# The Dynamic Composition of Independent Adaptations including Interferences Management

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*—Nowadays, software systems are becoming ever more complex and are likely to evolve quickly. A challenge is being able to adapt these systems and thus to integrate, swap or remove new functionalities. Compositional adaptation provides a way to tackle this at runtime. This situation, leads to satisfy new requirements in software adaptation. Moreover, adaptations entities may be developed independently and even can be specified by different designers. Accordingly, the set of all adaptations that may be deployed cannot be anticipated at design time. In such case, they may interfere when they are composed. In this paper, we propose an associative composition operation to support software adaptation. In our research, we consider that the adaptation of a running system is assimilated to the composition of the model of the initial application (called initial graph) with new model elements (graphs of adaptations).**

***Index Terms*—software composition; adaptive software; interference management; graph transformation.**

## I. Introduction

Nowadays, software systems must be developed with the capacity of being able to evolve quickly [1]. Software systems must react to environments changes and consequently integrate, swap or remove new functionalities. The compositional adaptation approach [2] allows to change software units. In compositional adaptation, the applications must be based on a modular architecture with a loose coupling between the software units composing it. The loose coupling between component and its execution environment control facilitates the dynamic reconfiguration of components assembly. Accordingly, Component Based Software Engineering is well-suited for compositional adaptation [3]. We will use throughout the paper the more general term *adaptation* or *adaptation entities* to point out the modification that will be done on the application.

In this paper, we present an adaptation mechanism that allows the *dynamic* composition of adaptation entities. In order to support such composition, existing approaches consist either in defining *explicitly* dependencies between adaptation entities or either in calculating all possible combinations between them in order to choose the most appropriate [4]. 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 $n!$. In this paper we propose an associative and a commutative composition mechanism that allows to minimize the number of combination by construction. The application resulting from adaptations composition will be always the same, regardless of their order of composition. Accordingly, there is no need for a designer to explicit the order when deploying adaptation entities or to calculate combinations. Using this mechanism, adaptations are added or removed independently of each other. More specifically, all resulting **interferences** (*interactions*) between adaptations entities are handled automatically at the composition step. 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"* [5]. These interferences should be managed in order to ensure the consistency of the application after the adaptation process. Moreover, since adaptation mechanism will modify only the structure of the application, we model software applications and adaptations entities by graphs. The main reason to choose graph as a basis formalism is that it can define software architecture easily as shown in [6]. The interference resolution step will operate on the graph by applying a set of graph transformation rules.

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

## II. Related Work

Systems adaptation can be triggered by several causes. Researchers have proposed a variety of methodologies for the development of self-adaptive systems. This objective can be achieved using reflective architecture-based mechanism [7]. To fulfill software adaptation objective, there is several representation of adaptations. As a consequence, the process of the detection and the resolution of interference are related to the used method. In this section, we mention some of works that have contributed to tackle the interference problem, all other related works are out of the scope of this paper. This problem is considered as the violation of the constraints defined at design time of the system [7]. A system change is valid only if the system satisfies the constraints after the change. Otherwise, the change will be canceled. *Barresi et al.* [8] define explicitly at design time the order in which adaptation should be applied. This requires the knowledge of all system adaptation at design time. But we have seen that we need unanticipated adaptation entities that will be performed at runtime and that can be loaded and unloaded at runtime. This requirement has been included in K-Component Model [9]. *Dowling et al.* use graph transformation rules to separate the adaptation code from the computational code and to perform architecture adaptation. Furthermore, they are not interested in the problem of interference. Similarly, *Guennum et al.* [10] use graph transformation rules to perform software adaptation and propose tree operators to compose rules. These operators allow defining an order between rules or select only one rule to be executed. These operators need to be defined at design time and as a consequence require the knowledge of all possible adaptations (contradictory with our constraint).

