# Communication par événements dans les modèles à objets

Emmanuel Lenormand

# THÈSE

présentée par

Emmanuel Lenormand

pour obtenir le titre de

Docteur de l'Université Joseph Fourier – Grenoble I

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité : Informatique

# Communication par événements dans les modèles à objets

Date de soutenance : 7 novembre 1996

Composition du jury :

| | |
|---|---|
| Sacha Krakowiak | Président |
| Jean-Marc Geib | Rapporteur |
| Claude Godart | Rapporteur |
| Christine Collet | Examinateur |
| Roland Balter | Directeur de thèse |

# THÈSE

présentée par

Emmanuel Lenormand

pour obtenir le titre de

Docteur de l'Université Joseph Fourier – Grenoble I

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité : Informatique

# Communication par événements dans les modèles à objets

Date de soutenance : 7 novembre 1996

Composition du jury :

| | |
|---|---|
| Sacha Krakowiak | Président |
| Jean–Marc Geib | Rapporteur |
| Claude Godart | Rapporteur |
| Christine Collet | Examinateur |
| Roland Balter | Directeur de thèse |

À Lix

Pour Augustin, Basile...

À l'heure où s'achèvent ces quatre années de thèse, mes remerciements les plus sincères vont à :

Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourier de Grenoble, qui a accepté de présider le jury chargé de juger ma contribution. Il m'a accueilli au sein de ce laboratoire et a suivi depuis plus de cinq ans l'évolution de mon travail : je lui en suis infiniment reconnaissant.

Monsieur Jean–Marc Geib, Professeur à l'Université Scientifique et Technologique de Lille, et Monsieur Claude Godart, Professeur à l'École Supérieure des Sciences et Techniques de l'Ingénieur de Nancy, qui ont accepté d'évaluer et de juger mon manuscrit et mon travail.

Madame Christine Collet, Maître de Conférence à l'Université Joseph Fourier de Grenoble, pour m'avoir fourni des références essentielles et pour avoir accepté de participer au jury de ma thèse.

Monsieur Roland Balter, Professeur à l'Université Joseph Fourier de Grenoble, qui a dirigé mon travail. Malgré ses multiples charges et responsabilités, il a su demeurer disponible pour m'encourager et me soutenir aux jours sombres.

Madame Fabienne Boyer–Dechamboux, Maître de Conférence à l'Université Joseph Fourier de Grenoble. Je sais combien le contenu de ce manuscrit doit à ses remarques et à nos discussions.

Messieurs Serge Lacourte, André Freyssinet et Marc Hermann, de la Société Bull, qui ont assuré, contre vents et marées, le support technique de OODE sans lequel je n'aurais pas pu réaliser les expérimentations nécessaires. Je les prie également de m'excuser pour le temps que j'ai pu leur prendre.

L'ensemble des personnes de ce qui fut jadis l'Unité Mixte Bull–IMAG et qui est devenu le projet Sirac. Tous ont contribué à créer l'ambiance de travail si particulière, faite de convivialité et de sérieux, qui régnait dans cette équipe.

# Introduction

## 1 Motivations

L'intérêt d'un travail mené en groupe et de manière coordonnée est avéré dans de nombreux domaines du monde économique, que ce soit pour la gestion de la complexité, la productivité ou l'efficacité, et il est naturel qu'il reçoive un écho de plus en plus intense dans la définition des outils informatiques. Deux facteurs expliquent et accentuent cette tendance. Le premier est la place grandissante prise par l'informatique dans les processus de décision, de gestion ou de production, où le travail en groupe est souvent une nécessité. Le second facteur réside dans l'accroissement de la puissance des systèmes et moyens informatiques qui permettent désormais la mise en place efficace d'outils adaptés aux nouveaux besoins.

Cet accroissement bénéficie certes de la puissance accrue des matériels et des réseaux, mais également de progrès réalisés dans le domaine des outils de production de logiciel. Les principaux apports vont de l'aide à la conception d'applications jusqu'à la définition de mécanismes avancés de synchronisation et de partage, en passant par la gestion de la répartition. Les modèles à objets ne sont pas étrangers à ces progrès. Initialement conçus et pensés en termes de langages pour la production de logiciels, leurs propriétés ont été utilisées dans de nombreux autres domaines. Plus particulièrement, les systèmes répartis ont largement bénéficié de cette approche. Notre laboratoire a acquis au cours du projet Guide une expertise dans ce domaine et contribué à ses avancées, notamment dans la définition d'un modèle à objets répartis, partageables et persistants [Balter 91]. Ainsi, les modèles et langages à objets répartis offrent une base solide pour la programmation d'applications appelées à fonctionner sur des réseaux locaux ou à grande distance. Les applications coopératives font partie de cette classe ; les techniques à base d'objets semblent donc appropriées pour leur conception et leur programmation et, prenant Guide pour base, nous nous sommes engagés dans cette voie.

Néanmoins, certains aspects du travail coopératif, et par conséquent de ces applications coopératives, ne sont que partiellement couverts par Guide et les modèles et langages à objets actuels. Nous avons évoqué la nécessité d'une coordination entre les membres d'un groupe au travail et cette coordination est justement un aspect primordial pour lequel les modèles à objets demeurent insuffisants sur quelques points. La coordination est affaire de communication et de

synchronisation et les schémas qu'elle met en œuvre sont extrêmement divers : demande de service ou diffusion de message, protocole de vote ou de négociation ou encore gestion cohérente d'un espace partagé. À défaut de mécanismes adaptés à un travail aisé pour le concepteur, ces schémas réclament un effort de programmation important compte tenu des abstractions fournies par les modèles et langages à objets.

Cette carence est particulièrement nette, entre autres, pour la communication asynchrone. En effet, le mode de communication le plus répandu dans les modèles à objets, répartis comme Guide ou non, repose sur l'appel de méthode, qui est fondamentalement un mécanisme synchrone. Cette technique ne permet pas de manière directe et simple de réaliser certains schémas de coordination tel que par exemple la diffusion d'un message. L'asynchronisme, découplage dans le temps, s'accompagne souvent dans ce contexte de l'anonymat, découplage dans l'espace : de même que les deux extrémités de la communication ne se synchronisent pas, elles ignorent leurs identités respectives. Or, pas plus que l'asynchronisme, l'anonymat n'est une caractéristique de la communication par appel de méthode.

Définir un complément qui, dans le cadre des modèles à objets en général et dans celui de Guide en particulier, remédierait à cette double lacune constitue par conséquent un objectif nécessaire pour la mise en œuvre d'applications coopératives avec des techniques à objets.

## 2 Objectifs

Le premier objectif de ce travail est par conséquent de compléter Guide : il offre un certain nombre de fonctions requises pour la programmation d'applications coopératives, tel que le partage, la synchronisation et la répartition, mais ne propose aucun mode de communication asynchrone et anonyme.

Dans ce cadre, les axes de notre travail concernent deux aspects..

La proposition et l'étude d'un tel modèle de communication dans le cadre général des modèles à objets constituent le premier aspect. En effet, les lacunes de Guide ne lui sont pas propres et concernent bon nombre d'autres modèles, ce qui justifie une étude générale. Cette étude comporte trois points :

1. définir les éléments du mécanisme et le principe de son fonctionnement ;

   Cette définition comprend à la fois l'identification des caractéristiques désirées du mécanisme et la définition d'un cadre susceptible de les remplir d'un point de vue fonctionnel.

2. examiner comment ces éléments peuvent être exprimés et intégrés au sein d'un modèle à objets ;

Deux aspects revêtent une importance particulière dans cette partie. Le premier aspect a trait à la définition des éléments qu'utilise le mécanisme, de leurs caractéristiques, de leur mode de désignation et de déclaration. Le second aspect concerne l'adéquation de cette définition avec les modèles à objets. Les points soulevés pour la définition précédente doivent être pris en compte, de même que l'impact de cette définition sur les caractéristiques canoniques – encapsulation, notion d'interface, héritage – des modèles à objets.

3. étudier le modèle d'exécution qui régit le fonctionnement du mécanisme.

   La description et la compréhension du comportement dynamique du système revêt une double importance. D'une part, ce comportement conditionne le type d'utilisation qui sera faite du mécanisme et d'autre part, le contexte des applications coopératives où de nombreuses activités s'exécutent en parallèle requiert que soient clairement définis les structures et schémas d'exécution propres du mécanisme. Enfin, nous souhaitons montrer combien ce modèle d'exécution est contraint par celui du modèle hôte.

Le second aspect de l'étude consiste à appliquer et à réaliser cette proposition dans le cadre du modèle Guide.

Cette partie du travail comprend en premier lieu l'adaptation des principes définis auparavant aux caractéristiques de ce modèle particulier. Cette adaptation obéit à des critères qui sont fonction à la fois des caractéristiques de Guide et de l'utilisation envisagée pour le mécanisme de communication asynchrone.

En second lieu, la réalisation d'un prototype expérimental mettant en œuvre le modèle proposé doit permettre d'évaluer le mécanisme dans le contexte choisi. Cette évaluation doit montrer l'utilité du mécanisme et son adéquation aux besoins.

# 3 Cadre du travail

Ce travail s'est déroulé à la charnière des projets Guide et Sirac.

Le projet Guide (Grenoble Universities Integrated Distributed Environment) a été lancé en 1986 comme un projet commun au Laboratoire de Génie Informatique de l'IMAG et au Centre de Recherche Bull et s'est poursuivi de 1990 à 1994 dans l'Unité Mixte Bull–IMAG.

L'objectif de ce projet [Balter 91] était la construction d'un système informatique permettant de développer, de mettre au point et d'exécuter des applications réparties sur un réseau local à haut débit interconnectant des postes de travail et des serveurs. Le système assurait une gestion globale et intégrée des ressources matérielles et des informations. Il devait permettre de développer des applications coopératives (génie logiciel et gestion de documents), en utilisant des méthodes de structuration à base d'objets. Une plate–forme d'exécution pour de telles applications à été développée. Un langage à objets donnant accès aux fonctions fournies par cette plate–forme a également été conçu.

Sirac (Systèmes Informatiques Répartis pour Applications Coopératives) est un projet mené au sein de l'INRIA. Il fait suite au projet Guide et s'appuie sur ses résultats. Son objectif [Balter 95] est de concevoir et de réaliser un environnement pour le développement et l'exécution d'applications réparties. Les recherches sont menées dans les deux thèmes suivants :

- Construction d'applications réparties. L'objectif est de fournir des outils répondant à deux besoins : a) construire des applications réparties en combinant des techniques de programmation à base d'objets et des techniques d'intégration de composants ; b) faciliter l'administration, la configuration et l'évolution de ces applications.
- Services pour le support d'objets partagés persistants répartis. L'objectif est de fournir un support adaptable et efficace utilisable pour la construction de plates–formes à objets répartis et de serveurs d'objets, en utilisant une mémoire virtuelle partagée répartie. Sa conception tient compte des nouvelles infrastructures matérielles (grandes mémoires virtuelles et réseaux rapides) et utilise des outils génériques pour le maintien de la cohérence.

Le travail de cette thèse prend place dans le premier thème, où il s'attache à compléter l'ensemble des modes d'interaction nécessaires à la construction des applications coopératives.

## 4 Plan de la thèse

Le **chapitre I** présente tout d'abord OLAN, l'axe de Sirac dans lequel s'insère ce travail. Il se poursuit par une étude de cas au cours de laquelle, à travers trois exemples, les besoins spécifiques des applications coopératives en terme de coordination sont mis en lumière. L'accent est mis au terme de cette étude sur ce qui constitue le cœur de la thèse, la **communication asynchrone et anonyme**.

Le **chapitre II** propose un panorama de différents mécanismes de ce type de communication dans des domaines variés de l'informatique. L'analyse de ces travaux est concentrée sur les caractéristiques mises en lumière à la fin du chapitre I. Le résultat de cette étude motive l'approche retenue pour offrir ce mode de communication dans un modèle et un langage à objets, à travers l'utilisation d'**événements**.

L'étude de la communication événementielle dans ce contexte d'un environnement à objets est présentée au **chapitre III**. La définition des différents éléments impliqués dans le mécanisme, leur déclaration et leur désignation sont examinées dans un cadre général avec un minimum d'hypothèses sur le modèle à objets hôte. De même, les aspects relatifs au modèle d'exécution du mécanisme sont discutés. Ce chapitre se conclut par l'examen des conditions d'intégration de ce mécanisme dans un modèle à objets, en particulier sur l'impact que cette intégration pourrait avoir sur les caractéristiques canoniques de tels modèles.

Le **chapitre IV** montre comment l'étude générale du chapitre précédent peut être appliquée dans l'environnement particulier de Guide, un modèle à objets passifs, persistants, répartis et partageables. Ces caractéristiques justifient les choix de conception réalisés face aux alternatives que laissait l'étude du chapitre III.

Ces propositions ont fait l'objet de réalisations qui sont décrites au **chapitre V**. Utilisant une plate–forme d'exécution conforme au modèle Guide, un prototype du service de communication événementielle a été conçu et utilisé dans une application. Ce chapitre est aussi l'occasion d'une évaluation des propositions et réalisations présentées.

La **conclusion** de ce manuscrit rappelle les objectifs que nous poursuivions en réalisant ce travail et la démarche que nous avons suivie. Nous y présentons ensuite les apports de notre étude, avant de discuter et d'évaluer les résultats obtenus. En dernier lieu, nous indiquons les perspectives qu'ouvre notre travail et les prolongements qui peuvent en découler.

L'**annexe A** présente des exemples de programmes qui concernent l'interface d'accès au mécanisme de communication événementielle et son utilisation dans des applications.

# Chapitre I

# Coordination et communication asynchrone

Les applications coopératives réparties constituent un champ d'investigation extrêmement vivant dans la communauté informatique et même au delà [Baecker 92]. Ces applications requièrent la mise en place d'infrastructures complexes et plus élaborées que d'autres applications [Ellis 91]. Elles utilisent en effet au plus haut degré les capacités de communication, de répartition, de gestion de données multimédia des systèmes sur lesquels elles sont réalisées. Un des objectifs du projet Sirac dans lequel s'intègre ce travail est de fournir un ensemble d'outils et de services pour faciliter la construction et l'administration de telles applications [Balter 95].

Nous présentons dans un premier temps l'approche retenue dans Sirac pour atteindre les objectifs du projet. Puis, après avoir indiqué notre définition d'une application coopérative, nous analysons dans la suite du chapitre les aspects liés à la coordination, c'est–à–dire aux interactions dans ces applications.

Même si à cette fin nous considérons la coordination dans toutes ses dimensions, notre intérêt portera plus particulièrement, et conformément à ce que nous indiquions en introduction, sur les problèmes relatifs à la communication asynchrone entre les intervenants des coopérations. Cet intérêt est motivé par le cadre dans lequel nous nous plaçons – les modèles à objets – et le peu de solutions offertes dans ce contexte.

Nous présentons trois exemples d'applications coopératives et les analysons en nous appuyant sur le modèle proposé dans Sirac. Cette étude nous permet alors d'exhiber les mécanismes système nécessaires au support de la coordination dans ce type d'applications, en insistant sur le besoin d'un mécanisme de communication asynchrone.

## I.1 OLAN

L'approche retenue au sein du projet Sirac est de construire une application répartie comme un ensemble de composants coopérants. La description globale d'une application repose sur la définition d'un modèle d'assemblage de **composants** fondé sur deux concepts : la description des composants et la description des interactions entre ces composants [Bellissard 96a]. Cette séparation entre composants et interactions au niveau de la description permet en particulier de configurer à la carte des applications en fonction de protocoles de leur choix.

### I.1.1 Composants

La description d'un composant distingue son interface et son implémentation. Les services offerts et les conditions requises par l'utilisation d'un composant sont définis par son interface, qui constitue la seule partie visible du composant. L'interface comporte en outre un ensemble d'attributs utilisés pour le contrôle et l'administration du composant.

### I.1.2 Connecteurs

Les interactions entre composants représentent les schémas de coordination dans lesquels les composants sont impliqués. Le terme **coordination** regroupe l'ensemble des techniques qui assurent la communication et la synchronisation entre des modules logiciels [Carriero 92]. Dans le cadre de notre modèle, la coordination entre composants est définie par des **connecteurs**. Un connecteur définit un ou plusieurs composants sources, un ou plusieurs composants cibles et décrit entre ces sources et ces cibles un protocole de communication et de synchronisation. Ces protocoles peuvent être simples – appel de méthode ou envoi de message – ou beaucoup plus complexes – protocoles de vote ou de négociation.

### I.1.3 Architecture et mise en œuvre

Le support d'exécution de ce modèle (cf figure Fig. 1.1) est défini par une machine virtuelle, dénommée OLAN, au dessus de laquelle des applications peuvent être construites. Cette machine virtuelle définit la représentation des éléments du modèle à l'exécution et les mécanismes qui le mettent en œuvre.

La définition de mécanismes pour le support des connecteurs est donc nécessaire à la conception de cette machine virtuelle. Des connecteurs sont fournis aux programmeurs d'applications et doivent leur permettre d'exprimer simplement les interactions qui existent entre les différents composants de leurs applications. Les mécanismes fournis permettent également aux programmeurs de définir leurs propres connecteurs en fonction de leurs besoins.

La description des applications au dessus de cette machine est faite dans un langage de configuration, le langage OCL [Bellissard 96b]. La configuration d'une application recouvre deux aspects : un aspect génie logiciel, avec la description de son organisation et de son mode de déploiement et un aspect administration. À l'aide de ce langage sont ainsi définis les composants et leur interface et les connecteurs qui les relient.

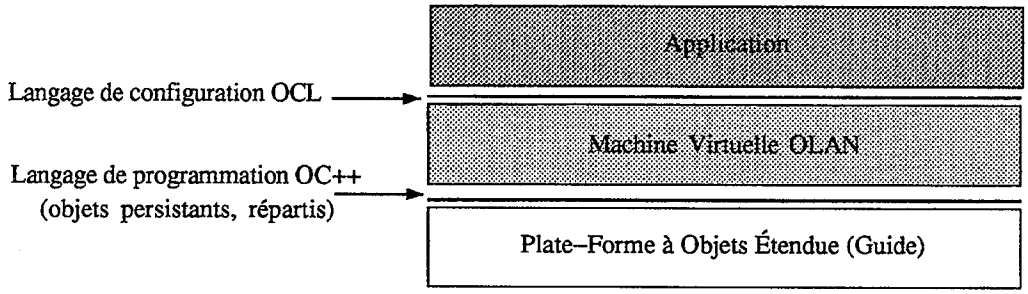


*Fig. 1.1 : Support d'exécution du modèle de composants*

La plate–forme support de cette machine virtuelle est une plate–forme à objets passifs, persistants, répartis et partageables, conforme au modèle Guide. Elle n'offre pas en l'état l'ensemble des fonctions nécessaires à la mise en œuvre des connecteurs et doit être étendue afin de répondre aux besoins que nous mettons en évidence dans la suite de ce chapitre. Parmi les extensions nécessaires figure un mécanisme de communication asynchrone qui complète les possibilités qu'offrent l'appel de méthode (synchrone) et les objets partagés.

Nous étudions dans la suite trois exemples significatifs d'applications coopératives : une application d'édition coopérative, une application de type « flot de travail » et un environnement intégré de développement de logiciels. Les descriptions que nous donnons sont volontairement simplifiées, notre objectif étant la mise en évidence de caractéristiques clés.

## I.2 Applications coopératives : définition

Une application coopérative est communément perçue comme un ensemble d'activités différentes menées par plusieurs utilisateurs et qui sont combinées pour réaliser une certaine tâche [Ellis 91]. Un éditeur coopératif permet par exemple à plusieurs personnes de travailler simultanément à la rédaction d'un document commun.

Néanmoins, nous pensons que cette définition peut être étendue à des applications qui plus généralement associent, non plus seulement plusieurs utilisateurs,

mais plusieurs modules ou entités ou composants logiciels en vue de combiner leurs actions. Un environnement de développement intégré, dans lequel plusieurs outils indépendants (compilateur, débogueur, éditeur, etc.) sont combinés pour assister un programmeur, établit des coopérations entre ces outils [Boyer 95]. Ainsi, dans cette nouvelle acception, une application coopérative met en jeu des modules logiciels répartis qui coopèrent.

Elle est représentée dans le modèle OLAN par un ensemble de composants liés entre eux par des connecteurs qui réalisent les interactions qui existent entre eux.

## I.3 Application d'édition coopérative

L'édition coopérative suppose un partage harmonieux de ressources communes et la gestion de leur cohérence.

### I.3.1 Description

Un éditeur coopératif permet à plusieurs utilisateurs de travailler sur un même document tout en assurant la cohérence du document partagé. Tous les utilisateurs peuvent voir le document évoluer au rythme des modifications de chacun. Cependant, si tous partagent le même document, tous n'ont pas la même vue sur ce document. C'est au seul niveau du document qu'est assurée la cohérence.

Cette contrainte de cohérence implique notamment que l'édition du document ne doit pas être anarchique : on doit gérer les conflits d'accès au document (ou à des parties du document si l'édition simultanée est autorisée) en édition et adopter une politique pour les régler.

Les problèmes que pose la coordination sont donc ici relatifs à la mise en cohérence des vues selon les modalités décidées par chaque utilisateur et à la négociation des droits sur les parties du document.

La mise–à–jour des vues des différents utilisateurs peut être réalisée de plusieurs façons. La première consiste à contraindre un processus de surveillance à scruter régulièrement pour le compte de chaque utilisateur le document partagé et à effectuer la mise–à–jour si besoin est. Cette méthode présente l'inconvénient de multiplier les accès peut–être inutiles à l'espace partagé. Une méthode moins coûteuse en accès consiste à informer tous les intervenants lorsqu'une modification a eu lieu. Ceux–ci peuvent alors la prendre en compte de la manière qui leur semble appropriée. Nous privilégions cette dernière méthode.

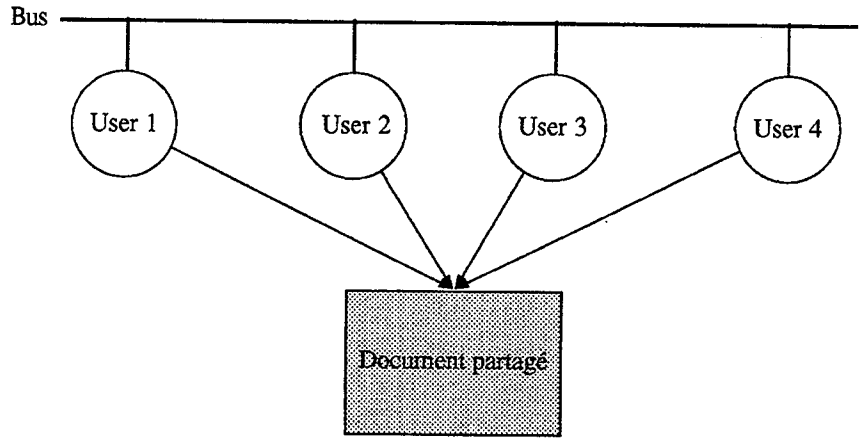


*Fig. 1.2 : Organisation logicielle d'une session d'édition coopérative*

La figure Fig. 1.2 donne une vision logique d'une session d'édition coopérative. Chaque utilisateur possède une instance de l'éditeur et accède à un document partagé entre toutes les instances. Cette vision n'est que logique car le document peut être physiquement dupliqué dans l'espace de travail de chaque utilisateur ou bien partitionné entre ces différents espaces. Entre les utilisateurs interviennent des communications qui n'utilisent pas le médium du document partagé mais un canal de communication que nous pouvons assimiler à un bus. C'est par ce biais que sont émises les notifications de modifications de document, ou les messages de négociation des droits sur le document. Les communications qui interviennent sur ce canal de communication sont asynchrones, car elles ne suspendent pas l'activité de l'émetteur.

### I.3.2 Application du modèle OLAN

Si nous adoptons pour une telle application une décomposition qui à chaque utilisateur associe un composant, les relations qui doivent être établies entre les composants de l'éditeur remplissent un certain nombre de fonctions. Ce sont ces fonctions que doivent réaliser les connecteurs.

- **Gestion d'un espace partagé (le document)**

  L'accès au document doit être possible de manière concurrente et la synchronisation de ces accès doit préserver la cohérence du document. Dès lors que l'espace partagé constitue un lien entre les composants, il peut être fourni sous la forme d'un connecteur.

  Ce connecteur a comme sources les composants qui partagent l'espace et comme cible le composant qui le représente. Les communications que le connecteur véhicule entre ses sources et sa cible sont

contrôlées et doivent respecter le mode de partage qu'il réalise (exclusion mutuelle ou lecteurs–rédacteurs par exemple)

- **Information sur les évolutions de l'espace partagé**

  Toute modification apportée au document doit être notifiée à tous les membres d'une même session d'édition en évitant une consultation continuelle de l'espace partagé. Ceci permet à chacun de rafraîchir si besoin est sa vue du document.

  Le connecteur qui permet de réaliser cette diffusion d'information a pour origine le composant qui notifie et pour cibles les composants à notifier. Il réalise une diffusion asynchrone de la notification fournie par la source vers toutes les cibles. Ces dernières peuvent alors utiliser le message diffusé à leur gré, soit pour rafraîchir la vue du document, soit pour simplement avertir qu'un changement est intervenu. De plus, chaque composant qui peut notifier des modifications est la source d'un connecteur de ce type.

- **Négociations sur les droits d'accès au document**

  Chaque coauteur dispose de droits particuliers sur les différentes parties du document. Ces droits peuvent faire l'objet de négociations entre utilisateurs et donc de relations entre composants. Ainsi, il peut s'agir d'un simple passage de jeton entre deux composants signifiant une transmission de droit d'écriture, mais ce peut également être un protocole de vote reliant tous les composants participant à la session qui délivre le droit au vainqueur du vote.

  Un connecteur réalisant de tels protocoles est plus complexe que ceux que nous avons décrit précédemment. Un connecteur de vote possède par exemple des sources et des cibles multiples, puisque les composants qui participent au vote doivent à la fois émettre et recevoir des informations. Il encapsule le protocole du vote : il contrôle les communications qui circulent et détermine le vainqueur en fonction des flots qu'il véhicule, en accord avec le protocole choisi.

De plus, il existe sur les composants des contraintes pour l'établissement des connecteurs entre eux. En effet, les composants ne doivent participer à certaines interactions que s'ils sont membres d'une même session d'édition : une notification de modification n'est pas transmise à un composant qui ne travaille pas sur le document modifié. De même, certains rôles particuliers dans la gestion du document (administrateur par exemple) doivent être pris en compte. Enfin, à chaque utilisateur peuvent être associés des droits sur tout ou partie du document. L'établissement des relations entre les composants dépend de ces droits.

Les applications de télé–conférence [Crowley 90][Ohmori 92] constituent un autre exemple d'application coopérative et présentent du point de vue de la coordination de grandes similitudes avec l'édition coopérative. Si l'on excepte les problèmes liés au multimédia, la gestion d'un espace partagé (présenté généralement sous le terme générique de « tableau blanc ») et celle du droit de parole dans la conférence posent des problèmes analogues au partage d'un document édité concurremment et à la négociation des droits sur un document.

## I.4 Application « Flot de travail »

Les applications de type « flot de travail » (en anglais « workflow ») sont utilisées le plus souvent pour modéliser les procédures administratives au sein d'une organisation.

### I.4.1 Description

Dans une application de « flot de travail », une activité élémentaire définit un enchaînement de tâches à effectuer par des utilisateurs – ou des rôles(1) – conformément à un schéma prédéfini qui traduit les éléments de la procédure administrative. Une tâche est un élément asynchrone d'une activité globale. Ces applications coordonnent les multiples actions à effectuer en fonction du but à atteindre et de l'organisation des ressources.

La figure Fig. 1.3 illustre un exemple très simple, dans lequel il s'agit pour une entreprise d'honorer la commande d'un client. À l'arrivée de la commande, celle–ci est traitée par le « Service clients », qui transmet aux « Services financiers » une demande de facturation et au « Magasin » une demande de fournitures. Chacun de ces services exécute alors les étapes nécessaires à l'accomplissement de sa tâche. Le « Magasin » émet par exemple des requêtes vers une base de données pour la gestion des stocks, tandis que les « Services financiers » demandent la validation de l'opération par un responsable habilité. Pendant ce temps, le « Service clients » peut procéder à d'autres traitements, mais il est toujours en attente du retour de ses demandes antérieures. Lorsque les services demandés reviennent sous la forme d'une facture et de fournitures à transmettre, le « Service clients » peut honorer la commande et demander au « Service expédition » d'envoyer le résultat de sa commande au client.

---

( 1 ) Un rôle est défini par une attribution de fonctions affectée à un ou plusieurs utilisateurs dans l'organisation et il peut être rempli par n'importe quel utilisateur auquel il a été attribué. Un utilisateur peut de plus avoir plusieurs rôles.

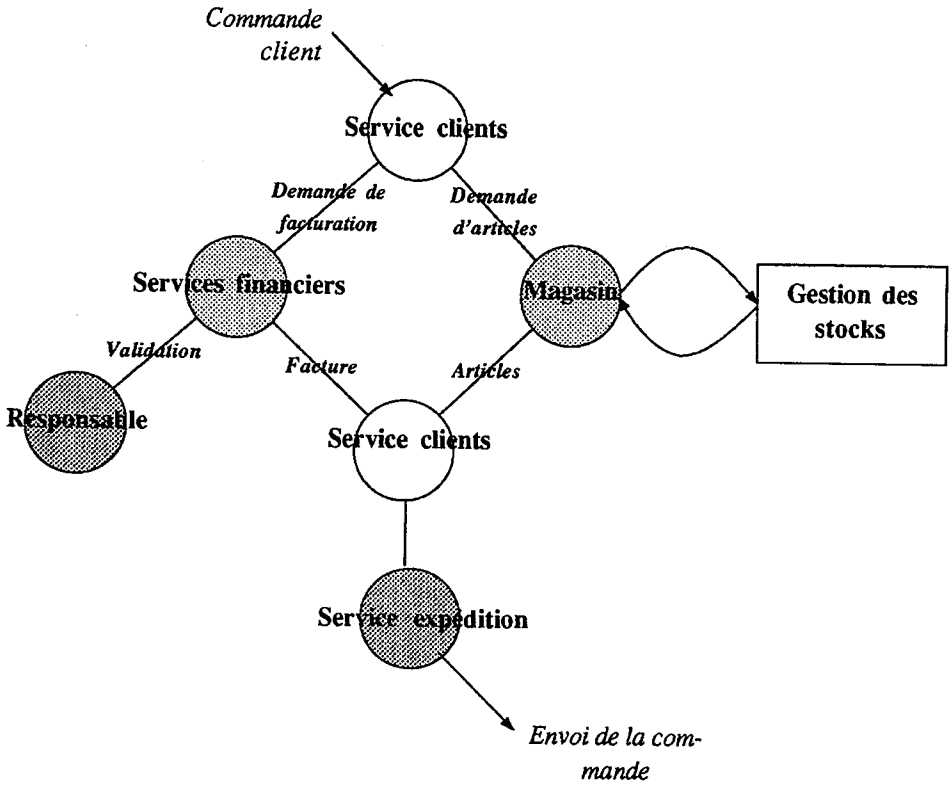


*Fig. 1.3 : Exemple d'organisation dans une application « workflow »*

Chaque tâche est ici définie et fait partie d'une activité plus large où elle possède une place déterminée. Par exemple, elle ne peut être démarrée que si telle autre tâche est terminée correctement.

Il existe des langages de description pour les schémas d'organisation de telles applications. Ces langages sont fondés sur des formalismes tels que les ICN [Ellis 79] [Ellis 93] ou celui décrit dans [Casati 95]. Nous ne nous intéressons pas pour notre part à ces aspects, mais à ceux qui concernent la configuration de l'application (décomposition en modules et mécanismes de transfert de contrôle entre modules).

### I.4.2 Application du modèle OLAN

Une manière directe de décrire à un niveau macroscopique une application de ce type en termes de composants et de connecteurs est de considérer chaque tâche à exécuter dans l'organisation comme un programme et d'associer un composant à chacun de ces programmes. Ainsi, l'exemple précédent sera représenté à l'aide de six composants : un pour le « Service clients », un pour les « Services financiers », un pour le « Magasin », un pour le « Service expédition », un pour la base de

données de gestion des stocks et un pour le responsable chargé du contrôle des opérations financières. Une fois précisé le comportement de chaque composant, la mise en place des connecteurs adéquats décrit la place de chaque rôle dans l'organisation et ses relations avec les autres.

