A Symmetric Compositional Approach for Adaptive Ubiquitous Systems

Sana Fathallah Ben Abdenneji, Stéphane Lavirotte, Jean-Yves Tigli, Gaëtan Rey and Michel Riveill
Laboratory I3S (University of Nice-Sophia Antipolis / CNRS)
B.P. 145 06903 Sophia-Antipolis Cedex - France
{fathalla, stephane.lavirotte, tigli, gaetan.rey, riveill}@unice.fr

***Abstract*—In ubiquitous computing, systems evolve surrounded by a heterogeneous smart-devices and software services, offering functionalities that enable new applications to be created. In such system, we need to consider the unpredictability of software infrastructure changes. To tackle the issue of the dynamic variation of the software infrastructure, compositional adaptation is now often used. The problem is that adaptation entities are independent-written. In such case, they may interfere when they are composed. In this paper, we propose a formal approach that allows composing applications at run time and managing these interferences. The formal model of the system and adaptations are defined in term of graphs. In particular, we demonstrate the symmetry property of our composition process.**

***Index Terms*—adaptations composition, adaptive software, interference management, graph transformation**

I. Introduction

A ubiquitous system offers a new opportunity to augment people's lives with new technologies that facilitate their everyday tasks. It relies on heterogeneous, variable and communicating devices that are scattered in our physical environments. The software infrastructure on which is based a ubiquitous system appears dynamically populated by functionalities of these devices. Due to the mobility of devices which may appear or disappear at any time, ubiquitous applications have to be adapted in order to consider these changes. Compositional adaptation [1] allows adding or removing software elements into the applications at runtime and is well suited to handle infrastructural changes [2]. As a consequence, adaptations can be designed and integrated later during execution. By nature, compositional adaptation requires applications based on a modular architecture with a loose coupling between software entities composing it.

In order to manage all these constraints, various paradigms are used in middleware for ubiquitous computing. Component-oriented middleware and service-oriented middleware[3] provide new ways of designing dynamic and heterogeneous applications. The loose coupling between components and their execution environment control facilitates the dynamic reconfiguration of components assemblies. Accordingly CBSE is well suited for compositional adaptation [4]. Moreover, services-oriented approaches provide solutions to manage the heterogeneous devices involved in the system. Services have also the characteristic of an autonomous existence without predefined information on their use in applications, which seems appropriate with devices blackbox property. Recently, new approaches combine components and services and use components assembly to compose preexisting services such as Service Component Architecture (SCA) [5] or SLCA [6]. In this paper, we will present an adaptation mechanism that can be applied on such approaches.

A. Requirements and constraints

All the previous challenges lead to the emergence of new requirements in software adaptation. First, the adaptation must be done *at run time* (without stopping the application) in order to consider infrastructure changes. Moreover, these adaptation must tolerate the *unpredictability* of these changes. Second, *different designers* with various skills may deploy new adaptations. Accordingly, a designer cannot predict all possible adaptation of a system and adaptation entities may be developed independently. As a consequence, they must respect the *separation of concern* principle. Such independent adaptation entities have to be **composed** together and with the base application.

There are also some constraints to consider during the adaptation process. It will compose n adaptation entities in order to obtain the final application. In such case, they may interfere when they are composed. Interference is defined as *"a conflicting situation where one adaptation that works correctly in isolation does not work correctly anymore when it is composed with other adaptation"* [7]. These interferences should be managed in order to ensure the consistency of the application after the adaptation process.

Classical approaches to manage interferences consist either in defining *explicitly* dependencies between adaptation entities or either in calculating all possible combinations between the adaptations in order to choose the most appropriate [8]. Since the result of the composition may depend on the order in which adaptations are made, the number of combinations to be calculated from n adaptations may be $n!$. However, interferences must be detected and managed automatically because the number of adaptation entities that may be deployed is finite but *not bounded* (we can add adaptations entities at any time). Another reason is that it's hard for a human to integrate a new adaptation entity with existing ones while verifying the good behavior of the adaptation system. Moreover, since the changes of the software infrastructure cannot be known in advance, the order of application of adaptation entities in reaction to these changes cannot be known either. The application resulting

from adaptations should always be the same, regardless of the order in which service have appeared.

