Document :	*Modélisation du contexte et Adaptation* dans CONTINUUM
Sous-tâches :	2.1 et 2.2
Numéro des Livrables :	D2.1 et D2.2
Date	13/09/2012
Rédacteurs :	*Sana Fathallah (I3S), Gaëtan Rey(I3S), Jean-Yves Tigli(I3S), Stéphane Lavirotte(I3S), Nicolas Ferry(I3S), Joëlle Coutaz(LIG), Emeric Fontaine(LIG), Fabrice Jouanot (LIG), Anis Benyelloul (LIG), Marie-Christine Rousset(LIG), Philippe Renevier(I3S), Anne-Marie Pinna-Dery (I3S),Vincent Hourdin (MobileGov),*
Coordinateurs :	Gaëtan Rey

10. La Gestion des interférences entre les Aspects d'assemblage

Cette annexe présente le mécanisme de résolution des interférences intégré dans le tisseur d'Aspects d'Assemblage.

10.1 Interférence : Définition

La gestion des interférences est une étape fondamentale dans le processus de la composition d'assemblages de composants. Elle permet de résoudre le cas ou la superposition des assemblages de composants s'appuie sur les mêmes points de jonction. Il existe alors un conflit d'utilisation de ces composants logiciels. La figure 1 illustre les deux cas d'interférences considérées dans nos travaux :

- Type 1 : Un port de sortie d'un composant qui est connecté à plusieurs composants
- Type 2 : Un port d'entrée d'un composant qui est appelé par plusieurs composants

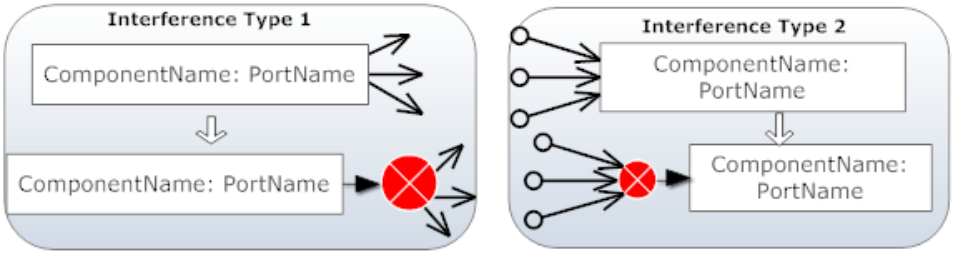

Figure 1 Les deux patrons d'interférence

Les nœuds ⊗ marquent les endroits où il y a un problème à résoudre. C'est à partir de ces nœuds que le mécanisme de résolution d'interférence va construire sa solution.

10.2 La résolution des interférences de Type 1

10.2.1 La résolution d'interférence dans la littérature

Nous étudions dans cette section les travaux sur l'adaptation par aspect et les solutions proposées pour la résolution des interférences et plus particulièrement ceux utilisant une modélisation de l'application à l'aide de graphe. (Zhang et al., 2006) proposent d'utiliser un mécanisme de précédence entre les aspects aux endroits où il y a des interférences. Cette précédence est définie au moment de la modélisation de l'application par le concepteur. Si on ajoute un aspect à l'application, l'utilisateur doit analyser l'interaction de cet aspect avec tous les autres aspects. Le problème est que cette opération devient de plus en plus complexe avec le nombre d'aspects présents dans l'application. L'approche ainsi proposée est explicite. Elle ne peut pas être utilisée en informatique ambiante car nous ne connaissons pas à l'avance les adaptations qui vont être en interférence. Dans le cadre de Continuum, la création des adaptations se fait au fil de l'utilisation du système et toutes les adaptations ne sont pas connues au moment de la conception. De plus, on ne pourrait demander à

l'utilisateur, à chaque nouvelle adaptation spécifiée, de venir insérer celle-ci dans l'ordre des priorités des adaptations existantes sur le système, tout en considérant l'ensemble des interactions possibles entre celles-ci.
Il existe d'autres travaux sur la résolution des interférences entre les aspects mais cette fois ci d'une manière dynamique. Dans (Greenwood et al., 2007), il existe deux cas d'interférence entre aspects : (1) un point de jonction commun entre les aspects et (2) un aspect nécessitant la présence d'un autre aspect pour fonctionner. Toutefois, la solution proposée est l'utilisation des contrats qui sont consultés par le tisseur à l'exécution pour respecter les relations ainsi définies entre les aspects. Par rapport à (Zhang et al., 2006), il a ajouté une autre stratégie qui considère le contexte d'exécution de l'application. Même si les contrats sont utilisés à l'exécution, la solution donnée reste explicite puisqu'elle nécessite l'intervention du concepteur de l'application. De plus, nous pouvons avoir un cas où les contrats spécifient des stratégies contradictoires. Dans ce cas de figure, ils sont tous ignorés et on perd alors les comportements spécifiés par ces aspects.
(Dinkelaker et al., 2009) ont proposé une approche de résolution dynamique des interférences. Pour ce faire, ils utilisent des comparateurs consultables et modifiables à l'exécution. Ces comparateurs peuvent être complexes c'est-à-dire combiner plusieurs stratégies selon les variables liées à l'environnement. Malgré la variété de stratégies de résolution utilisée dans ces travaux, l'approche proposée pour la résolution des interférences reste liée à l'intervention du concepteur. Il s'agit d'une résolution explicite.

Une application peut être vue comme un graphe modélisant les relations entre les composants ou services. Dans ce cadre, une adaptation est alors considérée comme une transformation de ce graphe. De ce fait, un aspect représente une règle de transformation de graphe qui est l'application. Dans ce cadre (Mehner et al., 2006) spécifient les aspects par un ensemble de règles de transformation de graphe. La détection des interférences entre aspect, donc entre les règles, est confiée au mécanisme Critical Pair Analysis (CPA) (Heckel et al., 2002) qui est capable d'analyser les règles et de déterminer celles qui sont en interférence ou bien dépendantes. Dans le même contexte, (Sadou et al., 2007) utilisent le mécanisme de transformation de graphe pour l'évolution structurelle des architectures logicielles à base de composants à plusieurs niveaux (conception, déploiement, exécution). La théorie de transformation de graphe est utilisée pour spécifier et appliquer les adaptations. En cas d'interférence, le gestionnaire d'évolution sélectionne plusieurs règles d'adaptation. Il fournit alors au concepteur la liste de ces règles d'évolution éligibles. Ce dernier doit alors intervenir pour le choix d'une seule règle d'évolution à déclencher. Il n'y a donc pas de mécanisme de résolution des interférences entre les adaptations.
Les travaux les plus récents dans ce domaine sont (Ciraci et al., 2010). La transformation de graphe est utilisée pour la détection des interférences sémantiques. Ils appliquent les aspects dans différents ordres. S'ils aboutissent à des résultats différents, c'est qu'il y a une interférence entre les aspects. Cette approche est très originale pour détecter les interférences comportementales, mais ils ne proposent pas de solution aux cas identifiés et une résolution automatique est nécessaire.

Selon cette étude il n'y a pas d'approche pour la résolution des interférences entre les adaptations, réalisée de manière implicite sans avoir recours à l'intervention de concepteur. Nous proposons de résoudre les interférences en produisant une solution en terme d'assemblage aux points marqués par ⊗ selon un ensemble de règles pré-définies. Ces

règles spécifient comment les assemblages de composants seront fusionnés. Dans la suite nous présentons les deux approches de fusion.