Another related area of research that relies on software adaptation is the use of AOP (Aspect Oriented Programming). *David et al.* [11] consider that software adaptation is a crosscutting concern of the application and use aspects to encapsulate adaptations code. In this area, the problem of interference was well defined and several solutions were proposed. Greenwood et al. [3] investigate a solution to interferences in the context of AO-Middleware platform. To do that, they define *interaction contract* which is used at runtime. These contracts express several strategies to resolve interferences such as priority, precedence and logical operators (to combine contracts). Despite the use of these contracts at runtime, their specifications are made by the developer who should include all dependent relationships between the adaptations. This will be 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.* [12] 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.* [13] propose a composition mechanism that repair interference problem in an anticipate manner. Since interference can occurs at input and/or output of components, they propose two composition mechanisms, based on two different languages for adaptations, 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.

Graphs is 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.* [14]. They use the 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.

Although all of previously described approaches propose software adaptations mechanism that support interferences detection and resolution, none considers a merge process for computing reasonable system adaptation from a large set of possible adaptation, which is the main focus in this article.

## III. PRELIMINARIES

To be able to present our composition mechanism, we introduce in this section the model of our software systems.

### A. Modelling Software architecture Assemblies Using Graphs

In order to represent structural specifications of software systems, several researchers use Business Process Model (BPM), which embodies also the strategies for accomplishing software evolution. It provides high-level specification that is independent of the target platform. There exist many notations to represent BPM [15]. In this paper, we abstract from any specific notation and represent a process model as a directed graph as per in the definition 1.

**Definition 1.** *A process 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.*

**definition 2** (Successor and Predecessor). *Let $G = (V, E)$ be a directed graph. For each vertex $v_i \in V$ : we define the set of its predecessor vertices as $[v_{0i}, ..., v_{li}]\bullet v_i$ where $\{\forall v_{li} \in V, (v_i, v_{li}) \in E\}$ , and the set of its successors (or output) vertices $v_i\bullet[v_{0i}, ..., v_{mi}]$ where $\{\forall v_{mi} \in V, (v_{mi}, v_i) \in E\}$.*

Each vertex has a type and depends on the target platform language that we represent. In the remainder of this paper we consider that applications are created as component assemblies (component can be instantiated when a device appears and destroyed when the device disappears). Vertices are classified into two subclasses: (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. We introduce **5** basic connectors: *PAR, SEQ, IF, CALL, DELEGATE.* $PAR\bullet[v_i, v_j]$ connector performs the concurrent execution of vertices $v_i$ and $v_j$. $SEQ\bullet[v_i, v_j]$ defines the order of execution; $v_i$ before $v_j$. $IF\bullet[v_c, v_i, v_j]$ is used to choose a path between $v_i$ and $v_j$. If the vertex $v_c$ has the value true, $v_i$ will be performed (or selected) otherwise $v_j$ will be executed. $CALL$ connector allows rewriting an existing edge. The connector $DELEGATE\bullet[v_i]$ specifies that the link to the vertex $v_i$ will be unique in case of interference. $CALL$ and $DELEGATE$ are special connectors that will never be instantiated in the final application because they are used only to modify some links. Figure 1 shows an example of the application according to our model. 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

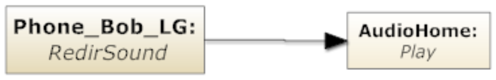

Figure 1. An example of component assembly graph

on the form $ComponentinstanceName$ : $PortName$. The white box vertex will be represented by a rectangle on which we add a label on the form : $ConnectorName$.

### B. Running Example

We will now present the scenario that will be used throughout this paper to present our approach. As an example, we define a ubiquitous 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. Inside his home, he has an adaptation that redirect the sound from his phone to any available audio device (Home cinema, speaker, etc.). 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 (reconfiguration) that will specify 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.*

