

Un service d'interactions : principes et implémentation

Mireille Blay-Fornarino, David Ensellem, Audrey Occello Anne-Marie Pinna-Dery, Michel Riveill, Jérémy Fierstone, Olivier Nano, Gilles Chabert

Laboratoire I3S, Université de Nice
Bat ESSI,
930, route des Colles
06903 Sophia Antipolis Cedex
email : rainbow@essi.fr

RÉSUMÉ. Pour faciliter une adaptabilité dynamique des applications à base de composants, nous proposons d'explicitier les interactions entre composants et de les considérer à la base de la connectivité des applications. Nous avons donc défini un service d'interactions qui permet l'adaptation dynamique des composants par la définition de schémas et la pose et la destruction d'interactions à l'exécution. L'utilisateur exprime alors les interactions au niveau de l'applicatif. Basée sur le langage ISL, la fusion des interactions assure la commutativité et la transitivité lors de la pose des interactions, ce qui permet une adaptation « cohérente » de l'application par plusieurs intervenants. Cet article présente au travers d'un exemple le service, puis décrit le langage ISL avant d'aborder plus précisément les aspects implémentation du service qui permettent de faire interagir des objets locaux et des composants RMI et EJB. L'implémentation est disponible à partir du site de l'équipe <http://rainbow.essi.fr>

ABSTRACT. In order to allow dynamic component-based application adaptation, we will describe component interactions and consider them at the base of application connectivity. We have defined an interaction service that allows dynamic component adaptation by using patterns and runtime interaction instantiation and destruction. To do so, the user defines interactions at application level. Based on the ISL language, interaction merging insures commutability and transitivity when interactions are instantiated. This allows a consistent adaptation of the application by several users. In this article, we introduce the service through an example, then describe the ISL language before dealing more precisely with implementation details of the service allowing local, EJB, or RMI object to interact with each other. Implementation is available from the team web site: <http://rainbow.essi.fr>

MOTS-CLÉS : interactions, adaptabilité dynamique, composants distribués, méta-programmation

KEYWORDS: interactions, adaptability dynamic, distributed components, meta-programmation

1. Introduction

L'avènement, ces dernières années, des modèles de composants industriels renforce l'idée que la programmation des applications ira de plus en plus vers une composition d'éléments logiciels à l'instar des composants électroniques. L'effort important mené par ces modèles autour d'une intégration presque automatique des services, conforte cette allégorie, le composant est « enfiché » dans l'infrastructure logicielle. Ainsi l'adaptabilité des composants doit être prise en charge au déploiement. Pourtant, le monde logiciel est un monde dynamique dans lequel le contexte d'exécution des composants est particulièrement changeant. La prise en compte des interactions entre un composant et les autres composants présents à un moment donné (composants logiciels, techniques ou physiques) est nécessaire à son adaptation. Le nomadisme, l'usage intensif du réseau, le travail collaboratif sont autant de facteurs qui accentuent ces besoins d'adaptation dynamique et donc de gestion des interactions entre tous les composants aussi bien métiers que techniques.

Le développement d'applications sur des plates-formes à composants « standards » tels que EJB [EJB00], CCM [Marvie02] ou .Net [Riveill01], force à prévoir à priori les interactions au niveau des interfaces et du code métier des composants. Si certaines interactions sont prises en charge partiellement par les différents modèles de ports des CCM, certains services (événements, notifications), l'utilisation de scripts, d'intercepteurs CORBA [OMG01] ou les conteneurs ouverts [Vadet02], la part de connectique et de contrôle induite par les interactions (synchronisation, conditionnelle, etc.) est toujours à la charge du programmeur. Certains travaux basés sur la programmation par aspects ([Pawlak01], [Riveill00], [Sarradin01]) proposent également d'explicitier les interactions, néanmoins leur programmation à un niveau plus méta en terme de réception et émission de requêtes et non pas au niveau applicatif rend non « naturelle » l'écriture de ces interactions. Nous défendons donc l'idée que ces techniques sont essentielles à la réalisation d'une adaptation dynamique des composants, mais qu'il est nécessaire de les abstraire pour faciliter leur programmation et aller vers des systèmes plus sûrs.