10.2.2 Approche 1 : Fusion ISL4Wcomp

L'opération de fusion d'assemblage de composants est régie par un ensemble de règles logiques à partir du modèle de Berger (Berger, 2001). Cette fusion ne dépend pas de l'ordre d'application des adaptations. Pour ce faire l'opération de fusion a été mise en œuvre de façon pour garantir la propriété de symétrie. Cette propriété est décomposée en trois sous propriétés: la commutativité, l'associativité et l'idempotence de (Cheung-Foo-Wo et al., 2009).

Les adaptations sont écrites dans le langage ISL4WComp. Dans ce langage, chaque comportement spécifié par un aspect d'assemblage est un arbre qui constitue la partie gauche des règles de réécriture. Ces entêtes représentent les ports des composants. Chaque opérateur est modélisé par un composant de sémantique connue. Le moteur de fusion est alors inspiré de celui de Berger. Il s'appuie sur la fusion de programmes écrits sous la forme d'assemblages de composants.

Limites d'ISL

Les adaptations, qui sont l'entrée du mécanisme de composition, sont basées sur un langage. Dans cette approche, la composition se base sur des arbres syntaxiques mais le résultat de la composition est un assemblage de composants qui va être ajouté à l'assemblage initial. Nous sommes donc obligés de passer d'un langage (représentation arborescente) à un assemblage de composants (représentation de type graphe).

D'autre part, nous savons faire la composition avec des composants de sémantique connue. Mais si nous souhaitons introduire des nouvelles sémantiques pour la composition dans l'adaptation des applications, les difficultés rencontrées pour réaliser cette tâche sont multiples.

Dans le cadre de cette thèse, nous proposerons une modélisation des assemblages de composants et des applications par des graphes. Une description des architectures basées sur les graphes présente, outre l'aspect formel rigoureux, l'avantage d'offrir une description visuelle et facilement compréhensible qui ne nécessite aucun pré-requis concernant les langages formels. Tout nouveau besoin sera modélisé par des composants de sémantique nouvelle et l'ajout de nouvelles règles dans le mécanisme de transformation de graphes.

10.2.3 Approche 2 : Graphes et règles de Transformation des Graphes

a) Le graphe d'Assemblage de composant

Pour être en mesure de présenter notre mécanisme de composition adaptation, nous présentons dans cette section le modèle de nos assemblages de composants. Afin de représenter les caractéristiques structurelles des systèmes logiciels, plusieurs travaux utilisent le modèle de processus métiers (BPM) qui englobe aussi les stratégies pour accomplir l'évolution des applications logicielles. Ce modèle fournit des spécifications de haut niveau, indépendantes de la plate-forme cible. Il existe de nombreuses notations pour représenter BPM (La Rosa et al., 2010). Dans ce rapport, nous faisons abstraction de toute notation spécifique et nous représentons un modèle de processus comme un graphe orienté selon la figure 2.

Un graphe est donné à deux niveaux : type (*Type Graph)* et instance. Au niveau type, nous

définissons les nœuds et les arcs ainsi que leurs attributs. Nous spécifions aussi les relations entre les différentes entités (cardinalité, contrainte,…). Toutes les instances de graphe doivent être cohérentes au *Type Graph* qui a été défini. Notre *Type Graph* est donné par la figure 1(a).

Le nœud *Comport* est une abstraction des entités impliquées dans une application. Il peut être un nœud de sémantique connue (opérateur) ou bien un nœud de sémantique inconnue (un composant ou service). En effet, dans une approche « boîte noire », les interfaces des entités de l'application sont constituées des ports des composants (ou interfaces de service).

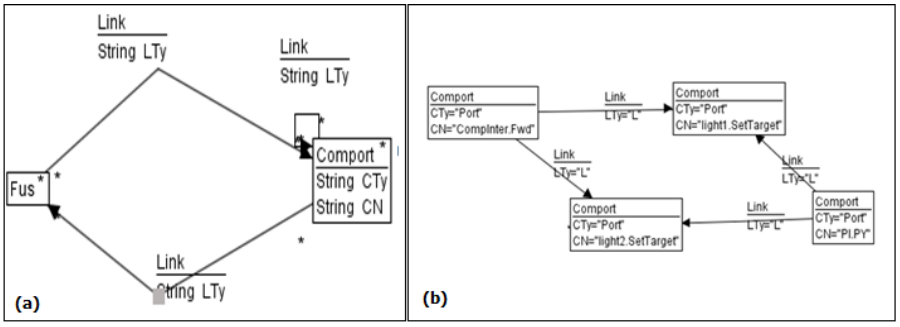

Figure 2. (a) *Type graphe de l'application* (b) *instance d'application*

Chaque nœud est spécifié par deux attributs : *CTy* est le type de nœud et *CN* est le nom de l'instance qui doit être unique. Comme exemple, nous définissons le type graphe d'une application à base de composants. Pour les nœuds de sémantique inconnue, nous représentons les ports des composants *CTy*='Port' et *CN* est sous la forme *Component_Name.PortName* (pour une interface sera *serviceName.interfaceName*). Pour les nœuds de sémantique connue, nous représentons les opérateurs. Les types d'opérateurs dans nos applications sont decrits dont le tableau suivant :

Le type CTy	Description
PAR	Définir deux actions en parallèle
IF	Exécution conditionnelle
SEQ	Définir un ordre entre deux actions
CALL	Réécriture d'un lien
DELEGATE	Definir une action dominante

Les arcs du graphe modélisent les interactions entre les entités impliquées dans nos applications. Chaque arc est libellé par son type. Par défaut, un arc porte le libellé « L » qui signifie qu'il s'agit d'un lien simple (appel d'une méthode). Par exemple le nœud *SEQ* possède deux sorties, *OUT1* qui sera exécuté avant *OUT2,* d'où sur les arcs sortant de ce nœud nous spécifions le numéro de la sortie concerné. La figure 1(b) est une instance d'un graphe selon le modèle ainsi défini. L'émission de l'événement *Fwd* du composant *CompInter* déclenche deux actions: light1.*SetTarget* et light2.*setTarget*.

Le nœud *Fus* servira à marquer les endroits dans le graphe où il y a une interférence entre les adaptations.

b) Processus de l'approche

Le mécanisme général de composition est donné par la figure 2. L'évaluation des points de coupe génère les points de jonction (endroits accueillant les modifications de l'application). Ces points de jonctions sont utilisés avec les greffons pour produire les sous-graphes à insérer dans le graphe de l'application de départ (*AdviceFactory*, Etape 1). Les sous-graphes sont superposés dans le but de produire un graphe final qui est l'union de ces sous-graphes (Etape 2). Nous obtenons donc $\mathbf{G_{Global}}$ qui représente l'application de tous les aspects (d'où les adaptations) qui ont été sélectionnés.

