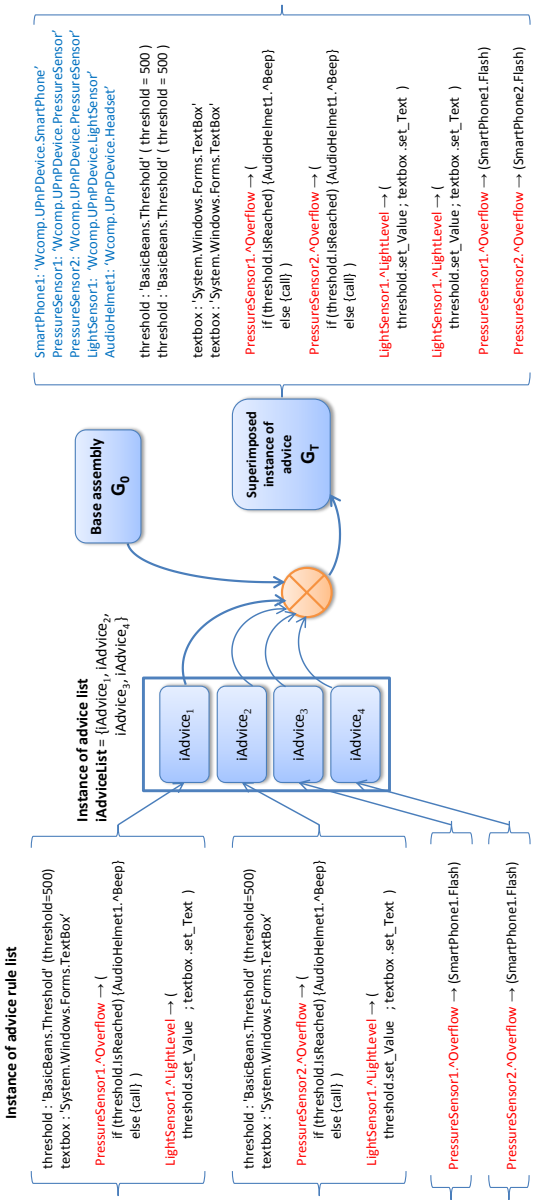


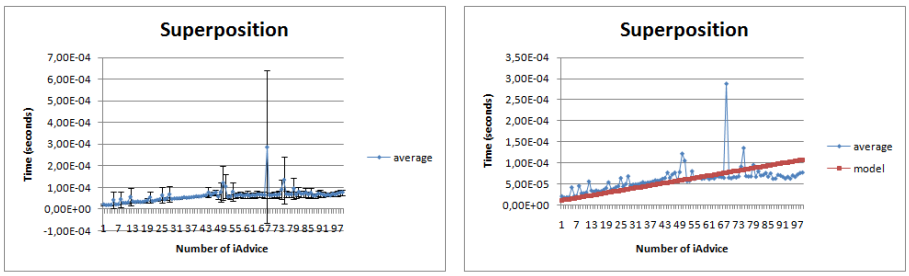
Fig. 20: Superimposition Example

Experiments. Our experiments involved a set of instances of advice, with their cardinality ranging from 0-100. These instances were composed of 7 rules, and relied on three joinpoints. Several experiments were made, and the curve presented in Figure 22a shows the average of these series, and the standard devia-

S : duration of the instance of advice superimposition
y : number of instances of advice
w : number of advice rules
g_0 : number of rules in the base assembly's instance of advice
$a1$;$a2$: model parameters

$$S = a1 \times \sum_{i=1}^{y}(w_i.g_0) + a2$$

Fig. 21: Duration of the superimposition of the instances of advice



(a) Average response time and deviation (b) Average response time and model

Fig. 22: Superimposition processing response time

tion between the values obtained. Figure 22b provides a comparison between the mathematical model and experimental values. From these experiments, we identified the following values for the model: $a1 = 1.3 \times 10^{-6}$ and $a2 = 1 \times 10^{-6}$. The superimposition operation is quite simple, and as we can see on the graphs, it takes generally less than 0.1ms to compute the superimposed instance of advice.

5.2 Interference management

The aim of interference resolution is to manage interferences occurring when several instances of advice are being woven on the same joinpoint, making what we call a *shared joinpoint*. To preserve the symmetry property of the weaving operation of AAs, it has been seen that AAs are always applied on a base assembly G_0 which is free of any adaptation. We have also seen that they cannot remove a component from an assembly and an AA cannot prevent another AA from being woven. This greatly reduces the potential for interaction between AAs. The main type of interaction is in fact the interference on a shared joinpoint. We address the automatic resolution of this kind of interference with the symmetry property of the merging operation. Consequently, the result of the weaving of several AAs is deterministic, whatever the order of their weaving.