Pour autoriser une adaptabilité dynamique des applications, nous proposons de considérer les interactions comme des entités de premières classes avec les propriétés suivantes :

- Un schéma d'interaction spécifie les dépendances comportementales entre les composants qui seront liés ; les interactions instances de ce schéma maintiennent cette cohérence locale,
- La définition d'un schéma d'interaction ne dépend pas des implémentations ; une interaction peut lier des objets ou composants définis sur différentes plates-formes,
- Un schéma d'interaction est décrit en se basant sur l'interface des composants, il peut être utilisé pour lier un ensemble d'instances de composants,
- Un schéma d'interaction et une interaction peuvent être créés et détruits en cours d'exécution,

- Une interaction ne peut pas contrôler des propriétés n'appartenant pas à l'interface du composant pour ne pas briser l'encapsulation ; l'interface d'un composant lié par des interactions n'est pas modifiée même si son comportement est modifié par l'interaction (de façon plus concrète, « *comportement* » représente pour nous l'exécution du code d'une méthode),
- La gestion des interactions est basée sur une composition des interactions qui respecte les propriétés de commutativité et de transitivité (cf. partie 3.2).

Afin de supporter ces différentes propriétés, nous avons défini un service d'interactions (disponible sous <http://rainbow.essi.fr>) qui présente les composantes suivantes.

- Un serveur d'interactions qui permet de :
 - définir dynamiquement de nouveaux « schémas d'interactions » qui constituent ainsi une bibliothèque,
 - de lier et de délier des composants par la création et la destruction d'interactions
 - de naviguer dans le graphe d'interactions
- Des composants « interagissants » ; c'est-à-dire des objets ou composants qui ont été préparés à interagir ; leur comportement peut être modifié dynamiquement.

L'organisation de cet article est la suivante. La partie 2 présente de manière intuitive le service d'interactions au travers des cas d'utilisation, la partie 3 décrit le langage permettant de décrire les interactions. La partie 4 expose des éléments des différentes mises en œuvre du service. La partie 5 présente nos perspectives de travail et conclut cet article.

2. Cas d'utilisation

Cette partie vise à donner une vue intuitive de l'utilisation du service d'interactions au travers de la présentation de cas d'utilisation du service d'interaction. Nous utilisons un exemple simple de gestion d'agendas.

Dans cette application, il existe une implémentation RMI d'un composant *Display* qui permet l'affichage de message, d'un *authentificateur* qui permet de valider les appels sur un composant, d'un composant simplifié *DataBase* qui permet de mémoriser les agendas dans une table de hachage en fonction du nom de l'utilisateur de l'agenda, et une implémentation EJB et RMI d'un composant *Agenda*. En cours d'exécution, les instances de ces différents composants sont liées et déliées par des interactions pour s'adapter au contexte d'exécution. Ainsi certaines instances du composant *Agenda* RMI peuvent être sauvegardées à chaque ajout d'un rendez-vous, un agenda de groupe est créé par la pose d'interactions entre plusieurs

agendas, un authentificateur est associé à certains agendas pour vérifier que seuls les utilisateurs autorisés accèdent à l'agenda. La figure 1 visualise un réseau de composants et d'interactions possibles à un moment donné de l'exécution de l'application.

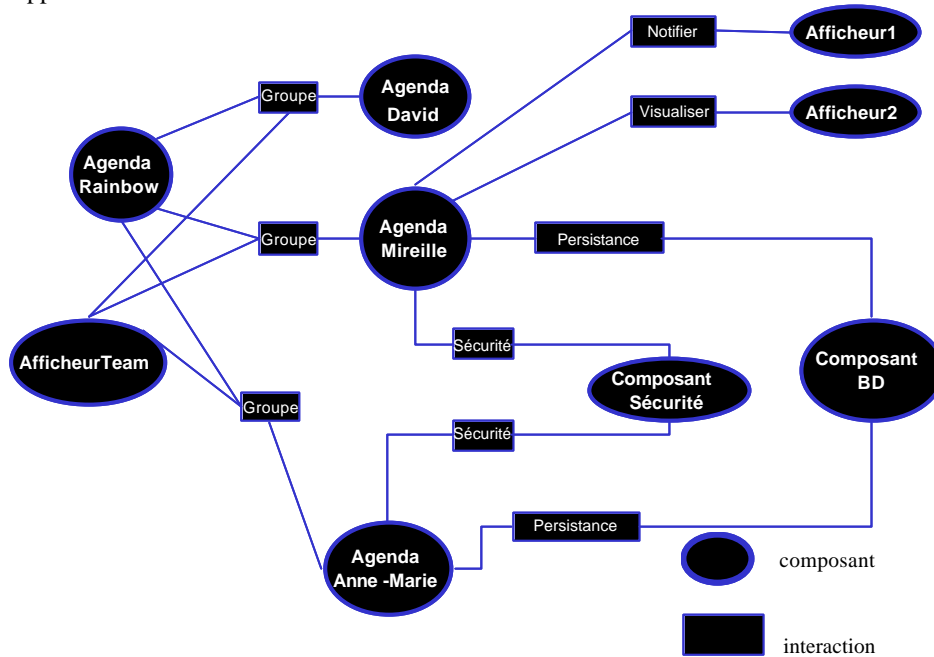


Figure 1 : Réseau d'interactions pour l'application Agenda.