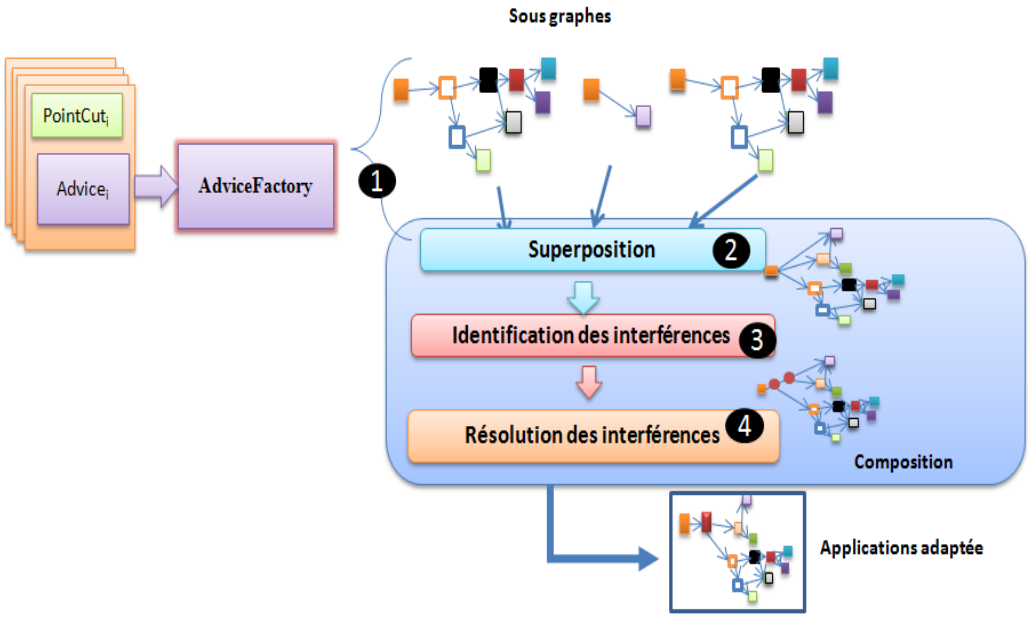

Figure 3. *Composition des aspects d'assemblage*

L'étape suivante est l'identification des interférences. En effet, les aspects peuvent avoir en commun des points de jonctions puisque au moment de leurs spécifications nous n'avons aucune visibilité sur les interactions possibles. De ce fait, pendant la phase de tissage, nous pouvons avoir des interférences entre eux. Une interférence se produit lorsque nous avons des sous-graphes à ajouter au même endroit. Il s'agit d'un nœud de sémantique inconnue qui possède plusieurs arcs sortants. A ce stade, nous ajoutons un composant spécifique (⊗) pour marquer ces points d'interférence (Etape3). Le mécanisme de résolution des interférences intervient dans les endroits marqués par un composant de fusion (⊗). La présentation de mécanisme fait l'objet de la section 4.3.

c) Support de résolution

i. Transformation des graphes

La réécrire d'un graphe G en un graphe G' revient à remplacer un sous-graphe L de G par un graphe R. G' est le graphe résultant de ces deux opérations. G est appelé graphe hôte ("*host graph*"), L est appelé graphe mère ("*mother graph*") et R est appelé graphe fille ("*daughter graph*"). Dans cet esprit, une règle de réécriture est sous la forme p: L →R. Ce type de règle est applicable à un graphe G s'il existe une occurrence de L dans G (Wermelinger *et al.*, 1999). Son application a pour conséquence la suppression de l'occurrence de L du graphe G et son remplacement par une copie (isomorphe) de R. L'application de la règle implique la

suppression du graphe correspondant à Supp=(L ∩ (L\R)) et le rajout du graphe correspondant à Aj= (R ∩ (L \ R)). Les arcs suspendus (arcs sans nœud de départ ou sans nœud d'arrivée ou sans les deux) sont supprimés.

ii. Règles de fusion

Durant la phase d'identification des interférences, nous avons marqué par un nœud *Fus* les endroits où il y a des interférences. L'approche que nous avons choisie pour la résolution de ces interférences est la ***fusion*** des comportements. Ceci est rendu possible par la connaissance de la sémantique d'éléments clés dans la spécification des règles d'adaptation. Donc, si nous avons plusieurs comportements à la sortie d'un nœud, le mécanisme de fusion produira un comportement final qui regroupe tous les comportements spécifiés sans avoir d'interférences dans le comportement résultant. De plus, ce mécanisme de fusion garantit la symétrie via trois propriétés : *l'idempotence*, *la commutativité* et *l'associativité* (Cheung-Foo-Wo *et al.*, 2009). De ce fait, l'ordre d'application des adaptations, donc l'ordre dans lequel la fusion des comportements est réalisée n'a pas d'importance. Nous avons défini un ensemble de règles de fusion qui dérivent de (Cheung-Foo-Wo *et al.*, 2009).

Pour illustrer le mécanisme de fusion, nous détaillons les règles de fusion de nos opérateurs.

- **La Fusion de deux appels à un Port**

La fusion d'un envoi de messages avec lui-même est décrite par la règle de la figure 4. Dans le graphe R on ne garde qu'un seul lien vers le Port x.

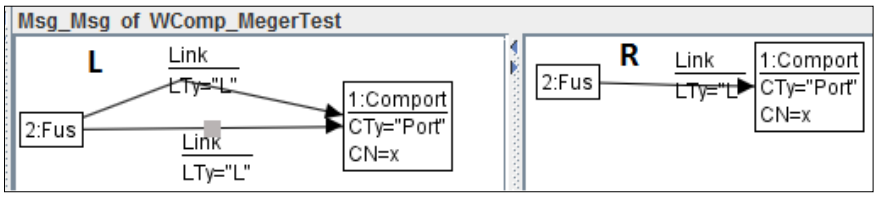

Figure 4. La Règle de fusion de deux appels d'un port

- **La Fusion du PAR avec un port**

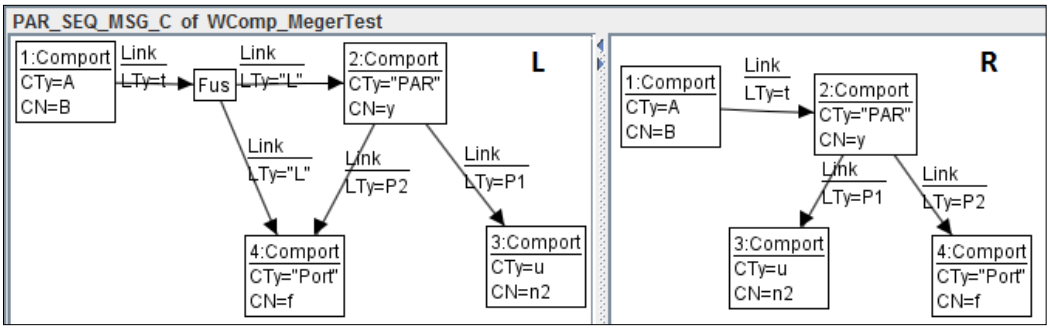

Figure 5. La Règle de fusion du PAR et un port

La fusion d'un PAR avec un port qui appartient à l'ensemble de ses successeurs donne le graphe R de la figure 5. Un seul lien (qui part du PAR) sera gardé vers ce port. Par exemple le résultat de fusion de : **PAR**(a,B)⊗a est **PAR**(a,B) avec CTy(a)= « Port » et B représente un autre comportement (un autre port ou opérateur).