B. Our proposal

In this paper, we present an adaptation mechanism that allows composing dynamically adaptations with the base application. It embeds an automatic mechanism to manage interferences. Accordingly, there is no need for a designer to manage explicitly these problems when deploying adaptation entities or to stop the adaptation mechanism. Thus, our mechanism can be used in self-adaptation control loops. Our adaptation operation should be **symmetric** because we cannot know the order in which adaptations will be applied and deployed. It means that the order in which adaptations are composed is not important. Thus the composition operation ensuring this property is presented in this paper. Moreover, since adaptation mechanism will modify only the structure of the application, we model applications and adaptations entities by graphs. The interference resolution step will operate on the graph by applying a set of graph transformation rules.

The limitation of designer's intervention makes the challenge of maintaining the consistency of adaptation's results even more difficult. However, it seems unavoidable considering the requirements and constraints previously presented. In practice, in the proposed approach, a designer deploys its adaptations on the fly and no longer manages the interferences among adaptations. However, it may intervene in cases that are not resolved by the adaptation mechanism. Accordingly, the produced application will always be consistent.

The remainder of this paper is organized as follows: next section introduces the model of our applications and details the example that will be used all along the paper to illustrate our approach. Then section III details formally our composition process and how it addresses the issue of interference management. Particularly, we show in the section IV the symmetry property that allows us to offer a deterministic solution when we compose adaptation entities. Then, we briefly describe our implementation (section V). Finally, we present some related works (section VI) before concluding (section VII) on the contribution of this work and its perspectives.

II. Formalising Software applications and their adaptations

A. Motivating Example

As an example, we define a domestic application that relies on variable and communicating devices which define its software infrastructure. This software infrastructure appears dynamically populated by the functionalities of these devices. As a consequence, the application has to be adapted during execution time in order to consider these changes.

As all young people, Bob listens to music all the time on his Smartphone. At home, he has an adaptation that redirects the sound from his phone to any available audio device (Home cinema, speaker...). Bob lives alone. Today his mother visits him. Bob's mother had recently hearing problems; she cannot withstand the high tones. For this reason, she has an adaptation that specifies the threshold of the sound level for audio devices in her surrounding physical environment. Bob doesn't know that his mother had hearing problems. He increases the level of music. This example shows an interference problem. What is the resulting behavior when both adaptations are relevant to be applied? In this case, the system will respect the threshold defined by the mother? Or will let Bob increases the level of sound independently of the threshold. This scenario will be used throughout this paper to present our approach.

Table I
The description of primitive connectors

Notation	Description
$PAR\bullet[v_i, v_j]$	The concurrent execution of vertices v_i and v_j
$SEQ\bullet[v_i, v_j]$	The order of execution; v_i before v_j
$IF\bullet[v_c, v_i, v_j]$	Choose a path between v_i and v_j
CALL	Rewriting an existing edge
$DEL\bullet[v_i]$	The link must be unique

B. Application design in ubiquitous environments

Software services are often used to encapsulate functionalities provided in ubiquitous environments, whether they are device-based or purely software services. The software infrastructure that we consider is a dynamic set of services. Our applications are created as an assembly of the available services of the infrastructure. In particular, components are instantiated when services appear and deleted when services disappear. An appropriate structural model for our software applications (which is a component assembly) are graphs. Components are represented by vertices and their links by edges in this model.

Definition 1. *A component assembly graph G is a set of vertices V and a set of directed edges E ($G = (V, E)$). A vertex v_i of V is defined by a tuple $(Id(v_i), Typ(v_i))$ where $Id(v_i)$ is the identifier of v_i and the attribute $Typ(v_i)$ is its type. An edge e_j of E is written as $e_j = (v_i, v_k, l_j)$ where l_j is a label.*

Each vertex has a type. We separate vertices into two groups: (i) *Black box* vertices ($Typ(v)$='*Port*' and $id(v)$='*ComponentName:PortName*'), representing component ports (event and method call); (ii) *White box* vertices (or *connectors*) which determine the flow of the execution when events are triggered. The attribute $Typ(v)$ of a white box vertex indicates the kind of connectors. The table I details the set of our basic connectors.

The vertices $CALL$ and DEL are special connectors that will never be instantiated in the final application because they are used only to modify some links. This latter must be used carefully because if the developers are using it without knowing the set of possible interferences, the correctness of the resulting application can be compromised. In order to facilitate the comprehensibility of this paper, we use a lightweight representation. Each black box component will be represented by a rectangle on which we add a label on the form *Component instance Name:Port Name*. The white box vertex

will be represented by a rectangle on which we add a label on the form *Connector Name*. Each directed edge represents the execution direction between two vertices. For our motivating example, we consider that the original application (called the base assembly) was built from a physical audio device that is represented by a Black box component *AudioHome* at software level.