The interference management requires first to locate the interferences on shared joinpoints. The superimposed advice G_T is browsed and modified by

introducing a specific component noted $\otimes$ as source component of link-based rules that interfere.

The merging mechanism then acts by replacing these components with components of well known semantics. Thanks to this mechanism, we can guarantee the application's consistent behavior. The merging operation can be achieved by means of various techniques, for example based on graph transformations [14], or on languages with specific operators, as ISL4WComp [8] that we present in this paper. Basically, the algorithm used to resolve interferences browses all $\otimes$ components in order to run a merging engine on them. The resulting assembly is G_F. This process is described more formally as follows (Algorithm 6):

Algorithm 6 InterferenceResolution(iAdvice)

for $s = 0$ to $card(List\otimes)$ **do**
  $Merge(List \otimes [s])$
end for

We will now present the merging mechanism associated with the ISL4WComp language, and describe how operators are merged when they interfere.

ISL4WComp merging mechanism. The shared joinpoint interference resolution mechanism is based on the operators listed in Table 1. Interfering rules are expressed in the form of semantic trees for destinations of links. Operators are the nodes of these trees and ports are their leaves. The merging of two trees consists in merging the operators, according to predefined rules as shown for some operators in Figure 23. These rules are defined in [8]. Their symmetry is the key of the symmetric merging of ISL4WComp's operators [8, 33]. The merging operation is then propagated to the leaves, solving the interference on each node.

This propagation is depicted in Figure 24, using the case study AAs as example, with the interference appearing on the *pressure* port. First, the $\otimes$ is merged with the `if` operator, by propagating in its two branches, the `then` and the `else`. In this example, there is a simple port expression in the `then` branch, with no operator. In that case, the merging stops and returns the port in the leaf. In the other case, like in the `else` branch, the merging is propagated and this has to be done recursively. The merging between the `call` and the port results in the port. It allows rewriting the existing link between *pressure* and *flash* (or any other) ports.

When two rules for the addition of two bindings do not use operators and are interfering, the result of the merging operation consists in adding a parallel operator between the two bindings. This also ensures the symmetry property of the merging operation. Finally, a rule which adds a black-box component cannot result in an interference, since an AA cannot reuse a component instantiated by another AA. Once the trees have been merged, they are transformed into

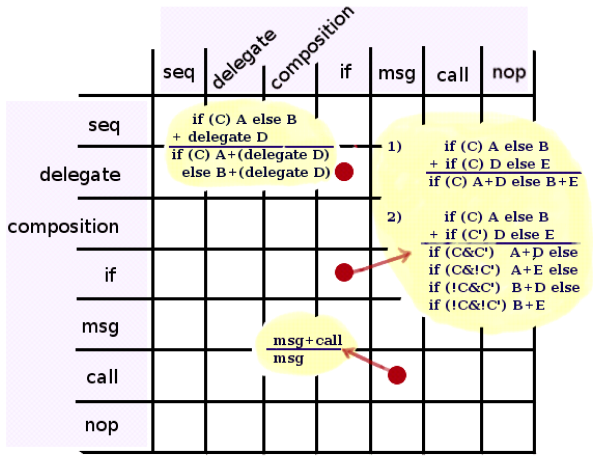


Fig. 23: Operators merging sample matrix [33]

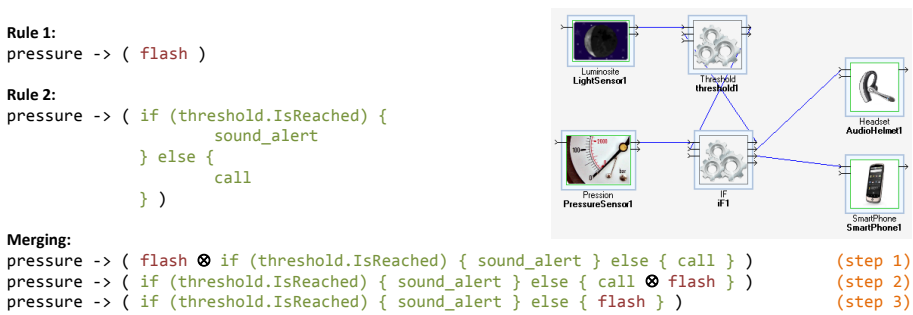


Fig. 24: ISL4WComp merging result

elementary instructions (add/remove component/binding), and the operators are then represented in the assembly by COTS with a well known semantics.

Model. Figure 25 defines the mathematical model representing the duration of the interference resolution mechanism.