2.1. Définition d'un schéma d'interaction

Dans l'exemple, le développeur de l'application d'agendas veut autoriser ses utilisateurs à lier à l'exécution un agenda à un «afficheur» afin d'être notifié chaque fois qu'un rendez-vous est ajouté dans l'agenda. Pour les agendas RMI, il peut être utile de gérer la persistance par stockage dans une base de données définie par l'utilisateur. Pour cela, le développeur définit des schémas d'interactions, qui sont fournis avec l'application (cf. figure 2, schémas *notification* et *persistance*). Ces schémas définissent des modèles d'interactions que l'utilisateur final pourra utiliser. En cours d'exécution, un utilisateur des agendas peut vouloir ajouter de la notification entre agendas, interaction non prévue par le développeur de l'application, pour par exemple créer des agendas de groupe. De la même manière que précédemment, il définit ses propres schémas d'interactions (cf. figure 2, schéma *group*).

L'utilisateur exprime les « schémas » d'interactions en utilisant le Langage de Spécification des Interactions : ISL¹ (cf. figure 2) qui est décrit plus amplement dans la section 3. Un **schéma** décrit plusieurs **règles** d'interactions qui expriment les contrôles qui doivent être opérés sur les objets liés. Une règle est composée d'une partie gauche qui exprime le **message notifiant** et d'une partie droite la **réaction** qui correspond à de la **réécriture** de code.

```

1  interaction notification(Object O, Display display){
2      O.* -> O._call //
3          display.notify(_call)
4  }
5
6  interaction group(Agenda team, Agenda member, Display display){
7      team.addMeeting(java.lang.String title)
8          -> team._call; member.addMeeting(title)
9      ,
10     member.addMeeting(java.lang.String title)
11         -> member._call; display.notify(_call)
12 }
13
14 interaction persistance(Agenda A, Database database){
15     A.addMeeting(java.lang.String title)
16         , java.lang.String owner
17         -> A._call;
18         owner:=A.getOwner(); database.store(owner,A)
19 }
20
21 interaction security(Object O, SecurityService service){
22     O.* -> if service.check(_call) then
23         O._call
24     else
25         exception "unauthorized user"
26     endif
27 }

```

28 : 1

Figure 2 : Schémas d'interactions

Le schéma *notification* (ligne1) peut lier un objet quel que soit son type et un composant de type *Display*. Il contient une règle (ligne 2 et 3). Elle exprime que tout message (utilisation de l'étoile) reçu par un objet ainsi lié, doit être exécuté dans un ordre indéterminé avec un envoi de message à l'objet *display* associé. Le schéma *group* (ligne 6)

¹ ISL=Interaction Specification Language

peut lier 3 composants. Il définit deux règles. La première stipule que le message *addMeeting* (message *notifiant*) des agendas correspondants au paramètre *team* entraînera la réaction (ligne 8) qui après l'exécution du message lui-même, ajoute le rendez-vous à l'agenda d'un membre. Dans le schéma *notification*, le mot clé *_call* est utilisé pour représenter l'appel du message notifiant (*O._call*), mais aussi pour représenter une réification du message notifiant lorsqu'il est utilisé seul (*_call*) comme paramètre d'une méthode.

Les schémas sont enregistrés auprès du serveur d'interactions par appel de la méthode *createPattern* qui prend pour paramètre le schéma d'interactions concerné. Le serveur d'interactions stocke ainsi les schémas qui peuvent être récupérés pour modification ou analyse en utilisant la méthode *getPattern(String)*. Une interface graphique facilite l'écriture des schémas (cf. figure 2).

2.2. Pose et destruction d'interactions

En cours d'exécution, le programmeur peut lier ou délier les instances de composants entre eux en utilisant les schémas qui sont enregistrés auprès du serveur. Pour cela, il s'adresse au serveur d'interactions par la méthode *createInteraction(String createInteraction String PatternName, Object targets[]) throws Exception*. Le serveur mémorise la nouvelle interaction, renvoie son nom et demande à chacun des composants liés par l'interaction et définissant des messages notifiant de fusionner les nouvelles règles qui le concernent. La prise en compte de ces modifications aura lieu à compter du prochain appel.

Une interface graphique permet également la pose d'interactions. A partir de la liste des schémas d'interactions enregistrés, l'utilisateur sélectionne le schéma d'interaction à instancier (*security* par exemple) . Une fenêtre propose alors le type des composants à connecter (*Object* et *SecurityService*) et les composants interagissant existants de ce type afin de poser une interaction.