C. Adaptation entities as Graph Transformation rules

In this section, we define adaptation entities introduced in our example as graph transformation rules [9]. In the following, we use a visual notation to define rules. Each rule is defined in a separate figure which consist of two parts: Left (L) and Right (R). A graph transformation rule has the form of $p : L \rightarrow R$ and is applicable to a graph G_{Base} if there is an occurrence of L in G_{Base}. It will replace the subgraph L of G_{Base} by a subgraph R.

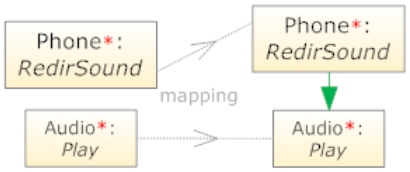

Figure 1. The graph of *RedirSound* describes the graph of the first adaptation

The *SoundRedirect* adaptation graph is depicted in the Figure 1. The vertex mapping indicate that vertices of the left side will be linked when the rule is applied. In order to be applied, this adaptation requires that there are instances of *Phone** and *Audio** components in the initial assembly (for example, if no component with audio capability is found, this adaptation will not be applied at all). A simple wildcard '*' is used in this example, but more complex regular expression can be defined (Component instances' types and component ports' names can both be wild-carded). This wildcard will be replaced by the real device type of the infrastructure. It allows these adaptations to be applied to real application although its configuration is not completely known at design time. This adaptation adds a new edge between the port *RedirSound* of the component *Phone** and the port *Play* of the component *Audio**. Through this link the *Audio** device will receive the music to play.

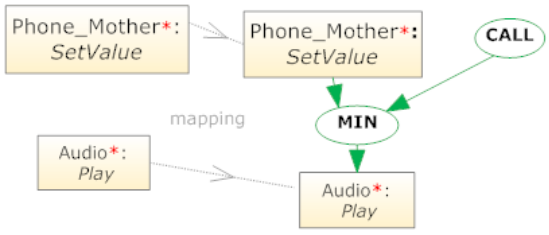

Figure 2. The graph of *Threshold Level* describes the graph of the second adaptation

The *Threshold Level* adaptation is depicted in the Figure 2. The adaptation that is not relevant to the application at the beginning, can become relevant only when its required components appear in the application assembly (components tagged in the figure by a '*'). This adaptation specifies that when an Audio device is detected, the user can define using his phone the threshold for the sound level. The Min vertex is a new connector which will be defined in the next section. It is used to forward the minimal received values from its input vertices. The *CALL* vertex allows rewriting existing links.

We consider that adaptation process will integrate a set of structural changes in the base component assembly.These changes are expressed separately by different designers and often introduce behaviors to be executed concurrently. The key idea of this contribution is to provide the formal composition of adaptation entities. This composition manages automatically interferences in a ***symmetric*** way. In the next section we show that these adaptations may interfere when they are composed. In such case, we demonstrate how *graphs* and *graph transformations rules* are used to detect and repair interferences.

III. Composition process including Interference management

A. Adaptation as Graph transformation rule execution

Previously, we have presented the adaptation entities as graph transformation rules. During the step of rule matching, The abstract description of the Left graph will be transformed to a concrete one (replacing '*' wildcard by the type of real components in the base assembly). If various components match the L graph of an adaptation rule, then it can be applied as many times as there are combinations of components instances. Each rule of adaptation is applied on the graph of the base assembly. Subsequently, all computed graphs need to be composed. The **composition** mechanism considers then these graphs, in order to generate a single graph representing the adapted assembly. Despite the order of graphs composition the final graph will be the same due to the **commutativity** and the **associativity properties**. There are two steps in the composition process: the first step *superimposes* all graphs and also identifies potential interferences. The superimposition operation builds a unique graph $G_T = (V_T, E_T)$ from the set of graph representing the initial assembly $G_{initial}$ and several graphs resulting from the *graph transformation rules applying*. When two graphs g_1 and g_2 share a vertex ($vg1$ and $vg2$ have the same Typ and Id) the superimposition operation: (1) keeps one vertex $vg1$ in the resulting graph (2) copies the incoming edges (respectively the outgoing edges) of the $vg2$ to $vg1$ (by modifying their target vertex and their source vertex to be $vg1$). Starting from this point, interferences may appear in the G_T graph.