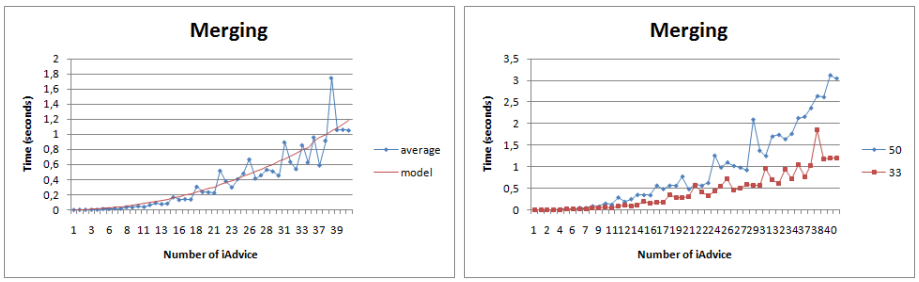
Experiments. Experiments on the merging mechanism are based on the implementation presented in [8], featuring the ISL4WComp language. They allowed us to define M as being proportional to the minimum height between the two trees to merge: $M = k_0.min(h_0; h_i)$ where k_0 is a model parameter depending on the underlying system [8], like $a1$ or $a2$ in our previous models.

Our experiments involved a set of instances of advice ranging from 0 to 50 elements. The curve shown in Figure 26a shows the experimental results of the interference resolution mechanism, with an interference probability of approxi-

F : duration of instance of advice merging
g_o : number of rules in the base assembly
y : number of instances of advice
w_i : number of advice rules
$a1$: model parameter
p_i : merging probability
$M(h_0, h_i)$: tree heights

$$F = a1.g_0 \times \sum_{i=1}^{y} w_i.p_i.M$$

Fig. 25: Duration of instance of advice merging



(a) Average response time and model

(b) Average response time with $p = 33\%$ and $p = 50\%$

Fig. 26: Interference resolution processing response time

mately 0.33 between two instances of advice. It also compares the mathematical model with the experimental values. From these experiments we identified the following value to the model: $a1 = 1 \times 10^{-6}$. The curves shown in Figure 26b describe the experimental results of the interference resolution mechanism, with interference probabilities about 0.50 and 0.33. These evaluations highlight the high cost of the merging mechanism, which represents approximately 85% of the total cost of the weaving process. The probability of interference between several instances of advice thus also plays a major role in the duration of the interference resolution mechanism.

6 Discussion of Results

We have presented models for each operation involved in processing and composition of adaptation rules in the AA weaver. The following formula (Figure 27) is the summary of the models presented in the two preceding sections. It models the time required to perform a complete weaving cycle. The algorithm chosen for joinpoint combinations in the model computes all possible combinations, which is the worst case since it creates the highest amount of duplications of each AA.

Δ_{adapt} : duration of a weaving cycle
g_o : number of rules in the base assembly
w_i : number of advice rules from AA_i
$a1$; $a2$: model parameters
p_i : merging probability
$M(h_0, h_i)$: tree heights
c : number of joinpoints
i : number of AAs
j_i : number of rules in pointcut from AA_i
p_{kj} : probability that a joinpoint satisfy a pointcut rule

$$W = a1.g_0 \times \sum_{k=1}^{i}[(j_k.c) + \sum_{z=1}^{j_k}(p_{kj}.c)^{j_k} + NBComb.w_k + \sum_{l=1}^{NBComb} w_k.p_k.M] + a2$$

$$NBComb = \prod_{r=1}^{j_k}(2^{(p_{km}.c)-1})$$

Fig. 27: Duration of the weaving process

When required, the merging process duration encompasses most of the weaving duration. The duration of a weaving cycle is equal to the sum of the durations of the various processes involved in the weaving mechanism. The AA processing flow (Section 4) is independent for each AA and can consequently be parallelized. The total AA processing time could then get closer to the time required to process a single AA processing flow. It again emphasizes the importance of the merging mechanism, which takes most of the time of a weaving cycle.

Figure 28 was calculated using the mathematical models described above. It shows the duration of a weaving cycle, according to the number of joinpoints present in the base assembly, for the two AAs of our case study. We consider all these joinpoints to satisfy the pointcut matching, and the combinations to be generated according to the affinity of the joinpoint's meta-data. The merging engine is involved in 33% of cases. These evaluation conditions are not representative of our complete scenario, because of the number of AAs involved, which is generally approaching 20 depending on the situation, and because there won't be so many joinpoints matching the same pointcut rules. However, we can use the results of our models, derived from evaluation conditions that are easier to measure, to give response times relevant to our scenario. *The weaving of 20 AAs, having an average of 3 pointcut rules and 2 advice rules instantiated one time, with interferences appearing less than for a third of them, takes approximately 0.5s.*

The response time of the AA weaver (Δ_{adapt}) presented in this paper is only the time spent to calculate the adaptation. It is surrounded by the time required to monitor or detect a change in the infrastructure (Δ_{infra}) and the time required for the actual reconfiguration of the application once the resulting assembly G_F is calculated (Δ_{reconf}). The sum of these three times (Δ), being the time spent from infrastructure change to the application adapted to this change. It has to be low enough to prevent the application from using unavailable services, and to prevent adaptation loops from occuring with a higher frequency