Dans notre exemple, si l'utilisateur demande la pose des interactions visualisées dans la figure 1, le comportement des instances de composants liés est modifié pour respecter le résultat de la fusion des différentes règles d'interactions. Le processus de fusion est brièvement décrit dans la partie 3.3.

Par exemple, à la méthode *addMeeting* de l'agenda *AgendaRainbow* est associée la règle suivante résultat de la fusion de la règle définie figure 2 à la ligne 7 instanciée successivement entre l'agenda *AgendaRainbow* et les agendas de *AgendaDavid*, *AgendaAnne-Marie* et *AgendaMireille* :

```
AgendaRainbow.addMeeting(java.lang.String _var0) ->
    AgendaRainbow._call;
    (AgendaDavid.addMeeting(_var0)
```

```
//2 AgendaAnne-Marie.addMeeting(_var0)
// AgendaMireille.addMeeting(_var0 )
```

Cette nouvelle règle exprime le fait que ajouter un rendez-vous dans l'agenda de groupe ajoute le rendez-vous dans un ordre non déterminé à chacun des participants.

Le retrait d'interactions consiste à demander au serveur d'interactions le retrait d'une interaction par `void removeInteraction(String interactionIdentifier) throws Exception`. Lors de la destruction d'une interaction, chacun des composants déliés comportant un message notifiant est prévenu afin de retirer la règle qui le concerne et de recalculer la règle résultante.

2.3. Rendre un composant interagissant

Seules les instances de composants « interagissants » peuvent être liées par des interactions comportant des règles où leurs messages apparaissent comme notifiants. Tout composant peut cependant apparaître dans la partie réaction d'une règle.

Pour rendre un composant EJB interagissant, l'utilisateur doit simplement le spécifier dans le fichier de déploiement dont la grammaire XML a été étendue. Pour Jonas [OBJ], le développeur insère la ligne suivante:

```
<jonas-interaction value="true"></jonas-interaction>
```

Pour rendre une classe Java interagissante, il suffit d'appliquer l'outil fourni avec le service d'interaction, « GENINT », qui modifie directement son byte-code.

Les impacts de ces préparations sont décrits dans la partie 4.

2.4. Exécution

Lorsqu'un composant interagissant reçoit un message qui se révèle être notifiant, la règle d'interaction qui lui est associée, est évaluée localement. Les appels entre les composants lors de cette évaluation sont alors directs et ne passent pas par le serveur d'interactions.

2.5. Navigation et Analyse du graphe d'interactions

Via le serveur d'interactions, et les objets qu'il mémorise, il est possible de naviguer dans le graphe d'interactions et de l'analyser pour déterminer des propriétés telles que l'absence de cycle ou de points de non-déterminisme. L'analyse du graphe est en cours et ne sera pas discutée dans cet article.

² Attention : la chaîne `'/'` ne note pas le début d'un commentaire mais le parallélisme des actions.

3. Langage de Spécifications des Interactions (ISL)

Le langage ISL présenté de façon intuitive dans les exemples précédents est basé sur une grammaire, incluant des opérateurs conditionnels (*if..then..else*) des opérateurs séquentiel (*;*) et parallèle (*//*), l'attente, la gestion d'exception, etc. A l'instar de l'OMG avec les approches CORBA, ce langage est indépendant des langages d'application. Si nécessaire, une séquence de code spécifique à un langage donné peut toujours être encapsulé dans un composant interagissant qui sera dynamiquement activé lors de la pose de l'interaction. La variable *this* dans un schéma d'interaction permet de désigner à l'exécution, l'interaction elle-même.

Dans la suite de cette section, nous présentons brièvement la sémantique du langage (la syntaxe est donnée en annexe) et expliquons ce qu'est le mécanisme de fusion.

3.2. Sémantique

Le langage d'interactions définit, de manière récursive, les comportements à l'aide de l'ensemble des opérateurs décrits ci-dessous (chaque opérateur décrit un type de comportement). Une description formelle peut être trouvée dans [Berger01].

- L'opérateur séquentiel (*noté ;*) spécifie que deux comportements doivent être exécutés l'un après l'autre.

- L'opérateur parallèle (*noté //*) spécifie que deux comportements peuvent être exécutés de façon concurrentielle

- L'opérateur d'invocation de méthode (*noté .*) dénote l'appel d'une méthode sur l'objet receveur.

- L'opérateur d'assignation (*noté :=*) spécifie que la valeur d'une variable est assignée à la valeur de retour d'un comportement d'envoi de message ou d'assignation.

- L'opérateur réactif d'attente (*noté _{X}*, *X étant une étiquette*) spécifie que l'exécution d'un envoi de message, d'une assignation ou d'une autre attente est contrainte par la fin de l'exécution d'un message provenant d'un autre comportement étiqueté par [X].