- **La Fusion du PAR avec tous les autres comportements**

La règle de fusion d'un opérateur concurrentiel avec un autre comportement est décrite dans la figure 6. L'arc libellé par la lettre « P » montre le sens de propagation de l'opérateur de fusion Fus. Il est calculé par la fonction ***Pivot***. Cette fonction permet de contrôler la manière dont la fusion s'opère. Le pivot sert à guider la fusion afin qu'elle se fasse en des points précis. Cette fonction recherche s'il existe des ports successeurs du PAR et qui ont aussi un prédécesseur le nœud x. Si c'est le cas, une fusion sera appliqué entre les nœuds x et f (il a été identifié par la fonction pivot).

Notion de *pivot*.

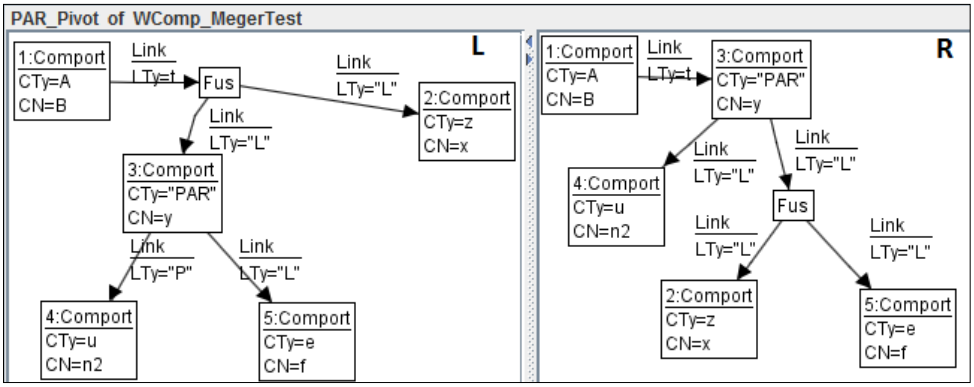

Figure 6. La Règle de fusion générique du PAR

- **La Fusion de deux IF**

La fusion de deux comportements IF est régie par plusieurs règles de fusion. Un comportement IF est sous la forme IF(m,A,B) ; avec *m* la condition à vérifier, *A* le comportement à mettre en place si m est vrai et *B* le comportement si m est faux. La fusion de IF(m1,A,B) ⊗IF(m2,C,D) dépend de *m1* et *m2*. S'il s'agit de la même condition, c'est-à-dire m1=m2, alors la fusion sera propagée sur les deux branches de IF (Figure 7). Nous aurons comme résultat IF(m1, A⊗C, B⊗D). En effet puisque nous avons la même condition à vérifier, il s'agit d'un seul IF d'où la fusion de deux branches vraie et de deux branches fausse de deux IF : c'est la propagation de l'opérateur de fusion. Par la suite la fusion des comportements A⊗C et B⊗D fait appel à d'autres règles de fusion. Dans l'autre cas c'est-à-dire si m1 !=m2 le résultat de fusion sera alors IF(m1,IF(m2,A⊗C,A⊗D),IF(m2,B⊗ C,B⊗D)).

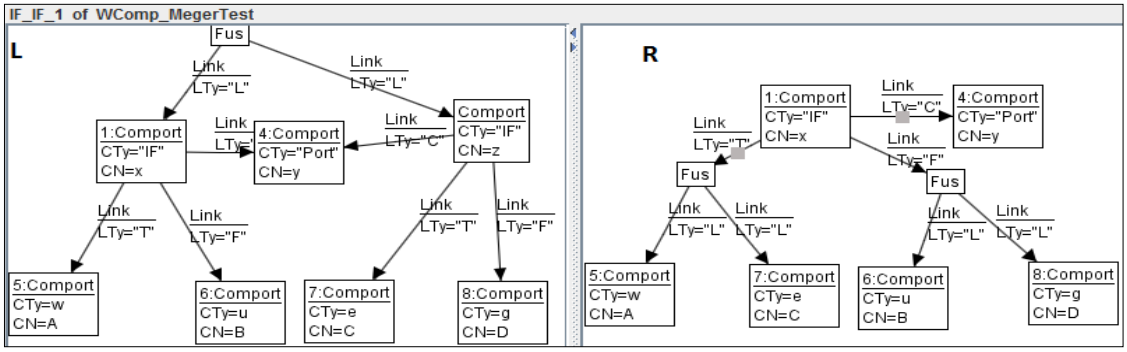

Figure 7. La Règle de fusion de deux IF : cas général

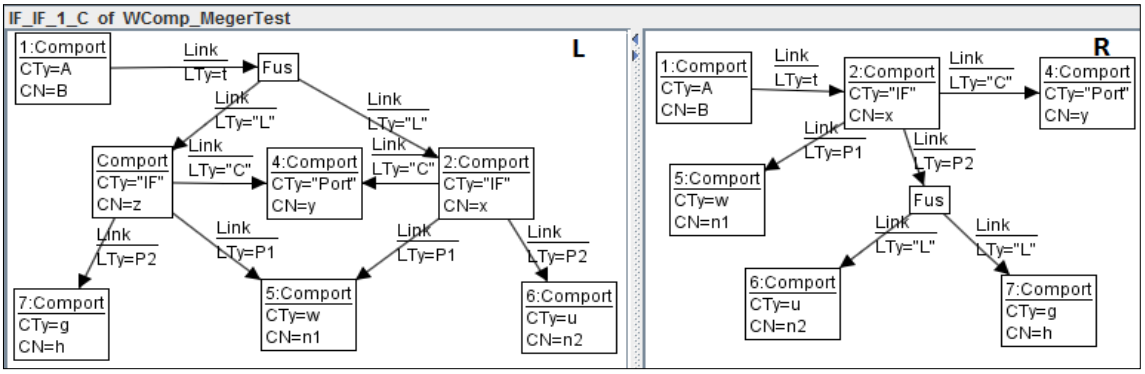

Figure 8. La Règle de fusion de deux IF : cas particulier

La règle précédente montre le cas général de la fusion de deux IF. Une autre configuration possible est le cas où les deux opérateurs conditionnels partage un nœud illustré dans la figure 8. Logiquement les deux opérateurs doivent partager un nœud de la même branche (puisqu'il s'agit de la même condition à vérifier). La variable P1 est utilisée pour désigner soit « T » ou bien « F ».

- **La Fusion d'un IF et un Port**

Les figures 9, 10 et 11 montrent que le comportement conditionnel « encapsule » le comportement séquentiel et les envoie de message. Ces derniers seront fusionnés avec les deux branches du comportement conditionnel. La fusion d'un IF avec un Port présente deux cas : Si le port appartient à l'un de deux branche de IF la règle de la figure 9 sera appliquée sinon la règle de la figure 10 sera mis en ouvre.