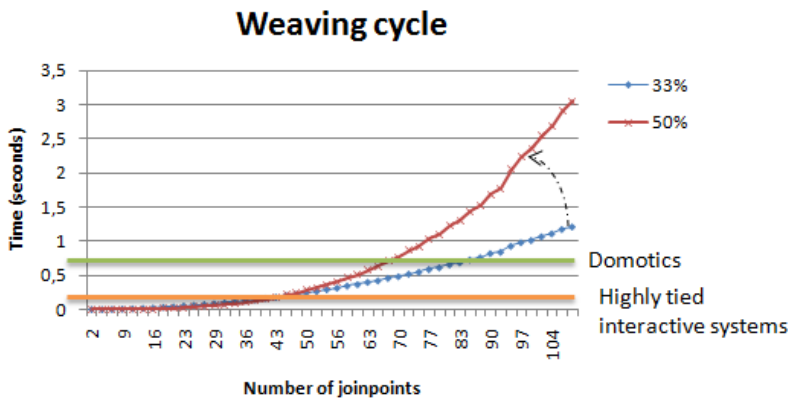


Fig. 28: Duration of weaving cycle, calculated from our model

than can be computed. During a loop, the application is not stalled for Δ, since Δ_{infra} and Δ_{adapt} can be executed by entities external to the application, in our case the AA weaver. The application may even not be completely stalled for its reconfiguration, for example parts not being reconfigured may still execute, which is the case with our component model SLCA.

$$\Delta = \Delta_{infra} + \Delta_{adapt} + \Delta_{reconf}$$
$$\Delta < T_{max}$$

In order to validate our approach, we must evaluate the conditions in which Δ is lower than the maximum acceptable response time T_{max}, and provide values for T_{max} depending on the application's domain. We were not able to find response time evaluations in other adaptation mechanisms, so we cannot directly compare our results to existing research. In the field of human computer interactions, the user latency is considered to be at most 100ms. Bérard [10] thus proposed that the latency for highly tied interactive systems should be two times smaller than the user latency: 50ms. These systems are not distributed, and infrastructural changes triggering an adaptation are local, for example, the user enabling a new functionality on his mobile phone. Δ_{infra} is thus negligible, so we have to evaluate the conditions in which $T_{max} = 50ms \Longrightarrow \Delta_{adapt} + \Delta_{reconf} < 50ms$.

Δ_{reconf} has already been evaluated for our component platform SharpWComp [34]. The instantiation of a component takes close to 3ms and the creation of a link between them takes around 4ms. When G_F is computed by the weaver to become the new assembly, chances are that a large part of it has already been created by previous weaving cycles. If we fix the application's reconfiguration to the adding of a component and three links, we get $\Delta_{reconf} = 15ms$, which leaves us with $\Delta_{adapt} \leq 35ms$. During that time, our model above allows us to state

that the AA weaver can process adaptations for an application of approximately 25 components.

On the other hand, ubiquitous computing does not necessarily require such low response times. A comparable field is home automation, in which the generally accepted latency is about 1 second. In this distributed environment, the time required to detect changes in the software infrastructure may already take a large part of that time. We thus have: $T_{max} = 1s \Longrightarrow \Delta_{infra} + \Delta_{adapt} + \Delta_{reconf} < 1s$. We have done experimentation with the UPnP service discovery, which is the protocol currently used in our SharpWComp platform. For a *device* providing one *service*, 180ms, with a 8.5ms standard deviation, are required to detect its appearance on a low traffic Wi-Fi network. It can be easily higher if the network suffers from lot of errors, or if there are more than one *service* in each UPnP *device*. To this value, we must add the time required to reflect the infrastructure change in the component assembly, by instantiating or destroying components. The first time a service is encountered, it may take up to 260ms to create the proxy component and load it in the application, but when a service disappears or a service appears again, it takes less than 10ms. It is thus difficult to evaluate Δ_{infra} properly. In good conditions it can be as low as 190ms, and in bad conditions it can be more than 2s, which is already higher than the expected Δ.

In good conditions, the scenario from the CONTINUUM application has the following values:

$$\begin{cases} \Delta_{infra} = 190ms, \text{ or } 440ms \text{ when new services are met} \\ \Delta_{adapt} = 500ms \text{ as explained above} \\ \Delta_{reconf} = 120ms \text{ for 10 components and 20 links created} \end{cases}$$

The sum fulfills the requirement $\Delta < 1000ms$ most of the time. When the network does not allow it, we probably have no control over it, but the response time of the adaptation cycle still has to be very fast to not add even more overhead to Δ. In fact, Δ_{adapt} should have the following relation to Δ, to ensure that the weaving cycle does not impact too much the overall response time:

$$\frac{\Delta}{\Delta_{adapt}} \geq 2 \Longleftrightarrow \Delta_{infra} + \Delta_{reconf} \geq \Delta_{adapt}$$