Les relations qui s'établissent entre les différents composants doivent être représentées par des connecteurs qui encapsulent les protocoles de communication et de synchronisation requis. Ces relations doivent permettre :

- **d'enchaîner une activité d'un composant vers un autre**

  Une fois qu'un rôle a terminé son travail, il transmet le contrôle de la procédure à l'étape suivante. Ainsi, le « Service clients » notifie au « Service expédition » qu'il peut envoyer la commande au client en lui fournissant les données nécessaires. Cette notification ne bloque pas le « Service clients », qui peut reprendre son travail sans attendre de réponse de son interlocuteur.

  Le connecteur qui réalise cette communication relie le composant « Service clients » au composant « Service expédition ». Lorsque la source utilise ce canal de communication, le connecteur prend en charge les données à transmettre et les achemine de manière asynchrone vers sa cible.

  La question se pose alors de savoir comment l'activité du « Service expédition » est déclenchée. Ce déclenchement peut en fait correspondre à plusieurs schémas d'exécution : mise en place d'une synchronisation, initialisation d'une nouvelle activité.

- **de diviser la tâche en n sous–tâches affectées à n composants différents**

  L'étape suivante est menée en parallèle par n composants différents. Le traitement d'une commande par le « Service clients » consiste à répartir les travaux de facturation et de recherche des articles à deux services (deux rôles) différents.

  Un connecteur qui réalise ce protocole a une source et n cibles. Prenant en entrée un message de la source, il diffuse à chacune des cibles les messages adéquats, qui peuvent faire partie de la spécification du connecteur.

- **de synchroniser la terminaison de plusieurs sous–tâches avant qu'un autre composant reprenne le contrôle de la procédure**

  Le début de la tâche suivante est subordonné à la terminaison de n autres tâches. Le « Service clients » doit attendre que la facture et les

articles soient revenus des services concernés avant de demander l'expédition.

Un connecteur qui réalise ce protocole reçoit en entrée les notifications des n composants à synchroniser et lorsque tous sont au rendez–vous, il notifie la cible.

- **de « sous–traiter » la réalisation d'un service à un autre composant en attendant le résultat**

  Le « Magasin » utilise pour la gestion des stocks une base de données, vers laquelle il émet des demandes de services. Ce mode d'interaction est un mode client–serveur.

  Un tel connecteur relie le service requis d'un composant au service fourni qui le réalise dans un autre composant et réalise l'appel de procédure synchrone entre ces deux composants.

- **d'informer un composant de l'accomplissement d'une tâche**

  Certains composants représentant un rôle particulier (chef de service, directeur, etc.) doivent être tenus informés de l'état d'avancement du travail.

  Le connecteur correspondant est analogue à celui utilisé pour l'enchaînement d'une tâche ; il met en œuvre une communication asynchrone.

De plus, la notion de rôle utilisée suppose que des composants d'identités différentes mais de rôle identique sont interchangeables. Ainsi, il peut exister plusieurs instances interchangeables du composant « Services financiers ». Si un seul est actif et traite les activités qui concernent son rôle à un instant donné, il peut être dynamiquement remplacé par une autre des instances sans que le fonctionnement global du système en pâtisse. Ainsi, les connecteurs doivent prendre en compte non pas seulement l'identité exacte des composants qu'ils lient, mais des propriétés que ceux–ci possèdent (le rôle ou un service fourni).

## I.5 Environnement intégré de développement de logiciel

Les environnements intégrés de développement de logiciel offrent un support pour toutes les étapes du cycle de vie et de production d'un logiciel.

### I.5.1 Description

Nous étudions ici un modèle d'environnement de développement inspiré du travail décrit dans [Boyer 94]. L'environnement est composé d'un ensemble d'outils (éditeur, débogueur, fouineur, compilateur,...) qui coopèrent afin de fournir le meilleur support possible au développement, en liaison avec une base de composants logiciels.

On peut par exemple demander, à partir du fouineur, d'éditer un composant logiciel. Le fouineur va alors s'adresser à un éditeur et lui demander d'éditer le composant requis. L'éditeur pour sa part adressera à la base e composant une requête pour obtenir ce composant.

Un éditeur peut également signaler aux autres outils de l'environnement la modification d'un composant. Ces outils peuvent alors réagir en fonction de cette modification. Le fouineur peut par exemple mettre à jour les dépendances de ce composant.

Deux types d'interactions sont mis en évidence dans ce contexte. Les requêtes correspondent à des demandes de services et sont synchrones, alors que les notifications sont informatives et asynchrones.

### I.5.2 Application du modèle OLAN

La réutilisation d'outils existants est un des objectifs de la définition du modèle OLAN. Une application telle qu'un environnement de développement de logiciel se prête donc bien à son utilisation puisqu'elle met en jeu des outils très divers (éditeur, débogueur, compilateur, base de sources, etc.). Dans ce contexte, chaque outil peut être encapsulé dans un composant.

Dès lors, les relations entre composants doivent prendre en compte les différents types de dépendances qui peuvent exister entre les outils. Par exemple, la terminaison du débogueur entraîne la terminaison de l'éditeur qui en dépend, ou l'arrêt sur une erreur du compilateur positionne un éditeur à la ligne de l'erreur.

En première approximation, les types de connecteurs à mettre en œuvre sont donc un connecteur « requête » de demande de service (client–serveur) et un connecteur « notification ». Ils sont analogues aux connecteurs que nous avons mis en évidence pour la communication synchrone et la communication asynchrone dans les exemples précédents.

## I.6 Caractérisation des besoins

Comme le montrent les exemples précédents, les connecteurs assurent une grande variété d'interactions entre les composants. Il ressort tout d'abord de leur étude qu'ils peuvent réaliser plusieurs niveaux de coordination. Des protocoles de négociation élaborés ou la gestion d'un espace partagé constitué de plusieurs documents sont des schémas de coordination de haut niveau, qui peuvent être réalisés à partir de mécanismes de base.

La définition de ces mécanismes assure la liaison entre les projets Guide et Sirac : elle constitue donc notre objectif prioritaire.

D'une manière générale, ces protocoles de coordination requièrent que les entités coordonnées puissent communiquer sur différents modes (par exemple synchrone pour les demandes de service, asynchrone pour les notifications), qu'elles puissent partager des données et se synchroniser, et enfin qu'elles puissent se désigner les unes les autres. Ce sont donc des besoins de base que nous devons satisfaire en fournissant les mécanismes adéquats. Ceux–ci serviront de fondations à la construction des connecteurs, du plus simple (par exemple demande de service d'un composant à un autre) au plus complexe (par exemple négociation entre n composants pour l'attribution d'un jeton).

Il est important de noter que la proposition OLAN prolonge les travaux menés dans Guide. Dans ce contexte particulier, les aspects relatifs à des mécanismes tels que le partage, la synchronisation ou la communication synchrone, qui peuvent être pris en compte par les connecteurs, ont fait l'objet d'études nombreuses et notre équipe y a acquis des compétences.

### I.6.1 Mécanismes de base

Ces mécanismes de base remplissent les fonctions primaires pour la communication et la synchronisation entre composants, ainsi que pour leur désignation. Ces fonctions peuvent être regroupées en trois familles.

- **Partage et synchronisation**

  Les composants doivent pouvoir partager des données de manière cohérente et communiquer par le biais de ces données. Pour un éditeur coopératif ou une application de télé–conférence, le support du partage et de la synchronisation des accès concurrents à l'espace de travail pour chaque utilisateur doit être fourni. Dans un premier

temps, nous ne représenterons pas ce mode particulier d'interaction par des connecteurs. Notre approche consiste à concevoir les données partagées comme un composant vers lequel les composants qui partagent émettent des requêtes.

- **Communication synchrone et asynchrone**

  Les communications qui apparaissent lorsqu'on analyse les exemples précédents sont de plusieurs types. L'un correspond à une communication synchrone, où tous les membres de l'interaction décident du moment où ils inter–agissent. Le mode client–serveur appartient à ce type de communication. L'autre type, qualifié d'asynchrone, correspond à un envoi de messages et possède un caractère plus impromptu. Les initiateurs et les exécutants ne se synchronisent pas pour la prendre en compte. Une forme anonyme de ce mode de communication, où les récepteurs ne sont pas connus de l'émetteur, est également nécessaire. La communication est alors complètement déconnectée. La modélisation des différentes formes d'interactions nécessite que ces modes de communication soient proposés au programmeur ou au concepteur de connecteurs.

- **Désignation**

  Les intervenants dans un protocole de coordination doivent pouvoir se désigner. L'attribution à chacun de ces intervenants d'un identificateur ou la possibilité de le désigner par un nom symbolique doit donc être permise. Cela ne nous semble cependant pas suffisant. Les exemples ont en effet montré que, bien souvent, un connecteur reliait des composants dont les identités respectives n'étaient pas nécessairement connues, et qu'en revanche ils devaient satisfaire certaines propriétés. Ainsi, la demande du droit de parole dans une application de télé–conférence doit toujours être adressée au gestionnaire de ce droit de parole, qui est un rôle attribué à un participant de la conférence dont l'identité peut varier. De même, la notion de rôle dans une application « workflow » se rattache plus à une propriété de celui qui le remplit qu'à son identité, qui n'a pas d'importance en l'occurrence. La désignation sur leurs propriétés des composants est donc nécessaire à la mise en place des connecteurs.

La réalisation de connecteurs pour OLAN nécessite que ces trois familles de mécanismes soient offertes.

### I.6.2 Communication asynchrone et anonyme

Comme nous l'avons indiqué à plusieurs reprises, notre intérêt se porte principalement sur un mécanisme de communication asynchrone et anonyme.

#### I.6.2.1 Motivations

La communication asynchrone dans les modèles à objets a fait l'objet de peu de travaux, et il nous a semblé pertinent d'apporter une contribution à ce domaine, en particulier sur l'intégration de ce mode de communication dans un modèle à objets. De plus, le caractère anonyme des communications nous paraît particulièrement important dans un environnement comme celui des applications coopératives. L'ensemble des intervenants potentiels dans les communications y évolue en effet dynamiquement et une communication anonyme permet de prendre en compte de manière transparente ces évolutions.

D'autre part, les aspects liés à la désignation nous semblent être un auxiliaire de la communication, qui permet d'en enrichir les possibilités en offrant des modes plus évolués pour nommer les intervenants de la communication. Ces aspects ont fait l'objet d'une autre étude au sein du projet [Marangozov 95], au cours de laquelle un mécanisme de désignation associative a été conçu. Ce mécanisme permet de désigner les objets non plus seulement par un identificateur unique, mais sur des propriétés qu'ils peuvent posséder.

La combinaison de la désignation associative et de la communication asynchrone permet de réaliser une communication asynchrone associative. La **communication asynchrone et anonyme** suppose qu'aucune description de destinataire, de quelque sorte qu'elle soit, n'est associée à une émission. En revanche, la **communication asynchrone et associative** permet de décrire un ensemble de destinataires caractérisés par des propriétés.

#### I.6.2.2 Caractéristiques

Nous résumons maintenant les caractéristiques que doit posséder le mécanisme de communication asynchrone dans le cadre que nous nous sommes donnés – la construction de connecteurs pour la coordination dans les applications coopératives.

| | |
|---|---|
| **Asynchronisme** | L'émetteur ne doit pas être bloqué et aucune opération explicite des récepteurs n'est nécessaire pour établir la communication. |
| **Anonymat** | L'émetteur n'a pas à connaître l'identité des récepteurs, et vice–versa. |
| **Diffusion** | La communication s'établit sur le mode de la diffusion : ce n'est pas une communication point à point. |

| | |
|---|---|
| **Répartition** | La communication prend place dans un environnement réparti. |
| **Réaction** | Le mécanisme doit permettre d'une part de décrire les réactions à l'occurrence d'un message et d'autre part de les faire évoluer dynamiquement. |
| **Intégration** | La communication asynchrone doit être un mécanisme complémentaire à l'appel de méthode synchrone au sein d'un modèle à objets. |

L'ensemble de ces caractéristiques nous permet par anticipation d'indiquer que les principes du mécanisme souhaité se rapprochent d'un mode de communication reposant sur des **événements** : émis de manière non bloquante, ils engendrent dans le système l'exécution de réactions. Nous confirmerons et complèterons cette correspondance au chapitre suivant.

# Chapitre II

# État de l'art

Nous avons montré au chapitre précédent l'intérêt que représentait un mode de communication asynchrone et anonyme dans la conception et la programmation d'applications réparties et coopératives. Nous allons à présent étudier des travaux connexes à cette problématique dans différents domaines de l'informatique. Il existe en effet dans la littérature de nombreuses présentations sur de tels modes de communication, qui concernent des domaines très variés autour desquels nous avons articulé notre étude. Ces travaux abordent des niveaux fonctionnels différents. Les uns concernent les aspects relatifs aux modèles et aux langages, tandis que les autres présentent des supports d'exécution ou des primitives accessibles par des interfaces de type API (Application Programming Interface). Les seconds peuvent d'ailleurs servir de support de réalisation et d'exécution des premiers.

En premier lieu, nous décrivons sur quels critères nous avons évalué ces différents travaux. Ces critères font écho aux besoins que nous avons énoncés au chapitre précédent.

Une fois ces critères précisés, le premier domaine que nous considérons est celui des langages et modèles de coordination, car il correspond à notre préoccupation initiale de fournir des services de coordination pour un environnement de programmation d'applications réparties.

Nous nous intéressons dans un deuxième temps aux infrastructures d'exécution qui peuvent constituer, entre autres, le support des modèles et langages précédents : les bus logiciels et les courtiers d'objets. Les bus logiciels comme outils d'intégration d'applications hétérogènes et les courtiers d'objets comme médiateurs de communication entre des objets répartis proposent des fonctions de communication qui correspondent à nos préoccupations. C'est également dans cette rubrique que nous classons les appels asynchrones de procédure à distance (RPC asynchrone), dont nous montrerons que le caractère asynchrone ne suffit pas pour le type de communication que nous souhaitons mettre en place.

À la suite des courtiers d'objets, nous recentrons dans un troisième temps notre étude sur les modèles à objets, pour lesquels il existe plusieurs travaux pertinents

autour de la communication asynchrone, et qui mettent notamment en avant la notion d'**événement**.

Ce troisième pan de notre étude met en lumière la profonde similitude des propositions avec celles que l'on trouve dans les bases de données actives. Certains systèmes de gestion de bases de données offrent en effet à travers la notion de règle active un mécanisme semblable à celui que nous envisageons et nous présentons quelques exemples significatifs pour conclure cette étude bibliographique.

Enfin, nous dressons une synthèse des apports de ces différents domaines sur la communication asynchrone et anonyme, ce qui nous permet de motiver l'approche retenue.

## II.1 Critères d'étude

Nous avons considéré dans l'étude qui suit plusieurs critères qui font référence aux conditions que doit remplir le mécanisme de communication asynchrone et anonyme que nous souhaitons mettre en place. Ces critères sont les suivants :

- **Mode de communication**

  Sous ce critère, nous examinons le caractère asynchrone de la communication. Cet asynchronisme doit se retrouver aux deux extrémités de la communication : l'émetteur n'est pas bloqué en attente de réponse et le ou les récepteurs n'ont pas d'opération explicite de réception (et donc de synchronisation) à réaliser.

  Nous examinons également les possibilités de diffusion offertes par les mécanismes et modèles étudiés. La diffusion sélective, qui permet de restreindre l'ensemble des récepteurs, constitue un pan de ces possibilités.

- **Désignation**

  Nous souhaitons mettre en place une communication anonyme, où la désignation des récepteurs n'est pas nécessaire. Le mode de désignation des entités qui interviennent dans les mécanismes étudiés détermine le degré d'anonymat et de découplage de la communication.

- **Description du comportement des récepteurs – Évolution**

  Nous souhaitons pouvoir exprimer le comportement des récepteurs lors de l'établissement d'une communication et faire évoluer dynamiquement la nature des réactions en fonction de l'évolution du système et du récepteur. Les fonctions qu'offrent les travaux étudiés

pour exprimer ces comportements et leur évolution sont donc prises en compte dans notre évaluation.

- **Modèle d'exécution**

  Nous ne souhaitons pas dissocier la communication du modèle d'exécution qui la régit. Ce modèle d'exécution concerne à la fois les opérations qui permettent d'initier la communication et l'éventuelle exécution des réactions induites chez les récepteurs.

  La possibilité d'utiliser le mécanisme dans un environnement réparti est également cruciale.

- **Intégration dans un modèle à objets**

  Notre objectif est de réaliser un modèle à objets étendu par des possibilités de communication asynchrone. La possibilité d'intégrer les mécanismes étudiés dans un modèle à objets constitue par conséquent un point crucial de notre étude.

Les quatre premiers critères sont détaillés pour chacun des travaux présentés, tandis que l'examen du dernier prend place dans les sections de synthèse associées aux différentes classes de travaux.

## II.2 Modèles et langages de coordination

La coordination, comme nous l'avons définie au chapitre précédent, permet de définir et gérer les relations et interdépendances entre des activités – applications, processus, agents. La description et l'expression de cette coordination a nécessité la conception de modèles et de langages adaptés à ces tâches [Carriero 92]. Ce dernier article introduit une séparation entre langages de programmation, pour l'expression des calculs, et langages de coordination, pour l'expression des interactions entre activités.

### II.2.1 Linda et ses héritiers

Linda [Carriero 89a] [Carriero 89b] est un modèle de coordination, précurseur de nombreux travaux dans ce domaine.

#### II.2.1.1 Description

L'idée de base de Linda consiste à considérer un espace partagé et réparti, dénommé « espace de tuples » qui contient des données structurées sous forme d'enregistrements. Les activités qui se coordonnent déposent et consomment des messages dans cet espace plat.

Un **tuple** est constitué d'une série de champs typés, par exemple `("une chaîne", 12, 15.1, "une autre chaîne")`. Le nombre de champs d'un tuple est quelconque. La production et la consommation de tuples dans l'espace partagé est réalisée par les activités du système à l'aide de quatre primitives :

- `eval` permet de créer une activité qui exécute un programme et produit un résultat sous la forme d'un tuple.
- `out` permet de produire et de déposer un tuple dans l'espace partagé.
- `in` permet de consommer et d'effacer un tuple dans l'espace partagé.
- `rd` permet de lire un tuple dans l'espace partagé.

La consommation des tuples se fonde sur des manipulations associatives : les consommateurs décrivent en paramètre des fonctions `in` ou `rd` le motif que doit satisfaire un tuple pour être lu. Si un tuple correspond à ce filtre, il est « reçu » par l'exécutant de la fonction `in` ou `rd`. L'existence d'un tuple n'est pas subordonnée à celle de l'activité qui l'a créé : un tuple peut subsister après la terminaison de son créateur.

Dans ce contexte, la base de la communication est le partage d'un espace et l'échange d'information via cet espace, qui est géré selon un modèle producteurs–consommateurs.

Un aspect de Linda sur lequel insiste ses concepteurs est sa généralité. Indépendant de tout langage de programmation, il peut s'adapter à différents contextes et coordonner des activités qui possèdent des modes d'exécution divers, éventuellement écrites dans des langages différents. Ainsi, il existe des modules d'accès aux fonctions de Linda écrits pour de nombreux langages, dont C et Prolog.

#### II.2.1.2 Examen des critères d'étude

**Mode de communication**

Le caractère asynchrone de cette communication résulte du fait que l'initiateur de la communication (activité qui exécute un appel à `out`) n'a pas à se préoccuper du moment où la ou les cibles liront effectivement les messages déposés (par un appel à `in` ou à `rd`). En revanche, les récepteurs des messages doivent réaliser une opération explicite, et donc se synchroniser, pour recevoir un message.

La diffusion en Linda n'est pas réalisable directement, ni de manière sûre. Un émetteur peut déposer un tuple destiné à de multiples destinataires, qui chacun le liront à l'aide de la commande `rd`, mais l'exécution d'une commande `in` –que rien n'interdit– sur ce tuple termine la possibilité de diffusion. Une telle diffusion n'est de plus pas explicite : on ne sait pas qu'un message est produit pour être diffusé. La diffusion sélective est encore plus

délicate à réaliser de manière certaine, car rien n'interdit à une activité qui n'est pas *a priori* destinataire d'un tuple d'exécuter une commande `in` dessus, et donc de violer le critère de sélection de la diffusion.

**Désignation**

Aucune désignation des intervenants de la communication n'est nécessaire en Linda. Seule la spécification d'un motif de filtrage est nécessaire côté récepteur. La communication induite est donc bien anonyme.

**Description du comportement des récepteurs – Évolution**

Linda ne se préoccupe absolument pas des traitements qu'engendre chez les récepteurs la réception d'un tuple. Ce sont les programmes exécutés par les flots coordonnés qui décrivent ces traitements. Il n'y a donc dans Linda aucun moyen de décrire ces traitements ni leur évolution.

**Modèle d'exécution**

Linda ne fait aucune hypothèse sur le modèle d'exécution que respecte les activités coordonnées. L'espace de tuples est une entité passive qui ne possède aucune caractéristique dynamique établie.

Linda offre au niveau du modèle une gestion transparente de la répartition. L'espace des tuples est réparti et toute activité dans le système peut y accéder.

De nombreux travaux dans le domaine de la coordination sont inspirés par Linda et en étendent les possibilités. Ainsi, [Gelernter 89] propose la création d'espaces de tuples multiples, ce qui permet en particulier de structurer cet espace initialement plat. Cette extension offre de surcroît la possibilité de définir des domaines de communication différents selon les programmes en cours, et de forcer ainsi la sélection des récepteurs. Nous présentons en II.4.1 une adaptation de ce schéma de communication asynchrone et découplée dans un environnement à objets concurrents et répartis.

### II.2.2 ActorSpace

ActorSpace [Agha 93][Callsen 94] est un paradigme de programmation qui fournit un modèle de communication dont la base est un adressage associatif. Ce mode d'adressage permet de décrire des motifs que doivent satisfaire les destinataires. Reposant sur le modèle Actors [Agha 86], il en étend les possibilités de communication point à point par une diffusion sélective.

### II.2.2.1 Description

Le modèle Actors propose un modèle de programmation concurrente dont la base est l'envoi asynchrone de message entre acteurs. Chaque acteur possède une adresse unique qui l'identifie et peut communiquer avec d'autres acteurs en envoyant des messages à leur adresse. Chaque acteur décrit également quel est son comportement à la réception d'un message qui lui est adressé. La communication est asynchrone, car l'émetteur n'est pas bloqué lors d'une émission. De plus, le récepteur n'a pas, à l'inverse de Linda, à se synchroniser pour recevoir le message. En revanche, la communication n'est pas anonyme, car l'adresse du destinataire doit toujours être précisée. C'est pour combler ce manque que le modèle ActorSpace a été conçu.

L'idée est d'étendre le mode de désignation par adresse avec un mode de désignation à l'aide de motifs qui caractérisent des attributs des acteurs. Un « espace d'acteurs » est un conteneur d'acteurs dont les membres rendent visibles dans cet espace certains de leurs attributs.

L'espace d'acteurs se comporte alors comme un contexte pour la reconnaissance de motifs d'adressage. Ces motifs décrivent à la fois le contexte (l'espace d'acteurs) dans lequel ils seront analysés et les propriétés des acteurs auxquels ils seront appliqués. Ils sont comparés aux attributs visibles des acteurs de l'espace cible et les messages ne sont délivrés qu'aux membres de l'espace qui correspondent aux motifs.

La structure des espaces d'acteurs n'est pas plate. Un espace peut être inclus dans un autre, deux espaces peuvent se superposer. Ainsi, un même acteur peut appartenir à plusieurs espaces à la fois, en rendant visibles dans chacun des ces espaces des attributs éventuellement différents.

Ainsi, il est possible de diffuser de manière asynchrone et anonyme des messages qui seront reçus par un ensemble d'acteurs qui réaliseront chacun une action qu'ils auront décrite dans leur comportement.

### II.2.2.2 Examen des critères d'étude

**Mode de communication**

Actors garantit une communication asynchrone sans opération de lecture explicite de la part des destinataires.

ActorSpace apporte par rapport à Actors la possibilité d'une diffusion sélective.

**Désignation**

La communication n'est pas réellement anonyme, mais associative : elle ne repose pas sur une identification exacte des destinataires, mais sur des motifs qu'ils doivent respecter évalués dans le contexte d'un espace d'acteurs. Un tel espace se comporte alors comme une unité de limitation de la portée de la désignation.

**Description du comportement des récepteurs – Évolution**

Pour chaque acteur, un traitement est associé aux messages qu'il peut recevoir. Le modèle offre les fonctions qui permettent de faire évoluer l'ensemble de ces traitements en fonction de l'état de l'acteur.

**Modèle d'exécution**

Les messages émis vers un acteur sont mis dans une file d'attente, avant d'être traités. Le traitement associé par l'acteur au message reçu est alors réalisé. Aucun ordre n'est garanti sur l'arrivée des messages chez leurs destinataires.

Actors et ActorSpace sont des modèles de programmation répartie et qui prennent ces aspects en compte de manière transparente.

### II.2.3 Linear Objects

Linear Objects [Andreoli 93] est un modèle de coordination qui se fonde sur l'utilisation de règles semblables à celles de la programmation logique. L'objectif du modèle considéré est de permettre dans un environnement réparti des interactions entre des sous–systèmes indépendants et définis localement.

#### II.2.3.1 Description

Les sous–systèmes que considère le modèle sont dénommés des **agents**. Un agent est caractérisé par un ensemble de **ressources**, qui constituent son état, et un ensemble de règles qui décrivent son comportement. À la différence des acteurs de Actors, un agent ne possède aucune adresse où il pourrait recevoir des messages.

Les règles définissent des transitions entre deux états d'un agent : la partie gauche décrit les ressources que consomme le déclenchement de la règle, la partie droite énumère les ressources produites par ce déclenchement. La création de nouveaux agents par duplication peut être spécifiée dans cette partie droite, ainsi que la diffusion de ressources vers l'extérieur.

C'est par le biais de cette diffusion qu'est réalisée la communication entre agents. Ce mécanisme permet d'introduire dans l'état des agents et depuis l'extérieur, de

nouvelles ressources. Ces ressources permettent alors le déclenchement éventuel de nouvelles règles et peuvent provoquer des « réactions en chaîne ». La diffusion est réalisée de manière anonyme à tous les agents du système. Elle est asynchrone, d'une part parce que l'émetteur n'est pas bloqué et d'autre part parce que le système n'a pas besoin d'être gelé pour déposer la ressource diffusée dans chaque agent avant de poursuivre l'exécution. Les auteurs introduisent la notion de « vague » qui atteint tous les agents une fois et une seule et leur dépose la ressource.

#### II.2.3.2 Examen des critères d'étude

**Mode de communication**

La communication entre agents est effectivement asynchrone. L'émetteur n'est pas bloqué en attente d'une réponse, et il n'y a aucun point de synchronisation explicite pour la réception.

Le mode de communication est une diffusion non sélective. Il n'y a aucun moyen par exemple de limiter la portée d'une diffusion.

**Désignation**

La communication est complètement anonyme. Aucun destinataire n'est spécifié lors d'une émission.

**Description du comportement des récepteurs – Évolution**

Le comportement des récepteurs est décrit par un ensemble de règles, et la réception d'une ressource peut déclencher une règle. Cependant, rien ne permet de faire évoluer cet ensemble de règles au cours de l'exécution.

**Modèle d'exécution**

Les aspects du modèle d'exécution relatifs à cette diffusion se limitent au déclenchement des règles correspondantes dans les agents atteints par une « vague » de diffusion.

Le modèle Linear Objects est le fondement de CLF (Coordination Language Facility)[Andreoli 94][Andreoli 96], un langage de coordination dont le « workflow » (cf. I.4) constitue le champ d'application privilégié. Ce langage met en œuvre un comportement dit « **proactif** » : un coordinateur qui gère un ensemble d'agents et un ensemble de règles émet vers ces agents des requêtes afin que ceux–ci produisent les ressources nécessaires au déclenchement des règles.

### II.2.4 Synthèse

Ces travaux offrent trois approches différentes pour la coordination, mais tous permettent de gérer des dépendances et des relations entre des composants indépendants. Ils sont donc utilisés en conjonction avec d'autres langages de programmation dans lesquels les composants coordonnés sont écrits.

La communication qu'ils offrent est une communication asynchrone anonyme. Cependant, si leur indépendance par rapport à tout langage peut être un atout dans certaines conditions, c'est pour nous une lacune importante, compte tenu de nos objectifs. En effet, aucun des trois modèles présentés ne peut être simplement **intégré dans un modèle à objets** qui offre les possibilités d'un langage de programmation classique. Cette difficulté tient au fait que Linda, ActorSpace et Linear Objects ont été conçus pour être utilisés indépendamment et en complément d'autres outils de programmation.

## II.3 Infrastructures d'exécution

Parmi les infrastructures d'exécution existantes, certaines offrent des fonctions de communication semblables à celles que nous souhaitons. Ces infrastructures se situent à un niveau autre que les modèles et langages que nous avons présentés en II.2. En fait, elles peuvent servir de support de réalisation et d'exécution à ces modèles.

### II.3.1 Bus de messages

Les bus de messages sont des outils de communication qui permettent à des applications hétérogènes de communiquer entre elles. Ce sont des plates–formes d'intégration dont le principe a été introduit par le projet Field. De nombreux prototypes de recherche et produits commerciaux existent à l'heure actuelle. Citons Field [Reiss 89][Reiss 90], The Information Bus [Oki 93], Koala [Beust 93] [Beust 96], ToolTalk [SunSoft 91] ou SoftBench [Cagan 90]. Le mode de communication offert majoritairement est un mode de communication asynchrone qui correspond à la diffusion d'information. Certains bus offrent par ailleurs un mécanisme de demande de service (communication synchrone).

#### II.3.1.1 Description

Les bus de messages offrent un mécanisme de communication asynchrone à travers la diffusion de messages. Le principe repose sur le dépôt asynchrone et la circulation de messages sur un bus auquel sont connectés les clients. L'ensemble de ces clients peut varier dynamiquement.

La réception des messages est subordonnée à l'**abonnement** des clients. À l'instar des techniques utilisées dans Linda et ActorSpace, l'abonnement correspond à la donnée d'un motif sur lequel les messages seront filtrés et reçus par les clients. Ces motifs peuvent concerner l'ensemble du message comme dans Field, ou seulement un champ particulier comme dans The Information Bus, où chaque message possède un sujet. Ce sujet est une chaîne de caractères qui sert au filtrage.

Certains bus, tels que SoftBench, The Information Bus ou Koala, proposent un mécanisme d'association qui permet à un client d'indiquer une action à exécuter à la réception de certains messages.

#### II.3.1.2 Examen des critères d'étude

**Mode de communication**

Les bus de messages offrent un mécanisme de communication totalement asynchrone : l'émission est non bloquante et aucune synchronisation n'est requise de la part des clients.

Les bus de messages permettent une diffusion asynchrone à de nombreux destinataires. On ne peut cependant pas restreindre à l'émission l'ensemble des destinataires potentiels.

**Désignation**

La communication est totalement anonyme. Il n'y a aucun moyen de désigner les destinataires d'un message, ni de décrire des propriétés qu'ils pourraient posséder.

**Description du comportement des récepteurs – Évolution**

Les clients s'abonnent à des messages : ils indiquent par un mécanisme de désignation associative quels messages ou types de messages ils souhaitent recevoir. Ces abonnements peuvent évoluer dynamiquement.

Certains bus de messages permettent également de décrire de manière dynamique des associations message–traitement au niveau des clients qui recevront les messages. L'évolution de ces associations permet de prendre en compte l'évolution de l'ensemble du système et de ses composantes.

**Modèle d'exécution**

Les clients sont maîtres de leur réactions. D'autre part, certains bus de messages fonctionnent de manière transparente dans un environnement réparti.

### II.3.2 Courtiers d'objets.

Les courtiers d'objets [Osher 92], dont l'exemple le plus significatif est la spécification CORBA (Common Object Request Broker Architecture) de l'Object Management Group (OMG) [OMG 91][OMG 96], sont des services qui permettent de décrire des objets répartis offrant via une interface une palette de services, et de gérer de manière transparente l'invocation de ces services par des clients locaux ou distants. Si le mécanisme de base défini dans CORBA est l'appel synchrone de méthode sur un objet, un certain nombre de services complémentaires construits autour de cette plate–forme de base ont été décrits. Parmi eux, le service de notification d'événements [OMG 93] se rapproche de ce que nous souhaitons réaliser.

#### II.3.2.1 Description

Le service de notification d'événements de l'OMG offre un mode de communication asynchrone sur la plate–forme CORBA. C'est le besoin d'une communication plus découplée que l'invocation de service de CORBA qui motive la définition de ce service.