- L'opérateur conditionnel (*noté if then else endif*) spécifie une exécution conditionnelle d'un comportement en fonction du résultat (un booléen) d'une exécution d'un comportement d'envoi de message.

- L'opérateur de traitement d'exception (*noté try catch*) spécifie un traitement à exécuter lorsqu'une exception est détectée dans un comportement réactif.

- L'opérateur d'exception (*noté exception*) permet de lever une exception. Il ne peut être utilisé que pour exprimer le refus d'exécution du message notifiant. L'exception alors levée si elle est retournée à l'utilisateur (non attrapé dans une règle d'interaction) est soit une instance de la classe spécifiée en paramètre quand

elle appartient à la signature de la méthode notifiante, soit une *InteractionException* qui encapsule l'exception ; il s'agit dans ce cas d'une runtime exception.

- L'opérateur de délégation spécifie qu'un comportement ne comportant pas le message déclencheur de la règle est traité comme étant ce message déclencheur. Cet opérateur n'est pas défini dans la syntaxe mais est implicitement ajouté lors de l'analyse sémantique.

L'ensemble des définitions décrivant à l'aide de la sémantique naturelle de *Typol*, pour chaque type d'opérateur, la méthode d'exécution λ qui lui est associée se trouvent dans [Berger01].

3.3. Fusion

La fusion de règles d'interactions est nécessaire lorsque plusieurs interactions sont appliquées simultanément sur le même message notifiant d'un composant³. Elle est gérée de façon dynamique à chaque ajout ou retrait d'une règle d'interaction sur un composant.

La fusion se formalise par un ensemble fini de règles de fusion définies en fonction des opérateurs du langage ISL et sur un ensemble fini de règles d'équivalence. La fusion des règles ISL respecte les propriétés suivantes:

- la cohérence de l'ensemble des règles est conservée, en particulier la fusion n'entraîne pas d'attente non explicite ;
- quel que soit l'ordre de pose des interactions, si la fusion est acceptée, le comportement du système doit être équivalent puisque plusieurs utilisateurs peuvent intervenir simultanément ; l'opération de fusion est donc commutative;
- la fusion des règles d'interaction est également transitive pour permettre la pose de l'ensemble des interactions dans un ordre quelconque.

Le mécanisme de fusion que nous proposons génère, quand la fusion est possible, une unique règle d'interaction qui sera exécutée à la place de l'ensemble des règles dont elle est la fusion et qui préserve la sémantique du tout. L'ensemble des règles de fusion est décrit dans [Berger01].

Lors de la pose des interactions, les règles sont partiellement instanciées (substitutions de valeurs aux variables) c'est-à-dire que les variables dénotant les composants participants à l'interaction sont connues, alors que les variables

³ La fusion n'intervient que pour des règles dont le message notifiant présente la même signature (même nom de méthode, même arité et types d'arguments) respectant la hiérarchie des types) et portent sur le même objet receveur. Dans ce cas, les règles sont dites unifiables.

dénotants les paramètres d'appels des messages ne le sont pas. Par exemple, pour la règle lignes 22 à 26 du schéma d'interaction *security* si nous l'instancions entre l'entité désignée par *ComposantSecurité* et l'agenda désigné par *AgendaAnne-Marie*, nous obtenons la règle partiellement instanciée suivante :

```
AgendaAnne-Marie.addMeeting( java.lang.String _var0 )->
  if ComposantSecurite.check(_call) then
    AgendaAnne-Marie._call
  else exception "unauthorized user"
endif
```

Les règles de fusion sont basées sur quelques principes énoncés dans la suite.

Les conditionnelles sont prioritaires à la fusion

Certaines règles d'interaction permettent de conditionner l'exécution du message notifiant. Dans ce cas nous avons choisi de rendre prioritaire, lors de la fusion, la vérification de la condition (cf. figure 4).

Si R1 est une règle partiellement instanciée dont la partie droite correspond à une conditionnelle que nous noterons : M-> if C then A1 else A2 et si R2 est une règle dont la partie droite ne correspond ni à la levée d'une exception, ni à une délégation de code, la fusion des deux règles est égale à une règle dont la partie droite est une conditionnelle dont la condition est celle définie dans R1 et dont les branches correspondent à la fusion de R2 avec respectivement les règles M-> A1 et M-> A2.

La fusion de deux règles conditionnelles est commutative par équivalence sous l'hypothèse que la vérification des conditions n'a pas d'effet de bord et donc que l'ordre des tests n'a pas d'influence.