The second phase is accomplished by the *Merging engine* which resolves these problems using other kind of *graph transformation rules*. The final graph that represents the new configuration of the system, will be exported to the adaptive execution platform.

B. Interference detection Step

The definition of interferences types is given in a graph representation depicted in Figure 3. Interference will occur only when adaptations share at least one port. We defined two types of interferences: (1) Control flow interference occurs when adaptation entities share an output port of the same component (2) Concurrent method call occurs when adaptation entities share a method call of a component. In our previous

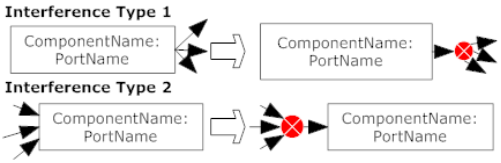

Figure 3. the two patterns of interference used by the superposition step. It adds ⊗ vertex to mark problems

work [10], we have considered only output port to detect interferences (type 1). In this paper, we introduce new *White box* vertices and we focus on a new type of interference that occurs when several adaptations try to access a shared component's ports (type 2) Therefore, we obtain a general approach that considers two interference cases: Output port and Input port.

As mentioned before, the *SoundRedirect* adaptation and the *Threshold Level* adaptation cause an interference problem when they are composed. Their superimposition illustrates an example of interference of type (2) presented above. They share the port $Play$ of the component $AudioHome$. Each adaptation sends a different value to this device. To tag this point in the graph G_T, a ⊗ vertex has been added in the Figure 4 (step a).In the next section we detail the resolution process of this problem.

C. Concurrent method call interference resolution

To resolve concurrent method call interference, we have defined new connectors (which have two predecessors vertex and one successor vertex). The first connector is $[v1, v2]\bullet FW$. When a data comes from v1 and/or v2, this connector forward it to its successor vertex (the method call). In other cases, it would be necessary to add an $[v1, v2]\bullet AND$ or $[v1, v2]\bullet XCHoice$ connectors (the $Select$ connector can be instantiated according to several strategies such as Min, Max, XOR,.. It allows to choose one data from its input vertices). Similarly to the Control flow interference resolution, adaptations entities should specify connectors that should be added for each method call. Otherwise, the merging engine will apply its default solution (FW connector).

In order to merge interfering vertices, we first need to get the list of ⊗ vertices from G_T graph. Each ⊗ vertex has exactly two predecessor' vertices. Then, we set $v1$ to the first predecessor vertex and $v2$ to the second predecessor vertex. Given two vertices $v1$ and $v2$, the merging engine starts by searching the correspondent graph transformation rule. If there is no graph transformation rule able to solve this problem, then the default solution will be applied. For example, if a designer has defined a *permissive policy*, all adaptations will be met at the same time (parallel composition). This is similar to applying a Forward *FW* connector to each adaptation graph. The default solution will replace the ⊗ vertex (which cannot be resolved) by a *FW* vertex (this rule is called *DefautRule*). Using our case study, we will show the merging operation steps.

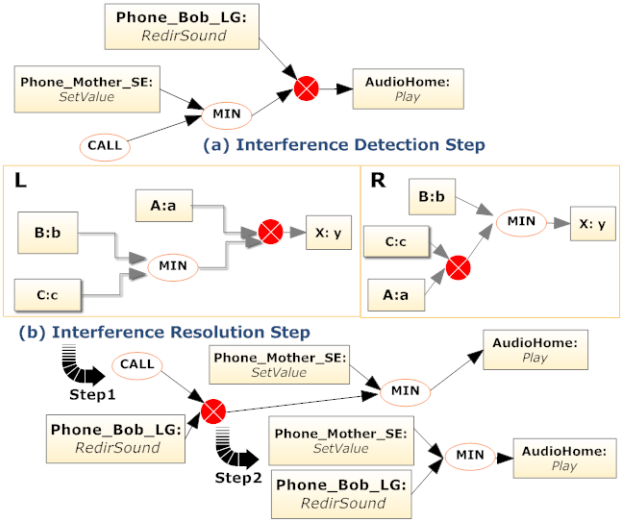

Figure 4. The Graph transformation rule (R1) and interference detection and resolution steps