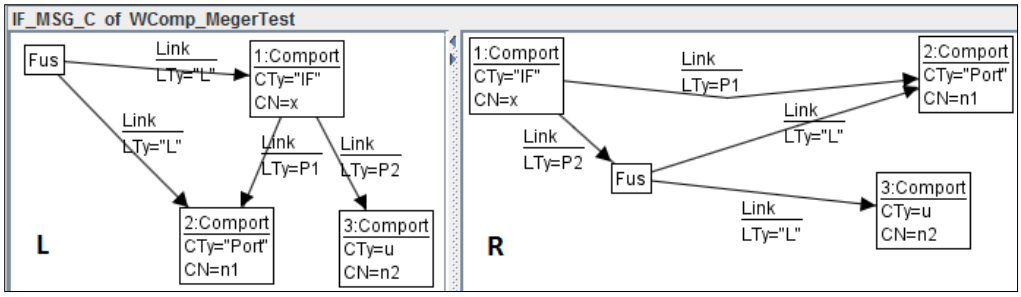

Figure 9. La Règle de fusion de IF et Port : cas particulier

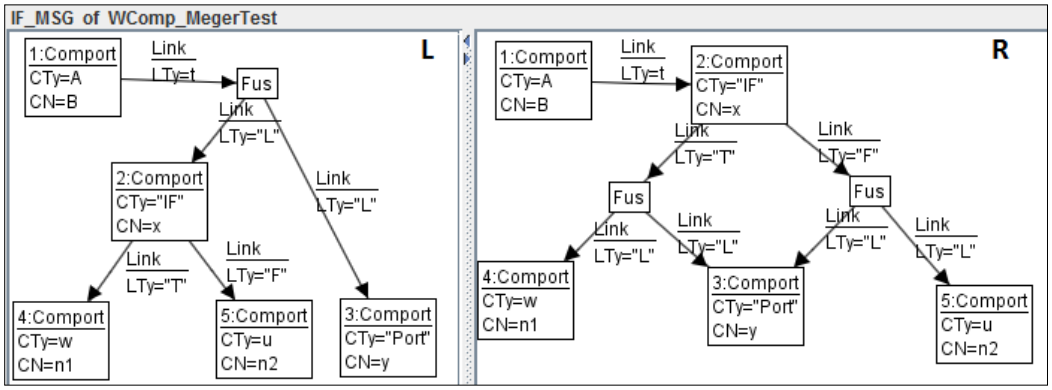

Figure 10. La Règle de fusion d'IF et Port : cas général

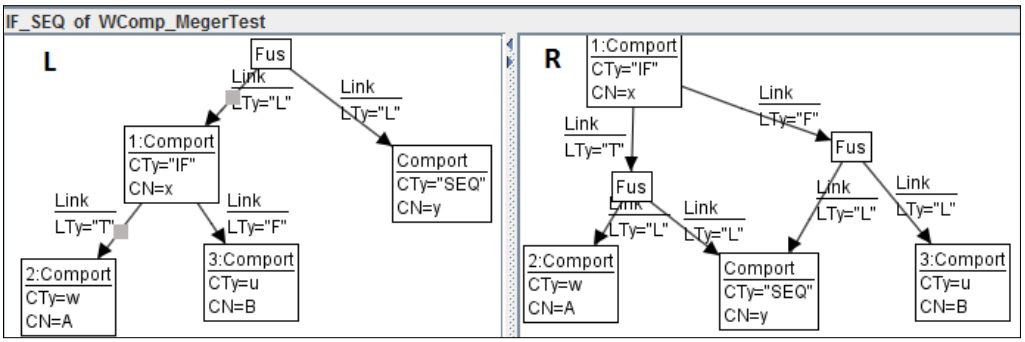

Figure 11. La Règle de fusion d'IF et SEQ

La règle de la figure 11 montre que l'opérateur de séquence va être aussi dupliqué dans les deux branches d'IF. La logique de cette fusion est comme suit : quelque soit la valeur de la condition à vérifier par le IF, le comportement qui a été spécifié par la séquence devrait être mis en place d'où la propagation de l'opération de la fusion dans le graphe R.

- **La Fusion de deux SEQ**

La fusion de deux séquences doit conserver les relations d'ordre établies au sein de chacune séparément. Il existe plusieurs configurations possibles pour deux SEQ. Dans la figure 7 on représente le cas où les SEQ partagent un nœud qui possède le même ordre. Les variables P1 et P2 désignent « OUT1 » et « OUT2 » (spécifient l'ordre entre les actions). Si on suppose que P1= « OUT1 » et P2= « OUT2 » alors le graphe L montre la fusion de **SEQ**(f,h) et **SEQ**(f,w). Intuitivement, la fusion de ces deux operateurs implique le résultat donné par le sous graphe R : La séquence de f et le résultat de fusion de h et w.

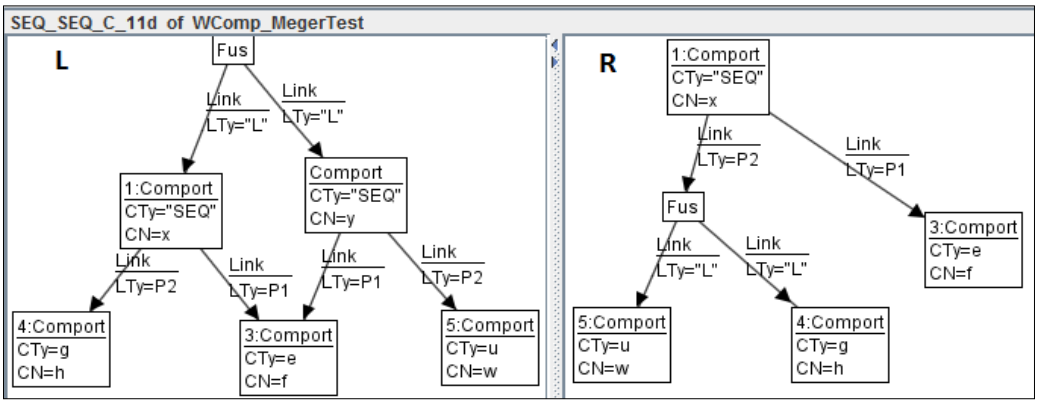

Figure 12. La Règle de fusion de deux SEQ avec Pivot : cas 1

Un autre cas de figure est lorsque les deux SEQ partagent un nœud qui possède un ordre différent (dans la figure 13 le nœud h est définit comme un OUT1 dans la première séquence et OUT2 dans la deuxième séquence), la règle de la figure 13 sera appliquée. La fusion de **SEQ**(h,w) et **SEQ**(f,h) implique comme résultat : **SEQ**(f,**SEQ**(h,w)).

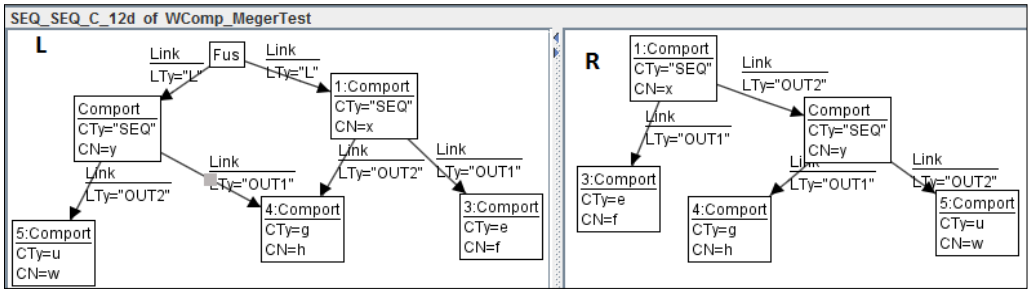

Figure 13. La Règle de fusion de deux SEQ avec Pivot : cas 2

Dans le cas où les deux séquences partagent un nœud mais ce dernier n'est pas un de leurs successeur immédiate (il existe un chemin de SEQ1 et un autre chemin de SEQ2 qui mènent vers ce nœud), les règles des figures 14 et 15 seront appliquées. La figure 14 montre le cas où il existe un nœud en commun entre les branches P1 P3 et entre les branches P2 P4.