La levée d'exception est absorbante pour la fusion

Une règle qui lève une exception exprime que l'exécution du message notifiant est incohérente ou source d'erreur. De ce fait, toute autre pose d'interaction est inhibée par la fusion (cf. figure 4). Deux règles qui lèvent des exceptions ne peuvent pas fusionner, à l'exécution, car nous ne saurions pas quelle exception lever.

L'agenda Anne-Marie est simultanément lié aux composants *ComposantBD*, *ComposantSecurité* et *AgendaRainbow* par les interactions respectivement instances des schémas *persistance*, *security* et *group*. La fusion des règles (22 à 26) et (15 à 18) donne la nouvelle règle (au nommage des variables près) :

```
AgendaAnne-Marie.addMeeting( java.lang.String _var0 ),
java.lang.String owner->
  if ComposantSecurite.check(_call)
```

```

then
  AgendaAnne-Marie._call ;
  owner := AgendaAnne-Marie.getOwner()
  ComposantBD.store(owner, AgendaAnne-Marie)
else
  exception "unauthorized user"
endif

```

La fusion sur la conditionnelle a d'abord été opérée puis, dans la première branche de la conditionnelle, c'est la fusion sur le message notifiant (élément neutre pour la fusion) qui a été utilisée dans la seconde c'est celle sur l'exception.

Figure 4 : Fusion des opérateurs de condition, d'envoi de message et d'exception

Séquence et concurrence

La fusion des opérateurs de séquence et de concurrence doit se faire en ne rajoutant pas d'attente non voulue. Lorsque la fusion peut introduire une séquence non voulue l'opérateur d'attente est utilisé (cf. figure 5). L'agenda *Rainbow* est simultanément lié aux composants *David*, *Mireille*, *Anne-Marie* par les interactions instances du schéma *group*. La règle résultant de la fusion de ces interactions qui a été présentée dans la partie 2.2. utilise les règles de fusion basées sur la fusion des opérateurs // et ;.

Transitivité

Les interactions sont posées à l'exécution les unes après les autres. Quel que soit l'ordre dans lequel elles sont posées, si elles sont simultanément cohérentes (c'est-à-dire si la fusion est possible) la fusion doit conduire à des comportements équivalents du système. Cette propriété a été démontrée dans [Berger01] par récurrence sur la profondeur des règles. Nous la présentons ici de façon empirique au travers de l'exemple suivant (cf. figure 5).

L'agenda de *AgendaMireille* est lié à plusieurs composants (cf. figure 1), voici le résultat de la fusion pour la méthode *addMeeting* :

```

AgendaMireille.addMeeting( java.lang.String 0 ),
java.lang.String owner
-> if ComposantSecurite.check(_call) then
  ([0] AgendaMireille._call
   ; AfficheurTeam.notify(_call))
  //Afficheur2.notify(_call)
  //Afficheur1.notify(_call)
  // (AgendaMireille.getOwner())_{0}

```

```

        ; ComposantBD.store(owner,DB)
    else exception "unauthorized user"
    endif

```

Figure 5 : Fusion et opérateur d'attente

4. Mise en œuvre

Il existe à ce jour, 2 versions du service d'interactions, qui partagent un grand nombre de packages. La première est écrite en Java pur et permet de poser localement (sans passer par le réseau) des interactions. Une seconde version en Java RMI permet de faire interagir des composants RMI et/ou composants EJBs.

4.1. Entités interagissantes

Un composant interagissant doit pouvoir remplir les tâches suivantes :

- fusionner ou (dé-fusionner) dynamiquement des règles d'interactions (cf. 3.2)
- donner la main à un contrôle local lors de la réception de messages « déclenchant »,
- adresser des messages directement aux composants liés sans passer par le serveur d'interactions.

Les composants doivent donc subir des transformations de code pour supporter les interactions, en particulier le contrôle de la réception de message et l'ajout des fonctionnalités d'attachement et de détachement des règles d'interactions i.e. (`int addRule(String methodName, RuleInstance rule), removeRule(String methodName, int ruleNumber)`).

Dans le cas d'un objet java pur ou RMI, le byte-code de l'objet est transformé par méta programmation en utilisant BCEL [BCEL] et un mécanisme de capture de message (wrapper) est mis en place pour déléguer quand cela est nécessaire l'exécution du message à l'interprétation d'un arbre de règles [Berger99, Bartorello01].

Dans le cas d'un composant EJB, les interactions sont prises en charge par les objets d'interposition générés au même titre que les autres services standards. Dans JOnAS, ils sont obtenus par l'outil de génération de code GenIC que nous avons modifié. Le code généré dépend des informations décrites dans le fichier de déploiement. Le programmeur écrit donc ses beans indifféremment du fait qu'ils supportent ou non les interactions. Il lui suffit de l'indiquer dans le fichier de déploiement dont la grammaire XML a été étendue. L'intégration aurait également pu se faire par modification du bean au moyen de la méta programmation au

détriment de la composition du service d'interactions avec les autres services standards puisque la logique de composition de services est située au niveau des objets d'interposition.