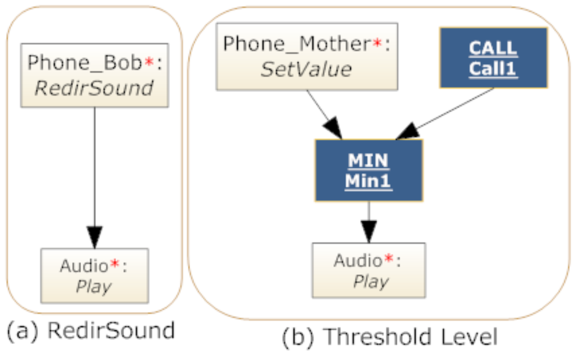

Figure 2. The graphs of adaptations defined in the running example

We consider that the original application (called the base assembly) is build from a physical audio device, which is represented by a Blackbox component *AudioHome* at software level. In our use case, there are two sub-assemblies (adaptations) linked to the *AudioHome* device. The *SoundRedirect* adaptation graph is depicted in Figure 2. In order to be applied, this adaptation requires that there are instances of *PhoneBob** and *Audio** components in the initial assembly (for example, if no component with audio capability is found, this adaptation will not be woven 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 *PhoneBob** and the port *Play* of the component *Audio**. Through this link the *Audio** device will receive the music to play. Figure 1 shows an example of instance graph of this adaptation.

The *Threshold Level* adaptation is depicted in 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 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.

In the final system both adaptations are running in parallel. Even though the adaptation of Bob's mother has delineated the threshold, when Bob will increase the sound, this threshold will be exceeded. This example illustrates a problem of a concurrent access to a shared resource (audio device). In this case, if we apply classical resolution approaches, such as a precedence strategy, the problem will not be resolved. If we apply the *Threshold Level* adaptation before or after the *SoundRedirect* adaptation the problem will persist.

## IV. A GRAPH-BASED APPROACH FOR ADAPTATIONS COMPOSITION

In this section, we detail our approach for interference resolution and the proof of the symmetry property.

### A. Approach Overview

Until now, we have presented the adaptation entities graphs. Since each adaptation entity is based on a set of required components, we need to determine the set of relevant adaptations (adaptations that can be applied). After that, we transform the abstract description of relevant adaptations into a concrete one (replacing '*' wildcard by the type of real components in the base assembly). If various components satisfy the requirements of an adaptation, then it can be applied as many times as there are combinations of components instances. The *combination function* computes all the places in the assembly where adaptations can be applied according to different strategies (build all possible combinations, combine according to components names, etc.). Subsequently, all these graphs need to be composed with the graph of the base assembly. 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 sub steps in the composition process: the first sub step *superimposes* all graphs and also identifies potential interferences. The second sub step is accomplished by the

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

The interference problem is defined through two patterns. The interference Type 1 is detected when there is a Black Box vertex that has two or more outgoing edges. The interference Type 2 is denied as a Black Box vertex that has at least two incoming edges. In our previous work [16], we have considered interference Type 1. In this paper, we introduce new *White box* vertices and we focus on a new type of interference that occurs when several adaptation try to access to a shared component's ports (Type 2).

### B. Superimposition and Interference Detection process

The superimposition operation builds a unique graph $G_T = (V_T, E_T)$ from the graph representing the initial assembly $G_{initial}$ and several graphs resulting from the *adaptation instantiation* step ($G_{Adap1}$,...,$G_{AdapN}$). $G_T = (((G_{Base} \cup G_{Adap1}) \cup G_{Adap2})... \cup G_{AdapN})$ is the graph with $V_T = V_{Base} \cup V_{Adap1} \cup ... \cup V_{AdapN}$ and $E_T = E_{Base} \cup E_{Adap1} \cup ... \cup E_{AdapN}$. 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.

Interference will occur only when adaptations share at least one port. This is due to the following constraints, which are needed to guarantee the associativity and the commutativity properties: (1) In order to preserve the independency of adaptations entities, these later can only share vertices of the base assembly, (2) Adaptations cannot remove Blackbox vertices or links explicitly (This may lead to the loss of associativity and commutativity properties). Vertices are removed only if the adaptation is withdrawn or components are not available. Thus, the execution of an adaptation cannot prevent or enable another adaptation to be performed. Thus, the potential for interference between adaptations is greatly reduced. 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 [16] we have already detailed the process of resolution of *interference Type 1*. In this paper, *(i)* we extend the set of white box vertices, *(ii)* we proof the associativity and commutativity properties for these new connectors, and *(iii)* we detail the process of resolution of interference of Type 2.