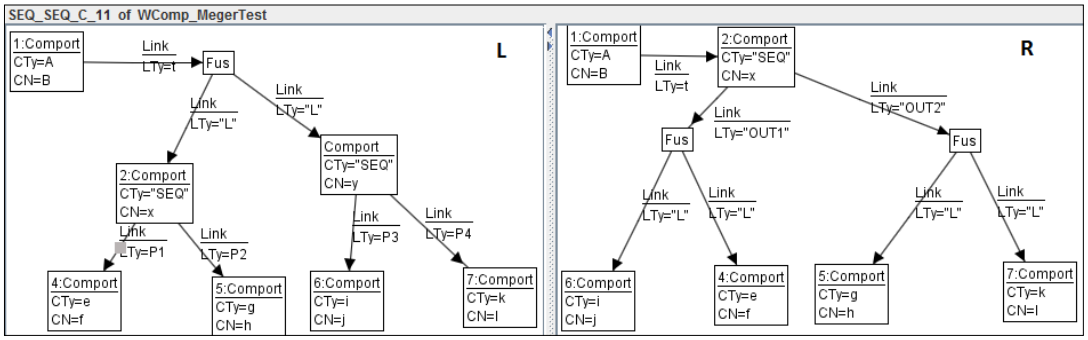

Figure 14 . La Règle de fusion de deux SEQ avec 2 pivots

Un autre résultat possible de pivot entre les deux SEQ est présenté dans la figure 15. Le nœud h est partagé par les deux branches de la deuxième SEQ qui porte le nom d'instance y (h appartient à la sous branches OUT1 et à la fois à la sous branche OUT2 de y). La fusion gardera le nœud f comme le premier nœud à être exécuté et propage l'opération de la fusion sur la première branche de la deuxième SEQ.

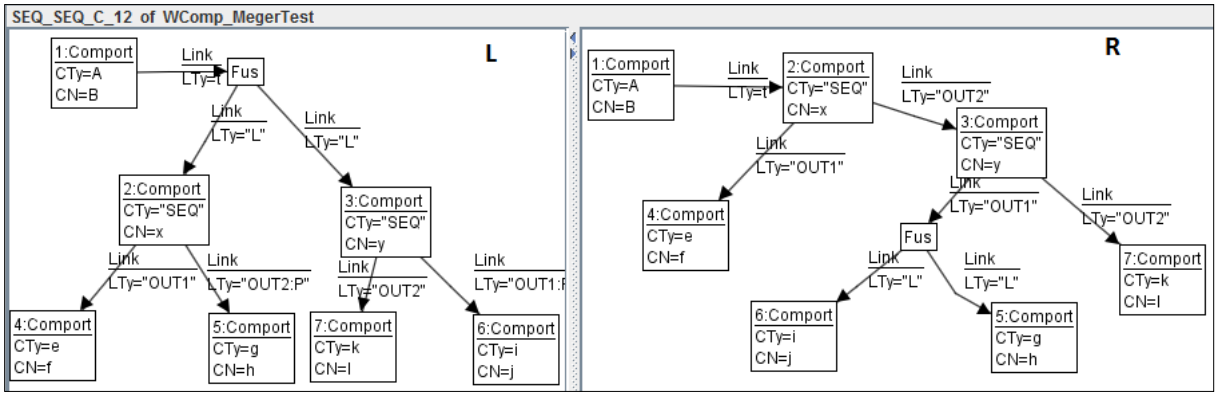

Figure 15 . La Règle de fusion de deux SEQ avec 3 pivots

- **La Fusion de CALL**

Nous avons déjà mentionné que cet opérateur est utilisé pour la récriture des liens. Les règles de fusion associée à CALL permettent alors de modifier les endroits où la fusion va être appliquée. La figure 16 illustre la fusion d'un comportement x qui possède comme

successeur un CALL avec un comportement e. le résultat de fusion sera alors la réécriture du lien vers le nœud e pour qu'il soit un successeur immédiat du CALL.

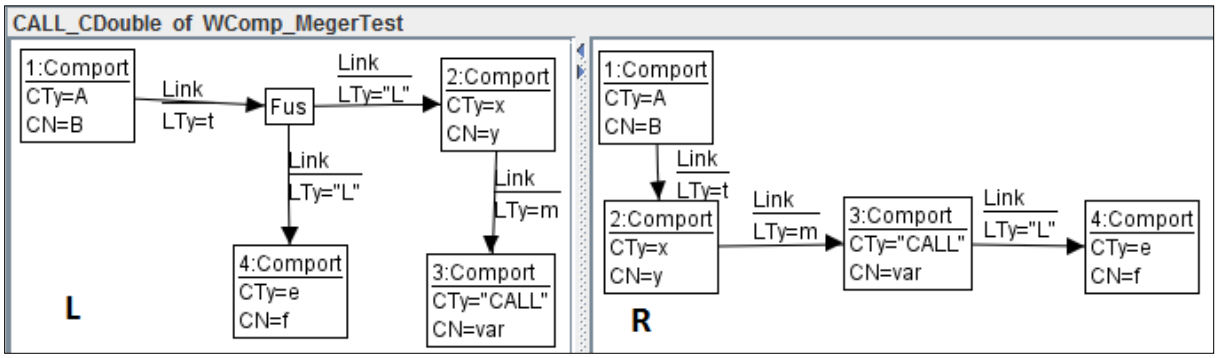

Figure 16. Règle de fusion de CALL

- **La Fusion de DELEGATE**

Cet opérateur est utilisé dans le cas où on voudra spécifier qu'un lien doit être unique. C'est-à-dire le nœud qui précède le DELEGATE (dans la figure 1 c'est le nœud de type A) ne devrait pas avoir d'autres liens vers d'autres nœuds (le lien vers le nœud x sera supprimé).

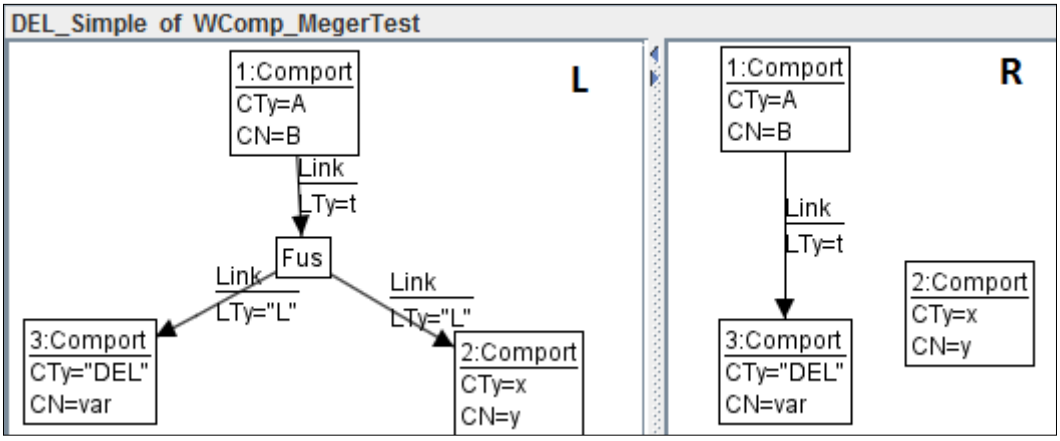

Figure 17. Règle de fusion de DELEGATE

- **La règle de fusion par défaut**

Les règles de fusion spécifiées permettent de fournir, dans le cas où cela est possible, le résultat de fusion des comportements qui sont en interférence. Mais parfois nous aurons des interférences que nous ne pouvons pas résoudre (pas de règle de spécification). Dans ce cas nous avons prévu une solution par défaut qui consiste à mettre en parallèle les comportements en ajoutant un composant « PAR » en sortie du nœud où l'interférence a été marquée. De ce fait, les comportements en interférence seront exécutés indépendamment les uns des autres. La figure 18 illustre cette règle de fusion.