This ratio is most of the time respected, but again, it depends on a number of environmental factors. Cases in which the weaving cycle may take the larger part of the adaptation appear when the infrastructure has very good properties, for example when using an Ethernet link, or when the reconfiguration of the application is very small while there are numerous AA selected. Besides, the temporal properties concern is well addressed only if $\Delta \ll T_{infra}$, T_{infra} being the average period of infrastructure changes that trigger a new adaptation. If the infrastructure relays on a slow physical channel, there should not be services appearing or disappearing too often because it would adapt the application too frequently. It is not a problem for the adaptation mechanism, but for the user, if the application changes too often, it may not be properly usable.

7 Related Work

7.1 Dynamic Adaptation Using Aspects

Many studies propose to use aspects, with the aim of achieving dynamic adaptations. Thus, Cheng *et al.* [39] propose a mechanism for the dynamic adaptation of applications, which were not designed to be adaptable. To achieve this, a two-stage process is implemented. The first stage is implemented at design-time, with mechanisms which thereafter permit runtime adaptations of the application. The second stage involves assessing, at runtime, when to adapt and then insert (remove) code into (from) the application. Such a two-step approach would be difficult to use in the field of ubiquitous computing. Indeed, to implement adaptations, some new unforeseen modifications would be required in order to repeat the first step.

Greenwood *et al.* [18] also propose an approach for the adaptation of applications, based on AOP, but it is fully dynamic, based on policies containing ECA rules. Initially, some monitoring aspects are deployed and policies are evaluated in order to trigger various action aspects. To achieve this, they use AspectWerkz [35], and aspects are applied to objects in an invasive manner, contrary to the approach used with AAs. The weaving condition of the policies, which does not allow reactions to infrastructural changes, describes how the adaptations must be triggered. Conversely, they can take background information related to the hardware resources (CPU, memory) into account. These types of policies could be used in combination with AAs, in order to condition their selection. In this study, as for example in SAFRAN [11], the weaving process can be triggered, unlike more conventional approaches such as in EAOP [12] or Munelly [26], by events based on application control flow which are external to the application. The adaptations can thus be triggered by events related to the context of the application, referred to as exogenous events.

In [26], Munelly *et al.* propose to decompose the context into categories, and to adapt an application, using aspects, according to these contexts. Aspects are used on top of objects. Such a decomposition allows several contexts to be considered separately. However, interactions between aspects are not managed, and the contextual information is a parameter of the adaptation. Aspects are thus triggered in a classical manner, and not according to changes occurring in the context. AAs are triggered by the operational context of the application to be adapted, or by the user. Other types of context could be considered by adding context-awareness concerns to the application, or by adding a mechanism to select AAs at runtime according to the context.

7.2 Ensuring The Component Blackbox Property

There are two major types of approach using aspects for adaptation: (1) invasive approaches and (2) non-invasive approaches. When invasive aspects are used, modifications are injected into the code of the target, which is seen as

a white-box. It is then possible to perform various parameterized or compositional adaptations. Several studies, such as [18] have proposed various mechanisms to achieve invasive adaptations, or make use of Reflex [31], which is a kernel for multi-language AOP in Java. Using Reflex, an aspect can add or remove a method, a field, an annotation, and so on. Adaptation capabilities are finer-grained than those of AAs. Another example is that of Dynamic Service Adaptation [21] whose aspects are used to integrate services or to correct mobile communication services, but which are not used to make structural reconfigurations of service workflows. However, in the field of ubiquitous computing, the entities composing the software infrastructure of an application are black-boxes, since they are provided by devices and are not intended to be editable. Moreover, they are not parameterized *a priori* for adaptation, and non-invasive compositional adaptations are required.

Non-invasive compositional aspects describe adaptations that manipulate elements of the target, and then change only the interactions between these entities. Among the studies having provided mechanisms used to achieve adaptations with aspects, only some make use of non-invasive compositional aspects. Among these, we cite FAC [28], or the Plugin Architecture of Charfi *et al.* [5]. The plugin architecture is based on AO4BPEL [6], which is an aspect oriented workflow language allowing dynamic adaptation of service compositions. Since these works are applicable to workflows, they do not consider the dynamic evolution of the software infrastructure. This is however a major feature for self-adaptive systems in ubiquitous and mobile computing. In the Plugin Architecture, the problem of managing interactions between aspects is not addressed dynamically. They are handled by using the standard AOP operators: *after*, *before*...