Le service repose sur la définition de **fournisseurs** et de **consommateurs** d'événements. Les fournisseurs produisent des événements et les consommateurs les traitent. Le service de l'OMG propose deux modes d'interaction (mode **push** ou **pull**) entre ces entités. Dans le premier mode, le fournisseur initie la communication en produisant un événement à l'intention d'un consommateur. Dans le second mode, le consommateur requiert d'un fournisseur la production d'un événement. Dans les deux cas, l'initiateur de la communication réalise un appel de méthode CORBA sur son correspondant. La communication est donc synchrone, puisqu'elle correspond à l'établissement d'un rendez–vous entre les deux parties.

C'est pourquoi il est également possible de connecter des fournisseurs et des consommateurs multiples par le biais de **canaux d'événements**. Ce sont des objets intermédiaires qui permettent la réalisation de la communication asynchrone et découplée désirée. Ils se comportent comme des courtiers d'événements. La figure illustre le principe. Le fournisseur émet de manière synchrone un événement vers le canal (mode **push**). Il est débloqué dès la fin de l'appel de méthode correspondant. Dès lors, le canal a la charge de transmettre l'événement aux consommateurs qui lui sont connectés (toujours avec des appels de méthode synchrone selon le mode **push**), réalisant ainsi le découplage entre fournisseur et consommateurs

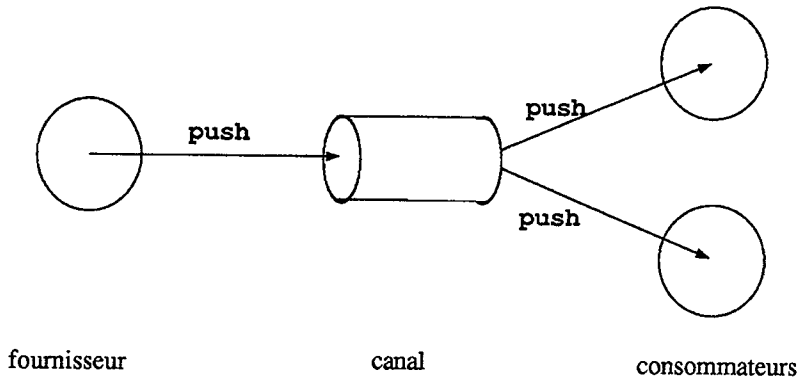


*Fig. 2.1 : Schéma de communication par canal d'événements*

Les canaux d'événements comme les autres objets sont créés et géré explicitement par le programmeur d'application.

### II.3.2.2 Examen des critères d'étude

**Mode de communication**

La communication en mode **`push`** (les fournisseurs initient la communication) à travers des canaux d'événements permet de réaliser une communication asynchrone. Les consommateurs potentiels n'ont pas à réaliser d'instruction explicite pour recevoir l'événement, puisque la réception correspond à un appel de la part du canal source.

La diffusion est possible vers plusieurs consommateurs si ces consommateurs sont connectés au même canal d'événements. Ceci suppose que les références des différents canaux d'événements sur lesquels des événements seront diffusés soient connues des consommateurs.

**Désignation**

Dès lors qu'on utilise les canaux d'événements, la communication entre fournisseurs et consommateurs est anonyme : les fournisseurs ignorent quels consommateurs sont connectés au canal d'événements qu'ils utilisent. En revanche, ils doivent préciser explicitement la référence du canal utilisé.

**Description du comportement des récepteurs – Évolution**

La réaction d'un consommateur à la réception d'un événement ne fait pas partie de la définition du modèle.

**Modèle d'exécution**

Le modèle d'exécution associé à ce service est celui de CORBA. Toutes les opérations sont réalisées à l'aide de requêtes CORBA.

### II.3.3 Appel asynchrone de procédure à distance

Certains systèmes proposent un mécanisme d'appel asynchrone de procédure à distance, ou RPC asynchrone. Un tel mécanisme, à l'inverse des mécanismes de RPC classiques, évite que le client qui demande la réalisation d'un service soit bloqué en attente d'un résultat. Ceci permet en particulier d'augmenter le parallélisme dans les clients sans avoir recours à la création de processus légers qui pénalisent les performances et rendent la mise au point difficile. On peut trouver dans [Ananda 92] une étude comparée de différents RPC asynchrones.

#### II.3.3.1 Description

Hormis le fait que l'appel de procédure n'est pas bloquant, un RPC asynchrone présente les même caractéristiques qu'un RPC synchrone : un client demande à un serveur distant l'exécution d'une procédure. Il lui passe pour ce faire le nom et les paramètres de la procédure.

Les auteurs de l'étude sus–citée distinguent deux types de mécanismes selon que l'appel retourne une valeur ou non. Dans le premier cas, la récupération du résultat est différée à une date ultérieure et le client exécute alors une opération explicite.

#### II.3.3.2 Examen des critères d'étude

**Mode de communication**

La communication entre client et serveur est asynchrone. D'une part, le client n'est pas bloqué lors de l'appel, d'autre part le serveur n'a pas à se synchroniser pour recevoir la requête. Cependant, dans le cas des RPC avec valeur de retour, le client peut être bloqué lorsqu'il récupère le résultat de sa requête.

Enfin, le RPC asynchrone est un mécanisme de communication point à point. Il ne permet pas la diffusion.

**Désignation**

Ce mécanisme n'est pas anonyme : il nécessite la connaissance complète du serveur qui sera invoqué. La communication ne peut donc pas être découplée.

**Description du comportement des récepteurs – Évolution**

Le comportement des récepteurs (serveurs) est déterminé par le contenu du message, qui contient le service à exécuter. Il n'y a donc aucune possibilité d'évolution dans le comportement du serveur.

**Modèle d'exécution**

Le modèle d'exécution utilisé est un modèle client–serveur en milieu réparti : le client émet des requêtes que le serveur distant exécute.

### II.3.4 Synthèse

Ces structures d'exécution offrent des services aux concepteurs d'applications et constituent des infrastructures d'exécution susceptibles de servir de base à des modèles tels que ceux que nous avons présentés en II.2.

Cependant, ils n'offrent pas en l'état les fonctions que nous désirons. Le RPC asynchrone n'est absolument pas anonyme et n'offre aucune possibilité de diffusion et nous l'écartons d'emblée. L'intégration des deux autres approches dans un modèle à objets n'est pas directe. Le service de notification est construit à partir de CORBA et utilise les fonctions décrites dans la spécification : il n'y est intégré ni du point de vue du modèle de données, ni du point de vue du modèle d'exécution. Les bus de messages sont des infrastructures d'exécution qui mettent en jeu des entités de grain trop important.

Cependant, nous retiendrons de cette étude et plus particulièrement de celle des bus de messages la notion d'abonnement : elle permet de décrire et de faire évoluer les interactions de type asynchrone dans lesquelles une entité peut être impliquée.

## II.4 Modèles à objets

Il existe dans le domaine des modèles et langages à objets des travaux sur la communication asynchrone. Nous examinons les caractéristiques de certaines de ces propositions en regard des objectifs que nous nous sommes fixés : fournir un mécanisme de communication asynchrone, anonyme, intégré à un modèle à objets qui pourrait servir de support à la définition de services de coordination.

### II.4.1 Communication par espace de tuples

[Matsuoka 88] propose l'utilisation d'une communication « à la Linda » dans le contexte d'un environnement à objets concurrents et répartis. Cette approche présente par rapport aux modes de communication couramment utilisés dans de tels environnements l'avantage d'être découplé dans le temps (asynchrone) et l'espace (l'émetteur ne connaît pas nécessairement le récepteur).

#### II.4.1.1 Description

Les adaptations proposées par l'auteur visent à homogénéiser les concepts des modèles à objets avec ceux de Linda. Ainsi, les tuples et les espaces de tuples

deviennent des objets. Les émetteurs créent des objets tuples qu'ils déposent dans un espace de tuples. Le protocole de lecture coté récepteur est également adapté, puisque les motifs d'adressage utilisés par le récepteur sont décrits dans un objet qu'il crée et dépose dans l'espace de tuples souhaité. Cet objet est ensuite confronté aux objets déposés par des émetteurs. Si une correspondance est trouvée, l'objet est modifié en fonction des champs effectifs du tuple correspondant et retourné au récepteur. Les espaces de tuples sont également des objets, ce qui permet d'une part de les manipuler (création, destruction, désignation) et d'autre part de construire un ensemble structuré de tuples. Cette structuration, à l'instar de ce qui est proposé dans [Gelernter 89] et que nous avons présenté en II.2.1, autorise un meilleur contrôle de la sécurité ainsi que de la portée de la communication.

#### II.4.1.2 Examen des critères d'étude

L'utilisation de cette approche pour réaliser les objectifs que nous nous sommes fixés présente les mêmes caractéristiques que Linda ou que ses dérivés (cf II.2.1). En particulier, la synchronisation côté récepteur est nécessaire. Comme nous l'avons dit, les apports par rapport au modèle original de Linda résident dans la structuration et le meilleur contrôle des communications qu'elle permet de mettre en place.

### II.4.2 Invocation implicite

Dans [Notkin 93], les auteurs proposent une extension des modes d'invocation de méthode pour trois langages à objets.

#### II.4.2.1 Description

Le principe de cette extension, dénommée **invocation implicite**, a été introduit dans [Garlan 91] et il est le suivant :

> Un composant(1) diffuse un ou plusieurs événements. D'autres composants enregistrent un intérêt pour un événement en lui associant une procédure(1). À l'occurrence d'un événement, le système invoque les procédures qui lui ont été associées.

Selon les auteurs, ce mécanisme permet une modification plus aisée des applications : des composants peuvent être ajoutés, modifiés ou intégrés avec peu ou pas de modification des autres composants [Sullivan 92]. En effet, aucun lien explicite du type *« le composant $C_1$ invoque le service S du composant $C_2$ »* n'est établi entre les composants. Par conséquent, la reconfiguration de l'application ne remet pas en cause les relations très lâches gérées par le système.

---

( 1 ) Les termes *composant* et *procédure* employés ici doivent être entendus selon une terminologie « objet » au sens d'*objet* et de *méthode d'un objet*.

L'approche des auteurs est, à la différence des bus de messages qui proposent le même type de mécanisme comme un service fourni aux applications, ou des propositions de l'OMG qui fournissent un service aux programmeurs, d'intégrer le mécanisme au niveau du langage. Cette intégration repose sur l'écriture de modules ou de classes qui sont utilisées dans des macro–fonctions fournies au programmeur. Ils illustrent leur proposition sur trois langages à objets : Ada, C++ et Common Lisp/CLOS.

### II.4.2.2 Examen des critères d'étude

**Mode de communication**

L'invocation implicite est asynchrone tant du point de vue de l'appelant qui n'est pas bloqué lorsqu'il initie la communication, que du récepteur qui n'a pas à se synchroniser pour recevoir le message.

L'invocation implicite réalise une diffusion des événements à l'ensemble des objets concernés. En revanche, il n'y aucun moyen de restreindre la portée de la diffusion.

**Désignation**

L'émetteur d'un événement n'a aucune connaissance des méthodes qui seront exécutées ni des objets sur lesquels elles porteront. C'est donc bien un mécanisme de communication anonyme.

**Description du comportement des récepteurs – Évolution**

La description du mécanisme permet d'une part d'associer une méthode à un événement mais également de faire évoluer pour un objet donné l'ensemble de ces associations.

**Modèle d'exécution**

Le modèle d'exécution n'est pas clairement établi, mais le mécanisme semble limité au niveau du processus : l'exécution effective des méthodes associées aux événements est dévolue à des flots de contrôles créés au sein du processus où l'invocation a eu lieu. Il ne permet donc pas la communication entre processus.

### II.4.3 Bacon

Dans [Bacon 95], les auteurs considèrent la programmation à l'aide d'objets répartis. Le point de départ de leur réflexion fait écho à ce que nous mentionnons au chapitre précédent : l'utilisation de la notification asynchrone d'événements qui déclenche des traitements est requise par de nombreuses applications, sans pour

autant que ce mode d'interaction prime sur l'invocation synchrone qui prévaut dans les modèles à objets.

#### II.4.3.1 Description

Ils proposent donc, dans le cadre d'un modèle de type client–serveur, de faire coexister ces deux modes. Ils étendent le modèle avec la description d'événements chez les serveurs, l'enregistrement pour certains événements des clients auprès des serveurs et la notification aux clients par les serveurs des événements pour lesquels ils se sont enregistrés.

Pour ce faire, ils étendent l'interface des services que publient les serveurs par des descriptions d'événements. Un événement est un objet instance d'une classe, qui décrit les paramètres de l'événement. La description de la classe d'un événement fait partie de l'interface du serveur qui peut l'émettre : les événements sont donc des services fournis par des serveurs. Un serveur peut créer un événement conformément à une classe d'événements de son interface et le notifier de manière asynchrone à ses clients enregistrés.

Les clients de leur côté enregistrent auprès des serveurs leur intérêt pour des événements que peuvent émettre lesdits serveurs. Cet enregistrement décrit un motif que doivent vérifier les événements et un traitant à exécuter à l'occurrence des événements adéquats. L'exécution de ces traitants est réalisée dans le contexte du client. Les clients peuvent d'autre part résilier un enregistrement.

#### II.4.3.2 Examen des critères d'étude

**Mode de communication**

La communication que propose ce mécanisme est asynchrone : l'émetteur n'est pas bloqué et aucune synchronisation du récepteur n'est requise.

Le mécanisme ne met pas en œuvre une diffusion générale, mais une diffusion du serveur vers ses clients déclarés.

**Désignation**

Un serveur de notification d'événement n'a pas à désigner les cibles des événements qu'il émet. En revanche, les clients doivent se connecter explicitement aux serveurs qui les intéressent.

**Description du comportement des récepteurs – Évolution**

Comme pour l'invocation implicite, il est possible à un client d'associer un traitement à un événement et de faire évoluer cette association.

**Modèle d'exécution**

Le modèle d'exécution que respecte l'exécution des traitants d'événements n'est pas décrit.

### II.4.4 Menon

Le travail présenté dans [Menon 93a] et [Menon 93b] est une étude complète sur l'utilisation d'événements émis de manière asynchrone dans un environnement à objets répartis. La motivation de cette étude réside dans le fait que l'utilisation des signaux asynchrones dans les environnements centralisés est une donnée acquise par les programmeurs, mais qu'en revanche, aucun support correspondant n'existait dans un environnement réparti. Le contexte de cette étude est un système à objets passifs répartis, auxquels accèdent des flots d'exécution concurrents.

#### II.4.4.1 Description

Dans ce contexte, les auteurs préconisent et présentent un modèle d'événements, qui sont notifiés de manière asynchrone et déclenchent des **traitants**. Ces traitants, qui selon notre terminologie sont des réactions, peuvent être associés soit à des objets soit à des flots d'exécution. Ainsi, les événements sont délivrés à des entités soit passives soit actives.

L'émission des événements comprend la spécification d'une destination qui est soit un objet, soit un flot d'exécution, soit un groupe de flots. Ceci permet de déterminer quels traitants seront exécutés à la réception de l'événement.

Un objet peut associer des traitants à différents événements. Ces traitants sont des méthodes de l'objet. Ils sont exécutés par un flot d'exécution du noyau.

Les traitants associés au niveau des flots de contrôle permettent de prendre en compte le caractère concurrent des exécutions et de différencier les réactions à un même événement pour des flots qui partagent un même objet. À la création d'un flot, des traitants sont associés à certains événements. Lorsqu'un événement est reçu, le traitant est exécuté par un nouveau flot soit dans le contexte de l'objet où l'événement a été signalé soit dans le contexte de l'objet où a eu lieu l'attachement.

Dans ces deux cas, l'attachement d'un traitant à un événement est irréversible. Les traitants associés aux flots d'exécution peuvent être chaînés : l'exécution d'une méthode par un flot peut associer pour ce flot un traitant à un événement qui en possède déjà. Plusieurs traitants peuvent ainsi être associés à un même événement pour un même flot.

#### II.4.4.2 Examen des critères d'étude

**Mode de communication**

Les événements sont délivrés de manière asynchrone : l'émetteur n'est pas bloqué et le récepteur n'a pas à se synchroniser. En revanche, le support à la diffusion se limite à l'émission vers un groupe de flots. Il n'est pas possible de réaliser la diffusion générale d'un événement.

**Désignation**

La communication induite n'est pas anonyme. En effet, l'opération d'émission comprend obligatoirement la mention d'une destination identifiée clairement et sans ambiguïté.

**Description du comportement des récepteurs – Évolution**

Pour chaque entité, on peut associer un traitant à un événement, ce qui détermine le comportement de l'entité lorsqu'elle reçoit l'événement. Cependant, pour les objets, ce comportement est fixé une fois pour toutes à leur création. D'autre part, les traitants associés aux flots d'exécution peuvent être chaînés, mais ne peuvent pas être révoqués.

**Modèle d'exécution**

L'émission des événements est réalisée au sein des objets par des flots d'exécution. L'exécution des traitants est dévolue à des flots créés à cet effet.

### II.4.5 Le modèle ECO

Le travail présenté dans [Starovic 95] est original. Il s'agit d'un modèle à objets dont le mécanisme d'invocation repose sur l'utilisation d'événements et d'un mécanisme semblable à l'invocation implicite (cf II.4.2). Ainsi, l'invocation synchrone de méthode n'est pas fournie.

#### II.4.5.1 Description

Dans ECO, les objets sont instances d'une classe qui définit les variables d'état et les méthodes qui permettent de manipuler cet état. Les interactions entre objets sont réalisées à l'aide d'événements émis de manière asynchrone.

Les événements sont définis de manière globale par un nom et une liste de paramètres. Chaque classe déclare quels événements ses instances peuvent recevoir (`inevents`) et peuvent émettre (`outevents`).

Les objets déclarent leur intérêt pour un événement en lui associant un traitant, qui est une de leurs méthodes. Les paramètres formels de la méthode doivent être

identiques à ceux de l'événement déclenchant. À l'occurrence de l'événement, la méthode est exécutée.

L'exécution des méthodes est contrôlée par des contraintes. Ces contraintes expriment notamment des conditions que doivent vérifier les événements pour être reçus et donc déclencher l'exécution des réactions. Elles permettent également de contrôler la synchronisation et les accès concurrents aux objets.

### II.4.5.2 Examen des critères d'étude

**Mode de communication**

ECO propose un mode de communication qui a les mêmes propriétés que l'invocation implicite décrite en II.4.2.

**Désignation**

Comme dans le cas de l'invocation implicite, la communication induite est anonyme.

**Description du comportement des récepteurs – Évolution**

Le comportement de chaque objet est déterminé par la correspondance entre événement et méthode exécutée à l'occurrence de l'événement. Cette correspondance peut être établie à la création de l'objet, mais peut évoluer dynamiquement au cours de l'existence de l'objet.

**Modèle d'exécution**

Comme nous l'avons mentionné plus haut, le modèle d'exécution associé n'est pas clairement défini, puisque la nature des objets (actifs ou passifs) n'est pas précisée.

Cette proposition est originale par son choix d'un unique mode d'invocation très particulier. Cependant, nous ne souhaitons pas substituer un mode de communication asynchrone au mode d'invocation synchrone classique, mais fournir ces deux modes simultanément.

## II.4.6 Synthèse

À l'exception du premier travail présenté, dont les lacunes font écho à celles relevées en II.2.1, l'ensemble de ces mécanismes décrivent des modes de communication qui correspondent aux principaux traits que nous recherchons : ils sont asynchrones, anonymes, ne nécessitent pas de synchronisation – ou d'attente – de la part des récepteurs. Ils obéissent de plus à un principe identique : des entités associent des traitements à des événements (notion d'abonnement analogue à celle

des bus de messages) et l'occurrence d'un tel événement dans le système entraîne l'exécution des traitements associés.

Les différences entre les différentes approches résident dans deux points :

- la prise en compte du modèle d'exécution dans la description du mécanisme et
- l'intégration du mécanisme dans le modèle hôte.

Le modèle d'exécution associé à ces différents travaux constitue une de leurs limites car il est souvent peu ou pas décrit.

L'intégration de ce type de communication avec les modèles à objets hôtes fait apparaître deux tendances. La première traduit une intégration faible et est représentée par les travaux présentés dans [Menon 93b] : les événements sont des signaux qui n'apparaissent pas dans les interfaces des classes. La seconde propose une intégration beaucoup plus forte où les événements sont des objets et sont parties intégrantes de l'interface de manipulation des classes. Elle est représentée par les travaux décrits dans [Bacon 95], [Notkin 93] et [Starovic 95].

Le principe de la communication ainsi mise en œuvre présente une grande similitude avec ce que proposent les systèmes de gestion de base de données, en particulier les bases de données actives. Nous nous intéressons donc dans la section suivante aux travaux menés dans ce domaine.

## II.5 Systèmes de gestion de bases de données

Il existe dans le domaine des systèmes de gestion de bases de données (SGBD) une activité très importante autour des bases de données actives. Ces systèmes permettent de décrire par des règles les actions à mener sur la base de données en fonction d'événements significatifs. Ce comportement correspond à celui que nous recherchons, puisque des traitements sont déclenchés de manière asynchrone et anonyme. Une synthèse des travaux menés dans ce domaine est présentée dans [Collet 96]. Les SGBD à objets n'échappent pas à cet intérêt. SAMOS [Gatziu 92], NAOS [Collet 94] ou Ode [Gehani 91] sont des exemples significatifs de ce type de système, que nous présentons dans la suite de cette section.

La base de la majorité de ces travaux est le modèle Événement–Condition–Action (ou E–C–A), dont nous indiquons le principe en préambule. Nous nous attacherons ensuite à décrire comment ses différents éléments ont été intégrés au modèle de données (modèle à objet) qu'utilisent les trois systèmes mentionnés plus haut.

### II.5.1 Le modèle Événement–Condition–Action (E–C–A)

Les règles actives qui décrivent le comportement et l'évolution des SGBD sont composées de trois parties : un Événement, une Condition et une Action. La sémantique associée à une telle règle est la suivante :

Lorsque l'*Événement* se produit

Si la *Condition* est satisfaite

Alors exécuter l'*Action*

On remarque la similitude de ce principe avec certains des travaux que nous avons présentés à la section précédente.

Dans de nombreux systèmes –dont ceux que nous présentons –, les événements sont soit **primitifs** soit **composites**. Les événements primitifs décrivent des changements dans le système, comme par exemple la modification des objets.

Ils peuvent être composés pour dénoter des changements plus complexes sous la forme d'événements composites. Ainsi, un événement composite peut être décrit par la conjonction de deux événements, ou par l'occurrence dans un ordre donné des ces événements.

Nous ne développons pas dans ce qui suit les aspects relatifs à la composition des événements.

### II.5.2 SAMOS

SAMOS [Gatziu 92][Gatziu 93][Geppert 95] est un SGBD à objets qui permet la définition de règles actives, construites sur le modèle E–C–A.

Les événements que permet de décrire SAMOS sont soit primitifs, soit composites. Des événements primitifs sont associés aux appels de méthodes, à la modification des valeurs d'un objet, à la gestion des transactions et à la gestion du temps. Le principe d'encapsulation impose que les événements de modification de valeur aient une portée limitée à la classe.

Deux types de règles peuvent être définis. Dans le premier cas, les règles sont attachées à une classe et font partie de sa déclaration. Les conditions et les actions peuvent alors manipuler directement l'état des instances de leur classe de déclaration. Dans le second cas, les règles sont déclarées indépendamment de toute classe. La manipulation directe des données des objets est alors proscrite, tant pour les conditions que pour les actions. De plus, les événements de modification de valeur ne peuvent pas déclencher de telles règles.

Les règles qui mettent en jeu des événements relatifs aux appels de méthode sont, quel que soit leur type, sensibles à l'héritage. En effet, elles peuvent être déclenchées par l'appel de la méthode concernée sur des instances de n'importe

quelle classe de la hiérarchie où est définie cette méthode. D'autre part, les règles déclarées de manière interne à une classe peuvent être héritées.

Enfin, règles et événements sont représentés comme des objets au niveau du modèle. Ils peuvent donc être manipulés comme tels à l'aide des méthodes définies dans leur classe.

### II.5.3 NAOS

NAOS [Collet 94][Collet 95] est un module de gestion de règles actives construit au dessus du SGBD $O_2$.

Comme pour SAMOS, NAOS distingue des événements primitifs et composites. Seules les opérations publiques peuvent générer des événements. Il n'y a donc pas comme en SAMOS la notion d'événement de modification de valeur.

Le niveau de déclaration des règles diffère de celui de SAMOS : elles sont définies au niveau des schémas de $O_2$, et donc au même niveau que les classes. En effet, $O_2$ permet la définition de schémas qui décrivent un ensemble de classes, liées entre elles par des liens de composition et d'héritage. L'exécution d'un programme $O_2$ se fait en référence à un schéma. Ainsi, les règles ne peuvent pas comme en SAMOS être attachées à une classe. Les règles peuvent être activées ou désactivées, ce qui permet d'adapter et de faire évoluer le fonctionnement des applications.

Les règles de NAOS sont sensibles à l'héritage comme le sont les règles définies hors des classes en SAMOS : un appel de méthode sur des instances de deux classes en relation d'héritage déclenche les même règles.

### II.5.4 Ode

Le SGBD Ode [Gehani 91][Gehani 92] offre la possibilité de définir des déclencheurs dans un environnement à objets. Les auteurs utilisent une variante des règles E–C–A, dans laquelle les règles décrivent l'association Événement–Action, tandis que les conditions sont déclarées indépendamment.

Les règles sont, comme les contraintes, exclusivement définies au niveau des classes. Ainsi, règles et contraintes peuvent faire l'objet d'un traitement homogène avec celui des autres éléments de la classe, ce qui conserve les caractéristiques du langage d'origine (C++ en l'occurrence).

### II.5.5 Synthèse

Les SGBD actifs à base de règles de type E–C–A présentent dans leur fonctionnement de nombreuses similitudes avec les travaux que nous avons présentés aux sections précédentes et avec le mode de communication que nous souhaitons mettre en place.

Ils permettent en effet de déclencher de manière asynchrone des traitements dans des points *a priori* indéterminés du système : les événements déclenchent les règles actives où qu'elles soient dans le système.

Les SGBD à objets ont par ailleurs abordé le problème de l'intégration de ces règles dans un environnement d'objets. Nous avons décrit trois approches différentes qui montrent en particulier que le lieu de déclaration des règles est important et détermine nombre de propriétés du mécanisme.

Le modèle E–C–A et les structures d'exécution associées ne constituent cependant pas un mécanisme de communication. Les systèmes de base de données permettent de communiquer par l'intermédiaire d'un espace partagé – la base de données – et le modèle E–C–A est un moyen d'assurer la cohérence de cet espace selon des critères fixés par les règles.

L'accent est également mis dans ces travaux sur la description des événements. En particulier, la description et la détection d'événements composites suscitent beaucoup d'intérêt. La détection en général est un problème crucial, car l'émission des événements n'est pas une opération explicite dans ces systèmes.

Nous ne nous sommes pas attardés dans cette étude sur les aspects relatifs au modèle d'exécution. En effet, les structures d'exécution des SGBD sont différentes de celles que nous considérons, puisqu'elles mettent essentiellement en jeu des transactions. Dans ce cadre, la question principale concerne les problèmes de **couplage**, c'est–à–dire le choix de la transaction au sein de laquelle une règle déclenchée sera exécutée et le moment auquel cette exécution aura lieu.

## II.6 Conclusion

Nous avons présenté dans ce chapitre différents travaux qui proposent des modes de communication asynchrone. Nous les avons analysés en fonction des besoins que nous avons mis en évidence au chapitre I.

Les critères que nous avons considérés caractérisent d'abord le caractère asynchrone et anonyme de la communication, ainsi que les possibilités d'évolution dynamique des interactions. Les possibilités d'intégration dans un modèle à objets constituent le deuxième type de critère que nous avons pris en compte.

Enfin, le modèle d'exécution associé revêt une grande importance, tant sa définition est nécessaire à une description précise du mécanisme.

Il ressort de cette étude que l'utilisation d'événements couplée à un mécanisme d'attachement offre des possibilités conformes à nos besoins : des entités, processus ou objets, s'abonnent à certains types d'événements en y attachant des réactions à exécuter et l'occurrence de ces événements déclenche l'exécution des réactions attachées. Ce mécanisme offre, outre ses caractéristiques d'asynchronisme et d'anonymat, une possibilité d'intégration qui est apparue à la fois en II.4 et en II.5.

Ainsi, nous choisissons ce principe pour réaliser notre proposition. Nous étudions donc au chapitre suivant les problèmes que posent la définition et l'intégration de ce type de communication dans un modèle à objets.

# Chapitre III

# Modèles d'événements

Nous présentons dans ce chapitre les éléments de discussion autour d'un mode de communication asynchrone qui repose sur des événements. L'étude vise à énumérer l'ensemble des choix offerts pour la réalisation d'un tel mécanisme dans un environnement à objets. Si l'environnement de référence demeure celui que nous avons décrit dans les chapitres précédents, nous nous efforçons cependant de ne pas restreindre la discussion à ce seul cadre, car la communication événementielle possède des caractéristiques génériques marquées. Ce choix implique cependant que les problèmes soulevés ne sont pas traités de manière complète : pour tous, nous donnons les principales questions à résoudre et les principes des solutions qui peuvent être envisagées.

## III.1 Communication événementielle : principes et apports

Nous donnons dans cette section introductive d'une part les principes sur lesquels repose la communication à base d'événements, que nous dénommons communication événementielle, et d'autre part ses apports lorsqu'on la compare à des modèles que nous avons présentés au chapitre précédent.

### III.1.1 Principes

La communication événementielle est fondée sur deux principes simples.

- Le premier principe déclare qu'un **événement** est un changement d'état du système qui entraîne l'émission d'un message. Par abus de langage, on désigne ce message comme l'événement lui–même. Cette émission est réalisée par une entité du modèle de programmation. Elle est asynchrone, car l'entité émettrice n'est pas bloquée en attente de réponses ou de délivrance du message.
- Le second principe est le principe de **réaction**. L'émission d'un événement entraîne dans l'ensemble du système l'exécution de réactions qui sont les réponses du système à un stimulus événementiel. Ces réactions ont été au préalable **attachées** à cet événement.

Ces principes permettent de modéliser l'évolution du système en fonction de stimuli qui sont pris en compte de manière asynchrone, mettant ainsi en œuvre un mode de communication découplée entre émetteurs et récepteurs.

L'évolution du système comprend à la fois l'évolution des données et celle des flots de contrôle et nécessite la mise en place de trois types de réactions. Les deux premiers types s'apparentent à des effets de bords. Le premier définit des réactions qui modifient les données. De telles réactions permettent par exemple de maintenir la cohérence entre des données dépendantes lorsqu'une partie en est modifiée. Le deuxième type concerne les flots de contrôle et permet de modifier leurs conditions (environnement, ressources) d'exécution. Enfin, le troisième type génère de nouveaux flots de contrôle. Il permet par exemple de lancer de nouvelles applications à l'occurrence d'événements donnés.

À titre d'exemple, considérons l'application d'édition coopérative que nous présentons au chapitre I. La notification de mise–à–jour correspond à un événement émis par l'instance d'éditeur où la modification a été validée. Les autres instances membres de la session ont pour leur part attaché à cet événement une réaction qui réalise les opérations souhaitées par l'utilisateur à l'occurrence d'un tel événement. Cela peut consister en un rafraîchissement automatique de la vue, dans le positionnement d'un témoin de modification ou en une action vide si l'utilisateur n'a aucun droit sur la portion modifiée du document.

### III.1.2 Apports

Notons d'ores et déjà que ce type de communication est différent de celui offert par un mécanisme d'appel asynchrone. En effet, lors d'un appel asynchrone, l'appelant connaît d'une part l'identité de l'appelé et d'autre part quel service ou quelle procédure sera exécuté sur cet appelé. En revanche, l'émetteur d'un événement ne sait *a priori* rien des traitements qui seront exécutés en réaction.

Dans un second temps, lorsque l'on compare ce mode de communication à celui qu'offre les bus logiciels [Oki 93] ou que propose les spécifications de l'OMG [OMG 93], l'apport essentiel réside dans la définition au sein même du modèle de la réaction qui sera exécutée lorsque la communication sera établie, ainsi que dans l'intégration plus forte du mécanisme dans le modèle.

En ce sens, le travail présenté est à rapprocher de ceux menés autour des bases de données actives qui, outre l'utilisation d'un modèle de description de type E–C–A, définissent les conditions d'exécution des actions à réaliser. La principale différence réside dans le fait que ce mécanisme n'est pas un mécanisme de communication dans les bases de données.