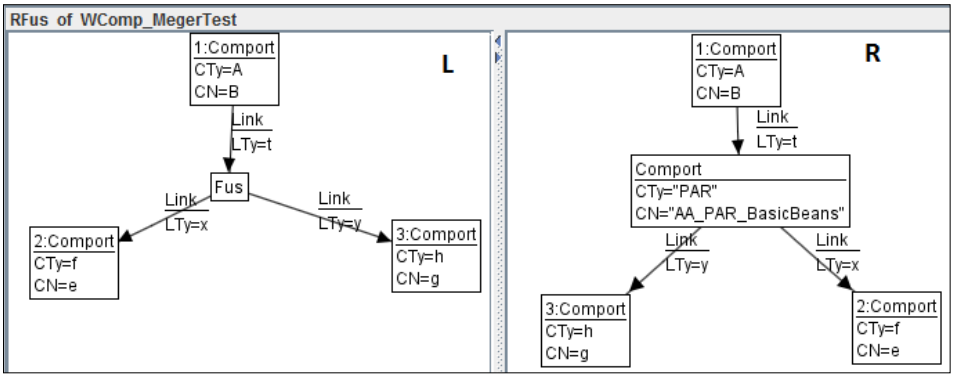

Figure 18 . La Règle de fusion

d) Evaluation et temps de réponse

Nous évaluons notre approche en termes de performance liée aux expériences de compositions assemblages des composants générés aléatoirement. Ils ont été réalisées sur un ordinateur personnel standard (Intel ® Core TM2, 3,06 GHz).

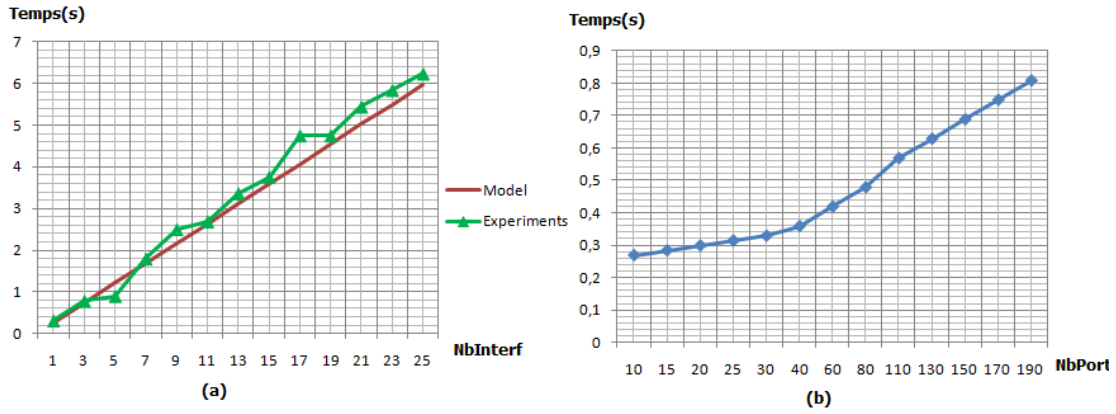

Figure 19. Temps de réponse de mécanisme de résolution des interférences

La complexité de l'implémentation actuelle est fortement liée au moteur de transformation de graphe utilisé AGG Attributed Graph Grammar (Taentzer et al., 2004) car la majorité du temps de composition est passé dans la phase de résolution des interférences. L'opération la plus complexe durant la phase d'application d'une règle de transformation de graphes est la recherche d'un match dans le graphe à modifier (rechercher s'il existe le sous graphe L de la règles de transformation) c'est de l'ordre de $O(2^{NNode})$ avec *NNode* est le nombre de nœud dans sa la partie gauche L de la règle. Si nous supposons que nous exécutons une règles de transformation pour chaque cas d'interférence le cout totale d'exécution du moteur de transformation de graph est : $CoutMTG=a1*NbInterf*2^{n_k}+a2$; avec nk est le nombre de nœud de la règle Rgk, NbInterf est le nombre d'interférence à résoudre et a1 et a2 sont les paramètres du modèle. Dans nos règles n_k est entre 3 et 13. La moyenne est $n_k=8$.

Le temps passé par AGG pour exécuter une règle de fusion varie entre 0,052 et 0,366 seconde. Ceci est lié au nombre de nœuds dans une règle. Il faut en moyenne 0,248 s pour exécuter une règle. La figure 19.a montre le temps de réponse du moteur de transformation fonction des nombre d'interférences. Selon ces expériences a2 = 0,02 et a1 = 0,446. Le nombre de composants présent dans l'application n'a pas d'influence sur le temps de réponse de la phase de résolution des interférences. Il affecte le temps de réponse global du

mécanisme de la fusion (utilisée pour calculer le pivot et l'export du graphe vers la plate-forme), Dans la figure 19.b, nous montrons l'influence du nombre de composants sur le temps de réponse total du moteur de fusion. Nous avons fait varier le nombre de composants instanciés (NbPort) dans l'application de 10 à 190. Dans cette expérimentation, nous avons utilisé un graphe d'application avec deux points d'interférences à résoudre.

10.3 La résolution des interférences de Type 2

Dans la section précédente, nous avons détaillé notre solution pour répondre aux problèmes d'interférences de Type 1. En ce qui concerne les interférences de type 2, il s'agit d'un problème d'accès concurrent à un port de composant. Nous avons une piste qui consiste à définir des connecteurs génériques qui servent à faire la coordination et la synchronisation entre ces accès. Le but est de pouvoir connecter un ou plusieurs composants émetteurs d'événements et un composant récepteur (où le conflit a été identifié). Ce type de connecteur est appelé "connecteur complexe". Dans un premiers temps on compte modéliser formellement le comportement de ces connecteurs par des automates à été fini. Ensuite, on définira les règles de fusion de ces connecteurs qui revient à faire le produit des automates (produit cartésien et produit synchronisé). Le mécanisme obtenu permettra dans le cas de la mise en œuvre simultanée de plusieurs applications (plusieurs assemblages de composants) de résoudre avec une logique maitrisée les conflits d'accès qui pourraient apparaître sur les dispositifs de l'application finale.

Un connecteur complexe est définit pat C_x=<P_e, p_r, A_{dec}, L_{fus}> avec :

- P_e un l'ensemble de ports de composants émetteurs
- un port de composants récepteur p_r
- un Automate décrivant la logique qui permet le déclenchement du p_r
- une fonction L_{fus} décrivant comment les données reçues par les ports d'entrée seront fusionnées.