In our example, interference has been detected on the port $Play$ of the component *AudioHome*. Thus, we will apply our merging process to resolve this problem. The ⊗ vertex has two predecessors vertices $v1 = Min$ and $v2 = PhoneBobLG : RedirSound$. The rule r that will be applied is given in the Figure 4.Throughout this rule the designer has defined that he wants rewrite an existent link (due to the CALL use). The trivial way to do this, is achieved by propagating $v2$ into the second branch of the Min connector. The rule will reconnect these vertices using a new ⊗ vertex. The rule R1 depicted in the Figure 4 can be applied because there is an occurrence of the left graph L on G_T. The matching step will unified the L graph variables as following: *A:a* is $PhoneBobLG : RedirSound$; *B:b* is $PhoneMotherSony : SetValue$ port. The Figure 4 (step b) shows the result of R1 execution (Step1). First, ⊗ vertex is propagated in Min branch. Since there are still ⊗ vertices, the merging algorithm will continue the resolution process. In this example, there is a CALL connector and a port. In that case, the merging will apply a graph transformation rules that will connect the port to the output vertex of the *CALL* connector. It was previously mentioned that this connector allows rewriting an existent link. The graph of components assembly of the final application is depicted in the Figure 4 Step 2. We note that $CALL$ connector will not be instantiated in the final assembly because it is used only for the interference resolution. When we define adaptation entities, this connector should be used carefully.

IV. The Symmetry property demonstration

More than a simple mechanism for compositional adaptation, our merging process guarantees the property of symmetry of the composition mechanism. This property provides a good independence between adaptation entities, that is to say that there is no need to define some explicit dependency between them since the composition process ensures the consistency of the result. Let ***C*** be the set of connectors defined in the section II. The symmetry is defined via three sub-properties:

- *Idempotency*: $\forall c1 \in \boldsymbol{C}$ and c1⊗c1=c1. Merging a connector with it self should return the same connector. This property is achieved by construction. We have specified a graph transformation rule that keep only one connector when we compose the same connector (ie., *PAR•[a,b]⊗PAR•[a,b]=PAR•[a,b] when a≠b*).
- *Commutativity* : $\forall c1, c2 \in \boldsymbol{C}$; c1⊗c2=c2⊗c1. It should not matter in which order connectors are merged.
- *Associativity*: $\forall c1, c2, c3 \in \boldsymbol{C}$; ((c1⊗c2)⊗c3) =(c1⊗(c2⊗c3)). If the merge operation is associative, then generalization to more than two connectors can be achieved merely by repeated merges, in any order.

Without these properties, the merging process must compute all possible combinations between adaptation, and then choose the more suitable combination. In ubiquitous computing since we cannot know the order in which components will appear or disappear from the assembly, we need a non-ordered composition mechanism. This is guaranteed by the symmetric property. Consequently, the result of adaptations weaving will be the same independently of their order. Particularly, we need to prove the associativity property to conclude that our composition is symmetric. In this section, we focus on the proof of this property. We will use the previous example. In order to resolve interferences at input port, we have extended the set of White Box vertices to support new semantic. Thus, for each new connector, we must prove the associativity property of its merging rules. Let $NewC$ be the new connector. $NewC$ do not necessarily need to be symmetric, but its composition with itself and other connector is symmetric (for example *SEQ* connector is not symmetric because: when *a≠b,* ***SEQ•[a,b]≠ SEQ•[b,a]***). In order to prove that the merging of $NewC$ is symmetric, we need to demonstrate that:

1) The merging of $Newc$ with itself is associative.
2) The merging of $Newc$ with all the existent connectors is also associative.

In this paper we will demonstrate this property for the rule R1 presented previously. This rule shows the merging of a port with a Min connector. Let $G1 = [a, CALL]\bullet Min, G2 = c, G3 = d$ be the set of input graphs of the composition operation where a, b, c and d are ports' components. We will compose theses graphs as fellows (Figure 5): $Comb1 = (G1 \otimes G2) \otimes G3$ and $Comb2 = G1 \otimes (G2 \otimes G3)$ and we will show that the resulting graph is the same. There are several configuration to consider during composition:

- case1: $c = a$ and $d \neq a$ and $d \neq c$
- case2: $c = d$ and $c \neq a$
- case3: $c \neq a$ and $d \neq a$ and $d \neq c$

The Figure 5 shows the merging result according to different order of composition for all defined cases.