Les travaux décrits dans [Notkin 93] sont encore plus proches de nos objectifs, car le mécanisme proposé est réellement un mécanisme de communication. En revanche, nous essaierons de remédier dans notre étude à l'imprécision que laisse planer les auteurs sur le modèle d'exécution.

### III.1.3 Organisation du chapitre

La définition de ce que sont les événements et les réactions, la description des attachements ainsi que celle du modèle d'exécution associé sont exposés dans ce chapitre. La présentation s'attache à couvrir l'ensemble des choix possibles pour la conception d'un tel mécanisme de communication au sein d'un modèle à objets.

D'autre part, chacune des sections consacrées aux éléments de description du modèle est conclue par une synthèse des points que nous considérons comme intangibles du modèle, qui ne sont pas sujet à modification, et de ceux qui en revanche peuvent faire l'objet de choix de conception. L'ensemble de ces points est repris et synthétisé dans la section finale. Nous indiquons à cette occasion quelles options nous privilégions pour l'application de l'étude dans le cadre de Guide.

## III.2 Événements

Comme nous l'avons indiqué en ouverture de ce chapitre, les événements représentent des changements dans l'état du système et que l'on souhaite diffuser afin de les exploiter. Nous étudions dans cette section les éléments de description des événements. Plus particulièrement, nous montrons ce qui définit un événement et comment il est déclaré et désigné.

### III.2.1 Définition

Un événement représente une action observable, un changement d'état dans le système et y déclenche des réactions. Il est caractérisé par un type, que nous dénommons *famille*, par un contenu et par une durée de vie.

Un événement n'est pas un objet manipulable au sens classique du terme. Une fois émis, il ne peut en particulier pas subir de transformations de son état qui pourraient en modifier la nature. Dans ces conditions, si les événements devaient être considérés comme des objets, les méthodes qui pourraient leur être appliquées réaliseraient exclusivement des consultations, mais pas de modifications de l'état. Ce seraient donc des objets non modifiables. La définition d'une classe dont les instances ne peuvent pas être modifiées après leur création nécessite la mise en place de contrôles statiques permettant de vérifier cette propriété. Ces contrôles doivent également être appliqués aux sous-classes : aucune méthode de

modification ne doit être ajoutée par héritage. La mise en place de tels contrôles n'est pas simple et c'est afin de les éviter que nous introduisons une notion de famille distincte de celle de classe. La famille peut donc être perçue comme une classe dont aucune des méthodes ne peut modifier l'état des instances. Elle peut dès lors et selon les choix du concepteur être une classe à laquelle sont appliquées des contrôles supplémentaires ou une structure nouvelle soumise à des traitements différents.

Afin d'éviter toute confusion ultérieure, d'une part nous ne considérons pas les événements comme des objets dans la suite de ce chapitre et d'autre part nous distinguons clairement la notion de famille de celle de classe. Cette dernière distinction ne doit pas cacher les similitudes entre la relation famille–événement et la relation classe–objet. Elle nous permet en outre d'envisager un traitement différent pour les familles notamment du point de vue de la déclaration

#### III.2.1.1 Familles d'événements

Un événement identifie la nature d'un changement survenu dans le système. Des changements de même nature sont donc dénotés par des événements qui ont une structure semblable. De ce fait, on peut les regrouper dans une même famille qui définit un format commun utilisé à chaque émission d'un de ses événements membres. Ainsi, le contenu d'un événement respecte la structure de la famille dont il est instance. De plus, chaque famille d'événements correspond à une famille de changements observables dans le système et une de ses instances sera émise à l'occurrence de ce changement.

> La mise-à-jour d'un fragment de document partagé dans une session d'édition coopérative est un type d'événement qui, quelle que soit la modification apportée, possède la même sémantique et la même structure. On peut donc lui associer une famille d'événements dénommée par exemple majEvent.

#### III.2.1.2 Contenu d'un événement

Un événement permet de propager et de diffuser des informations sur l'évolution du système. Dès lors, son contenu s'attache à décrire ce changement.

Cette description comprend nécessairement l'identification de la nature du changement survenu. Comme nous l'avons indiqué en III.2.1.1, cette information est équivalente à la mention de la famille de l'événement. De plus, dans la mesure où chaque réaction est attachée à un événement spécifique, il faut pouvoir identifier les événements sans ambiguïté. Nous appelons *identificateur* de l'événement ces informations sur sa famille et son identité propre et nous reviendrons sur leur définition en III.2.3.

> La modification d'un fragment de document dans une session d'édition coopérative est un type d'événement particulier qui déclenche dans l'environnement des réactions spécifiques, qu'un événement d'un autre type ne doit pas déclencher si elles ne lui ont pas été attachées. L'événement de modification doit donc être identifié pour déclencher les réactions adéquates.

La détermination du type du changement survenu à l'aide de l'identificateur de l'événement peut se révéler insuffisante pour certaines utilisations des événements. En effet, des informations sur le contexte d'occurrence du changement peuvent être nécessaires à sa prise en compte correcte dans les réactions. L'événement doit donc pouvoir véhiculer de telles informations et c'est au sein de sa famille qu'est décrite la structure des informations nécessaires.

> La réaction à une modification de fragment de document peut dépendre de l'identité de ce fragment. Il est donc souhaitable que l'événement correspondant à cette modification véhicule également cette information contextuelle qui peut être nécessaire, pour un rafraîchissement par exemple.

Ainsi, un événement comprend deux parties distinctes. La première, son identificateur, est la clé qui va permettre de déclencher les réactions attachées à l'événement en question. La seconde contient les paramètres de l'événement. Ce sont des données qui seront utilisées dans les réactions lors des traitements qu'elles ont à effectuer.

### III.2.1.3 Durée de vie d'un événement

Les changements qui surviennent dans un système marquent l'évolution de celui–ci et les événements permettent d'en rendre compte. Ces changements sont par nature transitoires, ils s'apparentent à une transition entre deux états. Cependant, il peut être souhaitable de conserver l'historique de ces changements pour certaines utilisations (techniques de journalisation). Les évolutions du système deviennent ainsi persistantes. Du fait que les événements sont associés à ces changements, cette dualité entre fugacité et persistance les concernent également. Afin de préciser cette dualité, nous introduisons la notion de *durée de vie* d'un événement.

Cette durée de vie est définie par la période pendant laquelle les réactions qui lui ont été attachées peuvent être exécutées. Un corollaire de cette définition est que les réactions qui sont exécutées sont celles qui ont été attachées avant l'émission de l'événement ou pendant sa période d'existence. Cette durée de vie est mesurée à partir de l'instant où le système a pris l'événement en compte et déterminé les réactions à exécuter à ce moment précis.

Un événement est **fugace** si sa durée de vie est nulle. Dans ce cas, il ne survit pas à sa prise en compte par le système et tout attachement ultérieur d'une réaction à cet événement ne sera pas honoré.

Une durée de vie non nulle définit un événement **persistant** qui peut déclencher des réactions pendant une période plus longue[1]. La détermination de la durée de vie pourrait se faire dans l'absolu –tel événement a une durée de vie de cinq jours deux heures et trente trois minutes– mais il nous semble peu plausible qu'une telle définition soit utile. En revanche, la subordination de la persistance d'un événement à celle d'éléments de l'environnement est un choix judicieux. Ainsi, un événement peut par exemple persister tant que le flot de contrôle qui l'a généré existe, tant que l'objet depuis lequel il a été émis persiste ou tant que tel autre événement n'a pas été émis. Il peut également disparaître sur une commande explicite d'annulation.

> Afin d'illustrer cette caractéristique des événements, nous considérons l'exemple de la connexion tardive d'un nouveau membre dans une session d'édition coopérative. Ce nouveau membre doit disposer d'une version du document à jour et nous envisageons deux politiques pour cela.
>
> La première politique consiste d'abord à récupérer la version du document telle qu'elle a été sauvegardée pour la dernière fois sur support permanent (disque dur par exemple), puis à rejouer toutes les modifications qui lui ont été apportées depuis, jusqu'à la nouvelle connexion. Dans ce cas, des événements de modifications persistants permettent la construction de ce journal et le nouvel arrivant n'a plus qu'à réagir à ces événements pour retrouver une version cohérente avec les autres coauteurs. De plus, les événements ne persistent que jusqu'à la sauvegarde suivante.
>
> La seconde politique est basée sur une copie depuis l'espace d'un des autres utilisateurs d'une version à jour du document. Dans ce cas, les événements antérieurs n'ont pas à être conservés et peuvent donc être fugaces.

Comme le montre l'exemple précédent, le choix entre fugacité et persistance dépend du cadre d'utilisation des événements.

---

( 1 ) Cette notion de persistance diffère de celle appliquée en général aux objets : un objet est persistant s'il survit au flot d'exécution qui l'a créé.

### III.2.2 Déclaration

La déclaration des événements comprend deux volets. Le premier concerne la déclaration des familles auxquelles ils appartiennent. Cette déclaration couvre les aspects statiques de la définition des événements. Le second a trait à la déclaration des événements eux–mêmes au moment de leur émission, et définit donc le mode d'instanciation des événements. Cette dualité entre aspects statiques et dynamiques de la déclaration est semblable à celle qui existe pour les classes et les objets.

#### III.2.2.1 Familles

Le rôle d'une famille d'événements vis–à–vis de ses membres est semblable à celui du modèle d'instance de la classe pour les objets. Comme une classe définit la structure d'un objet, une famille d'événements décrit la structure d'un événement, à savoir l'identificateur et les informations véhiculées (cf III.2.1.1).

La déclaration au même niveau que les classes de familles d'événements servant de modèle aux événements émis dans le système est donc un choix qui peut être adopté. Dans ce cas, chaque événement doit se conformer à un tel modèle déclaré au préalable et rendu visible à l'ensemble des entités qui souhaitent l'utiliser. L'avantage d'une déclaration globale comme celle–ci est de donner une visibilité explicite aux événements qui peuvent être émis au sein d'une application ou dans le système. Ces déclarations doivent être indépendantes des classes car des classes différentes, même sans lien d'héritage, peuvent utiliser les mêmes types d'événements. L'inconvénient majeur de cette approche est d'ajouter dans le modèle à objets de nouvelles constructions de haut niveau pour le support des événements.

Dans cette approche, la structure d'un texte source pourrait être la suivante :

```
FAMILY majEvent
    NAME sender;
    PARAMETERS
        RefFragment modifiedFragment;
        Modification modif;
END
...
CLASS myClass
EXPORT m
BEGIN
    METHOD m(...)
    {
        // Instanciation et émission d'un événement
        // de famille majEvent
    }
    ...
END myClass;
```

À un niveau intermédiaire, la déclaration des familles d'événements au sein même des classes constitue un autre terme de l'alternative. Avec cette

approche, chaque famille d'événements est associée à une classe donnée. Si cette classe ne rend pas visible la famille dans son interface, ladite famille ne peut être utilisée, tant pour l'émission d'événements que pour la déclaration d'attachements, que dans cette classe ou ses descendantes dans la relation d'héritage. Par suite, l'émission d'événements de cette famille ne pourra être réalisée que dans les méthodes de ces classes appelées sur leurs instances. Ainsi, les changements du système sont clairement associés aux changements dans l'état des objets, ce qui n'est en soit pas aberrant dans un environnement à objets. Cependant, deux classes indépendantes – c'est–à–dire sans ancêtre commun – ne peuvent alors pas utiliser les mêmes familles d'événements. Si, en revanche, la classe de déclaration fait apparaître dans son interface la famille d'événements, celle–ci est visible et utilisable partout où la classe est connue. Ainsi, des événements membres peuvent être émis ailleurs que dans la classe de déclaration.

Si nous reprenons l'exemple précédent, la structure d'un texte source serait alors la suivante :

```
CLASS myClass
EXPORT m
BEGIN
    FAMILY majEvent
        NAME sender;
        PARAMETERS
            RefFragment modifiedFragment;
            Modification modif;
    END
    ...
    METHOD m(...)
    {
        // Instanciation et émission d'un événement
        // de famille majEvent
    }
    ...
END myClass;
```

Une autre solution consiste à ne pas recourir à des déclarations globales pour définir des familles d'événements, mais à considérer qu'elles sont déclarées implicitement par la définition et l'utilisation d'un de leurs membres. Cette approche offre une flexibilité plus grande, puisque rien ne contraint *a priori* la définition d'un événement. D'autre part, elle évite l'introduction de constructions supplémentaires pour la déclaration des familles. En revanche, elle nuit à la visibilité des événements si aucun substitut n'est trouvé pour informer le programmeur de l'existence de telle ou telle famille d'événements.

### III.2.2.2 Événements

Un événement est déclaré et instancié selon le modèle défini par sa famille au moment de son émission. Selon que les familles d'événements sont ou non des éléments explicites et manipulables, la déclaration d'un événement peut prendre

plusieurs formes. Dans tous les cas, et quelle que soit l'unité où l'événement est déclaré –classe, programme ou autre séquence de code –, la famille de l'événement doit être accessible et connue dans cette unité.

Dans le cas d'une déclaration globale des familles, un événement doit se conformer au modèle d'une famille pré–existante et être déclaré comme tel. Sa déclaration s'apparente alors à celle d'un objet typé, ce qui permet la mise en place de contrôles statiques. Son instanciation respecte le modèle correspondant.

Le cas d'une déclaration dans les classes des familles d'événements offre deux choix possibles. Dans une première approche, les événements ne peuvent être déclarés que dans la classe où leur famille l'a été. La seconde solution est d'intégrer d'une manière ou d'une autre dans l'interface des classes les familles qu'elles définissent. Ainsi, visibles à l'extérieur, elles pourront être utilisées dans tout le système et leurs événements membres pourront être utilisés partout.

En revanche, si aucune déclaration de famille n'est faite, la déclaration d'un événement ne se conforme *a priori* à aucune contrainte. Cette déclaration peut parfaitement se conformer à un modèle décrit lors de la déclaration d'un autre événement, mais peut également en définir un nouveau.

Dans tous les cas, l'instanciation de l'événement à partir de sa famille est réalisée au moment de son émission. Un événement instancié et non émis n'a pas de sens en soi, car ce n'est pas un objet manipulable.

### III.2.3 Désignation

Comme les aspects déclaratifs, la désignation des événements concerne à la fois les événements eux–mêmes et les familles qu'ils représentent. Tant pour les événements que pour les familles, le problème de la désignation concerne la forme de l'identificateur utilisé pour désigner et la portée de cet identificateur.

#### III.2.3.1 Familles

Nous l'avons déjà mentionné, une famille d'événements est à bien des égards semblable à une classe d'objets. Les différences majeures entre ces deux types d'abstractions sont qu'une famille d'événements ne définit pas d'opérations sur ses instances et que son niveau de déclaration n'est pas forcément celui d'une classe (cf III.2.2.1).

La désignation des familles d'événements est nécessaire pour l'utilisation des événements comme la désignation des classes l'est pour l'utilisation des objets. Ainsi, un nom est attribué à chaque famille d'événements. La forme de cet identificateur peut également tirer parti du mode de déclaration de la famille. Ainsi, le nom de la classe où elle est définie peut être utilisé.

La portée de cet identificateur définit le sous–ensemble du système où ce nom est connu et où il désigne sans ambiguïté la famille considérée. La définition de cette portée dépend entre autres du mode de déclaration choisi. Nous avons évoqué ces problèmes de portée en III.2.2. Si la déclaration des familles est confinée au sein des classes, une portée naturelle de son nom est la classe de déclaration. Si l'interface de la classe rend visible cette famille à l'extérieur, l'utilisation du nom de la classe pour forger l'identificateur de la famille assure que la portée du nom de la famille est la même que celle du nom de la classe.

En revanche, la déclaration globale ou implicite des familles rend plus délicate la définition d'une portée. Il serait alors souhaitable d'avoir recours à des services externes d'administration qui permettraient au programmeur de gérer cette portée. Cette gestion relève alors de l'administration d'applications.

### III.2.3.2 Événements

L'identification des événements peut permettre de discriminer entre les événements eux–mêmes ou entre des familles d'événements. Dans le premier cas, deux événements distincts, instances de deux familles distinctes ou non, ont chacun un identificateur différent. Dans le second, ils ont même identificateur s'ils appartiennent à la même famille. La première solution permet de particulariser chaque événement afin d'exploiter des particularités de l'instance telles qu'un éventuel numéro d'ordre à l'émission ou la mention de son origine. L'autre approche concentre l'utilisation des événements sur la notification de types de changements identifiés, qui seuls importent alors.

Le choix entre ces deux possibilités est déterminé par l'utilisation des événements, et donc par les réactions qui leur sont associées. En effet, l'attachement d'une réaction à un événement se fait en utilisant l'identification de l'événement. Dans la première approche, la clé qui déclenche les réactions ne doit pas se limiter à la seule identification de l'événement, mais prendre en compte des motifs de son identificateur, typiquement l'identité de sa famille et du type de changement qu'il dénote, qui servent de clé au déclenchement des réactions. En effet, il serait dans le cas contraire impossible d'associer une réaction à l'ensemble des occurrences d'une famille donnée. Une alternative possible consiste à reporter dans la structure de l'événement, c'est–à–dire dans son contenu (paramètres), les informations qui le particularisent par rapport aux autres instances de sa famille. La distinction entre deux événements de même famille nécessite alors la mise en place de filtres sur leur contenu.

L'identification d'un événement ou d'une famille d'événements prend place au sein d'une certaine portée. Cette portée définit le domaine au sein duquel il n'y a aucune ambiguïté sur cette identification et par suite sur le rôle de l'événement. La

gestion de ces portées peut être définie à la déclaration des éléments désignés, en relation par exemple avec leur contexte de déclaration. Ainsi, une famille d'événements déclarée au sein d'une classe pourrait avoir une portée limitée aux objets de cette classe. Cette gestion peut également reposer sur un service de nommage des événements qui gère les conflits de noms d'événements. Un tel service ne présente aucune spécificité particulière par rapport à d'autres services de nommage utilisés par exemple pour des objets.

### III.2.4 Familles et héritage

Nous avons déjà mentionné la similitude entre les notions de famille et de classe : ce sont des modèles pour leur instances. Nous pouvons pousser plus avant la comparaison si nous introduisons entre les familles une relation d'héritage.

La sémantique associée à l'héritage est la suivante :

> La famille d'événements $F_2$ hérite de la famille d'événements $F_1$ si le type de changement dénoté par $F_2$ implique le type de changement dénoté par $F_1$.

Dès lors, si $F_2$ hérite de $F_1$, toute instance de $F_2$ dénote au moins le changement que dénoterait une instance de $F_1$. Elle peut donc déclencher les réactions associées aux instances de $F_1$.

Ainsi, un événement serait considéré comme une instance d'un ensemble hiérarchisé de familles et pourrait déclencher des réactions associées à ses super–familles. Cette propriété implique deux contraintes.

En premier lieu, les paramètres définis dans une sous–famille doivent être conformes à ceux de la super–famille. En effet, ces paramètres peuvent être utilisés dans les réactions associées aux événements. Des paramètres non conformes entraîneraient des erreurs à l'exécution.

Deuxièmement, l'identification des événements, qui permet de déterminer les réactions à déclencher, doit prendre en compte cette hiérarchie. Le format de l'identificateur des événements doit alors être adapté, car un événement peut déclencher des réactions qui ne sont pas associées à des événements de sa famille. Une reconnaissance purement syntaxique reposant sur la recherche d'un motif dans l'identificateur est une solution à ce problème.

En tout état de cause, l'utilisation d'une relation d'héritage complique la définition et le traitement des événements et le bénéfice en terme de fonctionnement est rien moins qu'évident.

### III.2.5 Invariants et éléments de choix

**Invariants**

1. Un événement est instance d'une famille.
2. Un événement est identifié par un nom, contenant au minimum le nom de sa famille.
3. Un événement a des paramètres dont les types sont définis par sa famille.

**Éléments de choix**

1. Format de l'identificateur d'un événement

   L'identificateur peut être unique pour chaque événement ou identique pour tous les événements d'une même famille. Une discrimination se ferait alors sur les paramètres. Nous privilégierons ce dernier choix.

2. Déclaration des familles : forme, lieu.

   Les deux choix extrêmes sont :

   - intégration forte avec le modèle hôte : les familles sont des classes auxquelles peuvent être appliquées les règles de l'héritage et de la conformité.
   - les familles ne sont pas des classes et leur espace est un espace plat. Ce choix est celui de la simplicité et c'est celui que nous privilégierons dans notre application.

## III.3 Réactions

Les réactions constituent les réponses du système aux stimuli événementiels. Leur déclaration et leur désignation, comme celles des événements qui les déclenchent, sont indispensables à la définition complète d'un modèle de communication événementielle.

### III.3.1 Définition

Une *réaction* est un programme exécuté à l'occurrence d'un événement. Un *programme* est défini par un point d'entrée, des données de travail et une séquence d'instructions.

La déclaration de ces réactions doit permettre de préciser quelle forme prennent les programmes qui les représentent, comment et où ils sont définis, alors que leur désignation définit le moyen d'identifier et de retrouver les programmes associés.

### III.3.2 Déclaration

Il existe dans un modèle à objets différentes formes pour définir un programme, et chacune de ces formes peut être utilisée pour déclarer une réaction. Ces différentes formes correspondent à différents niveaux de déclaration.

#### III.3.2.1 Forme d'un programme

Il existe dans le contexte du modèle où nous nous sommes placés quatre possibilités pour la déclaration des séquences d'instructions qui pourront servir de réactions à des événements.

- un binaire exécutable : un binaire exécutable peut être utilisé comme réaction à un événement. Il définit complètement les instructions qui le composent (appels de méthodes, tests, itérations) et les données qui lui sont nécessaires. L'intérêt principal de cette possibilité réside dans la récupération et la réutilisation d'applications existantes.
- une méthode : c'est la forme de programme la plus naturelle et la plus simple dans un modèle à objets. Toutes les données qui lui sont nécessaires sont accessibles depuis l'objet sur lequel elle est exécutée.
- une procédure : à la différence d'une méthode, une procédure n'apparaît pas dans l'interface d'un objet. Comme pour la méthode, les données utilisées sont celles accessibles depuis l'objet où elle est appelée.
- une séquence d'instructions quelconque : on peut considérer une suite d'instructions quelconque comme un programme, mais sous certaines conditions. Il faut en particulier lui associer un point d'entrée et des données sur lesquelles la séquence agira et qui seront accessibles quel que soit le contexte où s'exécutera la réaction en question.

Pour pouvoir être exécutés, chacun de ces types de programmes doit fournir un point d'entrée accessible. Si ce point d'entrée existe pour un exécutable et pour une méthode associée à un objet, sa définition pour une procédure et pour une séquence d'instructions quelconque recèle plus de difficultés.

### III.3.2.2 Niveau de déclaration

Il convient ensuite de déterminer à quel niveau une réaction doit être déclarée.

- au niveau d'une classe : les réactions sont définies dans le code des classes. Les méthodes ou les procédures définies dans la classe constituent dans ce cadre des formes naturelles de réaction. L'utilisation de séquences de code quelconques dans les classes suppose qu'elles soient particularisées, soit par l'introduction d'une construction supplémentaire du niveau de la méthode, soit en les associant explicitement aux opérations d'attachement.
- à un niveau global : les réactions sont des éléments de base du modèle et donc déclarées et définies à un niveau équivalent de celui des classes. Cette approche est la seule possible pour déclarer explicitement des programmes exécutables comme réactions. Elle suppose cependant l'introduction de nouvelles constructions dans le modèle pour déclarer les autres formes de réactions, plus particulièrement les séquences quelconques d'instructions.

### III.3.3 Désignation

Lorsqu'elle est attachée à un événement et au même titre que celui–ci, une réaction doit être identifiée sans ambiguïté. Le mode de désignation dépend des réponses apportées au niveau de la déclaration et des modes d'attachement retenus.

**Utilisation dans la classe de déclaration**

La portée d'une déclaration de réaction ne dépasse pas le cadre de la classe et de ses instances. Dans ce cas, l'identification d'une réaction se fait par un nom de méthode ou de procédure. Si la réaction est définie par une séquence quelconque d'instructions, il n'existe pas de moyen de la désigner et donc de la retrouver au moment de l'attachement. Le seul moyen est donc de la définir en même temps que l'attachement lui–même. De plus, les données sur lesquelles agira la réaction seront nécessairement celles de l'objet où est réalisé l'attachement.

Un programme exécutable ne peut pas être défini en tant que tel dans une classe, puisque c'est le produit d'une compilation et d'une édition de liens.

**Utilisation à un niveau global**

Une déclaration globale des réactions suppose que la portée de cette déclaration dépasse le cadre d'une classe et impose donc la définition d'identificateurs plus complexes.

Si la réaction est définie par une méthode, la désignation d'un objet sur lequel opérera la méthode est nécessaire. Le cas de la procédure est exclu, puisqu'une procédure est invisible à l'extérieur d'un objet.

Si la réaction est définie par une séquence d'instructions (autre qu'une méthode) dans une structure globale, elle doit être identifiée. Dans le cas particulier où la séquence considérée est celle d'un binaire exécutable, le nom de cet exécutable sert à identifier la réaction.

| | | Niveau d'utilisation | |
|---|---|---|---|
| | | au niveau de la classe | à un niveau global |
| Mode de déclaration | Méthode | nom de méthode seul | nom de méthode + nom d'objet |
| | Procédure | nom de procédure seul | – |
| | Binaire exécutable | – | identificateur global |
| | Séquence d'instructions | associée à un attachement | identificateur global |

*Fig. 3.1 : Modes de désignation d'une réaction*

### III.3.4 Invariants et éléments de choix

**Invariants**

1. Une réaction est un programme avec un point d'entrée.
2. Chaque type de réaction possible possède un mode de désignation adapté.

**Éléments de choix**

1. Certaines formes de programmes peuvent être abandonnées

   Les procédures et séquences de code quelconques peuvent poser des problèmes inutiles.

2. Problème de la déclaration

   Les réactions peuvent faire l'objet d'une déclaration spécifique, ce qui nécessite la définition de nouvelles structures, ou bien ce peuvent être des éléments existants du modèle hôte (méthodes, procédures).

L'option que nous retenons pour l'application au modèle Guide est faire correspondre les réactions aux méthodes et de ne pas introduire de structures de déclaration spécifique.

## III.4 Attachement

L'attachement d'une réaction à un événement est l'opération centrale dans le fonctionnement du mécanisme. C'est à travers elle que s'établit la communication. Cette opération met en jeu plusieurs éléments qu'il faut préciser.

### III.4.1 Éléments de description d'un attachement

L'opération d'attachement associe une réaction R à un événement e. Une fois cette opération réalisée, l'émission de e provoque l'exécution de R.

La définition d'un attachement suppose donc en premier lieu la donnée d'un identificateur d'événement et d'un identificateur de réaction conformes aux principes de désignation décrits en III.2.3 et en III.3.3. Cependant, ces informations ne sont pas suffisantes. En effet, elles ne définissent pas de droits sur la modification de l'attachement et ne précisent pas dans quel contexte la réaction sera lancée. La question des droits sur l'attachement, à savoir qui a le droit de le supprimer par exemple, est cruciale si on souhaite que l'ensemble des attachements puisse évoluer au cours du temps. Quant au contexte de lancement de la réaction, il est indispensable de l'associer à la réaction.

Considérons de nouveau l'exemple de la mise–à–jour d'un document partagé dans une session d'édition coopérative. Si chaque participant à la session a associé pour son instance d'éditeur une réaction à l'événement de mise–à–jour, cette réaction doit être lancée dans le contexte de cette instance. Dans le cas contraire, son

exécution ne produira aucun effet visible ou en produira dans un contexte impropre (celui d'un autre coauteur par exemple). La spécification de ce contexte doit donc être associée à l'attachement.

Ces deux aspects de la définition d'un attachement sont discutés dans la suite de cette section.

### III.4.2 Contexte de lancement

Le contexte de lancement d'une réaction décrit quelles sont les unités d'exécution qui en sont chargées. Il s'agit donc de définir un flot d'exécution qui réalisera la réaction et l'environnement, dénommé *contexte d'exécution*, dans lequel cette exécution aura lieu.

#### III.4.2.1 Flot d'exécution

Pour ce qui concerne le flot d'exécution associé à la réaction, nous distinguons deux cas :

- la réaction est exécutée par un nouveau flot d'exécution : le flot de contrôle est créé à seule fin de réaliser la réaction. Ce mode de réaction peut être utilisé pour réaliser des effets de bord ou pour générer de nouveaux flots de contrôle dont l'utilisateur a besoin.
- la réaction est exécutée par un flot d'exécution existant : il existe dans le système un flot d'exécution qui est chargé de réaliser la réaction. Deux cas se distinguent à nouveau pour la détermination de ce flot de contrôle. Il peut s'agir :
  - d'un flot particulier et bien identifié, et l'exécution de la réaction permet de modifier les conditions d'exécution du flot chargé de la réaction, ou bien
  - d'un flot quelconque non identifié, et le flot choisi n'a alors aucune importance et la réaction est un pur effet de bord. L'utilisation de flots dédiés à ce type de réaction constitue un choix de réalisation très acceptable, puisqu'il épargne alors la recherche d'un flot au moment d'exécuter la réaction.

  Nous examinons par la suite la question du moment auquel le flot d'exécution choisi exécutera effectivement la réaction.

#### III.4.2.2 Contexte

C'est le contexte dans lequel s'exécutent ces flots de contrôle qui détermine le type de réaction souhaité. Nous employons ici *contexte* dans le sens d'un ensemble de variables d'état partagé par des flots de contrôle. Le contexte est donc ici caractérisé par les objets partagés par les mêmes flots. Ainsi une application

détermine un contexte et tous les flots qui participent à son fonctionnement s'exécutent dans ce contexte. Dans le modèle Guide, la notion de tâche au sein de laquelle s'exécutent les activités être considérée comme un contexte. Dans un système comme Unix, la notion de processus multiprogrammé correspond également à cette définition.

De même que pour le choix du flot de contrôle, il existe une alternative pour celui du contexte d'exécution et cette alternative présente sensiblement les mêmes termes. Le contexte peut être entièrement nouveau, existant et bien déterminé ou existant et quelconque. Chacun de ces cas correspond à une variété de réaction :

- Un contexte nouveau matérialise la création d'une application. Une réaction exécutée dans un tel contexte l'est forcément par un nouveau flot de contrôle.

  > Dans un environnement de développement intégré, lorsqu'une erreur à l'exécution d'un programme intervient, un débogueur, c'est-à-dire une nouvelle application, est lancé.

- Un contexte existant et bien identifié matérialise une application existante au sein de laquelle la réaction sera exécutée. Cette réaction est donc spécifique à l'application considérée. Le flot chargé de cette exécution peut être un nouveau flot, qui correspond à un nouveau flot de contrôle nécessaire au bon fonctionnement de l'application. Ce peut être un flot existant mais quelconque, qui réalise alors un effet de bord local à l'application.

  > La gestion du nombre de participants à une session d'édition coopérative doit être gérée dans le contexte de cette session, mais l'identité du flot qui à la connexion ou à la déconnexion d'un éditeur va mettre à jour cette donnée n'importe pas.

  Enfin, ce peut être un flot bien identifié de l'application et dont on souhaite modifier les conditions d'exécution.

  > Si un événement notifie le changement d'une version de programme, les activités qui exécutent effectivement ce programme doivent elles-mêmes prendre en compte l'événement et y réagir.

- Un contexte existant mais quelconque représente certes une application mais son rôle n'a *a priori* aucune importance pour que les effets de la réaction soient perceptibles ou valides. En conséquence, les réactions qui s'exécutent dans un tel contexte sont nécessairement des effets de bord dont la portée est **globale** à l'ensemble du système. Enfin, le flot chargé de leur exécution ne peut pas être un flot bien identifié puisque la détermination du flot entraîne la détermination de son contexte englobant.

```
Dans un système de gestion de bases de données, le maintien
en cohérence de données qui doit se faire à l'occurrence
de certains événements peut être réalisé dans un contexte
quelconque.
```

L'ensemble des différentes combinaisons possibles pour la détermination d'un contexte de lancement et les types de réactions correspondants sont synthétisés dans le tableau Fig. 3.2.

| | | Contexte d'exécution | | |
|---|---|---|---|---|
| | | nouveau | quelconque existant | identifié existant |
| Flot d'exécution | nouveau | déclenchement d'application | effet de bord | génération d'un flot de contrôle |
| | quelconque existant | – | effet de bord | effet de bord local |
| | identifié existant | – | – | contrôle d'activité |

*Fig. 3.2 : Contextes de lancement et types de réactions*

### III.4.3 Abonnement et désabonnement

Un attachement est *actif* si l'occurrence de l'événement qu'il concerne provoque l'exécution dans le contexte défini de la réaction associée. Dans le cas contraire, l'attachement est *inactif*. L'activité d'un attachement constitue donc une partie de la condition de déclenchement d'une réaction et est par conséquent à rapprocher de la condition dans le modèle E–C–A des SGBD actifs.