We have seen in the previous example that the *SoundRedirect* adaptation and the *Threshold Level* adaptation provide an interference problem. These two adaptations share the port $Play$ of the component $AudioHome$ with the base assembly graph. Each adaptation sends a different value to this device. The superimposition of these graphs illustrates an example of interference of type (2) presented above. To tag this point in the graph $G_T$, a special vertex has been added in Figure 3.

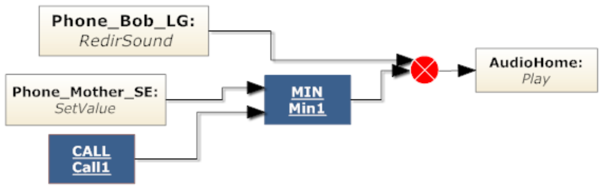

Figure 3. The Interference detection step adds a $\otimes$ vertex to mark the problem

Thus, 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. In the next section we detail the resolution process of this problem.

### C. Interference management using Graph transformation rules

The interference management operation addresses the automatic resolution of interferences with the *associativity* and the *commutativity* properties. To preserve this property, it has been seen that adaptation entities are applied on the base assembly $G_{Base}$, which is free of any adaptation. Consequently, the result of the interference management will be the same whatever the order of adaptations composition. The resolution mechanism builds its solution from $\otimes$ vertices (added previously in $G_T$ graph). We have seen that graphs are build from two subclasses of vertices: Black box and White box. Basically, the algorithm used to resolve interferences browses all $\otimes$ components in order to run a *merging engine* on them [16]. Since we work on graphs, the merging will be achieved by applying *graph transformation rules*.

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 $G_{Base}$. Using rules definition, the designer can control the manner in which adaptation graphs will be merged. Thus, the semantics of the deployed application will be affected by the logic of merging given by the designer. Given two vertices, $v1$ and $v2$ (they must share a $\otimes$ successor or predecessor vertex), we define a set of graph transformation rules that shows how they will be merged according to different possible configurations (for example if $v1$ and $v2$ share a predecessor vertex). Thus, we have defined the set of rules to merge each white box vertex with all other vertices. Compared with our previous work, we have extended the set of White box vertices and accordingly their graph transformation rules.

*1) Concurrent method call interference resolution:* This kind of interference occurs when at least two ports are linked to a port of another component in the same time. To resolve concurrent method call interference, we have defined new connectors (which have 2 predecessors vertex and one successor vertex). The first connector is $[v1, v2] \bullet FW$. When a data come 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 $XCHoice$ connector can be instantiated according 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). The merging algorithm is defined in algorithm 1.

In order to merge interfering vertices, we first need to get

**Algorithm 1** $Merging(Graph\ G_T)$

```
$ListFusvertex$ : is used to save the list of $\otimes$ vertex of $G_T$
$k$ : the number of $\otimes$ vertex in $G_T$

for $s = 0$ to $k = ListFusvertex.getSize()$ do
    $Pre$= $GetPredecessor(ListFusvertex(s))$
    Vertex $v1$= $Predecessor(0)$ ;
    Vertex $v2$= $Predecessor(1)$ ;
    Rule $r$=$SelectRule(v1, v2)$
    Boolean $res$=$ExecuteRule(r)$
    $ListFusvertex = getFusVertex(G_T)$
end for
```