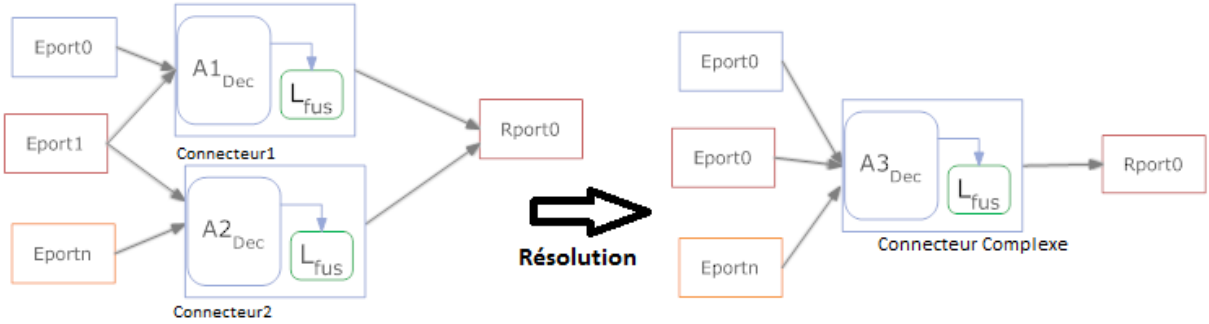

Figure 20. Fusion de deux connecteurs pour l'obtention d'un nouveau connecteur complexe

La figure 20 montre notre vision pour la fusion des connecteurs. Dans un premier lieu on part de l'hypothèse que tous les connecteurs possèdent la même logique de fusion de données reçu par les ports émetteurs (L_{fus}). L'étape principale dans notre solution est la détermination de la logique de déclanchement pour le connecteur résultant de la fusion. Dans la figure 20 l'automate $A3_{Dec}$ est obtenu soit par le produit cartésien ou bien le produit synchronisé des automates $A1_{Dec}$ et $A2_{Dec}$. Comme exemple, on considère les automates de la figure 21 qui représente la logique de déclanchement de chacun de deux connecteurs précédents. Le première connecteur implémente une logique de déclanchement **ET** entre ses ports

d'entrées alors que le deuxième utilise une logique de déclanchement **OU** entre ces ports d'entrées.

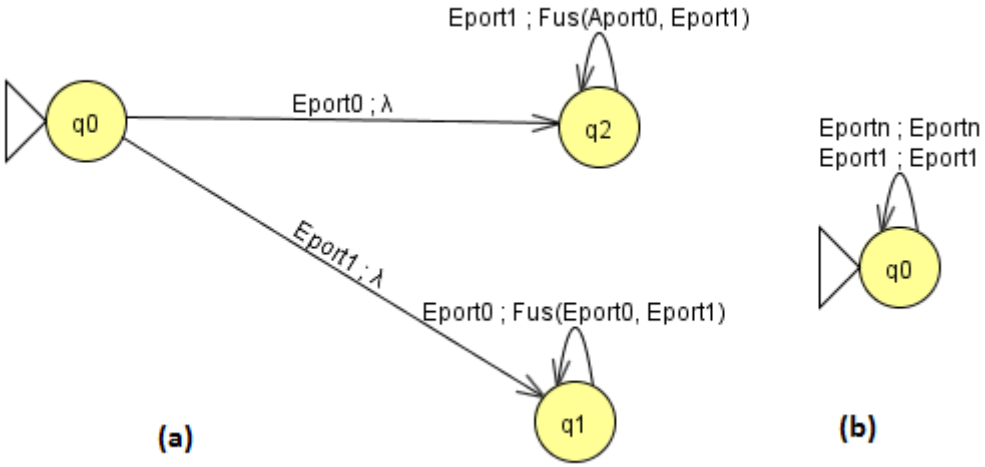

Figure 21. (a) $A1_{Dec}$ La logique de déclanchement de connecteur1 et (b) $A2_{Dec}$ La logique de déclanchement de connecteur2

Le résultat de fusion de ces deux automates est donné dans la figure 22. Ici on a fait recours au produit synchronisé de deux automates. Le port de sorti Rport0 ne sera appelé que dans le cas où les deux ports *Eport0* et *Eport1* sont activée ensemble ou bien lorsqu'il y a activation du port *Eportn*.

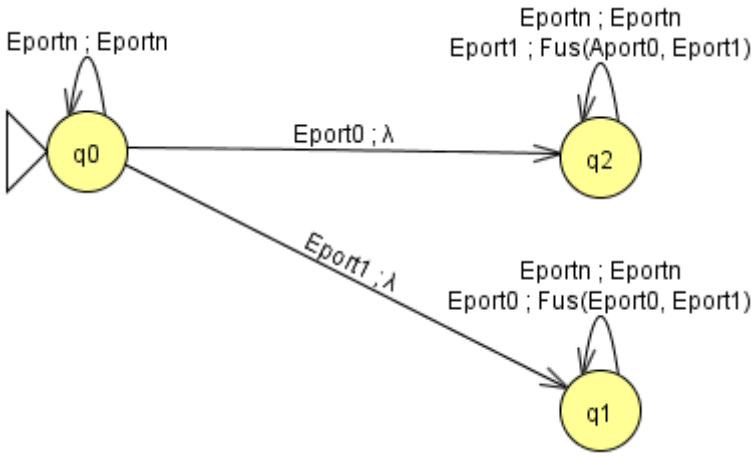

Figure 22. $A3_{Dec}$ L'automate qui représente la logique de déclanchement du connecteur complexe

Dans notre modèle on respecte l'hypothèse synchrone. Chaque réaction est atomique. Durant une réaction, la valeur des ports émetteurs reste inchangée. L'occurrence des événements ne pourra en aucun cas être simultanée (on ne peut pas recevoir deux événement de deux ports émetteurs au même instant).

Références

Berger L., Mise en Œuvre des Interactions en Environnements Distribués, Compilés et For-tement Typés : le Modèle MICADO. Thèse de doctorat, Université de Nice-Sophia Antipolis - Faculté des sciences et techniques, École doctorale STIC - Informatique, oct 2001.

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

Ciraci S., Havinga W., Aksit M., Bockisch C., Van den Broek P. «A graph-based aspect interference detection approach for UML-based aspect-oriented models». *Transactions on Aspect-Oriented Software Development* VII, pp. 321–374, 2010.

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

Greenwood P., Lagaisse B., Sanen F., Coulson G., Rashid A., Truyen E., and Joosen W., «Interactions in AO middleware». *In Proceedings of Workshop on ADI, ECOOP*. 2007.

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

Mehner K., Monga M., Taentzer G. «Interaction analysis in aspect-oriented models». *In Requirements Engineering, 14th IEEE International Conference*, pp. 69–78. IEEE, 2006.

Sadou N., Tamzalit D.,Oussalah M., . «SAEV, une solution à l'évolution structurelle dans les architectures logicielles». L'Objet. pp 45-80. Lavoisier. 2007

Taentzer G., «AGG: A graph transformation environment for modeling and validation of software» *Lecture Notes in Computer Science*, vol. 3062, pp. 446–453, 2004.

Wermelinger M., J. Fiadeiro, «Algebraic software architecture reconfiguration» *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 393–409, 1999.

Zhang J., Cottenier T., Van Den Berg A. and Gray J., «Aspect Interference and Composition in the Motorola Aspect-Oriented Modeling Weaver». *In Workshop on Aspect-Oriented Modeling at the 9th International Conference on Model Driven Engineering Languages and Systems*, Milan, Italy. 2006.