Jusqu'à présent, nous considérions sous le terme d'attachement la description d'une association événement–réaction–contexte de lancement et l'opération consistant à l'activer. Désormais, nous réservons cette dénomination à la description de l'association, et nous appelons *abonnement* l'opération qui active un attachement et *désabonnement* l'opération inverse. Ces deux opérations permettent par conséquent de faire évoluer l'ensemble des attachements actifs dans le système à un instant donné.

Cette distinction montre bien que la déclaration d'un attachement et son utilisation à travers les opérations d'abonnement et de désabonnement peuvent être distinctes.

### III.4.3.1 Nature des opérations

Nous distinguons deux formes pour l'opération d'abonnement. Ces formes se distinguent l'une de l'autre par le moment où elles ont lieu. La première forme, dénommée *abonnement initial*, est celle d'un abonnement effectué lors de la création d'une entité (objet, flot d'exécution ou autre). Un tel abonnement est implicite et peut être considéré comme un paramètre de cette création. La seconde forme est dénommée *abonnement dynamique* et peut intervenir à n'importe quel instant.

L'opération de désabonnement peut également prendre deux formes que nous qualifions d'*implicite* et d'*explicite*. Un désabonnement implicite désactive un attachement sans qu'il soit besoin d'exprimer cette suppression. Au contraire, un désabonnement explicite est une opération déclarée qui concerne un attachement clairement défini. Une conséquence de ce dernier point est qu'un attachement doit pouvoir être identifié et donc désigné sans ambiguïté.

### III.4.3.2 Type d'un attachement

La durée de vie d'un attachement est le temps qui sépare son abonnement de sa désabonnement, période pendant laquelle l'émission de l'événement qu'il concerne déclenchera la réaction associée. La détermination de cette durée s'appuie sur les différentes formes d'abonnement et de désabonnement que nous avons décrites en III.4.3.1 et induit l'introduction de plusieurs types d'attachements. Nous les détaillons dans la figure Fig. 3.3.

Un attachement est valide tant qu'un lien existe entre lui et le système. Ce lien est matérialisé en particulier par les éléments qui composent l'attachement – contexte de lancement, objet sur lequel une méthode est appelée en réaction, programme exécutable associé – et sans lesquels l'exécution de la réaction associée est impossible et incorrecte. Nous appelons ces différents éléments les *éléments constituants* de l'attachement. Nous proposons de caractériser un attachement par la possibilité de lui appliquer de manière licite les opérations d'abonnement ou de désabonnement.

| | Abonnement initial | Abonnement dynamique |
|---|---|---|
| Désabonnement implicite | attachement statique implicite | attachement dynamique implicite |
| Désabonnement explicite | attachement statique explicite | attachement dynamique explicite |

*Fig. 3.3 : Différents types d'attachements*

En premier lieu, il convient de noter que la disparition d'un de ses éléments constituants (contexte de lancement ou objet sur lequel porte la réaction par exemple) entraîne un désabonnement **implicite** de l'attachement. Par conséquent, un désabonnement implicite est toujours licite pour un attachement. Cette hypothèse posée, les types d'attachements possibles sont les suivants :

- attachement statique implicite : ce type d'attachement est associé à une entité pour toute la durée de son existence. Il ne peut pas faire l'objet d'un abonnement dynamique, ni d'un désabonnement explicite.
- attachement statique explicite : activé à la création d'une entité, un tel attachement peut être désactivé avant la disparition de l'entité.
- attachement dynamique implicite : un tel attachement fait l'objet d'un abonnement dynamique, mais ne peut pas être désactivé explicitement. C'est la destruction d'un de ses éléments constituants qui entraîne son désabonnement.
- attachement dynamique explicite : ce type d'attachement est activé dynamiquement et peut être désactivé explicitement avant la disparition de ses éléments constituants.

### III.4.3.3 Droits sur un attachement actif

L'opération de désabonnement explicite soulève la question des droits associés à un attachement. En effet, il n'est pas souhaitable d'autoriser n'importe quelle flot de contrôle à réaliser sur n'importe quel attachement actif une opération de désabonnement. La solution la plus simple consiste à n'autoriser le désabonnement que si l'utilisateur qui réalise la commande est celui qui a au préalable activé l'attachement.

Dans une approche plus évoluée, il faut pouvoir garantir que seules les personnes autorisées à le faire peuvent désactiver un attachement. Dans ce cas, un attachement doit également comprendre la description d'informations de protection. La protection des attachements peut être réalisée à l'aide de multiples techniques. Il peut s'agir par exemple de la gestion d'une liste d'utilisateurs à qui le droit de désactiver l'attachement est accordé. Une autre technique consiste à associer à chaque attachement une clé, que le créateur de l'attachement peut transmettre à ceux qu'il autorise à désactiver et qui doit être mentionnée pour toute opération de désabonnement.

### III.4.3.4 Informations dynamiques et statiques

L'étude précédente a montré qu'un attachement définissait deux types d'informations :

- Informations statiques : ce sont les informations invariantes de la définition d'un attachement. Elles concernent le nom de l'événement et le programme à exécuter en réaction.
- Informations dynamiques : ces informations peuvent varier en fonction du contexte d'utilisation de l'attachement, en particulier des opérations d'abonnement. Ce sont le contexte de lancement et le filtre sur les événements déclencheurs.

La distinction de ces deux types d'informations permet de déclarer statiquement un attachement unique par événement et réaction associée auquel pourront être affectés différents contextes et filtres, pour différents abonnements : une déclaration d'attachement unique permet de générer plusieurs abonnements distincts par leurs informations dynamiques.

### III.4.4 Contrôle des attachements

L'utilisation des attachements peut être ou non contrôlée. En particulier, la question est de savoir si n'importe quelle réaction peut être attachée à n'importe quel événement dans n'importe quel contexte. En fonction des modes de représentation des réactions et de la portée donnée à ces représentations, il existe différentes possibilités d'activer des attachements et de limiter ou non les attachements possibles.

#### III.4.4.1 Contrôle statique

L'attachement d'une réaction à un événement est soumis à des contraintes qui peuvent être vérifiées à la compilation. Nous nous attachons à détailler les contraintes liées à l'événement et à la réaction. Ces contraintes sont les suivantes :

- **Validité de l'événement et de la réaction.**

  Les données de l'opération d'attachement sont un identificateur d'événement et un identificateur de réaction. De plus, leur portée doit inclure le contexte de déclaration de l'attachement. Il n'est en effet pas licite d'utiliser ces identificateurs hors de leur portée.

- **Valorisation des paramètres de la réaction**

  Les éventuels paramètres de la réaction doivent être valorisés. Il existe deux possibilités : ou bien ils sont figés lors de l'attachement, ou bien ils seront valorisés d'après le contenu de l'événement. Par conséquent, il faut vérifier que la structure du contenu de l'événement permet de déduire les paramètres non complètement spécifiés de la réaction.

Au delà de ces vérifications que nous qualifierons de minimales, certaines peuvent être imposées par des restrictions de l'utilisation des attachements. En effet, rien n'empêche d'attacher à n'importe quel événement n'importe quelle réaction, pour peu que les conditions citées plus haut soient remplies. Cependant, il peut être souhaitable de déterminer *a priori* quels sont les attachements possibles dans le système ou dans un contexte particulier, et de n'autoriser que l'abonnement de ces attachements pré–déclarés. Dans ce cas, la déclaration et l'abonnement des attachements sont nécessairement distincts.

- **Attachements libres**

  Le programmeur est libre d'attacher n'importe quelle réaction à n'importe quel événement dont il peut avoir connaissance dans un certain contexte (problèmes de portée à la fois pour la déclaration des événements et pour celle des réactions).

  Ceci ne peut se faire bien sur dans le cas des objets que dans les limites imposées par le code de leur classe. En effet, seules les opérations prévues dans la classe peuvent être appliquées à ses instances.

- **Attachements contraints**

  Il est également possible de contraindre à la déclaration les attachements autorisés, en indiquant dans le code quelles associations événement–réaction il est possible de réaliser. Cette restriction permet de sortir du code des méthodes ou des constructeurs la déclaration des attachements et de la factoriser au niveau d'une classe.

### III.4.4.2 Contexte d'abonnement et attachement

Les contrôles statiques présentés précédemment se réfèrent uniquement à la définition de l'attachement lui–même, mais pas du tout au contexte dans lequel celui–ci sera activé. Ce contexte d'abonnement est défini par un objet `O` et par un flot de contrôle `F` qui exécute sur cet objet une méthode `m`. C'est au sein de cette méthode qu'a lieu l'opération d'abonnement. Nous avons jusqu'à présent ignoré l'influence que pouvait ou devait avoir ce contexte sur les attachements. Cette influence concerne essentiellement deux aspects particuliers de l'attachement.

Le premier point où ce contexte d'abonnement peut influer est la définition du contexte de lancement de la réaction lorsque celui–ci concerne un flot de contrôle existant et identifié. Le problème qui se pose l'est en ces termes : est–il licite que le flot de contrôle défini dans le contexte de lancement soit différent de celui du contexte d'abonnement ?

Le second aspect sur lequel peut influer le contexte d'abonnement est la définition de la réaction attachée lorsqu'elle est définie par une méthode à appeler sur un objet O'. Cet objet peut-il alors être différent de l'objet O du contexte d'abonnement ?

En première analyse, la réponse à ces deux questions est affirmative. En effet, dès lors que les entités – objets ou flots de contrôle – sont manipulables au travers d'appels de méthode ou d'autres mécanismes, il semble correct d'autoriser sans restriction l'utilisation de ces mécanismes dans le cadre de notre modèle de communication événementielle. Néanmoins, nous verrons au chapitre suivant que certaines caractéristiques du modèle hôte nécessitent que ces possibilités soient limitées ou interdites.

### III.4.4.3 Contraintes dynamiques

L'utilisation des attachements induit certaines contraintes lors de l'exécution, en particulier lorsque plusieurs attachements concernant le même événement sont actifs.

**Attachements multiples**

Dans le cas où plusieurs attachements faisant intervenir le même événement peuvent être actifs en même temps, l'occurrence de cet événement entraînera l'exécution de plusieurs réactions. Rien n'interdit alors à un même effet de bord d'être reproduit plusieurs fois pour un unique événement. De même, si la réaction est définie par une méthode à appeler sur un objet, il est plausible qu'un même événement engendre l'exécution de plusieurs méthodes sur un même objet. Il semble souhaitable dans certains cas de figure d'éviter cette multiplicité.

> Considérons un objet partagé par plusieurs contextes et une méthode de cet objet qui incrémente d'une unité un compteur dans son état. Ce compteur peut représenter par exemple le nombre de requêtes d'un certain type dans les différents contextes. Si cette méthode est déclarée comme réaction à un événement et que dans les différents contextes, l'attachement correspondant a été activé, l'occurrence de l'événement doit néanmoins ne provoquer qu'un appel de méthode sur l'objet et l'incrémentation d'une seule unité.

Ce contrôle peut assurément être explicitement défini, en désactivant systématiquement, mais explicitement, les attachements antérieurs. Selon nous, de tels cas méritent un traitement implicite, associé à l'attachement qui assure l'unicité des réactions. Cette unicité de réaction pour un événement donné peut être contrôlée sur différents modes :

- une seule réaction dans tout le système,
- une seule réaction par contexte, ou
- une seule réaction avec un programme donné.

Dans ces conditions, des conflits peuvent apparaître lors d'opérations d'abonnement si l'unicité doit être assurée sur un des modèles précédents.

**Politiques de résolution des conflits**

Le problème de l'unicité de la réaction et donc de l'attachement correspondant dans les trois cas ci–dessus peut être assuré de différentes manières. La première et la plus simple consiste à réaliser lors de chaque abonnement qui engendrerait des attachements multiples un désabonnement implicite des attachements existant avant l'opération, afin que le dernier attachement soit le seul activé.

Une deuxième solution consiste à gérer pour chaque événement et dans chacun des contextes énumérés plus haut une pile d'attachements dont seul le sommet est actif. Un abonnement à l'événement et dans le contexte correspondants met un nouvel attachement au sommet de la pile. Le désabonnement d'un des éléments et en particulier du sommet supprime ledit élément de la pile.

Il est enfin difficile de contrôler statiquement le contexte de lancement d'une réaction. En effet, les contrôles qui pourraient être faits portent sur l'existence d'une tâche ou d'une activité précise qui définissent le contexte de lancement. Or ces vérifications ne sont possibles qu'à l'exécution.

### III.4.5 Invariants et éléments de choix

**Invariants**

1. La structure d'un attachement est : nom d'événement, nom de réaction, contexte de lancement. Ces éléments se divisent en deux parties : une partie statique requise seulement à la déclaration, une partie dynamique requise à l'abonnement.
2. Un attachement peut être activé, par une opération d'abonnement, ou désactivé, par une opération de désabonnement.

**Éléments de choix**

1. Gestion des droits sur les attachements.

   On peut associer des droits à des attachements, de telle sorte que seul celui qui a activé un attachement ou ceux qu'il y a autorisé puissent le désactiver.

2. Relations entre contexte d'abonnement et contexte de lancement

   Ces deux contextes peuvent être forcément identiques, ce qui implique des contraintes et un contrôle forts. L'option extrême consiste à n'assurer aucun contrôle sur ces deux contextes : l'abonnement peut être réalisé dans un contexte et la réaction dans un autre.

L'option retenue pour Guide est de n'assurer aucun gestion spécifique de droits sur les attachements et de forcer l'identité entre contexte de lancement et contexte d'abonnement.

## III.5 Modèle d'exécution

La manière dont se comporte un mécanisme de communication événementielle à l'exécution présente une grande quantité de possibilités, tant du point de vue de l'émission et de l'acheminement des événements que de l'exécution des réactions.

### III.5.1 Émission

L'émission d'un événement est une opération non bloquante. Le flot d'exécution qui émet un événement n'est pas bloqué en attente de réponses et continue après cette opération le fil normal de son exécution. C'est en partie ce qui justifie le caractère asynchrone de la communication événementielle. Cette opération porte sur un événement qui a été instancié au préalable. Son rôle est de diffuser cet événement afin que les réactions qui lui ont été attachées soient exécutées.

L'émission est caractérisée par une portée, qui définit le sous–ensemble du système dans lequel l'événement est susceptible de générer des réactions. Cette portée peut être contrôlée ou non. Dans ce dernier cas, toutes les entités du système sans restriction peuvent recevoir l'événement. Dans le premier cas, il existe plusieurs possibilités pour réaliser ce contrôle.

- La portée est exprimée lors de l'opération d'émission : l'émission est paramétrée par la définition d'une portée. Les entités réceptrices doivent être dans cette portée.
- La portée est gérée par des services d'administration qui définissent des domaines de diffusion pour les événements. Ces domaines de diffusion peuvent être déterminés à partir de la structuration des applications. Aucune mention n'en est faite au moment de l'émission.

Dans le cas de la dernière option, les difficultés concernent essentiellement la structure donnée à l'espace de ces domaines. Il est en effet possible de définir des domaines qui se recouvrent ou des hiérarchies de domaines. La diffusion des événements au sein de ces structures mérite alors des adaptations, notamment sur la diffusion entre des domaines. D'autres types de problèmes concernent la manipulation de ces structures de limitation de la portée par le programmeur.

Les solutions à ces problèmes sont multiples. ActorSpace [Callsen 94], de même que certaines extensions de Linda [Gelernter 89][Matsuoka 88] en proposent dans un contexte qui n'est pas celui de la communication événementielle . Nous nous attacherons lors de l'application à Guide à proposer une solution adaptée.

### III.5.2 Exécution des réactions

Les paramètres de l'exécution d'une réaction à l'occurrence d'un événement sont définis par l'attachement qui les lie. Cet attachement définit notamment le contexte de lancement de la réaction, qui va en grande partie déterminer son mode d'exécution. Le contexte de lancement détermine d'une part quel flot de contrôle ou quel type de flot assurera l'exécution de la réaction et dans quel contexte elle aura lieu. L'analyse menée en III.4.2 identifiait les différents cas pour la détermination du contexte de lancement d'une réaction et en particulier pour celle du flot de contrôle chargé de l'exécution. Nous distinguions alors deux cas : le flot chargé de l'exécution de la réaction est un nouveau flot créé spécialement et uniquement pour cette exécution, ou ce flot est choisi parmi les flots existant dans le système.

#### III.5.2.1 Création de flot

Le modèle d'exécution associé au premier cas consiste à créer puis à lancer un nouveau flot qui exécute la réaction spécifiée (programme ou appel de méthode sur un objet). Cette création et ce lancement ne nécessitent *a priori* aucune synchronisation avec d'autres flots dans le contexte d'exécution : l'événement asynchrone est pris en compte indépendamment des autres flots. Une éventuelle synchronisation peut être mise en place dans le code de la réaction, mais son initialisation reste indépendante.

### III.5.2.2 Modes d'exécution par un flot existant

L'utilisation d'un flot existant pose d'autres types de problèmes. En premier lieu, il convient de retrouver le flot qui assurera l'exécution. En second lieu, il faut déterminer comment ce flot existant va exécuter la réaction. Il existe pour ce dernier cas deux solutions :

a) l'exécution de la réaction est différée jusqu'à la fin du traitement qu'effectue le flot choisi au moment de l'occurrence de l'événement, ou bien

b) cette terminaison n'est pas attendue et le flot est interrompu et dérouté pour exécuter la réaction, à l'issue de laquelle il reprend éventuellement le cours initial de son exécution. Ce mode d'exécution est donc analogue à celui d'un traitant d'interruption. L'événement s'apparente alors à une interruption logicielle.

Avant de poursuivre et de détailler plus avant ce mode de réaction, la notion de *traitement* que nous utilisons mérite un éclaircissement. Elle ne se limite en effet pas à ceux spécifiés par le programme originel exécuté par le flot : si ce programme se termine, le flot disparaît et ne peut donc plus rien exécuter à la fin de son traitement. Nous étendons donc cette notion à des unités algorithmiques indépendantes que le flot peut prendre en charge dynamiquement. Ce peuvent être dans le cas qui nous intéresse le programme initial exécuté par le flot ou des réactions. Ainsi, le flot peut demeurer en veille et recevoir l'ordre d'exécuter une réaction, un nouveau traitement. À l'issue de ce traitement, le flot exécute le traitement suivant ou se remet à nouveau en veille.

Les deux modes d'exécution présentés soulèvent d'autre part le problème de la définition d'un point interruptible, où un flot peut être arrêté, dérouté puis repris. Dans le cas a), un point interruptible est explicitement défini par la fin d'un traitement, exécution d'une entité algorithmique indépendante. Dans le cas b) en revanche, le point interruptible doit être défini implicitement.

### III.5.2.3 Discussion et application

Nous avons détaillé l'utilisation d'un flot existant pour exécuter une réaction en deux possibilités : le flot choisi est quelconque et son identité n'importe pas, ou bien c'est un flot précis et bien identifié. Chacun de ces cas correspond à un type de réaction particulier (cf III.4.2).

Si nous examinons le cas d'un flot quelconque, il apparaît que le choix de ce flot parmi ceux qui peuvent exister dans l'ensemble du système ou même dans un contexte particulier n'est pas forcément simple. Il suppose notamment de définir des critères pour guider ce choix. Aussi, il nous semble adéquat de faciliter ce choix en introduisant dans le modèle des flots de contrôles dont le rôle est précisément d'exécuter les réactions de ce type. Chaque contexte pourra inclure ce type de flot.

Dans la mesure où ces flots sont dévolus à l'exécution de réactions, le mode d'exécution b) décrit en III.5.2.2 semble peu adapté dans leur cas. En effet, la demande d'exécution d'une réaction interrompt l'exécution de la précédente et diffère d'autant sa terminaison. Une exécution sérialisée des réactions selon le mode d'exécution a) paraît alors plus adéquate.

En revanche, le choix d'un flot précis pour exécuter une réaction implique un contrôle à appliquer au flot lui–même (cf III.4.2 et table de Fig. 3.2). En conséquence, cette opération ne peut pas attendre la fin du traitement et l'éventuelle disparition du flot, à un moment où le contrôle devient inutile. Il faut donc interrompre le flot et appliquer le mode d'exécution b). Ce mode d'exécution, qui réalise en définitive une interruption logicielle, implique certaines contraintes. En particulier, le problème de la terminaison de la réaction est à soulever ; une réaction qui ne se termine pas se substitue en effet de manière définitive au traitement en cours lors de l'interruption. Cette possibilité autorise en fait une « reprogrammation » du flot en cours et offre deux modes de réactions pour les flots interrompus : une modification de son environnement avec une réaction qui se termine, ou un déroutement définitif vers d'autres traitements.

L'interruption pose également le problème des interruptions en cascade : un flot est interrompu, se déroute pour exécuter la réaction, puis en cours de cette exécution est de nouveau interrompu et ainsi de suite. Il existe deux solutions à ce problème. La première consiste à fournir une opération de masquage des réactions, qui permet explicitement d'indiquer la non–interruptibilité du traitement. Dans la seconde solution, l'interruption des réactions qui modifient les conditions d'exécution de la réaction sans en changer définitivement le cours est proscrite. Il faut alors que les réactions exécutées par des flots identifiés soient classées au moment de l'attachement en deux catégories.

Il ressort de ce qui précède que les choix réalisés dans ce contexte sont en grande partie dictés par les caractéristiques du modèle hôte. Il en est ainsi pour tous les éléments du modèle d'exécution associé à la communication événementielle.

### III.5.2.4 Environnement d'exécution

L'environnement dans lequel s'exécute la réaction est également décrit dans le contexte de lancement. Comme pour les flots et comme il l'a été indiqué en III.4.2, plusieurs choix sont possibles et nous détaillons l'impact de chacun de ces choix sur le modèle d'exécution.

La création d'un nouveau contexte pour accueillir le flot qui exécutera la réaction se fait, comme la création de ce flot, de manière asynchrone. Le principal problème réside dans la détermination du propriétaire du contexte ainsi créé. En effet,

l'identité de ce propriétaire doit pour des raisons de sécurité être la même que celle de l'utilisateur qui a activé l'attachement honoré par la réaction en cours.

De même que pour les flots de contrôle, la détermination d'un contexte quelconque pour l'exécution d'une réaction suppose la définition de critères de choix. Aussi, nous adoptons pour les contextes la même politique que pour les flots, à savoir d'introduire dans le modèle un contexte particulier dans lequel seront effectuées toutes les réactions ne nécessitant pas un contexte précis.

Enfin, l'utilisation d'un contexte bien déterminé suppose qu'il contienne bien le flot d'exécution chargé de la réaction. Cette propriété est assurée *de facto* lorsque le flot est bien identifié. L'introduction de flots chargés de l'exécution des réactions dans chaque contexte résout le problème des flots quelconques. En revanche, la création de flot est plus délicate. Il convient en effet que ce soit au sein du contexte que soit pris la décision de créer et de lancer ce flot. Le contexte devient donc un délégué du système pour l'exécution des réactions.

### III.5.3 Invariants et éléments de choix

**Invariants**

1. L'émission est une opération asynchrone.
2. La réception équivaut à l'exécution d'une réaction. Elle ne nécessite aucune opération explicite au moment de l'établissement de la communication[(2)].
3. Le contexte d'exécution d'une réaction est défini par un flot de contrôle et un environnement d'exécution pour ce flot.

( 2 ) Il est possible cependant de réaliser une attente sur la réception d'un événement. Le principe consiste à s'abonner à l'événement attendu puis à se bloquer immédiatement. La réaction associée doit alors débloquer le flot. L'utilisation de sémaphores permettrait par exemple ce type de fonctionnement.

**Éléments de choix**

1. L'émission peut être explicite ou implicite.
2. Un filtre (description d'un ensemble de destinataires) peut être appliqué lors d'une émission.

   Cette possibilité permet d'enrichir la partie « Condition » pour le déclenchement des réactions.
3. Le choix parmi les modes d'exécution des réactions peut être réduit.

   En particulier, l'interruption des flots de contrôle peut être problématique.

Les choix que nous privilégions pour Guide sont ceux d'une émission explicite, de la possibilité de spécifier un ensemble de destinataires et d'offrir le maximum de possibilités au niveau des modes d'exécution.

## III.6 Intégration dans un modèle à objets

Nous avons présenté dans les sections précédentes des éléments de description générique d'un modèle de communication événementielle. L'intégration d'un tel modèle dans une structure existante requiert une étude supplémentaire. Cette étude doit prendre en compte les caractéristiques canoniques des modèles à objets [Nierstrasz 89][Wegner 90]. C'est pourquoi nous examinons dans cette section les conséquences que pourraient avoir la communication événementielle sur ces caractéristiques, qui sont l'encapsulation, l'héritage et la notion d'interface.

Comme pour toutes les autres question soulevées dans ce chapitre, le cadre général où nous nous sommes placés ne permet pas de donner des réponses exhaustives. Néanmoins, nous énumérons les différents problèmes soulevés et y proposons des solutions générales.

### III.6.1 Encapsulation

L'encapsulation dans un langage à objets vise à n'autoriser des modifications des données internes d'un objet par un utilisateur quelconque qu'au travers des méthodes fournies dans l'interface de cet objet. Ce principe vise à interdire la manipulation directe et éventuellement incorrecte de l'état des objets.

Cette propriété des langages à objets est un argument supplémentaire en faveur de la limitation des types de réactions aux seules méthodes et programmes exécutables. En effet, l'utilisation comme réactions de procédures, qui n'appartiennent pas à l'interface des objets, ou de séquences quelconques d'instructions déclarées dans les classes, exécutées sur des objets, ne respecte pas ce principe : il est possible de commander depuis « l'extérieur » d'un objet des actions qui ne font pas partie de son interface.

Si les types de réactions sont limités aux méthodes et aux programmes exécutables, l'encapsulation des données sera respectée. La notion de méthode est en effet un des moyens à l'aide desquels l'encapsulation est réalisée. De plus, un programme exécutable qui accède à des objets le fait nécessairement par des appels de méthodes et ne peut donc pas violer le principe d'encapsulation.

### III.6.2 Notion d'interface

L'interface définit dans un modèle à objets l'ensemble des propriétés et opérations visibles pour un objet. Elle définit donc la vue externe de l'objet et la manière dont il peut être utilisé. C'est d'autre part la seule source d'information sur cet objet. La question se pose alors de savoir si l'utilisation au sein d'un objet de la communication événementielle implique que son interface comporte des informations sur son utilisation, et, dans le cas d'une réponse positive, quels seraient les éléments à y faire figurer ? À notre sens, et à la lumière de ce que nous avons présenté dans les sections précédentes, ces éléments pourraient être les suivants (selon les choix de conception) :

1. *familles d'événements définies dans la classe*

   Nous avons mentionné en III.2.2.1 que les familles d'événements définies dans une classe pouvaient faire partie de son interface afin que l'extérieur en ait connaissance.

2. *familles des événements susceptibles d'être émis au sein des instances de la classe*

   Ce sont des informations sur le comportement des instances de la classe, qui peuvent être exploitées à des fins de réutilisation.

3. *familles des événements qui peuvent être « reçus » par les instances de la classe*

   Cette propriété est plus particulièrement adaptée au cas où les attachements sont contraints par des déclarations faites au niveau de la classe et dont les réactions portent sur l'instance au sein duquel l'attachement est activé. Dans ces conditions, l'émission des

événements correspondants peut engendrer des traitements sur cet objet. Ces événements sont alors des moyens de manipuler les instances de cette classe et à ce titre peuvent apparaître dans son interface.

L'interface possède certes un rôle informatif, mais elle sert également de base au contrôle de la conformité. La question est donc de savoir dans quelle mesure les éléments introduits plus haut peuvent et doivent affecter une relation de conformité entre deux interfaces. Cet aspect peut également présenter un intérêt dans le cadre de l'utilisation des événements dans les connecteurs. Un connecteur de communication événementielle lie deux composants dont les interfaces doivent être conformes à la spécification du connecteur.

Un autre aspect de l'interface réside dans sa réalisation. Cette réalisation doit effectivement correspondre aux spécifications de l'interface. Un des éléments cités plus haut peut à cet égard créer un dilemme. En effet, si les événements qui peuvent être émis depuis les instances d'une classe apparaissent dans l'interface, faut–il assurer que des instructions d'émission sont bien présentes dans le code des méthodes ? Une telle vérification octroierait à l'interface la force d'un contrat : tout ce qui est dans l'interface est effectivement réalisé.

Notons cependant que les termes du contrat concernant les événements sont différents de ceux qui concernent les méthodes par exemple. Dans le cas d'une méthode, il est assuré par contrat que l'appel de cette méthode sur une instance de la classe est réalisé et donc que cette méthode est un service effectivement fourni par l'interface. Le cas des événements émis (point 2. de la liste ci–dessus) est différent puisqu'on ne peut pas garantir que ces événements seront effectivement émis. La seule garantie porte sur la possibilité de cette émission. Pour les événements reçus (point 3. de la liste ci–dessus), le contrat stipule qu'une instance des familles d'événements présentes dans l'interface peut engendrer un traitement sur l'objet, mais que ce traitement est subordonné à l'existence d'un attachement activé.

### III.6.3 Héritage

L'héritage est un mécanisme qui permet de définir une classe à partir d'une ou plusieurs autres en reprenant leur description et en l'enrichissant. Une description de classe se compose de divers éléments qui sont classiquement des méthodes, présentes dans l'interface, et des procédures, invisibles de l'extérieur, des variables d'état, visibles ou cachées. L'introduction des événements dans un modèle à objets n'a donc un impact sur l'héritage que si de nouveaux éléments de description sont ajoutés aux classes.

La question qui se pose alors est la suivante : si ces éléments figurent dans la définition de la classe, leur prise en compte dans le mécanisme d'héritage pose–t–elle des problèmes insolubles ? Ces problèmes peuvent être de deux natures :

- la définition dans une (super) classe des éléments dont hérite une autre classe peut être invalide dans le contexte de cette dernière. Cette définition peut par exemple faire référence à un élément qui n'est pas hérité.
- les noms utilisés dans une ou plusieurs (super) classes sont en conflit avec des noms d'une classe qui en hérite.

Ces deux types de problèmes ont reçu des solutions dans le cadre des modèles à objets classiques. Si nous examinons l'analyse des modèles d'événements présentée dans les sections précédentes, les nouveaux éléments peuvent exister et concerner les aspects suivants :

- **Familles d'événements définies dans la classe** (cf III.2.2.1)

  Ces éléments peuvent être utilisés pour déterminer la portée de définition d'une famille et apparaître dans l'interface. S'ils sont hérités, il est équivalent de dire que la famille est également définie dans la sous–classe, et peut alors y être utilisée.

  Les éventuels conflits de nom peuvent être arbitrés simplement en privilégiant toujours la famille déclarée dans la sous–classe. Si les familles sont intégrées au contrôle de la conformité entre classes et qu'elles apparaissent dans l'interface de celles–ci, la surcharge d'une famille doit être conforme, comme l'est la surcharge des méthodes.

- **Familles des événements qui peuvent être émis depuis une instance de la classe** (cf III.6.2)

  Ces informations font partie de l'interface de la classe. Si l'interface représente un contrat qui assure que des événements de ces familles peuvent être émis depuis des instances de la classe, l'héritage peut être problématique. En effet, une surcharge des méthodes où ses émissions avaient lieu dans la super–classe peut rendre obsolète le contrat ; il suffit que les émissions n'aient plus lieu dans les méthodes surchargées.

- **Description des attachements qui sont licites pour une instance de la classe** (cas des attachements contraints, cf III.4.4.1)

  La description de ces attachements est toujours licite dans la sous–classe, sauf si ces attachements spécifient des méthodes qui

ont été surchargées, avec modification de signature, dans la nouvelle classe. Il faut alors redéfinir de tels attachements.

Le cas de l'héritage multiple est tributaire du traitement appliqué aux méthodes. En effet, les éventuels conflits de noms qui peuvent apparaître entre deux familles héritées peuvent recevoir le même traitement que celui des méthodes. Pour le cas des attachements qui mettent en jeu des méthodes héritées, les solutions adoptées pour résoudre d'éventuels conflits – renommage, choix explicite de la méthode conservée dans la sous–classe – équivalent à des surcharges et nécessitent *a priori* la redéfinition des attachements concernés.