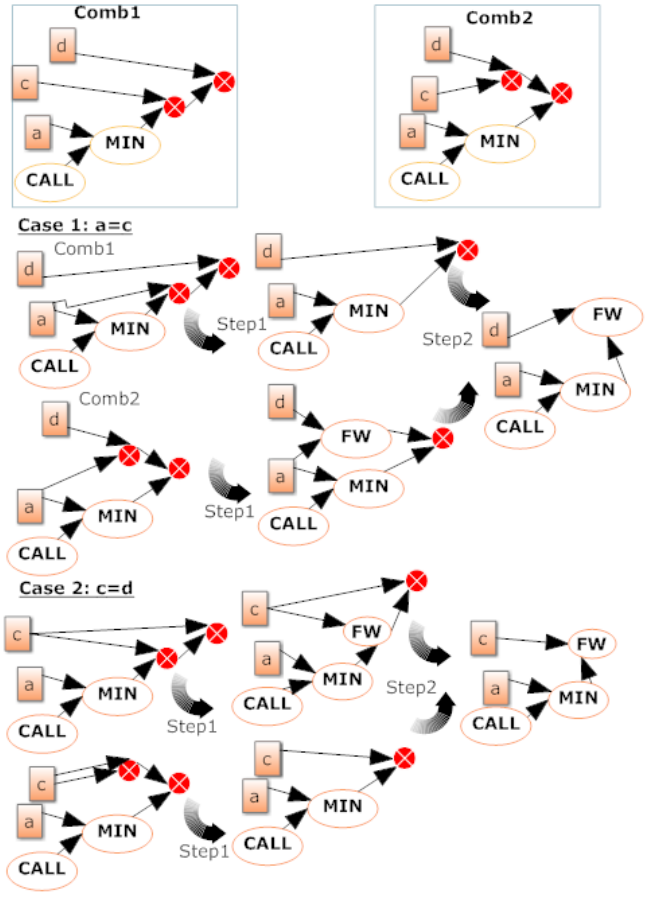

Figure 5. Proof of associativity propriety for the defined example

We conclude that the rule R1 is associative. The proofs of the associativity property is the same for all other graph transformation rules. The proof of other rules is out of the scope of this paper. In order to facilitate this task, we have implemented a tool for associativity automatic proof. The input of this tool is the definition of the $Newc$ connector and its graph transformations rules. As output, it will show in which case this property is failed (if it exists). Thus, the designer can modify the set of rules that cause problems to guarantees the property.

V. Practical Issues

The example given in this paper show that the dynamic interferences management is crucial for the adaptation in the field of ubiquitous environment. The implementation of the compositional adaptation mechanism presented in this paper is an extension of the Aspect of Assembly [11] weaver of our WComp framework, which is based on the dynamic and lightweight component model SLCA (Service Lightweight Component Architecture)[6]. Components allow the management of the black box properties of devices. The interaction

is limited to the use of their required and provided ports (the direct access to implementation is forbidden). The merging engine implementation has been presented in [10]. The merging engine uses graph transformation mechanism in order to merge vertices where interferences have been detected. Currently, the framework use AGG (Attributed Graph Grammar) [12].
Performance is a decisive factor in self adaptive systems (ubiquitous systems need to be quickly adapted to consider their infrastructure changes). For that reason, we measured the execution time of the composition operation. Then we can conclude in which case our solution can be used. The ruling that the response time is acceptable or not depends on application's domains. The results of our experiments are briefly presented in [13].

VI. Related Work

Many studies propose to use AOP Aspect Oriented Programming), with the aim of achieving dynamic adaptations. In this area, the problem of interference was well defined and several solutions were proposed. *David et al* [14] consider that software adaptation is a crosscutting concern of the application and use aspects to encapsulate adaptations code. Greenwood et al. [4] define several strategies to resolve interferences such as priority, precedence and logical operators. The strategies specifications are made by the developer who should include all dependent relationships between the adaptations. It is a complex task because we consider multi designers approach. Moreover, the strategy of interference resolution may depend on the runtime state of application. *Dinkelaker et al.* [15] propose to dynamically change the resolution strategies according to the application context. They define an extensible ordering mechanism which can be modified at runtime. Such approach still suffers limitation in term of software adaptation because interference management at runtime needs to be anticipated. In that direction, *Cheung et al.*[16] propose a composition mechanism that repairs interference problem in an anticipate manner. Since interference can occur at input and/or output of components, they propose two composition mechanisms, based on two different languages for adaptations (ISL4WComp and BSL), to handle these problems separately. But we have shown previously that these problems need to be resolved together and not separately. Their solution can resolve interference either for the output of component or for the input of component but not for both interferences types.
Graph are not only intuitive representation of software architecture but are also used to identify errors on the analysis level. The integration of the paradigm of graph with the aspect-oriented paradigm has been proposed by *Cirarci et al*[17]. They use graph formalism to identify interference. Graphs represent the several states of a program according to different order of aspect weaving. Interference is detected if the final state changes according to the selected order.