7.3 Managing Aspect Interactions

In these previous approaches, the management of interactions between several aspects occurs explicitly, and sometimes during implementation. However, many works provide mechanisms to dynamically detect these interactions. In [31], three types of interactions, specific to structural changes, are identified. A step-wise approach is proposed to the designer, to identify these interactions and to then give him the option of solving them explicitly. Among the existing interactions, there is one particular case which can be a source of difficulties, occurring when an aspect tries to change something that is in the pointcut of another aspect. Using AAs, this type of interaction cannot occur, since the symmetry property implies the absence of order between aspects, and all of the aspects are woven onto the same base assembly. The proposed step-wise approach allows this type of interaction to be solved in various ways. However, the explicit declaration of dependencies remains difficult to imagine, in the field of ubiquitous computing. Indeed, a designer cannot predict the order in which devices, which allow a feature of the system to be provided, will appear.

Other studies also focus on the management and detection of interactions between aspects, sometimes using approaches based on graph transformation,

and using critical pair analysis. Most of them are concerned with behavioral interaction between aspects. As in EAOP [12], the authors propose mechanisms to define aspects of aspects. This mechanism allows aspects to be applied to other aspects, including a mechanism to manage recursive calls. This is done using a monitor that applies aspects sequentially. The monitor observes events from the execution of the application, and distributes them to all aspects. The architecture is sequential, such that when the base application generates an event and involves the monitor, it is stopped. Aksit *et al.* [1] suggest, for example, a mechanism to identify interference issues, in particular those on shared joinpoints. This approach is language independent. It consists of simulating and representing the various states of a program in the form of a graph, and then identifying behavioral interactions between aspects, in particular with respect to the execution order of aspects. It provides a mechanism for the detection of interference, which is more complex than that of AA's. This is partly because AA pointcuts are not concerned with the execution flow of the application but with the structure of the component assembly to be applied. A further explanation arises from the fact that compliance with the symmetry property reduces the number of different types of interaction that can be reached. For example, an AA cannot remove a component, and therefore cannot prevent the application of another AA.

This type of explicit approach to the resolution of interference between aspects is also proposed in many other studies: [18, 31, 38]. In the first, many types of interactions are considered, and are addressed explicitly into policies. Similarly, in most of the work dedicated to ubiquitous computing adaptation, this is achieved by establishing at design-time the set of configurations which can be attained by the system. As already discussed, this type of approach is hardly imaginable in the context of ubiquitous computing. We thus proposed to address the issues of order and interferences on a shared joinpoint automatically, in a non-explicit way.

7.4 Self-adaptation using aspects

In SAFRAN [11] adaptation is identified as a crosscutting concern. SAFRAN is an extension of the Fractal hierarchical component model, which was devised to facilitate the design of adaptive applications. To do this, adaptation aspects are used, which can be added or removed at runtime. The architecture of SAFRAN is comprised of 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, referred to as WildCAT. An adaptation controller is integrated into the membrane, which controls non-functional properties and interactions with Fractal components, allowing these two parts to be linked together, according to the rules, and explicitly managing adaptations dependencies.

In [2, 30], the authors propose an adaptation model which allows the selection of aspects and components, in order to reconfigure ambient systems. This selection is based on the context of the application and on various QoS criteria. This mechanism is planning based, and considers aspects as components. Although

the approach is platform independent, the aspects are intended to modify the code of components. Such an approach implies that the designer is able to write these configurations in the form of compositions of aspects and services, and the composition mechanism must consider all such configurations. This selection process is time-consuming and again, requires knowledge about all possible configurations.

In [16], Fleissner *et al.* propose to use aspects to link an application and a reasoning mechanism used for self-recovery. Aspects make it possible to make the application visible and controllable by the reasoning system. Aspects provide information concerning the application to the self-recovery system, and provide it with various mechanisms which can be used to maintain and adapt the application. To achieve this, they modify the structure of the application. However, they are based on the existence of the reasoning mechanism and are not independent adaptation entities.

7.5 Adaptation With Controlled Response Times

Response time is often ignored by projects requiring complex context processing like ontologies, for which execution time is unbounded [4], and sometimes requires several seconds to be processed [20]. This is often due to the use of knowledge bases containing the contextual information. They involve expensive and time-consuming processes to evaluate the context, whatever the type of changes occurring in the context, whereas consideration of the operational context is a prerequisite for the construction of any ubiquitous application. Indeed, the changes occurring in the operational context generally have a major impact (such as the loss of a service) on applications, and should take priority. On the other hand, the use of independent adaptation entities allows each entity to focus only on relevant information. This prevents them from having a context exploiting mechanism that can become a bottleneck.

8 Conclusions

This paper describes the Aspect of Assembly approach applied on a motivating scenario and a temporal model of the adaptation process. Aspects of Assembly use a compositional approach for adaptation of component assemblies at runtime in reaction to variations of the software infrastructure. The AA's weaver embeds a merging mechanism in order to manage shared joinpoint interferences between AA and allows maintaining the symmetric property of the weaving operation to deal with independence of concerns. The main contribution of the paper consists in the definition of a model of the duration of the adaptation process. A study of the weaver-time performance was achieved and the model of the complete duration of the weaving process allows us to predict the response time of an adaptation. We can thus predetermine limitations, in terms of computational entities involved in our dynamic adaptation, to meet specific and temporal requirements for real applications.