the list of $\otimes$ vertices ($ListFusvertex$) from $G_T$ graph. Each $\otimes$ vertex has exactly two predecessor' vertices (saved at $Pre$). Then, we set $v1$ to the first predecessor vertex and $v2$ to the second predecessor vertex. Given two vertices $v1$ and $v2$, the $SelectRule(v1, v2)$ function starts by searching the graph transformation rule in the rule database in order to merge these vertices (using $Typ(v1)$ and $Typ(v2)$ attributes). 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 execution). This is similar to applying a Forward *FW* connector to each adaptation graph. The default solution will replace the $\otimes$ vertex (which cannot be resolved) by a *FW* vertex. Next the function $ExecuteRule(r)$ applies the selected rule $r$ to $G_T$. This rule merges $v1$ and $v2$ and removes the current $\otimes$ vertex. After this, the merging engine selects the next $\otimes$ vertex from the list $ListFusvertex$. Using our case study, we will show that the merging operation can be propagated to the successors vertices of $v1$ or $v2$ in order to solve the interference for each vertex. In such case we need to get the new list $ListFusvertex = getFusVertex(G_T)$ because some $\otimes$ vertices have been added to the graph.

Figure 3 shows an interference on the port $Play$ of the component *AudioHome*. Thus, we will apply our merging algorithm to resolve this problem. The $\otimes$ vertex has two predecessors vertices $v1 = Min$ and $v2 = PhoneBobLG : RedirSound$. The rule $r$ that will be applied is given in the top of Figure 4.

Throughout this rule the designer has defined that he wants rewrite an existent link (due to the CALL use). The trivial

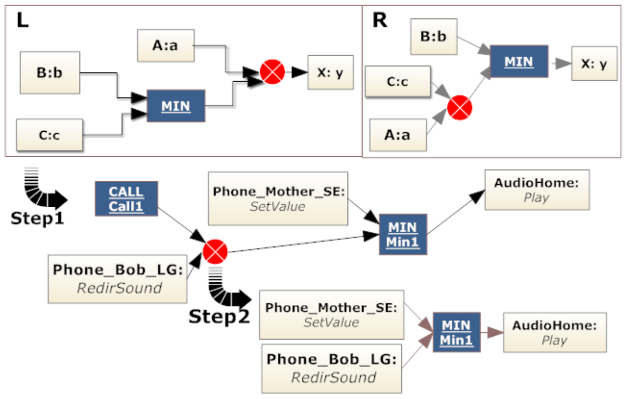

Figure 4. The Graph transformation rule (R1) and the two steps for interference resolution

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 $\otimes$ vertex. R1 rule can be applied in our example 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.

Figure 4 shows the result of R1 execution (Step1). First, $\otimes$ vertex is propagated in $Min$ branch. Since there are still $\otimes$ vertices, the merging algorithm will continue the resolution process. In this example, there are 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. We have seen that this connector allows rewriting an existent link. The graph of components assembly of the final application is depicted in 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.

## D. The symmetry propriety

Associativity and commutativity properties guarantee 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. The result of adaptations weaving will be the same whatever the order of composition. The interference resolution process must also guarantee this property when it merges interfering adaptations. Let ***C*** be the set of connectors. The symmetry is defined via three sub-properties:

- *Idempotent*: $\forall c1 \in$ ***C*** and c1$\otimes$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 (i.e., *SEQ•[a,b]⊗SEQ•[a,b]=SEQ•[a,b] when a≠b* ).
- *Commutativity* : $\forall c1, c2 \in$ ***C***; c1$\otimes$c2=c2$\otimes$c1. It should not matter in which order connectors are merged. The

function *SelectRule* presented previously, select the same graph transformation rule independently of the order of its parameter (it means *SelectRule*(c1,c2)= *SelectRule*(c2,c1)). This property is ensured at composition level by construction.
- *Associativity*: $\forall c1, c2, c3 \in$ ***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.