VII. Conclusion

The aim of this paper is to address the problem of interferences that may occur when we compose independently developed adaptations (but jointly deployed). In order to ensure the consistency of the final application, we propose an automatic composition process, which is able to manage interferences at run time. In this paper, we have shown that the order in which adaptations entities are composed is not important and the final result will be the same. This property allows facing the unpredictability of adaptations and the order of their deployment. As future works, we will focus on others properties for the composition operation in order to address the semantic of adaptations entities when they are composed.

References

[1] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng, "A taxonomy of compositional adaptation," *Rapport Technique numéroMSU-CSE-04-17, juillet*, 2004.

[2] A. Bouzeghoub, C. Taconet, A. Jarraya, N. Do, and D. Conan, "Complementarity of Process-oriented and Ontology-based Context Managers to Identify Situations," in *Int. Workshop on Context Modeling and Management for Smart Environments (CMMSE)*, July 2010.

[3] V. Issarny, M. Caporuscio, and N. Georgantas, "A perspective on the future of middleware-based software engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 244–258.

[4] P. Greenwood, B. Lagaisse, F. Sanen, G. Coulson, A. Rashid, and E. Truyen, "Interactions in ao middleware," in *Proc. Workshop on ADI, ECOOP*, 2007.

[5] D. Chappell, "Introducing sca," *Avalaible at http://www. davidchappell. com/articles/Introducing_SCA. pdf*, 2007.

[6] V. Hourdin, J. Tigli, S. Lavirotte, G. Rey, and M. Riveill, "Slca, composite services for ubiquitous computing," in *Proc. of the International Conference on Mobile Technology, Applications, and Systems*. ACM, 2008, p. 11.

[7] F. Sanen, E. Truyen, and W. Joosen, "Modeling context-dependent aspect interference using default logics," in *Fifth workshop on Reflection, AOP and Meta-data for Software Evolution*, no. 5, 2008, pp. 1–5.

[8] F. Munoz and B. Baudry, "Validation challenges in model composition: The case of adaptive systems," *ChaMDE 2008*, p. 51.

[9] H. Ehrig and G. Engels, *Handbook of graph grammars and computing by graph transformation: Applications, Languages and Tools*. World Scientific Publishing Company Incorporated, 1999, vol. 2.

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

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

[12] G. Taentzer, "Agg: A graph transformation environment for modeling and validation of software," *Applications of Graph Transformations with Industrial Relevance*, pp. 446–453, 2004.

[13] S. Fathallah, S. Lavirotte, J.-Y. Tigli, G. Rey, and M. Riveill, "Adaptations interferences detection and resolution with graph-transformation approach," in *the 6th International Conference Sciences of Electronic, Technologies of Information and Telecommunications(SETIT)*, ser. , Sousse, Tunisia, Nov.

[14] P. David and T. Ledoux, "An aspect-oriented approach for developing self-adaptive fractal components," in *Software Composition*. Springer, 2006.

[15] T. Dinkelaker, M. Mezini, and C. Bockisch, "The art of the meta-aspect protocol," in *Proc. of the 8th ACM international conference on Aspect-oriented software development*. ACM, 2009, pp. 51–62.

[16] D. Cheung, J. Tigli, S. Lavirotte, and M. Riveill, "Wcomp: a multi-design approach for prototyping applications using heterogeneous resources," in *Rapid System Prototyping. Seventeenth IEEE International Workshop on*, 2006, pp. 119–125.

[17] S. Ciraci, W. Havinga, M. Aksit, C. Bockisch, and P. van den Broek, "A graph-based aspect interference detection approach for uml-based aspect-oriented models," *Trans. on aspect-oriented software development VII*, pp. 321–374, 2010.