## III.7 Synthèse

Nous avons présenté dans ce chapitre les différentes composantes nécessaires à la conception d'un mécanisme de communication événementielle, ainsi que les différentes solutions qui peuvent être adoptées pour chacune de ces composantes. Cette section constitue une brève synthèse de ces éléments, dont l'ambition est de fournir une vision globale du problème et des choix offerts aux concepteurs pour intégrer la communication événementielle dans un modèle à objets hôte.

### III.7.1 Invariants de la communication événementielle

La famille de modèles que nous avons décrite est soumise à des invariants, qui sont vérifiés quel que soit le modèle instancié. Ces éléments intangibles étendent les principes énoncés en III.1.1 sur la base de ce que nous avons mis en évidence dans ce chapitre. Ils sont les suivants :

- **Événements**
  1. Un événement est instance d'une famille.
  2. Un événement est identifié par un nom, contenant au minimum le nom de sa famille.
  3. Un événement a des paramètres dont les types sont définis par sa famille.
- **Réactions**
  1. Une réaction est un programme avec un point d'entrée. La méthode est le type de réaction naturel.
- **Attachement**
  1. La structure d'un attachement est : nom d'événement, filtre sur l'événement, nom de réaction, contexte de lancement. Ces éléments se divisent en deux parties : une partie statique requise

seulement à la déclaration, une partie dynamique requise à l'abonnement.

2. Un attachement peut être activé ou désactivé

- **Modèle d'exécution**
  1. L'émission est une opération asynchrone.
  2. Le contexte d'exécution d'une réaction est défini par un flot de contrôle et un environnement d'exécution pour ce flot. La définition de ce contexte est dépendante du support exécutif du modèle hôte.

### III.7.2 L'attachement : notion centrale

Comme il a été mentionné en III.4, l'attachement est la notion centrale de la famille de modèles que nous considérons. D'une part, dans sa définition et son utilisation se retrouvent les différents éléments décrits dans ce chapitre Ainsi, la définition d'un attachement comprend notamment :

1. un nom d'événement,
2. d'éventuels filtres sur l'événement, son contenu est ses destinataires,
3. une réaction à exécuter,
4. un contexte de lancement pour la réaction, et
5. un mode d'exécution pour la réaction.

D'autre part, le type d'utilisation du mécanisme est directement traduit dans la structure et les informations portées dans l'abonnement. En conséquence, la forme d'un attachement et les opérations qu'il sous–tend conditionnent les choix de conception relatifs aux différentes composantes du modèle (cf Fig. 3.4).

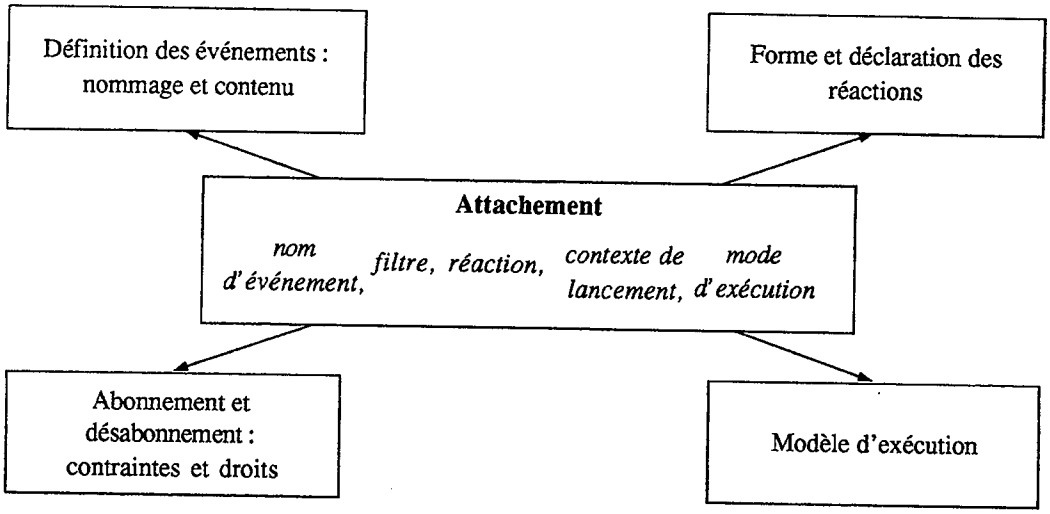


*Fig. 3.4 : Attachement, notion centrale*

### III.7.3 Structures d'exécution

La description des structures d'exécution d'un mécanisme de communication événementielle est d'abord celle du modèle d'exécution, et ensuite celle des programmes exécutés – les réactions.

Le modèle d'exécution est d'ailleurs un point crucial dans la définition du mécanisme, puisque ses possibilités sont contraintes par celles du modèle hôte ou de son support d'exécution. C'est d'autre part lui qui détermine une grande part de la puissance du mécanisme.

La présence du modèle d'exécution dans un attachement est marquée par le contexte de lancement et le mode d'exécution de la réaction. L'intégralité des possibilités que peut offrir ce modèle est résumée par la table Fig. 3.2 et nous avons détaillé les mécanismes correspondant en III.5.2. Cependant, ces possibilités sont contraintes par celles du modèle hôte, tout spécialement en ce qui concerne la manipulation des flots et des contextes qui n'est pas nécessairement possible.

### III.7.4 Le problème de la déclaration

À de fréquentes occasions dans les sections qui ont précédé, nous avons posé le problème de la déclaration des éléments tels que événement, famille, réaction ou attachement. Dans ces différents cas, nous avons en règle générale identifié deux possibilités :

- **Introduction de nouvelles constructions du même niveau que les classes ;**

  L'élément à intégrer dans le modèle hôte est déclaré dans une structure spéciale indépendante des classes. Les avantages de cette approche résident dans une plus grande liberté dans la déclaration des structures hors du cadre contraignant de la classe, une homogénéité plus grande dans le traitement des concepts

- **Déclaration au sein des classes.**

  La classe reste la principale structure de déclaration, sa structure est enrichie par de nouvelles constructions. Les avantages de cette approche résident dans la lisibilité conservée du modèle – la classe demeure la seule structure de déclaration – et dans sa forte intégration avec les objets, instances des classes.

# Chapitre IV

# Application : Extension du modèle Guide

Nous avons présenté au chapitre précédent différents choix que le concepteur d'un mécanisme de communication événementielle pouvait faire. Nous allons dans ce chapitre effectuer ces choix et instancier un modèle d'événements adapté au modèle à objets Guide. Après une brève présentation de ce modèle à objets répartis et persistants, nous détaillerons dans ce contexte les critères pris en compte pour intégrer les événements à Guide. Ces critères induisent les choix que nous avons faits pour la définition en Guide des éléments introduits au chapitre précédent : événements, réactions, attachement et modèle d'exécution. D'autre part, ce cadre nous permettra de donner des exemples de solution à certains points, notamment les problèmes de portée, qui n'ont pu être que partiellement abordés au chapitre III.

## IV.1 Présentation du modèle Guide

Le modèle Guide définit un cadre pour la description d'applications réparties. À cette fin, il offre à la fois un modèle de données, construit sur la notion d'objet, et un modèle d'exécution, qui décrit les structures d'exécution au sein desquelles évoluent les objets. Ce modèle a servi de base à la réalisation de prototypes. Ces prototypes offraient à la fois un langage qui permettait de manipuler le modèle de données et une plate–forme d'exécution qui réalisait les structures du modèle d'exécution [Balter 91] [Balter 94]. Cette plate–forme mettait en jeu un ensemble de stations de travail interconnectées par un réseau local.

### IV.1.1 Modèle d'objets

Dans le modèle Guide, le paradigme objet constitue la base de la représentation et de la manipulation des données. Ainsi, les éléments canoniques associés classiquement à ce paradigme – classe, héritage, encapsulation, interface – sont parties intégrantes du modèle Guide, de même que la notion de type qu'induit celle d'interface. Le domaine visé nécessitait l'enrichissement de ces éléments de base par

des caractéristiques moins courantes. Ainsi, les objets sont répartis, partageables et persistants.

#### IV.1.1.1 Classes et objets

Les objets Guide sont des instances d'un modèle appelé **classe**. La classe définit d'une part les structures de données internes de l'objet (son *état*) et d'autre part les fonctions qui permettent d'accéder à cet état (les *méthodes*). En vertu du principe d'encapsulation, seule l'invocation de méthode permet de manipuler l'état d'un objet.

L'ensemble des opérations licites sur les instances d'une classe est défini dans une interface dénommée **type**. Ces types constituent le fondement d'un contrôle statique de la conformité dans les programmes. La conformité respecte la règle de la contravariance [Cardelli 85], qui assure une exécution correcte des programmes vérifiés statiquement.

L'état des objets comporte des champs dont les types peuvent être divers : types de base, chaînes de caractères ou références vers d'autres objets.

Guide offre un mécanisme d'héritage simple qui permet de spécialiser la description des classes. Cet héritage est conforme, c'est–à–dire que le type correspondant à une sous–classe est conforme à celui de sa super–classe. Ceci impose notamment des règles strictes sur la surcharge des méthodes (principe de la contravariance cité plus haut). Le contrôle de la conformité entre type est réalisé à la compilation.

Enfin, à chaque objet peuvent être associés des droits d'accès qui régissent l'utilisation que tel ou tel utilisateur peut en faire. Ces droits sont exprimés à l'aide de listes d'accès associées à chaque méthode de l'objet et qui définissent des vues. Chaque vue détermine un ensemble d'opérations autorisées.

#### IV.1.1.2 Persistance, répartition, partage et synchronisation

Les objets Guide sont persistants, c'est–à–dire que leur durée de vie est supérieure à celle du flot d'exécution qui les a créés. La conservation des objets persistants est assurée par un service de stockage, dénommé Mémoire Permanente d'Objets (MPO).

Les objets sont également partageables, c'est–à–dire que plusieurs invocations de méthodes peuvent être réalisées concurremment sur un même objet.

Chaque objet possède un identificateur unique, appelé **référence**. Cet identificateur est invisible au niveau utilisateur, typé, global à tout le système et indépendant de la localisation de l'objet. C'est par son intermédiaire que l'objet peut être retrouvé et manipulé par n'importe quel flot d'exécution pendant toute la durée de sa

vie et dans l'ensemble du système. C'est donc également cet identificateur qui rend possible les accès concurrents aux objets.

Les objets sont toujours localisés sur un seul site. Ils peuvent migrer d'un site à un autre mais ils ne sont ni fragmentés, ni dupliqués. Le partage est donc un vrai partage concurrent. Cette propriété impose que soit fourni un mécanisme de gestion de la cohérence des objets manipulés concurremment.

Cette synchronisation des accès est exprimée au niveau de la classe par des conditions d'activation qui forment une clause de contrôle pour la classe. Chaque condition d'activation est associée à une méthode et doit être satisfaite avant que la méthode puisse être exécutée. L'absence de condition d'activation pour une méthode signifie qu'aucune contrainte n'est imposée sur son exécution. Les conditions d'activation mettent en jeu les variables d'état de l'objet, les paramètres effectifs de la méthode et des compteurs de synchronisation. Ces compteurs caractérisent pour chaque méthode de chaque objet le nombre d'appels de la méthode effectués, terminés, en cours, en attente, etc. Ces compteurs permettent notamment de mettre en œuvre des politiques de synchronisation simples, comme l'exclusion mutuelle entre méthodes ou un protocole de producteurs consommateurs.

Si nous considérons par exemple une classe Buffer qui dispose de deux méthodes Get et Put qui doivent mettre en œuvre un protocole de lecteur-rédacteur, elle peut se déclarer de la manière suivante :

```
CLASS Buffer
    ...
    METHOD Get(...);
    METHOD Put(...);
    ...
    CONTROL // bloc de déclaration des conditions
            // d'activation
        Get : current(Put) = 0;
        Put : current(Put) = 0 AND current(Get) = 0;
END Buffer.
```

La clause **CONTROL** indique que la méthode Get ne s'exécute pas quand la méthode Put s'exécute et que la méthode Put ne s'exécute pas lorsque Get ou Put sont en cours d'exécution. Le langage offre la possibilité d'écrire cette clause sous la forme plus synthétique suivante :

```
CONTROL
    READER(Get),WRITER(Put)
```

### IV.1.2 Modèle d'exécution

Le modèle d'exécution de Guide est celui d'Unix étendu à un environnement réparti.

### IV.1.2.1 Tâches et Activités

La notion de processus est représentée par ce que nous appelons des **tâches**. Une tâche s'étend sur plusieurs sites et correspond à l'exécution d'une application. Au sein de cette tâche, plusieurs flots de contrôle, appelés **activités**, s'exécutent en parallèle.

L'exécution d'une activité consiste en une succession d'invocations de méthode sur des objets. Ces invocations peuvent être réalisées sur n'importe quel site du système. À chaque invocation, l'objet correspondant est localisé, chargé et lié dynamiquement dans la mémoire virtuelle de la tâche. L'ensemble de tous les objets liés dans une tâche est appelé son **contexte**.

Comme les objets sont partageables, ils peuvent être liés dans plusieurs tâches. Les objets partagés permettent donc de réaliser une mémoire virtuelle répartie et partagée, qui constitue le seul médium de communication entre activités.

Cette mémoire virtuelle répartie, ou Mémoire Virtuelle d'Objets (MVO), est en fait composée de l'union des contextes de toutes les tâches présentes dans le système. C'est également l'union des mémoires virtuelles des différents sites du système.

### IV.1.2.2 Invocation de méthode

L'opération de base dans Guide est l'**invocation** de méthode sur un objet. Une invocation précise la référence de l'objet appelé, le nom de la méthode et ses paramètres effectifs.

Si l'objet n'est pas chargé dans la mémoire virtuelle du site d'appel, un défaut d'objet est décelé. L'objet fautif est alors retrouvé grâce à sa référence dans la MPO. La MPO détecte alors deux cas : soit l'objet est déjà chargé en MVO, mais sur un autre site, soit il ne l'est pas.

Dans le second cas, l'objet est dans la majorité des cas chargé dans la mémoire virtuelle du site d'appel. Certains objets en revanche sont toujours chargés sur leur site de création. Le site de chargement de l'objet est choisi pour recevoir l'exécution de la méthode, qui est invoquée sur ce site comme un appel de procédure. Toutes les phases de ce mécanisme sont exécutées dynamiquement à chaque invocation.

L'invocation de méthode est une opération synchrone, c'est–à–dire que l'activité appelante est bloquée tant que l'exécution de la méthode n'est pas achevée.

### IV.1.3 Mécanisme d'exceptions

Le modèle Guide propose également un mécanisme d'exceptions [Lacourte 91a] que nous présentons plus précisément car il présente certaines similitudes avec la communication événementielle. Nous distinguons dans notre

présentation les aspects déclaratifs des exceptions et les éléments correspondants du modèle d'exécution.

Le mécanisme d'exceptions permet d'associer à des exécutions incorrectes de méthodes (traduites par des levées d'exception) des traitants de reprise. Ces traitants permettent de réaliser un traitement systématique et propre des erreurs d'exécution.

### IV.1.3.1 Expression

Les déclarations associées au mécanisme d'exceptions dans Guide apparaissent au niveau de la définition des méthodes. La définition de la signature d'une méthode comprend les exceptions qu'elle est susceptible de lever.

Ainsi dans l'exemple suivant, la déclaration de la méthode `LitCaract` d'une classe `Page` et qui retourne un caractère est susceptible de lever les exceptions `erreur`, `fin_de_ligne` et `fin_de_page`. On voit que l'exception est identifiée par un simple nom.

```
type Page is
    method LitCaract: Char;
        signals erreur, fin_de_ligne, fin_de_page;
```

De même, les traitants des exceptions sont déclarés au niveau du code des méthodes. À ce niveau sont associés les exceptions qui peuvent être levées lors de l'exécution de la méthode (y compris dans les méthodes invoquées depuis cette méthode) et le traitant d'exception. Ce traitant est une séquence de code quelconque.

### IV.1.3.2 Modèle d'exécution

Lorsqu'une exception est levée durant l'exécution d'une méthode, c'est l'activité courante, celle par laquelle l'exception a été levée, qui est chargée du traitement. Les traitants associés à l'exception sont retrouvés et exécutés de manière synchrone par cette activité.

## IV.2 Choix pour un modèle d'événements dans Guide

Notre proposition d'intégration de la communication événementielle dans le modèle Guide obéit à différents critères. Ces critères subordonnent en partie les choix que nous avons faits pour instancier un mécanisme d'événements à partir des éléments présentés au chapitre précédent.

### IV.2.1 Critères et contraintes

L'extension du modèle que nous venons de présenter avec un mode de communication à base d'événements obéit à différents critères et contraintes que nous nous fixons et qui conditionnent les choix conceptuels que nous ferons.

#### IV.2.1.1 Critères de choix

Nous retenons quatre critères de choix principaux pour l'intégration dans Guide de la communication événementielle.

**Critère 1**

Le premier critère que nous retenons est celui de la **fidélité au modèle initial** tel que nous l'avons présenté en IV.1. En effet, nous ne souhaitons ni modifier profondément les habitudes des programmeurs, ni payer le coût d'une réorganisation des différents concepts et entités manipulés. Cette exigence vaut aussi bien pour les aspects déclaratifs et le modèle d'objets que pour le modèle d'exécution.

**Critère 2**

Le second critère qui conditionne nos choix est l'utilisation que nous souhaitons faire du modèle étendu. Il s'agit d'assurer un support pour la mise en place de services de coordination pour la programmation d'applications coopératives. Dès lors, il convient de fournir à ces services la **gamme de fonctions** la plus large et la plus complète possible, dans les limites que fixe le premier critère énoncé.

**Critère 3**

Un troisième critère vient tempérer le précédent. Nous souhaitons que le **contrôle** sur l'exécution, que ce soit dans les tâches ou dans les objets, ne soit pas délégué. En d'autres termes, le comportement d'un objet vis–à–vis des événements est déterminé dans cet objet, ou dans sa classe et l'exécution de réactions au sein d'une tâche est commandé et contrôlé par celle–ci. L'objectif de ce critère n'est pas de limiter l'indéterminisme inhérent à l'accès concurrent aux objets par des activités parallèles, mais de le contrôler malgré tout, comme l'accès concurrent est contrôlé par des clauses de synchronisation.

**Critère 4**

Un dernier critère est également pris en compte qui est celui de la **simplicité**, en particulier du point de vue de l'utilisation du mécanisme par un programmeur. Cette simplicité devra se traduire par l'absence de structures déclaratives supplémentaires trop nombreuses et trop complexes.

### IV.2.1.2 Contraintes induites

Ces critères induisent un certain nombre de contraintes sur les choix que nous allons faire. Ces contraintes sont les suivantes :

- **La classe, seule entité de déclaration**

  La classe est en Guide la seule entité de déclaration utilisée pour décrire le comportement des objets, les contraintes de synchronisation ou la gestion des exceptions. Dès lors, nous adopterons également la classe comme seule structure où pourront être déclarés et utilisés les différents éléments de la communication événementielle.

- **Les unités d'exécution ne sont pas manipulables**

  En Guide, les unités d'exécution, en particulier les activités, peuvent certes être créées ou détruites, mais elles ne peuvent pas être désignées et leur état, leur environnement ou leur conditions d'exécution ne peuvent pas être modifiés. En particulier, l'interruption d'une activité puis la reprise au point d'arrêt sont impossibles dans la réalisation actuelle. Nous conservons en vertu du premier critère cette caractéristique. Le corollaire de cette contrainte est que nous écartons des modes d'exécution licites pour les réactions celui où elles doivent être exécutées par des activités bien identifiées et interruptibles.

- **Utilisation de toutes les possibilités du modèle**

  Le second critère de choix impose que toutes les fonctions qui peuvent être construites en utilisant le modèle Guide doivent l'être et qu'il ne faut pas présumer de l'utilisation qui en sera faite. En particulier, le modèle d'exécution très riche de Guide doit être exploité.

- **Restriction sur les contextes de lancement**

  Le critère de contrôle implique des limitations sur les contextes de lancement des réactions. En effet, ce critère interdit qu'un objet soit « abonné » par un autre ou que l'exécution d'une réaction dans une tâche soit demandée depuis une autre tâche. En conséquence, le contexte d'abonnement (objet et tâche où est exécutée l'opération d'abonnement) limite fortement le contexte d'exécution des réactions associées.

### IV.2.2 Choix

Nous présentons maintenant les principaux choix effectués en fonctions des critères et contraintes que nous nous sommes fixés. Ces choix concernent la nature des événements, la définition et le contrôle des attachements et des opérations associées, et le contexte d'exécution des réactions. Nous présentons dans une section ultérieure un choix de conception pour la gestion de la portée.

#### IV.2.2.1 Événements fugaces et non manipulables.

**Choix**

Les événements que nous intégrons dans le modèle Guide sont fugaces et non manipulables. Ce ne sont donc pas des objets au sens de Guide : ils ne sont pas persistants et ne possèdent pas un état qui peut être manipulé explicitement dans le code des classes.

**Motivation**

Nous avons expliqué au chapitre précédent pourquoi les événements n'étaient pas selon nous à considérer comme des objets. Nous pensons ensuite que la persistance des événements est une propriété peu utilisée et qu'elle peut par ailleurs être réalisée en utilisant les possibilités des objets Guide, grâce auxquels la notion d'historique d'événements peut être fournie au coup par coup selon les besoins.

Le choix d'une durée de vie nulle pour les événements évite d'une part de faire ce choix au moment de leur déclaration, ce qui la surchargerait, et d'autre part d'avoir à gérer au niveau du modèle des politiques multiples de destruction des événements.

#### IV.2.2.2 Familles et événements

**Choix**

Les familles d'événements sont déclarées implicitement : la déclaration d'un événement équivaut à celle de sa famille. Un événement est identifié par le nom de sa famille, qui est en fait sa signature (nom et types des paramètres).

**Motivation**

La déclaration implicite des familles est motivée par la simplicité. En effet, nous ne souhaitons pas surcharger la classe avec la déclaration de structures dont les instances sont de plus non manipulables (cf IV.2.2.1). Ce même critère de simplicité intervient dans le choix de désignation fait pour les événements. De plus, nous privilégions ainsi le type du changement dénoté

par l'événement par rapport à l'occurrence particulière de ce changement. Comme le déclenchement d'une réaction par un événement est avant tout subordonné à son identificateur, il nous semble judicieux de limiter le contenu de cet identificateur à l'identification du type du changement pour ne pas le surcharger d'éléments de discrimination la plupart du temps inutiles. Une éventuelle discrimination peut être réalisée par des filtres sur les paramètres de l'événement.

#### IV.2.2.3 Attachements : contraintes

**Choix**

Les attachements qu'il est possible de réaliser au sein d'un objet instance d'une classe sont ceux déclarés au sein de cette classe, et seulement ceux–là. De plus, les réactions sont soit des programmes exécutables, soit des méthodes (fonctions publiques) qui seront appelées sur l'objet courant. On ne peut pas abonner un objet « contre son gré ».

**Motivation**

Ce choix relève du critère 3 et permet un contrôle renforcé du comportement des objets, que l'on peut faire évoluer, mais dans un cadre bien connu et limité par les attachements licites déclarés statiquement. Le code défini au sein de la classe de l'objet décrit ainsi complètement son comportement possible. On peut alors dire qu'un objet s'abonne lorsqu'une activité active un attachement en son sein.

#### IV.2.2.4 Abonnement et désabonnement

**Choix**

L'activation et la désactivation sont des opérations dynamiques et explicites. Seul le cas de la destruction d'une entité entraîne la désactivation de tous les attachements qui en dépendent.

**Motivation**

Nous avons indiqué qu'une des motivations de notre proposition pour Guide était de permettre l'évolution du comportement d'une application au cours du temps. C'est pourquoi nous souhaitons assurer un caractère dynamique aux opérations qui peuvent faire évoluer ce comportement. Ensuite, le critère de simplicité nous incite à ne pas multiplier les modes d'abonnement et de désabonnement.

### IV.2.2.5 Contexte d'exécution des réactions

**Choix**

Le contexte d'exécution d'une réaction est déterminé par une tâche et une activité dans cette tâche. La tâche correspond à l'environnement d'exécution et l'activité au flot d'exécution chargé de la réaction tels qu'ils ont été introduits en III.5.2.

La tâche est alors soit une tâche quelconque non identifiée – qui peut être nouvelle –, soit la tâche dans laquelle l'attachement qui est à l'origine de son déclenchement a été activé (cf IV.2.1.2). De plus, dans le cas d'une tâche identifiée, l'activité est soit une activité quelconque, soit une nouvelle activité. Ce choix détermine quatre modes d'exécution pour les réactions.

**Motivation**

Ce choix est une conséquence des critères 1 et 3. En effet, comme les activités ne sont pas manipulables, on ne peut pas modifier leurs conditions d'exécution pour leur faire exécuter des réactions. Cette restriction implique l'abandon des modes impliquant pour l'exécution des réactions des flots de contrôle (ici des activités) identifiés (cf III.5.2). Ensuite, le renforcement du contrôle (critère 3) justifie la restriction des contextes licites.

## IV.3 Structures de déclaration

Nous décrivons dans cette section les structures de déclaration mises en place et conformes aux choix que nous avons effectués et présentés en IV.2. Ces structures permettent la description des événements (et de leur famille) et des attachements. Comme il l'a été mentionné plusieurs fois, c'est au sein de la classe que sont utilisés les différents éléments que nous présentons.

### IV.3.1 Familles et événements

Rappelons les choix que nous avons faits pour les événements et leurs familles :

- les événements sont fugaces et non manipulables,
- les événements sont identifiés par leur nom de famille (une signature) et
- les familles d'événements sont déclarées implicitement.

En vertu de ce dernier choix, toute famille d'événements est implicitement définie par l'utilisation de l'un de ses membres, soit pour une émission, soit comme déclencheur d'une réaction.

### IV.3.1.1 Déclaration implicite des familles

La principale raison du choix d'une déclaration implicite est le souci de ne pas surcharger le modèle original avec des constructions nouvelles qui en accroissent la complexité. Ainsi, une famille peut être définie à deux occasions : soit à l'émission d'une de ses instances, soit à la déclaration d'un attachement qui la met en jeu. Nous verrons lorsque nous aborderons ces deux aspects la forme donnée à la déclaration.

Les éléments qui définissent une famille d'événements sont :

- son nom : il s'agit d'une chaîne de caractères semblable à un identificateur. Lorsque nous parlerons dans la suite de nom d'un événement, nous nous référerons à ce nom.
- le type de ses paramètres : la structure des événements instances d'une famille est décrite par des paramètres typés. L'ordre de ces paramètres dans la déclaration est pertinent. Les types de paramètres licites ne sont pas limités dans le modèle. Ce peuvent être des types de base, des chaînes de caractères ou des types construits (types d'objets compris).

L'identificateur d'une famille est constitué d'une chaîne de caractères qui contient le nom et un codage des types des paramètres. Par conséquent, la donnée de cet identificateur suffit pour définir la famille. Dans la suite de ce chapitre, nous prendrons pour convention qu'une famille de nom `familyName`, dont les types des N paramètres ont pour noms `<typeName1, ..., typeNameN>` possède un identificateur de la forme :

```
familyName_typeName1_..._typeNameN
```

Réciproquement, la donnée de cet identificateur définit la famille en question.

Cet identificateur est global et unique. Sa portée n'est *a priori* pas limitée à la classe où elle est utilisée. Ceci implique par conséquent que deux familles de même identificateur définies dans deux classes différentes sont confondues. Des ambiguïtés entre familles définies par deux programmeurs différents peuvent en résulter, mais nous pensons que la définition d'un **espace d'événements plat et global** est suffisante pour nos besoins. Cette propriété rend très utile la présence dans son interface des familles des événements émis dans une classe : cette information permet en effet de connaître d'une part les événements susceptibles d'être émis, mais également leur provenance.

### IV.3.1.2 Événements

Les événements ne sont pas des entités manipulables et ont de plus un caractère fugace. Par conséquent, leur déclaration peut coïncider avec et se limiter à leur utilisation, c'est-à-dire leur émission.

Cette émission coïncide également avec la définition de leur famille. En conséquence, la déclaration de l'émission d'un événement doit faire apparaître l'identificateur de la famille et les paramètres effectivement insérés dans la structure familiale.

Considérons l'événement associé à la mise-à-jour d'un document partagé. Cet événement est caractérisé par son type, l'identité du coauteur qui a effectué la modification, la référence du document modifié et le contexte de la modification - quel élément a été modifié. Dans ces conditions, lors de son émission, cet événement est déclaré comme suit :

```
Send modifyDoc(
        OBJ myIdentity,
        OBJ document,
        OBJ modifCtxt)
```

Cette commande indique qu'un événement va être émis, que son nom de famille est modifyDocEvt et que ses paramètres formels sont trois références sur des objets qui représentent dans l'ordre l'auteur de la modification, le document modifié et l'élément modifié dans ce document. Ces différents éléments définissent implicitement la famille d'événements correspondante.L'identificateur de cette famille est :

```
modifyDoc_OBJ_OBJ_OBJ
```

### IV.3.2 Attachements et clauses réactives

Les attachements décrivent pour une instance de la classe en cours de déclaration les associations licites événement–réaction. Les choix que nous avons effectués réduisent par rapport à la discussion du chapitre précédent les types de réaction possibles. Ainsi, un attachement se compose de deux parties qui sont :

- le nom de famille d'événement : comprend la signature (type des paramètres)
- la méthode à exécuter (sur l'objet courant) ou le programme à exécuter

#### IV.3.2.1 Déclaration des attachements

La déclaration des attachements précise leurs caractéristiques statiques. Ces caractéristiques sont :

- l'identificateur de la famille des événements déclencheurs et
- la réaction à exécuter à l'occurrence de l'événement.

Le choix d'une déclaration implicite des familles implique également qu'une déclaration d'attachement équivaut à une déclaration de famille. C'est la donnée de l'identificateur de la famille qui équivaut ici à sa déclaration.

Les réactions peuvent être soit des méthodes de la classe courante, soit des programmes exécutables. L'utilisation d'une méthode comme réaction ne peut se

faire que sur l'objet courant (cf IV.2.2.5). La seconde forme est requise pour permettre le lancement de nouvelles applications à l'occurrence d'un événement. La méthode est identifiée par son nom dans l'interface de la classe et le programme par son chemin d'accès.

La forme déclarative d'un attachement est la suivante :

```
ON <nom_d_evenement> <liste_de_types_de_parametres>
DO <reaction>
```

De la première partie de la déclaration, on déduit l'identificateur de la famille des événements déclencheurs. Cette déclaration équivaut donc à une définition de cette famille.

La réaction peut comporter des paramètres, déduits de ceux de l'événement ou indépendants. Il faut alors spécifier dans la déclaration de l'attachement les règles de correspondance entre paramètres de l'événement et paramètres de la réaction.

**Remarque**

Cette possibilité n'est pas offerte actuellement dans notre modèle. La correspondance entre les deux ensembles de paramètres est bijective : la signature de la réaction doit être identique ou conforme à la signature de l'événement. Les paramètres conservent le même ordre.

### IV.3.2.2 Clauses réactives

Le comportement d'un objet vis–à–vis des événements est déterminé par l'ensemble des attachements actifs à un instant donné. Le regroupement de plusieurs attachements peut alors revêtir un grand intérêt, puisqu'il permet de caractériser une partie du comportement des instances de la classe. C'est pourquoi nous introduisons la notion de *clause réactive* : une clause réactive est un regroupement d'attachements qui sont activés et désactivés en même temps. Outre la caractérisation d'un type de comportement, l'intérêt de la clause réactive est qu'elle permet l'activation de plusieurs attachements simultanément sans avoir à activer chaque attachement explicitement.

Les clauses réactives sont nommées et sont donc composées d'un ensemble d'attachements. La forme de leur déclaration est donc la suivante :

```
CLAUSE <nom_de_clause>
    ON <nom_evt_1> <param_1> DO <reaction_1>
    ON <nom_evt_2> <param_2> DO <reaction_2>
    ...
    ON <nom_evt_n> <param_n> DO <reaction_n>
```

L'ensemble des clauses réactives définies pour une classe sont déclarées dans une section spéciale de celle–ci, au même niveau que la déclaration des méthodes et des variables d'état. Ainsi, la structure d'une classe sera la suivante :

```
CLASS <nom_de_classe>
BEGIN
    <liste_des_variables_d_etat>
    <liste_des_methodes>
    COORDINATIONS
        CLAUSE <nom_clause_1>
            ON ... DO ...
            ON ... DO ...
            ...
        CLAUSE <nom_clause_2>
            ON ... DO ...
            ...
        ...
        CLAUSE <nom_clause_n>
            ON ... DO ...
            ...
    END_COORDINATIONS
END <nom_de_classe>;
```

Au sein des méthodes de la classe, chaque clause est alors connue par son nom et peut être activée ou désactivée – ce qui entraîne l'activation ou la désactivation des attachements qui la composent. Nous reviendrons sur ces opérations en IV.4.1.