The weaver has been functionally decomposed and each process has been formally presented, a model of the duration of their execution has been proposed and compared to some experiments. This decomposition demonstrates that the response time is largely dependent on the merging operation. To a lesser extent, the joinpoint combination mechanism is also time consuming compared to other processes. However these two treatments should be performed at runtime since all the services from the infrastructure as well as the set of AA can vary dynamically and cannot be anticipated at design-time.

Of course, the quality of the adaptation depends on the duration of the reconfiguration of the application, but also on the relevance of its trigger. On the one hand, future work will be devoted to improve the duration of the weaving process and therefore the response time of adaptation. Secondly, we will study, more relevant triggers for adaptation.

9 Acknowledgments

Thanks to Daniel Cheung-Foo-Wo[7] for his early works on AA and evaluation of performances in his PhD Thesis who initiated this paper. This work is part of the Continuum Project (French National Research Agency) ANR-08-VERS-005. We also thank Carlos A. Varela from University of Illinois for his comments on earlier drafts.

References

[1] Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: Proceedings of the 8th ACM international conference on Aspect-oriented software development. pp. 39–50. ACM (2009)

[2] Alia, M., Beauvois, M., Davin, Y., Rouvoy, R., Eliassen, F.: Components and aspects composition planning for ubiquitous adaptive services. Software Engineering and Advanced Applications, Euromicro Conference 0, 231–234 (2010)

[3] Berger, L.: Implementation of Interaction in Distributed, Compiled and strongly typed Environments: the MICADO model. Phd thesis, University of Nice-Sophia Antipolis (Oct 2001)

[4] Bouzeghoub, A., Taconet, C., Jarraya, A., Do, N., Conan, D.: Complementarity of Process-oriented and Ontology-based Context Managers to Identify Situations. In: Int. Workshop on Context Modeling and Management for Smart Environments (CMMSE). pp. 222–229 (Jul 2010)

[5] Charfi, A., Dinkelaker, T., Mezini, M., Darmstadt, S., Darmstadt, G.: A plug-in architecture for self-adaptive web service compositions. In: IEEE International Conference on Web Services, 2009. ICWS 2009. pp. 35–42 (2009)

[6] Charfi, A., Mezini, M.: Aspect-Oriented Web Service Composition with AO4BPEL. The European Conference on Web Services (ECOWS'04) (Sep 2004)

[7] supported by CSTB during his Ph.D. in I3S laboratory

[7] Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems, Lecture Notes in Computer Science, vol. 5525, pp. 1–26. Springer (2009)

[8] Cheung, D., Tigli, J., Lavirotte, S., Riveill, M.: Wcomp: a Multi-Design Approach for Prototyping Applications using Heterogeneous Resources. In: Proceedings of the 17th IEEE International Workshop on Rapid System Prototyping, Chania-Crete (2006)

[9] Cheung-Foo-Wo, D.: Dynamic adaptation using aspect weaving. Ph.D. thesis, University of Nice-Sophia Antipolis (2009)

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

[11] David, P.C., Ledoux, T.: An aspect-oriented approach for developing self-adaptive Fractal components. In: 5th International Symposium on Software Composition (SC'06). Lecture Notes in Computer Science, vol. 4089. Springer-Verlag, Vienna, Austria (Mar 2006)

[12] Douence, R., Südholt, M.: A model and a tool for event-based aspect-oriented programming (EAOP). Techn. Ber., Ecole des Mines de Nantes. TR 2(11) (2002)

[13] Fathallah, S., Lavirotte, S., Tigli, J.Y., Rey, G., Riveill, M.: MergeIA: A Service for Dynamic Merging of Interfering Adaptations in Ubiquitous System. In: Proceedings of the Fifth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies(UBICOMM). , Lisbon, Portugal (Nov 2011)

[14] Fathallah Ben Abdenneji, S., Lavirotte, S., Tigli, J.Y., Rey, G., Riveill, M.: MergeIA: A Service for Dynamic Merging of Interfering Adaptations in Ubiquitous System. In: Proceedings of the Fifth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM). pp. 34–38 (Nov 2011)

[15] Ferry, N., Hourdin, V., Lavirotte, S., Rey, G., Tigli, J.Y., Riveill, M.: Models at Runtime: Service for Device Composition and Adaptation. In: 4th International Workshop Models@Run.Time at Models 2009 (MRT'09). pp. 51–60 (Oct 2009)

[16] Fleissner, S., Baniassad, E.L.A.: Epi-aspects: aspect-oriented conscientious software. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. pp. 659–674. OOPSLA '07, ACM, Montreal, Quebec, Canada (2007)