4.2. Interopérabilité et interactions

Un des défis auquel nous avons été confronté était de permettre à des composants situés sur des plates-formes différentes d'interagir. Les objets interagissants peuvent être des objets locaux, des objets RMI, des beans, il s'agit de pouvoir les manipuler de la même façon tant au niveau du serveur que dans les communications directes inter-composants. Pour cela, il s'agissait à la fois de les désigner de façon uniforme et également de minimiser les différences entre les différentes implémentations. En effet, le parsing, la gestion de l'arbre, la fusion sont autant d'éléments communs à toutes les versions réalisées jusqu'à présent.

Pour cela, nous avons encapsulé les références dans des objets « `interactingObject` » qui gèrent les envois de message et présentent une même interface. Ces manipulations sont bien sûr transparentes du point de vue de l'utilisateur.

Une modification du protocole de communication pour passer à RMI-IIOP devrait nous permettre de proposer l'interopérabilité avec les objets C++ que nous avions initialement dans le monde Corba [Berger99].

4.3 Surcoût d'une réception de message dans le modèle EJB.

Pour tous les composants interagissant, le surcoût lié au mécanisme d'interaction, dans le cas d'une interaction vide est celui d'un test. Dans le cas où l'interaction est non vide, le message lui-même est exécuté en utilisant la méthode d'invocation dynamique de l'API de réflexion de Java.

5. Conclusion

Nous avons proposé un service d'interactions qui permet l'adaptation dynamique des composants par la définition de schémas et la pose et la destruction de règles à l'exécution. L'utilisateur exprime alors les interactions au niveau de l'applicatif, d'une manière que nous pensons « naturelle ». Basée sur le langage ISL, la fusion des interactions assure la commutativité et la transitivité lors de la pose des interactions. Plusieurs intervenants peuvent demander la pose ou le retrait d'interactions. Quelque soit l'ordre des requêtes le résultat sera équivalent. Les incohérences entre règles sont également détectées par ce mécanisme de fusion. Deux implémentations du service qui permettent de faire interagir des objets locaux

et des composants RMI et EJB ont été brièvement présentées. Ces implémentations sont disponibles sur le site <http://rainbow.essi.fr/noah>. Les interactions ont été utilisées pour aider à la gestion dynamique de bases de connaissances [Dery01, Dery02], à l'aide active dans la manipulation de frameworks [Rapicault02] et à l'intégration de services [Nano02, Blay02] et une étude pour la gestion de l'adaptabilité dans le cadre d'applications nomades est en cours.

En explicitant les interactions entre les composants non seulement la programmation est plus facile, l'interopérabilité naturelle, mais il devient possible de naviguer dans les graphes d'interactions, de les analyser et de vérifier la compatibilité des interactions au moins localement. L'évolution du serveur d'interactions de composants RMI à composants EJB, nous permettra de lui adjoindre des services tels que la persistance et les transactions. L'introduction de nouveaux opérateurs dans le langage ISL, comme l'appel asynchrone, le déclenchement d'une réaction après observation d'un pattern, t à l'étude avec bien sûr comme objectif de préserver les propriétés relatives à la fusion. L'analyse des graphes d'interactions est un autre point sur lequel nous travaillons actuellement. En effet les interactions apportent une couche supplémentaire d'abstraction et les outils permettant de la contrôler nous semblent en effet indispensables [Kienzle02].

6. Bibliographie

- [BCEL] *Byte Code Engineering Library*, <http://bcel.sourceforge.net/>
- [Bartorello01] M. Bartorello, H. Maguin, M. Blay-Fornarino, A.-M. Dery, M. Riveill, *Adjonction de services au sein d'un serveur EJB*, Journées composants : flexibilité du système au langage, 25 et 26 octobre 2001, Besançon
- [Berger99] BERGER L., « Évaluation Dynamique des Messages dans un Environnement Compilé et Distribué : Application aux Interactions entre Objets Distants », *Colloque Langages et Modèles à Objets (LMO'99)*, Villefranche-sur-Mer (France), janvier 1999, p. 131-146.
- [Berger01] BERGER L., « Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés: le modèle MICADO », PhD thesis, Université de Nice, 2001.
- [Berger02] BERGER L., « Support des interactions dans les systèmes objets et componentiels », *Numéro spécial de L'OBJET : Coopération dans les systèmes à objets*, vol. 7, 2001-2002.
- [Blay02] M. Blay-Fornarino, A.M. Dery, M. Riveill, *Towards Dynamic Configuration of Distributed Applications*, Workshop Aspect Oriented Programming for Distributed Computing Systems, July 2-5, Vienna, ICSCS 2002
- [Dery02] Dery, A.M. and Blay-Fornarino, M. and Moisan, S., *Apport des interactions pour la distribution des connaissances*, Numéro spécial de L'OBJET : Systèmes distribués et connaissances, Volume 8 n° 4, 2002
- [Dery01] Dery, A.M. and Blay-Fornarino, M. and Moisan, S. *Distributed access knowledge-based system: Reified Interaction Service for Trace and Control*, 3rd International