Comme nous l'avons montré au chapitre précédent, il n'existe aucune contre–indication fonctionnelle à hériter des attachements d'une classe. Ainsi, au même titre que les méthodes, les clauses réactives sont héritées et peuvent être surchargées. Comme l'héritage dans Guide est conforme, la modification de la signature d'une méthode n'implique aucune modification des attachements dans lesquels elle est impliquée, et donc pas de redéfinition des clauses incriminées.

## IV.4 Modèle d'exécution

Le modèle d'exécution associé à la communication par événements détermine le mode d'émission des événements, l'effet des opérations d'abonnement et de désabonnement et le mode d'exécution des réactions. Nous ne nous attarderons pas sur l'émission car Guide ne propose aucune caractéristique propre à compléter ou modifier la présentation de l'émission faite au chapitre précédent.

En revanche, les deux autres points méritent dans le cadre de Guide et de son modèle d'exécution une attention particulière. Selon les objectifs que nous nous sommes fixés en IV.2.1, le modèle choisi doit être le plus complet possible dans la limite des possibilités de Guide. Nous avons également mentionné les choix que nous faisions :

- abonnement et désabonnement sont des opérations dynamiques ;
- le contexte d'exécution des réactions est limité.

Étant donnés ces choix, nous détaillons comment ils sont réalisés dans notre proposition.

### IV.4.1 Abonnement, désabonnement et contexte d'exécution

Les opérations d'abonnement et de désabonnement sont des opérations dynamiques. Elles consistent à activer ou désactiver un ensemble d'attachements regroupés dans une clause réactive.

#### IV.4.1.1 Contrôle des attachements actifs

Dans la section III.4.4 du chapitre précédent, nous avons proposé différentes politiques de contrôle des attachements multiples pour un événement donné. La politique que nous adoptons pour Guide est de n'autoriser pour un événement qu'un attachement actif par contexte (ou tâche). Ainsi, on assure que pour tout événement émis, il y aura au plus une réaction exécutée dans chaque tâche du système. Nous avons écarté l'empilement des attachements car cette possibilité ne nous a pas parue nécessaire pour les utilisations que nous visions.

L'unicité est assurée grâce à un **désabonnement implicite** : une demande d'activation sur un attachement mettant en jeu un événement `e` dans un contexte `C` provoque le désabonnement de tout attachement impliquant `e` et actif dans `C`.

#### IV.4.1.2 Définition du contexte d'exécution

Ces attachements ne permettent que rarement de déterminer tout ou partie du contexte d'exécution. Le cas le plus simple est lorsque la réaction associée est un programme à exécuter. Le contexte d'exécution est alors celui d'une nouvelle tâche créée à cette occasion.

Dans les autres cas, seul l'objet sur lequel la méthode sera appelée est connu. Il faut déterminer les autres éléments du contexte d'exécution au moment de l'abonnement. Les éléments à déterminer sont les suivants :

- le contexte (tâche) où aura lieu la réaction est–il quelconque ou bien identifié ?
- dans le cas d'une tâche identifiée, l'activité est–elle quelconque ou identifiée ?

En conséquence, nous distinguons trois formes d'abonnement possibles, qui correspondent aux différents contextes d'exécution. Nous les illustrons en donnant une expression possible de l'opération effectuée sur une clause réactive `Clause`.

1. Exécution dans une tâche quelconque.

```
SUBSCRIBE Clause
```

2. Exécution dans une tâche identifiée par une activité quelconque

```
SUBSCRIBE CONTEXT Clause
```

3. Exécution dans une tâche identifiée par une nouvelle activité

```
SUBSCRIBE NEW_ACT Clause
```

Chaque attachement de la clause est activé selon les directives de l'opération (avec le contexte d'exécution associé), excepté si la réaction associée est un programme à exécuter, auquel cas le contexte d'exécution est obligatoirement une nouvelle tâche.

#### IV.4.1.3 Désabonnement

L'opération de désabonnement consiste à désactiver un attachement. Cependant, dans la mesure où un attachement peut être actif dans plusieurs contextes simultanément, il faut définir la politique choisie pour la désactivation.

Le choix que nous avons fait est le suivant : un attachement actif dans un contexte donné ne peut être désactivé que par une activité de ce contexte. D'autre part, les attachements actifs mais dont le contexte n'est pas précisé (contexte quelconque) peuvent être désactivés dans n'importe quelle tâche. Aussi, nous distinguons deux formes de désabonnement.

1. Le désabonnement porte sur un attachement activé dans un contexte quelconque. L'opération correspondante, toujours exprimée sur une clause réactive, peut s'exprimer de la manière suivante :

   ```
   UNSUBSCRIBE Clause
   ```

   Les attachements décrits dans cette clause sont alors désactivés dans la tâche qui représente les contextes quelconques et seulement dans celle–là.

2. Le désabonnement porte sur un attachement activé dans le contexte courant. Dans les mêmes conditions qu'au point précédent, l'opération s'exprime comme suit :

   ```
   UNSUBSCRIBE CONTEXT Clause
   ```

   Les attachements décrits dans cette clause et actifs dans le contexte courant sont désactivés. L'effet est nul pour tous les autres contextes.

Enfin, lorsqu'une entité (tâche ou objet) est détruite, tous les attachements actifs qui mettent en jeu cette entité dans le contexte d'exécution de la réaction associée sont implicitement désactivés. Tenter de satisfaire ces attachements à l'occurrence d'un événement alors que le contexte d'exécution n'existe plus conduirait immanquablement à des erreurs.

### IV.4.2 Contexte d'exécution

Les contraintes que nous nous sommes fixés à la section IV.2.1 sur le modèle d'exécution restreignent les possibilités décrites au chapitre III. Nous avons présenté en IV.2.2.5 les choix que nous faisions pour Guide. Le modèle d'exécution de Guide a été exploité pour réaliser ces choix.

### IV.4.2.1 Tâches et exécution des réactions

Les tâches dans lesquelles peuvent s'exécuter des réactions doivent pouvoir recevoir des directives qui leur permettent de créer dynamiquement des activités ou faire exécuter par une de leurs activités les réactions demandées. Cette propriété nécessite un complément à la définition des tâches qui devront être « instrumentées » pour réagir à de telles directives.

En particulier, l'exécution par une activité quelconque dans une tâche donnée réclame des aménagements. Conformément à la suggestion que nous faisions en III.5.2.3, ce rôle est dévolu à une activité particulière que chaque tâche possédera.

Cette activité, outre l'exécution de réactions à exécuter dans la tâche par un flot « quelconque », aura pour rôle de créer dynamiquement les activités chargées des autres réactions (exécutées par un nouveau flot).

Ainsi, les directives d'exécution de réactions destinées à une tâche seront toutes traitées par cette activité.

### IV.4.2.2 Tâches quelconques

Le modèle d'exécution retenu propose également l'exécution des réactions dans une tâche quelconque, qui n'est pas spécialement créée à cette fin. À l'instar de ce que nous proposions au chapitre III.5.2.4, nous déléguons ce rôle de tâches « quelconques » à un ensemble de tâches spécialisées. Toute réaction à exécuter dans une tâche quelconque l'est dans une de ces tâches spécialisées. Le choix de cette tâche est déterminé par le contexte où l'abonnement a été réalisé ; nous développerons cet aspect dans la section IV.5 où nous traiterons des problèmes de portée.

## IV.4.3 Problèmes d'ordre

Nous n'avons pas encore considéré les problèmes liés à l'ordre sur les événements. Cet aspect concerne essentiellement l'ordre d'exécution des réactions aux événements.

Nous adoptons la position suivante :

> *L'ordre des exécutions des réactions à des événements respecte l'ordre causal sur les événements.*

L'ordre causal que nous considérons est un ordre partiel et qui est déduit de la relation de dépendance causale entre les événements. Cette relation est définie comme suit :

> *Soient* `e1` *et* `e2` *deux événements.* `e2` *dépend causalement de* `e1` *si une réaction à* `e1` *entraîne l'émission de* `e2`.

Ainsi, d'après notre choix, si l'événement `e2` dépend causalement de l'événement `e1`, nous assurons que toutes les réactions à `e1` dans l'ensemble du système débutent avant toutes celles à `e2`.

Nous ne prenons en compte que le début des exécutions, car l'attente de la terminaison de toutes les réactions rendrait le système synchrone. En effet, les dépendances causales ne sont pas décelables statiquement, du fait du caractère dynamique des opérations d'abonnement et de désabonnement. Aussi, il faudrait appliquer à tous les événements le même traitement : attendre la terminaison des exécutions de toutes les réactions avant de traiter les événements suivants, au cas où une dépendance causale serait détectée. Cette solution n'est pas acceptable, car elle fait perdre au mécanisme une grande part de son asynchronisme et réduit le parallélisme.

## IV.5 Portée des événements : notion de domaine

La notion de domaine de communication événementielle que nous introduisons répond à un double objectif de structuration de l'espace des interactions et de limitation de la portée. Cette notion se rapproche des espaces d'acteurs [Callsen 94] que nous avons présentés en II.2.2.

### IV.5.1 Définition et fonctionnement

La structuration en domaines se traduit par la définition d'ensembles d'abonnements qui marquent la participation des objets qui les ont souscrits à certaines interactions. Les objets dont au moins un abonnement appartient à un tel ensemble constituent le domaine associé à cet ensemble. Ces objets sont appelés les **membres** du domaine. La limitation de la portée d'un événement est assurée en l'émettant vers un tel domaine. Il ne sera alors reçu que par les membres du domaine qui lui sont abonnés. Un domaine est donc du point de vue de l'émetteur d'un événement un **groupe de diffusion** à qui il s'adresse. L'événement est ensuite diffusé au sein de ce groupe en respectant un filtre (un objet ne reçoit un événement que s'il y est abonné).

Notons d'ores et déjà quelques propriétés des domaines. La première est qu'un objet peut appartenir à plusieurs domaines : il peut souscrire de nombreux abonnements qui marquent sa participation à des interactions très différentes. La seconde propriété est que l'émetteur d'un événement n'a pas à appartenir au domaine (groupe de diffusion) où il émet.

### IV.5.2 Opérations événementielles et domaines

Toute opération événementielle effectuée par un objet est associée à un domaine donné : un attachement est activé dans un domaine, un événement est émis vers un domaine. Ce domaine est appelé **domaine d'opération**. Sa détermination peut revêtir deux formes.

La première consiste à associer à chaque objet un domaine d'opération courant dans lequel toutes les opérations événementielles seront effectuées. Ce domaine peut changer au cours de la vie de l'objet. La seconde consiste pour chaque opération à associer explicitement un domaine d'opération. Ces deux modes peuvent être combinés : sauf indication explicite d'un domaine lors d'une opération, le domaine d'opération associé est le domaine par défaut.

```
Dans le cas d'une session d'édition coopérative, le domaine
    par défaut est celui associé à la session, et sauf
    spécification contraire, toutes les opérations
    événementielles réalisées au sein de la session sont
    effectuées dans ce domaine.
```

Nous précisons maintenant quels sont les effets des opérations événementielles dans le contexte de leur domaine d'opération.

- l'émission d'un événement par un objet est en fait une diffusion vers le domaine d'opération. L'objet émetteur n'a pas à être membre du domaine d'opération.
- l'opération d'abonnement d'un objet à un événement e dans un domaine correspond à l'enregistrement dans ce domaine de la structure d'abonnement correspondante. Si un abonnement à $e_2$ a été contracté dans un domaine $D_1$, que le domaine courant de l'objet abonné est $D_2$ et que e est émis dans $D_1$, la réaction spécifiée dans $D_1$ sera exécutée. En revanche, si e est émis dans $D_2$, la réaction spécifiée dans $D_1$ ne sera pas exécutée. Ainsi, un objet peut s'abonner au même événement e dans plusieurs domaines $D_1, \ldots, D_n$ différents et l'émission de e dans un des domaines $D_i$ entraînera la réaction spécifiée lors de l'abonnement dans $D_i$ (cf Fig. 4.1).
- le désabonnement se rapporte toujours à un abonnement contracté dans le domaine d'opération. Si l'abonnement en question n'appartient pas au domaine spécifié, l'opération est une opération nulle.

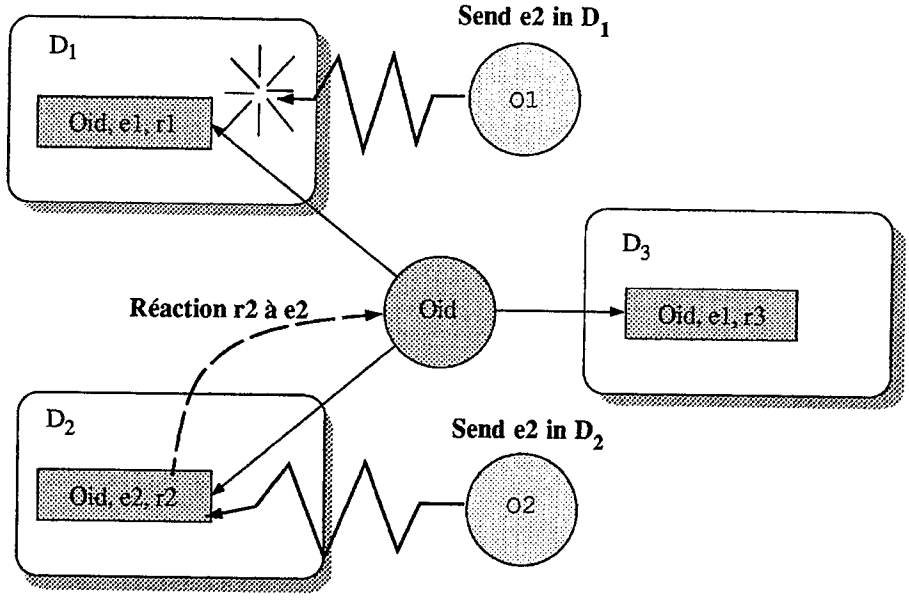


*Fig. 4.1 : Abonnements et émissions dans un espace de domaines*

La figure Fig. 4.1 illustre certaines de ces opérations. L'objet de référence `Oid` est abonné aux événements `e1` dans les domaines $D_1$ et $D_3$ et `e2` dans le domaine $D_2$. Lorsque `e2` est émis depuis un objet `O1` (qui n'appartient pas nécessairement aux domaines) dans le domaine $D_1$, aucune réaction n'est déclenchée. En revanche, si `e2` est émis dans le domaine $D_2$, la réaction `r2` est exécutée.

Pour qu'il puisse effectuer des opérations dans un domaine, un objet doit disposer d'un moyen de l'identifier. Nous proposons d'identifier un domaine par un nom symbolique non ambigu, à partir duquel pourront être retrouvées les structures qui le décrivent et permettent de le gérer.

### IV.5.3 Structuration de l'espace des domaines

L'intérêt principal des domaines est qu'ils déterminent une portée pour les événements. Cependant, cette notion peut également constituer la base pour une structuration des applications. Chaque application peut être décomposée en plusieurs domaines en interaction, et les applications ainsi décomposées peuvent également interagir.

Dès lors, des relations entre domaines peuvent être établies. Ces relations reposent à la fois sur les communications (événementielles) qui s'établissent entre domaines, mais aussi sur la place relative des domaines les uns par rapport aux autres. Afin d'explorer ces aspects, nous nous intéressons donc au schéma de communication inter–domaine et à la définition d'une hiérarchie de domaines.

### IV.5.3.1 Communication inter-domaines

Nous avons indiqué que la portée d'un événement était limitée au domaine où il était diffusé. Hors, il est possible que cet événement doive avoir une portée plus générale et puisse intéresser d'autres domaines. Le coût d'émissions supplémentaires dans ces autres domaines peut être évité si un moyen de propager les événements d'un domaine à un autre est fourni.

L'établissement de telles communications doit obéir aux critères suivants :

- la propagation ne doit pas être systématique : certains événements concernent plusieurs domaines, mais d'autres ne doivent pas dépasser les limites de leur domaine d'émission.
- ces relations ne doivent pas être établies statiquement : les relations entre domaines, comme celles entre objets, doivent pouvoir évoluer au cours de l'exécution.

Une manière de réaliser ces critères est de recourir à un mécanisme proche de l'abonnement où un domaine $D_1$ déclare son intérêt pour tel ou tel événement auprès d'un autre domaine $D_2$. Lorsque ledit événement est émis dans $D_2$, il est également propagé dans $D_1$. Cet intérêt peut être révoqué afin de permettre la dynamicité des relations.

### IV.5.3.2 Hiérarchie de domaines

Si la communication entre domaines est un moyen de structuration utile, une approche hiérarchique offre également de riches possibilités. Le principe d'une hiérarchie de domaines repose sur leur composition : un domaine D peut contenir plusieurs autres domaines $sD_i$ appelés **sous-domaines**. L'ensemble des membres de D est alors l'union de ceux qui lui sont propres (qui n'appartiennent à aucun des $sD_i$) et des membres de tous les $sD_i$.

Toute émission dans le domaine D se traduit alors par une émission identique dans chacun des $sD_i$, et de leurs propres sous-domaines. Ceci revient à dire que tout objet membre d'un $sD_i$ est membre de D. En revanche, tout objet membre de D n'est pas nécessairement membre d'un $sD_i$. Ainsi, une opération d'abonnement ou de désabonnement dans D n'est pas propagée à tous les $sD_i$.

La composition permet notamment la propagation à l'ensemble des domaines d'une hiérarchie d'un événement émis dans le domaine racine de cette hiérarchie. Cette propriété est illustrée à la figure Fig. 4.2. Un domaine $D_0$ contient deux sous-domaines $D_1$ et $D_2$, qui contiennent eux-mêmes deux sous-domaines chacun, $D_{11}$ et $D_{12}$ d'une part, $D_{21}$ et $D_{22}$ d'autre part. Comme indiqué sur la figure, l'émission de l'événement $e_0$ dans $D_0$ équivaut à l'émission de $e_0$ dans tous les

domaines inclus dans $D_0$, quel que soit leur niveau dans la hiérarchie. En revanche, l'émission de $e_1$ dans $D_{11}$ est effectivement limitée à ce seul domaine puisqu'il n'en contient aucun autre.

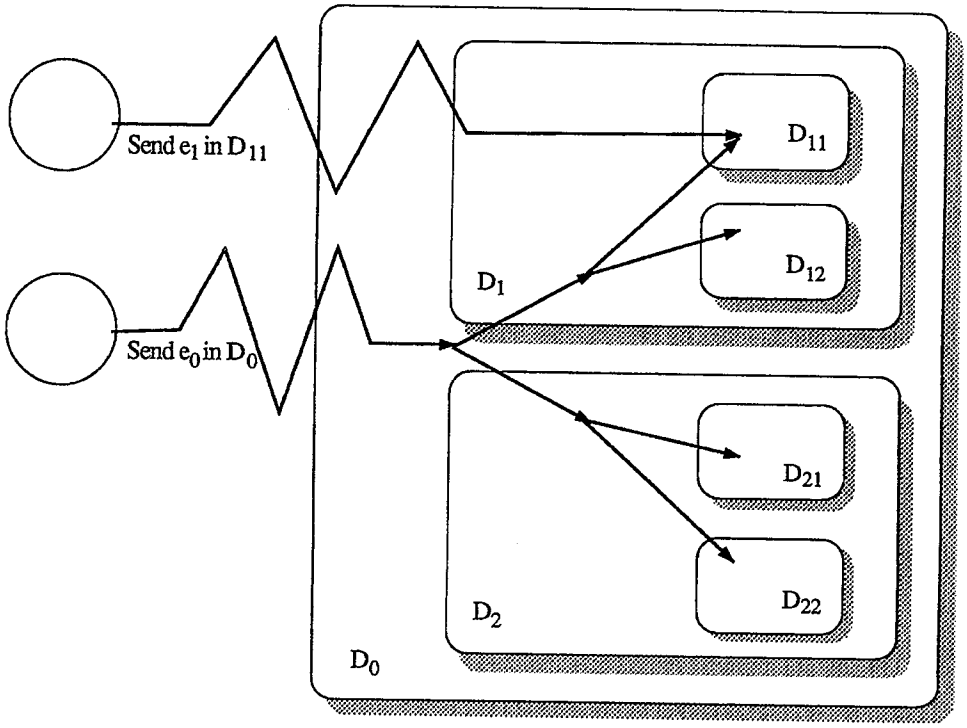


*Fig. 4.2 : Propagation dans un espace de domaines hiérarchisé*

Grâce à cette structure hiérarchique, il est possible de décomposer les applications en plusieurs domaines et de les inclure dans un domaine qui représente l'application. Les événements propres à chaque composante de l'application seront émis dans le sous–domaine correspondant, alors que les événements d'intérêt général, tel que le signal de terminaison de l'application, seront émis dans le super–domaine, puis propagés à toutes les composantes. L'approche peut être généralisée au système entier : un domaine représente l'ensemble du système et contient tous les domaines applicatifs. Les événements généraux, comme le signal d'arrêt du système, diffusés dans ce domaine, sont répercutés le long des liens hiérarchiques à l'ensemble des domaines du système.

### IV.5.4 Problèmes ouverts

La notion de domaine offre une double fonction. C'est d'abord une unité de limitation de la portée des événements, ensuite une unité de structuration des applications. Les éléments que nous avons donnés aux sections précédentes précisent les propriétés des domaines, mais ouvrent également un certain nombre de

problèmes que nous abordons dans ce qui suit. Définition d'une portée. Hiérarchie. Diffusion au sein de la hiérarchie. Critères de structuration.

### IV.5.4.1 Super–domaines multiples

Nous n'avons pas écarté en IV.5.3.2 la possibilité qu'un domaine possède plusieurs super–domaines, c'est–à–dire qu'il soit simultanément inclus dans plusieurs domaines. La hiérarchie des domaines n'est plus alors un arbre, mais un graphe orienté acyclique. Ainsi, un domaine ayant plusieurs super–domaines pourrait recevoir les événements de toutes les hiérarchies correspondantes. Un objet s'abonne une seule fois à un événement dans ce domaine, et l'émission de cet événement dans les niveaux supérieurs entraîne sa réaction. On évite ainsi un abonnement par super–domaine. L'intérêt de cette structure est donc principalement de factoriser les abonnements à des événements qui peuvent être émis indifféremment dans plusieurs chaînes d'inclusion distinctes.

Néanmoins, il faut garantir qu'une instance d'un événement n'arrive qu'une seule fois dans un domaine. Hors, l'introduction de super–domaines multiples ouvre une brèche dans la politique de diffusion des événements des domaines à tous leurs sous–domaines. En effet, si cette politique est appliquée, deux domaines peuvent diffuser à leurs sous–domaines communs une même instance d'événement pour peu que cette instance provienne d'un de leurs ancêtres communs (le domaine représentant le système par exemple). La gestion de cette diffusion est donc considérablement compliquée par rapport à une diffusion dans un arbre de domaines. Les moyens techniques à mettre en œuvre peuvent s'avérer trop importants pour le gain qu'apporte une telle structuration.

En fait, la diffusion d'un événement dans l'espace des domaines revient à suivre un chemin dans un graphe orienté acyclique. La propriété que nous souhaitons assurer est que pour chaque événement, un seul chemin est suivi parmi tous ceux qui sont possibles. Dans le cas où chaque domaine ne peut avoir qu'un seul super–domaine, le graphe représentant l'espace des domaines est un arbre, et quel que soit le domaine d'émission D et ses sous–domaines $D_i$ où l'événement doit être diffusé, il existe un chemin unique de D à chaque $D_i$ dans l'arbre des domaines. Dès lors que les super–domaines multiples sont autorisés, l'unicité de ce chemin n'est plus garantie.

En définitive, le choix doit être fait entre un mode de structuration qui permet une politique de diffusion simple mais offre moins de souplesse dans la définition de la hiérarchie, et un autre mode qui permet la mise en place de structures qui peuvent être utiles mais qui augmentent considérablement le coût de gestion de cette structure. Nous nous plaçons pour l'instant dans le premier cas et nous interdisons les super–domaines multiples.

### IV.5.4.2 Désignation

Les domaines doivent être désignés pour être manipulés. La désignation doit permettre au programmeur d'accéder simplement au domaine qu'il désire manipuler. L'utilisation d'un nom symbolique est donc nécessaire. D'autre part, dans l'éventualité d'une hiérarchie arborescente de domaines, une désignation par chemin d'accès dans un arbre de domaines faciliterait l'appréhension de cette structure.

Cependant ce nom symbolique est insuffisant pour réaliser un accès efficace aux domaines. Il convient de leur associer un identificateur interne, à l'image d'une référence d'objet.

### IV.5.4.3 Description et utilisation des domaines

L'introduction des domaines dans le mécanisme complique sa mise en place et le travail du programmeur. Les difficultés concernent deux points : la description des domaines et de leur structure d'abord, leur utilisation ensuite.

La description des domaines peut être soit intégrée au code de l'application, soit dévolue à des outils externes. Dans le premier cas, le programmeur doit au sein de ces programmes créer ses domaines et établir entre eux les relations. Dans le second cas, cette création et cette structuration sont indépendantes du processus de programmation. Nous avons adopté la seconde approche pour ne pas surcharger les programmes qui utilisent la communication événementielle.

D'autant plus que ces programmes devront nécessairement manipuler ces domaines pour les opérations événementielles ou pour établir des relations entre eux. Nous privilégions là–encore l'option qui nécessite le moins de référence possible aux domaines dans le code et qui consiste à définir un domaine par défaut dans lequel la majorité des opérations sont effectuées. Les références exceptionnelles à d'autres domaines sont permises et utilisent pour cela le nom symbolique du domaine.

### IV.5.4.4 Règles de structuration

Nous avons montré comment les domaines pouvaient être utilisés pour structurer des applications. Les règles qui régissent cette structuration sont cependant à définir. Nous avons indiqué à titre d'exemple des décompositions reposant sur des caractéristiques fonctionnelles, mais les critères qui peuvent être utilisé sont plus nombreux. Nous en énumérons quelques uns.

- décomposition fonctionnelle : un domaine est associé à chaque composant identifié d'un application.
- la répartition : on peut associer à chaque site du système un courtier chargé des opérations sur ce site, quelle que soit l'application cliente.

- les utilisateurs : un domaine peut être associé à chaque utilisateur. Toute application exécutée par cet utilisateur et qui utilise les événements passera par ce domaine.
- un type de fonction : un domaine particulier peut être utilisé pour certaines fonctions identifiées du système, telles que la gestion des imprimantes ou des disques présents sur le réseau.

Ces critères peuvent de plus être combinés pour la composition d'une hiérarchie de domaines.

## IV.6 Synthèse

Nous avons présenté comment nous pouvions appliquer l'étude générale du chapitre précédent à un modèle particulier. Nous tirons deux principaux enseignements de cette étude. Le premier est qu'elle est applicable à d'autres modèles à objets, mais que des adaptations sont nécessaires. Certaines caractéristiques de Guide influent en effet fortement sur les choix de conception. Le second réside dans la notion de domaine et dans le moyen qu'elle offre pour structurer des applications.

### IV.6.1 Modèle à objets partagés répartis

Le modèle que nous avons considéré possède des propriétés caractéristiques : il définit et manipule des objets qui sont persistants, partagés et répartis. De ce fait, il répond en partie à certaines conditions que nous avons mentionnées au chapitre I pour le support des applications coopératives (partage et répartition). En revanche, le seul mode de communication offert est l'appel de méthode synchrone et le partage d'objets entre activités (flots de contrôle). Guide était par conséquent un cadre adapté à l'extension que nous projetions.

La nature des caractéristiques de Guide a influé sur notre description d'un mécanisme de communication événementielle. Nous examinons dans ce qui suit leur impact et les conséquences pour une adaptation du mécanisme à un autre environnement.

La répartition n'est pas intervenue dans la définition des structures de notre mécanisme. Le caractère transparent de la répartition dans Guide nous a permis de nous affranchir de ces aspects dans la description de notre mécanisme. En conséquence, il peut être adapté sans modification à un environnement centralisé.

Le caractère partageable des objets est un aspect primordial de notre mécanisme, puisqu'il garantit que plusieurs flots d'exécution peuvent s'exécuter concurremment au sein d'un même objet. En conséquence, plusieurs réactions, générées par

plusieurs événements, peuvent être exécutées en parallèle dans un objet sans avoir à mettre en place de mécanisme particulier. Dans un environnement où ce partage n'est pas fourni, le contrôle des exécutions serait plus ardu : il faudrait d'une part sérialiser les exécutions et d'autre part, garantir que les flots principaux d'exécution ne sont pas altérés par l'exécution des réactions.

La persistance assure qu'un objet survit au flot d'exécution qui l'a créé. Cette propriété implique que les attachements actifs doivent également posséder cette propriété de persistance. Dans un autre environnement n'offrant pas cette propriété, les éléments persistants de notre proposition deviennent fugaces.

Enfin, le modèle d'exécution est celui d'Unix généralisé à un environnement réparti. Comme nous l'avons dit, la répartition dans Guide est transparente et n'est pas intervenue dans notre proposition. L'environnement d'exécution d'Unix offre par conséquent les structures nécessaires à la réalisation des différents modes d'exécution que nous avons proposés.

### IV.6.2 Structuration des applications

La notion de domaine que nous avons introduite offre un cadre de définition de la portée des événements, mais constitue également un moyen de structuration des applications que supporte le système.

Nous avons indiqué en IV.5.4.3 des règles ou critères qui pourraient être utilisés pour diriger cette structuration. Nous pensons de surcroît que les travaux menés au sein du projet Sirac autour du modèle OLAN (cf I.1) pourront apporter des indications précieuses.

# Chapitre V

# Mise en œuvre et évaluation

Les propositions du chapitre précédent ont fait l'objet d'une mise en œuvre expérimentale sur une plate–forme à objets répartis. Outre le prototype du système de communication événementielle, des applications ont également été conçues à fin d'évaluation. Nous décrivons dans ce chapitre les principes qui ont régi ces différentes actions de mise en œuvre. À la suite de cette description, nous tirerons les conclusions de ces expérimentations. Ces conclusions ont principalement trait à l'évaluation du mécanisme de communication événementielle et de son utilité.

## V.1 Environnement de développement : OODE

La plate–forme de mise en œuvre utilisée pour notre évaluation est un prototype pré–industriel dénommé OODE et développé au sein de la Société Bull [Bull 94]. Fortement inspiré par les résultats du projet Guide, il en reprend en particulier le modèle d'exécution. Cette plate–forme offre un environnement de développement et d'exécution pour applications réparties dont les caractéristiques principales sont les suivantes :

- un modèle d'exécution réparti qui offre une gestion totalement transparente de la répartition : un objet est toujours invoqué par le programmeur de la même façon, qu'il soit local ou distant ;
- la programmation persistante : le stockage et le chargement des objets sont pris en charge par la plate–forme de manière transparente au programmeur ;
- le partage simultané d'objets entre plusieurs utilisateurs et/ou applications.

### V.1.1 Langage et environnement

L'environnement de développement repose principalement sur un langage appelé OC++, qui est une extension de C++. Cette extension consiste en l'ajout d'un unique mot clé, **`distributed`**. Ce mot clé précède la déclaration des classes dont les instances pourront être manipulées à distance et qui pourront demeurer sur support

persistant. Les classes correspondantes, appelées **classes distribuées**, coexistent avec des classes standard C++, mais sont soumises à quelques restrictions. En particulier, l'utilisation de pointeurs y est prohibée, puisque les objets peuvent être liés dans les espaces d'adressage des différentes machines qui composent le système. Cette limitation vaut aussi bien pour l'utilisation au sein des classes, dans les variables d'état par exemple, que pour les paramètres des méthodes des classes distribuées. Seules les références à des objets distribués, appelées **pointeurs distribués** et qui sont indépendantes de la localisation de l'objet, peuvent être utilisées. D'autre part, l'héritage sur les classes distribuées est un héritage simple.

À chaque classe distribuée est associée une description de son interface écrite en IDL, le langage d'interface de CORBA. À partir de cette description, un compilateur génère le code des talons et des représentants locaux des objets par l'intermédiaire desquels les appels distants peuvent être réalisés. Cette description permet également l'interopérabilité avec CORBA.

### V.1.2 Plate–forme d'exécution

Le modèle d'exécution de OODE reprend celui proposé dans Guide et offre les mêmes abstractions. Réalisé sur AIX et DCE, OODE fait correspondre aux tâches la notion de processus et aux activités la notion de processus léger (« thread ») de DCE.