To grantees the determinism of the computed application and also ensures the consistency of the composed system at the implementation level, we need to prove the associativity property. 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*. A fundamental question to answer is that *"how associativity can be proved?"* To answer to this question we define steps to be done for that end. We choose as example the *new* connector $Min$ (we use the notation $Newc$). Connectors do ***not necessarily need to be symmetric*** (for example *SEQ* connector is not symmetric because: when *a≠b,* ***SEQ•[a,b]≠ SEQ•[b,a]*** but its composition with itself and other connector is symmetric). For the proof, we will require demonstrating that the merging of $Newc$ with any other connector is associative. The first step to do is the definition of graph transformation rules that specify the way of merging of the $Newc$ connector with each other connectors (we have already done the proof for all existent connectors). After that we need to prove that:

- The merging of $Newc$ with itself is associative. It means that ($Newc$⊗ $Newc$)⊗ $Newc$ = $Newc$⊗($Newc$⊗ $Newc$). For example, using our graph transformation rules, we obtain {*[a,b]•Min*⊗ *[a,c]•Min*}⊗ *[c,b]•Min=[a,b]•Min* ⊗ {*[a,c]•Min*⊗ *[c,b]•Min*}= *[a,[b,c]•Min]•Min* when *a≠b, a≠c and b≠c* (there are also other combinations to compute such as [b,c]•Min, [a,d]•Min...).
- The merging of $Newc$ with all the existent connectors is also associative. That is to say: $(Newc \otimes Oldc_1) \otimes Oldc_2 = Newc \otimes (Oldc_1 \otimes Oldc_2)$ *where both* $Oldc_1$ *and* $Oldc_2 \in$ {*PAR, IF, SEQ, DELEGATE, CALL, FW*}

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: $Comb1 = (G1 \otimes G2) \otimes G3$ 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 $d \neq c$
$Comb1 = ([a, CALL]\bullet Min \otimes a) \otimes d = [a, CALL \otimes a]\bullet Min \otimes d = [a, (CALL \otimes a) \otimes d]\bullet Min = [a, a \otimes d]\bullet Min = [a, [a, d]\bullet FW]\bullet MIN$
$Comb2 = [a, CALL]\bullet Min(a \otimes d) = [a, CALL]\bullet Min \otimes [a, d]\bullet FW = [[a, CALL \otimes [a, d]\bullet FW]\bullet Min = [a, [a, d]\bullet FW]\bullet Min$
we conclude that $comb1 = Comb2$

case2: $c = d$ and $c \neq a$
$Comb1 = ([a, CALL]\bullet Min \otimes c) \otimes c = [a, CALL \otimes c]\bullet Min \otimes c = [a, (CALL \otimes c) \otimes c]\bullet Min = [a, c]\bullet Min$
$Comb2 = [a, CALL]\bullet Min(c \otimes c) = [a, CALL]\bullet MIN \otimes c = [[a, CALL \otimes c]\bullet Min = [a, c]\bullet MIN$
we conclude that $comb1 = Comb2$

case3: $c \neq a$ and $d \neq a$ and $d \neq c$
$Comb1 = ([a, CALL]\bullet Min \otimes c) \otimes d = [a, CALL \otimes c]\bullet Min \otimes d = [[a, (CALL \otimes c) \otimes d]\bullet Min = [[a, c \otimes d]\bullet Min\ [a, [c, d]\bullet FW]\bullet MIN$
$Comb2 = [a, CALL]\bullet Min(c \otimes d) = [a, CALL]\bullet Min \otimes [c, d]\bullet FW = [[a, CALL \otimes [c, d]\bullet FW]\bullet Min = [a, [c, d]\bullet FW]\bullet Min$
we conclude that $comb1 = Comb2$

We conclude that the rule R1 is associative. The proofs of the associativity property of the merge function on other new graph transformation rules are completely identical to the proofs of the associativity shown for the fusion rule R1. We do not detail the proof of other rules. To demonstrate that the merging of each new connector is associative, we compute all the possible composition combinations (connector with itself and with existent connectors). 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 new connector and its graph transformations rules (we defined the graph of $Min$ connector and all graph transformation rules of its merging). 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. Using this tools we have proved that the composition of $Min$ connector with all previous operators is associative.