Symposium on Distributed Object Applications (DOA 2001), Rome, Italy, September 17-20, 2001

- [EJB00] «Enterprise Javabeans Specification. Version 1.1», janvier 2000, Sun Microsystem Inc. <http://java.sun.com/products/ejb/docs.html>
- [Kienzle02] Jörg Kienzle, Rachid Guerraoui *AOP:Does It Make Sense? The Case of Concurrency and Failures*, (Ed.):ECOOP 2002, LNCS 2374 ,pp.37 –61, Springer-Verlag Berlin Heidelberg 2002
- [Nano02] Olivier Nano, Mireille Blay-Fornarino, Anne-Marie Dery, Michel Riveill *An abstract model for integrating and composing services in component platforms*, Seventh International Workshop on Component-Oriented Programming (in conjunction with ECOOP'2002), Malaga, Spain, June 10, 2002
- [OBJ] OBJECTWEB, «JOnAS: Java™ Open Application Server», <http://www.objectweb.org/jonas>, ObjectWeb Open Source.
- [OMG01] OMG, « CORBA 2.4.2 Interceptors chapter », February 2001, OMG Document formal/01-02-57, <http://www.omg.org>.
- [Pawlak 01] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, *JAC: A Flexible Framework for AOP in Java.*, Reflection'01, Kyoto, Japan.
- [Rapicault02] Pascal Rapicault, *Modèles et techniques pour spécifier, développer et utiliser un framework : une approche par méta-modélisation*, PhD thesis, Université de Nice, 25 mai 2002
- [Riveill 00] RIVEILL M., BRUNETON E., *JavaPod: "an Adaptable and Extensible Component Platform"*, *RM'2000, Workshop on Reflective Middleware*, New York, USA, April 2000.
- [Riveill 01] RIVEILL M., «Microsoft NET, programmer le Web », *Développeur Référence*, vol. 10, Mars 2001.
- [Sarradin01] SARRADIN F. ET LEDOUX T., « Adaptabilité dynamique de la sémantique de communication dans Jonathan », *Colloque Langages et Modèles à Objets (LMO'01)*, *Hermès Science, L'objet-7/2001*, Le Croisic, France, janvier 2001.
- [Vadet01] Mathieu Vadet, Philippe Merle, 'Les conteneurs ouverts dans les plates-formes à composants', Journées Composants 2001, Besançon, 25-26 Octobre.

ANNEXE:

```

Interaction      →      'interaction' ClassName '(' Member { ','
Member } ')' [Extensions] [Class] '{ Rules }'
ClassName       →      ident { '.' ident }
Member          →      ClassName ident
Extensions      →      'extends' Extension { ',' Extension }
Extension       →      ClassName '(' ident { ',' ident } ')'
Class           →      'implements' ClassName
Rules           →      Rule { ',' Rule }
Rule            →      LeftSide '->' Reaction
LeftSide        →      NotifyingMessage { ',' Variable }
NotifyingMessage →      Selector '.' '*' | MessageDecl
MessageDecl     →      Selector '.' '(' [ FormalParameter { ','
FormalParameter } ] ')'
FormalParameter →      ClassName ident
Variable        →      ClassName ident
Reaction        →      ReactionBloc SEQ Reaction | ReactionBloc
'/' Reaction
ReactionBloc   →      [ FrontQualifiers ] ReactionBody [
BackQualifiers ]
ReactionBody   →      Message | Assignment | Conditionnal | '('
Reaction ')'
FrontQualifiers →      '[' ident ']' { '[' ident ']' }
BackQualifiers →      '_' '{ ident { ident } }'
Exception      →      'exception' ClassName
Assignment     →      ident ':=' Message
Conditionnal   →      'if' Message 'then' Reaction 'else' Reaction
'endif'
Message        →      Invocation | CallMessage
Invocation     →      Selector '(' [Parameter { ',' Parameter } ]
')'
CallMessage    →      Selector '.' '_call'
Selector       →      ident '.' ident | 'this' '.' ident
Parameter      →      ident | const | '_call'
Const          →      int | string | 'TRUE' | 'FALSE'

```