[17] Geihs, K., Reichle, R., Wagner, M., Khan, M.U.: Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments. In: Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems, pp. 146–163. Springer-Verlag (2009)

[18] Greenwood, P., Blair, L.: A framework for policy driven auto-adaptive systems using dynamic framed aspects. Transactions on Aspect-Oriented Software Development II pp. 30–65 (2006)

[19] Greenwood, P., Lagaisse, B., Sanen, F., Coulson, G., Rashid, A., Truyen, E., Joosen, W.: Interactions in AO Middleware. In: Proceedings of the Workshop on Aspects, Dependencies, and Interactions (2007)

[20] Gu, T., Pung, H., Zhang, D.: Peer-to-peer context reasoning in pervasive computing environments. In: Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2008). pp. 406–411. IEEE (2008)

[21] Hirschfeld, R., Kawamura, K.: Dynamic service adaptation. Software: Practice and Experience 36(11-12), 1115–1131 (2006)
[22] Katz, E., Katz, S.: Incremental analysis of interference among aspects. In: Proceedings of the 7th workshop on Foundations of aspect-oriented languages. pp. 29–38. FOAL '08, ACM, Brussels, Belgium (2008)
[23] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. SpringerVerlag (1997)
[24] MacKenzie, I., Ware, C.: Lag as a determinant of human performance in interactive systems. In: Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems. pp. 488–493. ACM (1993)
[25] McKinley, P., Sadjadi, S., Kasten, E., Cheng, B.: A taxonomy of compositional adaptation. Tech. Rep. MSU-CSE-04-17, Michigan State University (2004)
[26] Munnelly, J., Fritsch, S., Clarke, S.: An aspect-oriented approach to the modularisation of context. In: Pervasive Computing and Communications, 2007. PerCom'07. Fifth Annual IEEE International Conference on. pp. 114–124. IEEE (2007)
[27] Munoz, F., Baudry, B.: Validation challenges in model composition: The case of adaptive systems. First International Workshop on Challenges in Model-Driven Software Engineering (ChaMDE 2008) in MoDELS p. 51 (2008)
[28] Pessemier, N., Seinturier, L., Coupaye, T., Duchien, L.: A model for developing component-based and aspect-oriented systems. In: Proceedings of the 5th International Symposium on Software Composition (SC'06). Lecture Notes in Computer Science, vol. 4089, pp. 259–273. Springer Verlag, Vienna, Austria (Mar 2006)
[29] Romero, D., Rouvoy, R., Seinturier, L., Loiret, F.: Integration of Heterogeneous Context Resources in Ubiquitous Environments. In: 36th EUROMICRO International Conference on Software Engineering and Advanced Applications (2010)
[30] Rouvoy, R., Eliassen, F., Beauvois, M.: Dynamic planning and weaving of dependability concerns for self-adaptive ubiquitous services. In: Proceedings of the 2009 ACM symposium on Applied Computing. pp. 1021–1028. ACM (2009)
[31] Tanter, É.: Aspects of composition in the Reflex AOP kernel. In: Software Composition. pp. 98–113. Springer (2006)
[32] Tian, K., Cooper, K., Zhang, K., Yu, H.: A classification of aspect composition problems. In: Proceedings of the 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement. pp. 101–109. SSIRI '09, IEEE (2009)
[33] Tigli, J.Y., Lavirotte, S., Rey, G., Hourdin, V., Cheung-Foo-Wo, D., Callegari, E., Riveill, M.: WComp Middleware for Ubiquitous Computing: Aspects and Composite Event-based Web Services. Annals of Telecommunications (AoT) 64 (Apr 2009)
[34] Tigli, J.Y., Lavirotte, S., Rey, G., Hourdin, V., Riveill, M.: Lightweight Service Oriented Architecture for Pervasive Computing. International Journal of Computer Science Issues (IJCSI) 4, 1–9 (Sep 2009)
[35] Vasseur, A.: Dynamic AOP and Runtime Weaving for Java—How does AspectWerkz Address It? In: Proceedings of the 2004 Dynamic Aspect Workshop (DAW'04). pp. 135–145 (2004)
[36] Wagner, M.: Context as a service. In: Proceedings of the 12th ACM international conference adjunct papers on Ubiquitous computing. pp. 489–492. ACM (2010)
[37] Weiser, M.: The computer for the twenty-first century. Scientific American 265(3), 94–104 (Sep 1991)
[38] Whittle, J., Jayaraman, P.: Mata: A tool for aspect-oriented modeling based on graph transformation. Models in Software Engineering pp. 16–27 (2008)

[39] Yang, Z., Cheng, B., Stirewalt, R., Sowell, J., Sadjadi, S., McKinley, P.: An aspect-oriented approach to dynamic adaptation. In: Proceedings of the first workshop on Self-healing systems. pp. 85–92. ACM (2002)