## V. Practical Issues

The compositional adaptation mechanism presented in this paper has been implemented in the WComp platform, which is a middleware for adaptation based on a dynamic composition of services based on the SLCA (Service Lightweight component Architecture) model [17] . 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). In order to manage adaptation in a transversal way to produce simultaneous modifications in different points of the application, we use a paradigm named Aspect of Assembly(AA)[18] (based on AOP principles). It produces components assembly (graphs) that will be composed. The merging engine implementation has been presented in [16] and [19]. The merging engine uses graph transformation mechanism in order to merge vertices where interferences have been detected. Various graph transformation tools exist but the most used is AGG (Attributed Graph Grammar) [20].

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 the rest of this section. Figure 5 shows the execution time of the merging step. From these experimentations, we conclude that our approach can be used in application's domain that don't require a very low response time to be adapted. For example, our composition process can be used in the area of domestic application since it composes adaptations about 1 second.

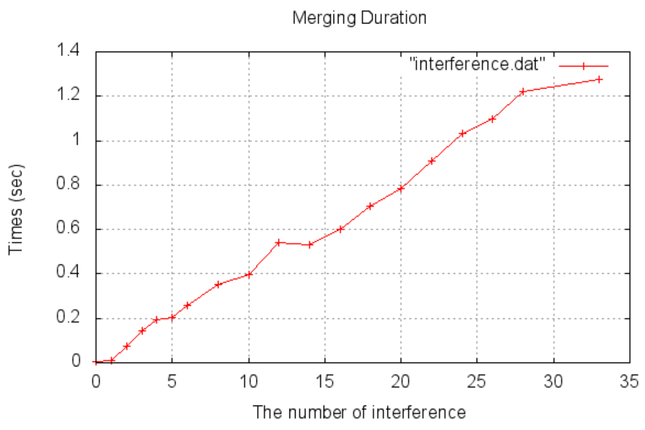

Figure 5. The execution time of the Merging process

## VI. Conclusion and future work

The work described in this paper is derived from our experience in composing independently developed adaptations (but jointly deployed). In this paper, we introduced a new connector $Min$ that enables our composition mechanism to handle the concurrent method call interference problems. Using graph representation, we formally defined the model of the running system and the set of its adaptations. Our approach performs some predefined graph transformation rules that will merge special vertices of the graph in order to resolve problems. We provide that this operation is associative as the order of adaptations compositions does not matter.

Immediate perspective of this work is to provide a well defined representation of connectors behavior. Model checking techniques will be applied to ensure that the composition of synchronous connectors does not violate the behavior of the target application.

## VII. Acknowledgments

This work is supported by the French ANR Research Program VERSO in the project ANR-08-VERS-005 called CONTINUUM.

## References

[1] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, and B. Cukic, "Software engineering for self-adaptive systems: A research roadmap," *Software Engineering for Self-Adaptive Systems*, pp. 1–26, 2009.

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

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

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

[5] 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.

[6] M. Wermelinger, A. Lopes, and J. Fiadeiro, "A graph based architectural (re) configuration language," in *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5. ACM, 2001, pp. 21–32.

[7] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[8] L. Baresi, R. Heckel, S. Thöne, and D. Varr´ o, "Style-based modeling and refinement of service-oriented architectures," *Software and Systems Modeling*, vol. 5, no. 2, pp. 187–207, 2006.

[9] J. Dowling, V. Cahill, and S. Clarke, "Dynamic software evolution and the k-component model," in *Workshop on Software Evolution*, 2001.

[10] M. Guennoun, "Architectures dynamiques dans le contexte des applications à base de composants et orientées service," 2006.

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

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

[13] 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.

[14] 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," *Transactions on aspect-oriented software development VII*, pp. 321–374, 2010.

[15] M. La Rosa, M. Dumas, R. Uba, and R. Dijkman, "Merging business process models," *On the Move to Meaningful Internet Systems: OTM 2010*, pp. 96–113, 2010.

[16] S. Fathallah Ben Bbdenneji, S. Lavirotte, J. Tigli, G. Rey, and M. Riveill, "Mergeia: A service for dynamic merging of interfering adaptations in ubiquitous system," in *UBICOMM 2011, The Fifth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, 2011, pp. 34–38.

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

[18] 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.

[19] 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.

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