Le système offre la possibilité de créer et de manipuler jusqu'à un certain point les activités et les tâches. Ces fonctions ont été utilisées pour réaliser les différents modes d'exécution des réactions présentés en IV.4.2. Activités et tâches sont représentées par des objets OODE, instances de classes distribuées fournies avec le système. Elles peuvent donc être dynamiquement créées et lancées.

Une activité est en particulier décrite par une référence d'objet distribué et une méthode de cet objet. Le lancement de l'activité correspond à la création d'un thread DCE qui exécute cette méthode sur cet objet.

La définition d'une tâche comprend la donnée d'un programme exécutable, sous la forme d'un chemin d'accès dans une arborescence de fichiers, programme qui sera exécuté dans un nouveau processus au lancement de la tâche.

Le lancement des activités et des tâches est réalisé selon un schéma « fork–join » : l'activité (respectivement la tâche) mère crée une activité (respectivement une tâche) fille qui poursuit son exécution de manière asynchrone. La mère peut ensuite se synchroniser sur la terminaison de sa fille.

Des besoins spécifiques de notre prototype ont conduit les concepteurs de OODE à rajouter au système un moyen de réaliser un appel de méthode dynamique : la

référence de l'objet et le nom de la méthode appelés ne sont pas connus au moment de la compilation, mais déterminés dynamiquement au moment de l'exécution. Ce mécanisme est rendu indispensable par le caractère dynamique de la communication événementielle et de sa composante réactive en particulier. Il est en effet impossible de déterminer statiquement quelle réaction sera exécutée à un instant donné, et donc de déterminer quelle méthode sera appelée sur quel objet. Comme la manipulation de pointeurs est prohibée dans les objets distribués, l'éventuelle utilisation de pointeurs de fonctions telle qu'elle existe en C++ est impossible. Il a donc fallu mettre en place ce mode d'invocation dynamique qui réalise sur OODE cette notion d'appel par pointeur de fonction.

### V.1.3 Désignation associative

Notre prototype utilise également une première version réalisée sur OODE d'un service de désignation associative décrit dans [Marangozov 95]. Ce service permet de désigner des objets sur des propriétés qu'ils ont publiées. Ainsi, nous pouvons définir pour un événement un ensemble d'objets destinataires en décrivant des propriétés qu'ils doivent satisfaire.

Ce service utilise des objets distribués pour représenter une base de données où sont enregistrées les propriétés publiées des objets. Pour chaque objet, le programmeur choisit et définit quelles propriétés sont enregistrées. Ces propriétés peuvent être des propriétés statiques – identificateur d'objet, nom de la classe – ou dynamiques – valeur d'un attribut.

La désignation repose sur la définition de groupes, dont la composition est définie par une expression sur les propriétés de ses membres. Les groupes sont nommés et définis dans le code des méthodes. Ils peuvent être rendus persistants et peuvent alors être utilisés en dehors de leur contexte de déclaration. À partir du nom d'un groupe, l'expression qui le décrit est retrouvée et une requête à la base de données est construite. Le résultat de cette requête est la liste des objets membres du groupe, c'est–à–dire ceux pour qui l'expression qui définit le groupe est vérifiée.

Le programmeur peut alors appliquer à ces objets le traitement qu'il désire.

## V.2 Mise en œuvre du prototype

Le principe de réalisation du prototype repose sur l'utilisation de **courtiers d'événements**. Ces courtiers sont en fait des serveurs auxquels les objets s'adressent pour toutes les opérations événementielles effectuées en leur sein. De plus, ce sont eux qui réalisent le traitement des événements. Nous détaillons la structure et le fonctionnement de ces courtiers dans la suite de cette section.

Au préalable, il convient de signaler que ce choix de conception (utilisation de courtiers serveurs) est commandé par un défaut de maîtrise sur la plate–forme utilisée, compilateur et machine d'exécution. Nous reviendrons en fin de chapitre sur le support spécifique qui pourrait être mis en place pour une intégration complète de la communication événementielle dans la plate–forme.

### V.2.1 Choix logiciels

Le prototype est caractérisé par l'utilisation massive des classes distribuées, pour la définition tant des données nécessaires à son utilisation que de ses structures d'exécution.

#### V.2.1.1 Utilisation de classes distribuées

Le prototype est écrit en OC++, mais n'utilise quasiment aucune classe standard (non distribuée). Il y'a deux raisons à cette utilisation exclusive des classes distribuées.

La première est le caractère réparti du mécanisme. Les entités et structures d'exécution manipulées doivent évoluer de manière transparente dans un environnement réparti. Le support de la répartition en OODE est fourni au travers des classes distribuées. L'accès aux instances de ces classes se fait indifféremment localement ou à distance.

La seconde raison est en fait une limitation imposée par OODE. L'utilisation systématique d'objets peut en effet nuire aux performances du prototype, mais l'emploi de pointeurs pour la représentation de structures de données temporaires et d'accès rapide est prohibé par OODE au sein de structures d'exécution réparties. Les objets distribués sont alors la seule alternative.

Ce choix induit quelques inconvénients, dont le principal découle de la persistance des objets distribués. En effet, si cette persistance est souhaitable pour la représentation des abonnements par exemple, elle constitue une gêne dès lors que l'on souhaite définir des entités fugaces, tels que les événements. C'est pourquoi la récupération et la destruction des objets après utilisation ont reçu une attention particulière.

Enfin, le mécanisme ne peut être utilisé que dans des classes distribuées : l'émission des événements, la définition de clauses réactives et les opérations d'abonnement qui s'y rapportent sont prohibées dans les classes standard C++.

#### V.2.1.2 Représentation interne des entités manipulées

Ce choix d'utiliser les classes distribuées se retrouve dans la représentation des données nécessaires à l'utilisation du mécanisme, telles que nous les avons introduites au chapitre IV : événements, attachements et clauses réactives.

Les événements sont représentés par des objets de classe `T_Event`, dans lesquels sont décrits leur identificateur (nom de famille), leurs paramètres et l'éventuelle description d'un groupe de destinataires. Ce groupe de destinataires est décrit grâce au service de désignation associative présenté en V.1.3.

La description statique des attachements au niveau des classes est traduite dans des objets de classe `Reaction`. Les variables d'état définies dans cette classe sont l'identificateur de la famille d'événements déclenchants et le nom de la méthode associée. Rappelons que la signature de cette méthode doit être semblable à celle des événements déclenchants.

Lorsqu'un attachement est activé, un objet de classe `Subscription` est créé. Cet objet prend en compte la description statique de l'objet `Reaction` correspondant et les indications données lors de l'opération d'abonnement. Ces indications concernent essentiellement le mode et le contexte d'exécution. Le mode indique s'il s'agit d'une nouvelle activité, tandis que le contexte identifie la tâche dans laquelle la réaction doit avoir lieu.

Les clauses réactives regroupent sous un même nom plusieurs attachements qui seront activés et désactivés ensemble. Les objets correspondants, instance de la classe `Clause`, contiennent donc ce nom et la liste des attachements (objets de classe `Reaction`) associés.

L'ensemble des clauses réactives associées à une classe (et des attachements qui correspondent) est stocké sous la forme d'une liste unique d'objets en mémoire persistante. Cette liste est créée et stockée à la première instanciation de la classe. Elle possède un identificateur unique et non ambigu, fonction de la classe à laquelle elle est associée, et qui permet de retrouver l'ensemble des clauses réactives de cette classe. Tout objet instance d'une telle classe dispose d'une référence sur cette liste à partir de laquelle sont retrouvées les caractéristiques des attachements activés. Lorsqu'une clause réactive est activée ou désactivée, la liste est parcourue séquentiellement jusqu'à ce que l'objet `Clause` correspondant soit trouvé. La recherche s'effectue sur le nom de la clause.

La politique adoptée pour la gestion de l'héritage est la suivante : la queue de la liste de clauses réactives de la sous–classe est la liste correspondante de sa super–classe. Ainsi, si un conflit de noms de clause entre sous–classe et super–classe survient, c'est toujours la clause de la sous–classe qui est prise en compte.

### V.2.2 Courtiers d'événements

L'utilisation des événements est dans la réalisation actuelle subordonnée à l'existence dans le système de courtiers d'événements. Ces courtiers sont des serveurs qui gèrent tous les aspects de la communication événementielle. Nous décrivons dans la suite le rôle de ces courtiers, leur architecture et leur fonctionnement.

#### V.2.2.1 Rôle du courtier

Le courtier est la représentation au niveau du support d'exécution du domaine que nous avons introduit en IV.5. Toutes les opérations relatives aux événements sont réalisées par ou auprès d'un courtier. Le rôle d'un courtier se compose donc des attributions suivantes :

- **Gestion des abonnements**

  Lorsqu'un abonnement ou un désabonnement sont effectués par une activité au sein d'un objet, le courtier dépose ou retire dans une base d'attachements l'attachement correspondant.

- **Émission des événements**

  Lorsqu'un événement est émis, il l'est auprès d'un courtier.

- **Traitement des événements**

  Lorsqu'un courtier reçoit un événement, il le traite : il recherche dans sa base d'attachements actifs quels sont ceux qui correspondent à l'événement et déclenche les réactions associées.

Nous détaillons dans ce qui suit comment le courtier remplit ces différents rôles.

#### V.2.2.2 Architecture

Un courtier est avant tout l'instance d'une classe distribuée, la classe `EventBroker`. L'interface de cette classe offre les différentes fonctions des courtiers (cf V.2.2.1). C'est ensuite un ensemble de flots de contrôle qui opèrent au sein de cet objet et réalisent les traitements nécessaires au fonctionnement du mécanisme.

L'architecture d'un courtier comprend quatre parties. Les trois premières correspondent aux trois attributions d'un courtier que nous avons énumérées en V.2.2.1, tandis que la quatrième correspond au ramasse–miettes. Cette architecture est représentée à la figure Fig. 5.1.

**Base d'attachements**

La base d'attachements est le module où sont stockés les attachements actifs dans le domaine que définit le courtier. Nous parlerons indifféremment de base d'abonnements, puisque l'opération d'abonnement consiste à activer un attachement. Une opération d'abonnement effectuée dans un objet – l'opération d'abonnement porte sur une clause réactive – se traduit par le dépôt dans la base d'abonnements des attachements correspondants, avec la définition du contexte de lancement des réactions associées (objets de classe `Subscription`). De même, l'opération de désabonnement consiste à supprimer de la base les abonnements correspondants. La recherche dans cette base est réalisée selon différents critères qui correspondent à différentes utilisations de la base (cf V.2.2.3). Par exemple, la recherche des abonnements qui concernent un événement donné est facilitée par l'utilisation au niveau de la structure même de la base d'une indexation par nom d'événement. De même, une indexation par identificateur d'objet, par nom de clause réactive, par contexte de lancement est également fournie. Nous détaillons ces différents points en V.2.2.3.

**File d'événements**

La file d'événements constitue le réceptacle des événements émis dans le domaine du courtier : toute émission se traduit par le dépôt en queue de cette file de l'événement correspondant. La file est gérée selon un protocole FIFO avec un consommateur et n producteurs.

**Traitement des événements**

Le module de traitement des événements consomme les événements en tête de la file des événements et réalise à partir de ceux–ci les opérations nécessaires de recherche des abonnements correspondants, puis de lancement des réactions associées. C'est donc une tâche de fond du courtier et une activité demeure en écoute permanente sur l'arrivée d'éventuels événements. D'autre part, associé à ce module, un « pool de réactions » permet d'exécuter des réactions au sein du courtier. Nous reviendrons plus en détail sur le fonctionnement de ce module en V.2.3.

**Ramasse–miettes**

Le ramasse–miettes est également une tâche de fond du courtier. Il a la double attribution de détruire les événements traités – rappelons que les événements sont fugaces dans notre modèle – et de supprimer les attachements désactivés lorsque les éventuelles réactions qui leur

correspondent et qui ont pu être lancées avant la désactivation sont toutes terminées.

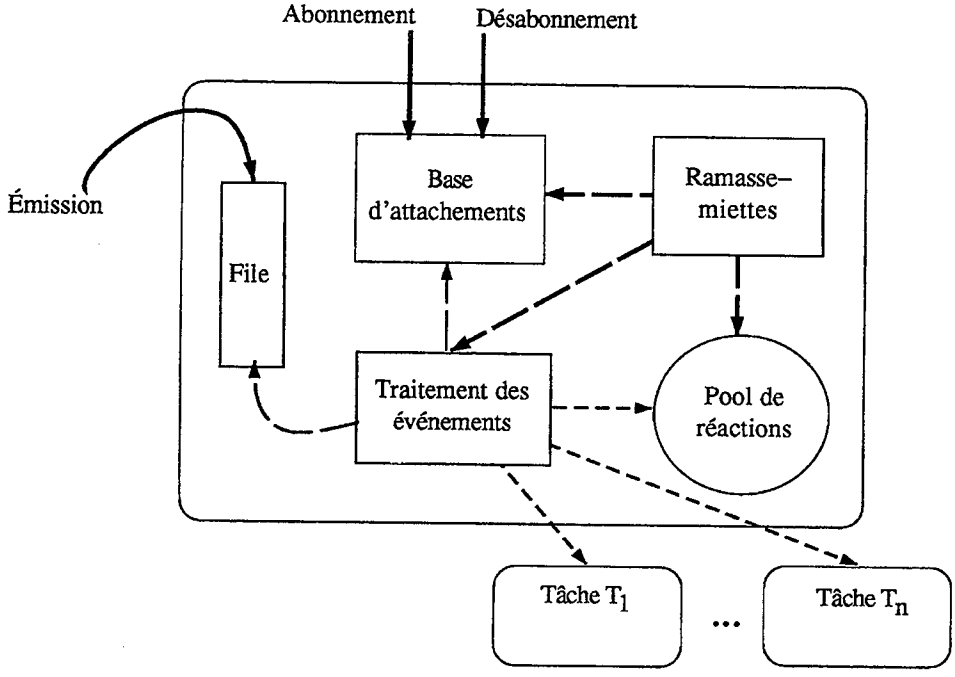


*Fig. 5.1 : Architecture d'un courtier d'événements*

### V.2.2.3 Base d'attachements

La base d'attachements permet de stocker et de gérer des attachements actifs. Chaque attachement est inséré dans plusieurs tables de hachage, avec une clé qui correspond à un critère de recherche particulier. Chaque critère de recherche est défini en fonction de certains besoins d'interrogation de la base. Ces différents critères sont les suivants :

- **Nom d'événement**

  Le nom de l'événement auquel est associé l'attachement est la première clé de recherche. Elle permet, étant donné un événement émis, de retrouver rapidement l'ensemble des abonnements contractés auprès du courtier pour cet événement et ainsi de déterminer quelles réactions doivent être exécutées.

- **Identificateur d'objet**

  L'identificateur de l'objet au sein duquel l'abonnement sera réalisé et sur lequel la réaction sera exécutée constitue la deuxième clé. L'utilité de cette clé est de supprimer de la base tous les attachements contractés au sein d'un objet qui est détruit. Ceci évite

notamment de conserver dans la base des attachements qui ne pourront pas être honorés, puisque l'objet a été détruit, et dont le traitement entraînerait une erreur.

- **Nom de clause réactive et identificateur d'objet**

  Le nom de la clause réactive dont l'activation a engendré l'abonnement est combiné à l'identificateur de l'objet au sein duquel elle a été activée pour former le troisième type de clé. Ce type de clé permet, lors d'une opération de désabonnement, qui met en jeu une clause réactive nommée et un objet, de retrouver l'ensemble des abonnements associés à ce couple et de les supprimer.

- **Nom de clause, identificateur d'objet et de contexte**

  La combinaison de ces trois champs dans une clé permet de vérifier et d'assurer rapidement l'unicité des abonnements d'un objet dans un contexte donné, en conformité avec les choix que nous avons faits au chapitre précédent (cf IV.4.1.1).

### V.2.3 Traitement des événements

Après avoir décrit l'architecture d'un courtier, élément central de notre mise en œuvre, nous précisons son mode de fonctionnement, en particulier la manière dont les événements sont traités.

Ce traitement comporte quatre phases. La première phase est la consommation de l'événement dans la file du courtier, la deuxième la recherche des attachements associés dans la base, la troisième l'analyse de ces attachements afin de déterminer s'ils doivent être honorés ou non et enfin la dernière phase consiste à déclencher les réactions.

Ces différentes phases doivent respecter un certain nombre de contraintes et de propriétés. Les deux principales sont l'ordre sur les événements et le respect des différents modes d'exécution.

#### V.2.3.1 Structures d'exécution

Les structures d'exécution mises en œuvre dans le courtier pour le traitement des événements consistent en deux activités de fond. La première activité est l'activité consommatrice des événements, la seconde réalise les autres phases du traitement.

La première activité consomme les événements dans la file de réception (gérée en FIFO). Elle les dépose ensuite dans une liste intermédiaire où la seconde les consommera. Ce découplage permet de vider plus rapidement la file de réception

puisque l'activité consommatrice n'a pas à traiter les événements. Cette vitesse de consommation accrue permet de réduire les risques de bloquage des clients.

La seconde activité consomme les événements de la liste intermédiaire – toujours en FIFO – et les traite. La description de ce traitement fait l'objet des sections suivantes.

### V.2.3.2 Recherche et analyse des attachements

L'analyse des attachements est la phase au cours de laquelle les éventuels filtres sur les événements sont évalués. Ces filtres portent essentiellement sur les objets abonnés et leur état et leur définition utilise le service de désignation associative. Le schéma suivi pour cette analyse est le suivant :

a) La première étape de l'analyse consiste à retrouver les objets qui correspondent au filtre spécifié. Si aucun filtre n'est donné, l'étape suivante est réalisée. Si la liste des objets correspondants est vide, aucune réaction ne doit être déclenchée et le traitement de l'événement suivant peut commencer.

b) La deuxième étape du traitement consiste à retrouver les abonnements associés à l'événement courant, grâce à une requête sur la base d'abonnements qui utilise l'identificateur de l'événement comme clé. Le résultat est une liste d'attachements actifs. Si cette liste est vide, il n'y a aucune réaction à exécuter.

c) Chaque attachement de cette liste est ensuite traité. Si l'objet associé à l'attachement n'est pas dans l'éventuelle liste des destinataires calculée à l'étape a), l'attachement suivant est analysé. Dans le cas contraire ou si aucun filtre n'a été défini, l'exécution de la réaction correspondante est commandée en fonction du mode d'exécution décrit.

### V.2.3.3 Déclenchement des réactions.

Nous avons retenu au chapitre précédent (cf IV.2.2.5) quatre modes d'exécution différents pour les réactions : dans une nouvelle tâche, dans une tâche identifiée et par une activité quelconque ou par une nouvelle activité, dans une tâche quelconque et par une activité quelconque. Pour réaliser ces différents modes nous avons utilisé les possibilités offertes par OODE pour créer des tâches et des activités au sein des tâches. Il a fallu par ailleurs instrumenter les tâches susceptibles d'exécuter des réactions. En effet, une activité ne peut pas être créée dans une tâche par un flot de contrôle extérieur, mais uniquement par une activité de cette tâche.

- **Création d'une nouvelle tâche**

  Le courtier crée une nouvelle tâche à l'aide des primitives OODE et le programme exécuté est celui qui est spécifié dans l'attachement.

- **Dans une tâche identifiée**

  La tâche en question doit être instrumentée. Cette instrumentation consiste en une file d'attachements et en une activité qui scrute cette file et traite les attachements qui y sont déposés (cf IV.4.2.1). Lorsqu'un attachement requiert une réaction dans une tâche identifiée, le rôle du courtier est de déposer dans la file de la tâche l'attachement et les paramètres de l'événement déclencheur. L'activité scrutatrice de la tâche visée consomme l'élément de la file et, suivant le mode précisé dans l'attachement, exécute elle–même la réaction ou crée une activité chargée de l'exécution.

- **Dans une tâche quelconque**

  Lorsque la tâche d'exécution n'importe pas, c'est la tâche du courtier qui est choisie. Le choix de l'activité d'exécution doit alors être un compromis entre la disponibilité du courtier pour consommer les événements émis et la charge induite par l'exécution des réactions. Les deux solutions extrêmes à ce problème sont les suivantes :

  – l'activité consommatrice exécute elle–même les réactions. Elle n'est plus disponible pour consommer un événement pendant cette exécution.

  – l'activité consommatrice crée autant d'activités que de réactions à exécuter. Les limites de la plate–forme d'exécution peuvent être rapidement atteintes en cas d'affluence d'événements.

  Notre choix est un compromis entre ces deux extrêmes. Il consiste en l'utilisation de deux activités. La première consomme les événements et les passe par l'intermédiaire d'un tampon partagé à la seconde qui fait exécuter les réactions, soit selon les deux modes précédents – création d'une tâche ou tâche instrumentée –, soit en lançant des activités au sein du pool représenté à la figure Fig. 5.1. Le nombre d'activités présentes dans ce pool est limité statiquement.

### V.2.4 Réseau de courtiers

Nous n'avons décrit pour l'instant que le fonctionnement d'un courtier isolé. Or le courtier, en tant que réalisation de la notion de domaine, évolue au sein d'un réseau d'autres courtiers, qui représentent d'autres domaines, avec lesquels il établit des relations. Ces relations sont en particulier matérialisées par la propagation d'événements de courtier à courtier. L'existence de ces relations doit être prise en compte

à la fois dans le fonctionnement et dans l'architecture des courtiers. En effet, la propagation des événements nécessite des adaptations et la représentation des relations au niveau des courtiers doit être discutée.

### V.2.4.1 Adaptation du fonctionnement

L'existence des relations impose une adaptation du fonctionnement des courtiers. Elle concerne essentiellement le traitement des événements et la gestion au niveau de chaque courtier de ses relations avec ses semblables.

Nous avons envisagé deux solutions pour ce dernier point :

1. les courtiers peuvent être, au même titre que des objets ou des applications, clients d'autres courtiers. Ainsi, si un événement doit être diffusé par un courtier vers d'autres courtiers, ceux–ci s'abonnent à cet événement auprès du courtier concerné. La réaction associée est l'opération d'émission de l'événement auprès du courtier abonné.
2. les courtiers jouent un rôle particulier par rapport aux objets et aux applications et cette particularité se retrouve dans la gestion des relations inter–courtiers. Ils sont traités à part.

L'avantage de la première approche réside dans l'homogénéité conférée au traitement des événements et de leur diffusion. Cependant, en raison des contraintes imposées par l'ordre causal sur la réception et le traitement des événements, cette solution ne peut pas être adoptée. En effet, si un événement $e_1$ n'est pas diffusé d'emblée aux courtiers reliés, les réactions engendrées peuvent poster chez ces courtiers des événements, qui dépendent causalement de $e_1$, puisqu'ils ont été engendrés par des réactions à $e_1$, mais qui seront reçus et donc traités avant $e_1$ dans ces courtiers.

La diffusion aux courtiers doit par conséquent être particularisée dans le traitement des événements. La solution que nous avons adoptée consiste à propager les événements aux autres courtiers avant toute autre opération de traitement. Ainsi, on assure qu'aucune réaction ne sera lancée avant la diffusion complète, et donc qu'aucune émission d'événement causalement dépendant ne pourra précéder cette diffusion dans l'ensemble des courtiers du système.

### V.2.4.2 Représentation de la topologie du réseau

Les données nécessaires à la diffusion sont gérées au niveau de chaque courtier et représentent les relations qu'il entretient avec d'autres courtiers. Ce sont ces données qui sont exploitées pour déterminer l'éventuelle diffusion des événements. Ce sont encore ces données qui définissent la topologie du réseau des courtiers (et donc des domaines).

Comme nous l'avons indiqué au chapitre précédent, de nombreuses questions subsistent sur la gestion de ces relations. Le prototype ne permet pas pour l'heure de définir des domaines hiérarchiques, en raison des difficultés liées à la gestion de telles relations (cf IV.5.4). Seules les relations de pair à pair entre courtiers (pour la propagation de certains événements) sont supportées.

Ces relations sont en fait établies par des abonnements entre courtiers, comme nous le suggérions à la section précédente. Cependant, ces abonnements ne sont pas gérés dans la base d'abonnements du courtier mais dans une structure indépendante. Cette séparation découle du traitement particulier qui est associé à ces relations et que nous avons présenté en V.2.4.1. Ainsi, chaque courtier s'abonne auprès des autres pour tous les événements qu'il souhaite recevoir d'eux, en associant comme réaction la méthode `Send` de la classe `EventBroker` qui permet d'émettre un événement chez un courtier.

Dans la réalisation actuelle, il existe une instance de courtier par site connecté au système. Ainsi, le critère de structuration retenu pour l'espace des domaines est un critère de localité. Chaque objet est associé au courtier de son site de création. Un courtier dont les objets souhaitent recevoir certains événements s'abonnent pour ces événements auprès des autres courtiers.

### V.2.5 Mode d'utilisation

L'utilisation du service de communication événementielle ne doit pas briser les habitudes de programmation et s'intégrer le plus harmonieusement possible au langage hôte, OC++ en l'occurrence. La solution idéale consiste donc à modifier le compilateur afin de réaliser effectivement l'intégration souhaitée. Malheureusement, ces modifications nous étaient impossibles. En conséquence, nous avons fait le choix d'utiliser les possibilités de l'héritage afin d'offrir aux classes distribuées le mécanisme de communication événementielle.

#### V.2.5.1 Utilisation de l'héritage

Une classe distribuée, dénommée `EventService`, définit et réalise par ses méthodes l'interface d'accès au mécanisme : émission, abonnement, désabonnement. Les classes qui souhaitent utiliser les événements doivent alors hériter de `EventService` pour les utiliser de manière transparente, comme s'ils faisaient partie du langage. La référence au courtier auprès duquel sont réalisées ces opérations est également gérée au niveau de la classe `EventService`.

### V.2.5.2 Éléments syntaxiques

D'autre part, la déclaration et la manipulation des structures au sein des programmes OC++ obéit à une syntaxe particulière qui simplifie le travail du programmeur, puisqu'une phase préliminaire à la compilation génère le code OC++ correspondant. Ainsi, la structure et la représentation interne des structures de déclaration est cachée au programmeur.

Le défaut de contrôle sur le compilateur nous a également empêché de mettre en œuvre une gestion complète des paramètres des événements. La prise en compte de tous les types de paramètres en dehors du schéma de compilation est une tâche dont la difficulté est sans commune mesure avec les bénéfices tirés. Aussi, nous avons restreint le nombre des types de paramètres autorisés à trois : ce peuvent être des entiers, des chaînes de caractères ou des pointeurs distribués. Les pointeurs distribués possèdent tous une structure identique qui permet de masquer le type des objets qu'ils référencent.

Les structures syntaxiques introduites concernent les trois aspects suivants :

- **la déclaration des clauses réactives et des attachements ;**

  Le programmeur définit une section particulière dans la déclaration de sa classe pour déclarer les clauses réactives et les attachements associés. Le programme suivant donne la structure d'une classe distribuée contenant une telle section.

```
distributed class aClass : public EventService
{
private:
    ...
public:
    ...
    void method1(int i);
    void method2(T_String str, dvoid *obj);
    ...
    COORDINATIONS
        CLAUSE clause1
            ON e1 INT DO method1
            ON e2 STR OBJ DO method2
        CLAUSE clause2
            ...
    END_COORDINATIONS
}
```

  Chaque clause est nommée (ici `clause1` ou `clause2`) et regroupe un ou plusieurs attachements. Les événements qui apparaissent dans les attachements sont identifiés par leur nom de famille (ici `e1` ou `e2`) et par la suite des types de leurs paramètres, dénotés **INT**, **STR** et **OBJ** pour entier, chaîne de caractères et pointeur distribué respectivement. Les signatures des méthodes prises comme réactions doivent correspondre à cette suite de paramètres.

- **l'abonnement et le désabonnement ;**

  Les opérations d'abonnement et de désabonnement sont réalisées dans le code des méthodes. L'abonnement peut prendre trois formes selon que la réaction doit être exécutée dans la tâche courante ou non et par une activité quelconque ou nouvelle. Dans l'exemple ci–dessous, la première ligne correspond à une réaction dans une tâche quelconque ou nouvelle – selon que les réactions définies dans la clause sont des méthodes ou des programmes –, la deuxième à une réaction dans la tâche courante dans un activité quelconque, et la troisième à une réaction dans une nouvelle activité de la tâche courante.

```
SUBSCRIBE clause_name
SUBSCRIBE_IN_CONTEXT clause_name
SUBSCRIBE_IN_CTXT_NEW clause_name
```

  L'opération de désabonnement offre la même distinction, selon que l'on désactive tous les attachements activés ou seulement ceux qui l'ont été dans la tâche courante.

```
UNSUBSCRIBE clause_name
UNSUBSCRIBE_IN_CONTEXT clause_name
```

- **l'émission des événements.**

  L'émission des événements est caractérisée par le mot **SEND** et prend plusieurs paramètres : nom d'événement, paramètres (types et valeurs) et éventuellement groupe de destinataires. Les types des paramètres sont dénotés par les mots **INT** pour entier, **STR** pour chaîne de caractères et **OBJ** pour pointeur distribué. Les groupes de destinataires sont définis indépendamment (conformément aux spécifications du service de désignation associative) et identifiés par un nom. L'instruction suivante réalise l'émission d'un événement dont le nom de famille est `classEvent`, avec trois paramètres de types entier, chaîne de caractères et pointeur distribué dans cet ordre, lesquels paramètres prennent les valeurs données entre parenthèses. Cette émission est réalisée vers un groupe d'objets dont le nom est `destGroup`.

```
SEND classEvent INT(1) STR("une chaine") OBJ(this)
     TO destGroup
```

Un programme qui comporte de telles structures de déclaration est traité par un filtre qui traduit les éléments ajoutés. Cette traduction consiste en un remplacement par du code OC++ qui génère les objets distribués nécessaires (événements, attachements, clauses réactives) et réalise grâce aux méthodes héritées de la classe

`EventService` les fonctions désirées (émission, abonnement ou désabonnement).

## V.3 Application : agenda coopératif et « flot de travail »

Nous avons utilisé la plate–forme OODE enrichie du prototype décrit auparavant pour réaliser une application de démonstration, qui mettrait en valeur les différents concepts que nous avons introduits. Nous décrivons cette application dans cette section, en insistant sur l'utilisation des événements.

### V.3.1 Description de l'application

L'application que nous avons mise en place comporte deux parties. La première est un agenda réparti et partagé qui permet de gérer l'emploi du temps de plusieurs personnes et la réservation de salles de réunions. La seconde est un système de gestion de commandes au sein d'une entreprise. Ces deux composantes utilisent OODE et le service de communication événementielle. Elles offrent toutes deux une interface graphique réalisée en Tcl–Tk [Ousterhout 94].

#### V.3.1.1 Agenda

L'environnement dans lequel fonctionne l'agenda que nous avons réalisé est celui d'un groupe de personnes – une entreprise, un laboratoire – qui peut être divisé en équipes. Certaines personnes possèdent des rôles particuliers, tels que responsable d'équipe. Chaque personne possède son agenda propre, mais les rendez–vous qui y figurent peuvent être partagés entre plusieurs personnes – les rendez–vous pour les réunions d'équipe par exemple. Ces rendez–vous peuvent avoir lieu dans des salles qui doivent être réservées. L'agenda assure également cette gestion.

Les fonctions de base de l'agenda sont la création, la suppression et la notification de rendez–vous et la réservation de salle. La création d'un rendez–vous correspond au protocole suivant :

> Une personne propose un rendez–vous dans lequel sont impliqués un groupe ou des individus. Cette proposition comprend un lieu, une date et une heure.
>
> La soumission du rendez–vous débute par un contrôle de la disponibilité de la salle choisie. Si elle est disponible, une pré–réservation de la salle est réalisée. Sinon, l'utilisateur peut modifier sa proposition.
>
> Après cette pré–réservation, les disponibilités des différents invités sont demandées. Si une personne est disponible, un pré–rendez–vous

est inséré dans son agenda, ce qui permet de réserver temporairement la plage proposée.

Si tous les intéressés sont disponibles, toutes les réservations sont validées. Dans le cas contraire, elles sont avortées : les pré–rendez–vous sont supprimés des agendas.

Lorsque la date d'un rendez–vous est atteinte, une notification est envoyée à tous les intéressés.

### V.3.1.2 Système de gestion de commandes

Le système de gestion de commandes que nous avons mis en place reprend les grandes lignes de l'exemple analysé en I.4. Il s'agit de traiter des commandes qui parviennent à une entreprise. Reçues par le service de la clientèle, elles suivent ensuite un circuit bien défini au sein de l'entreprise. Ce circuit est décrit à la figure Fig. 5.2. Ce service de gestion des commandes est complété par d'autres services, dont un service du personnel, qui permettent de gérer les rôles attribués aux différentes personnes de l'entreprise.

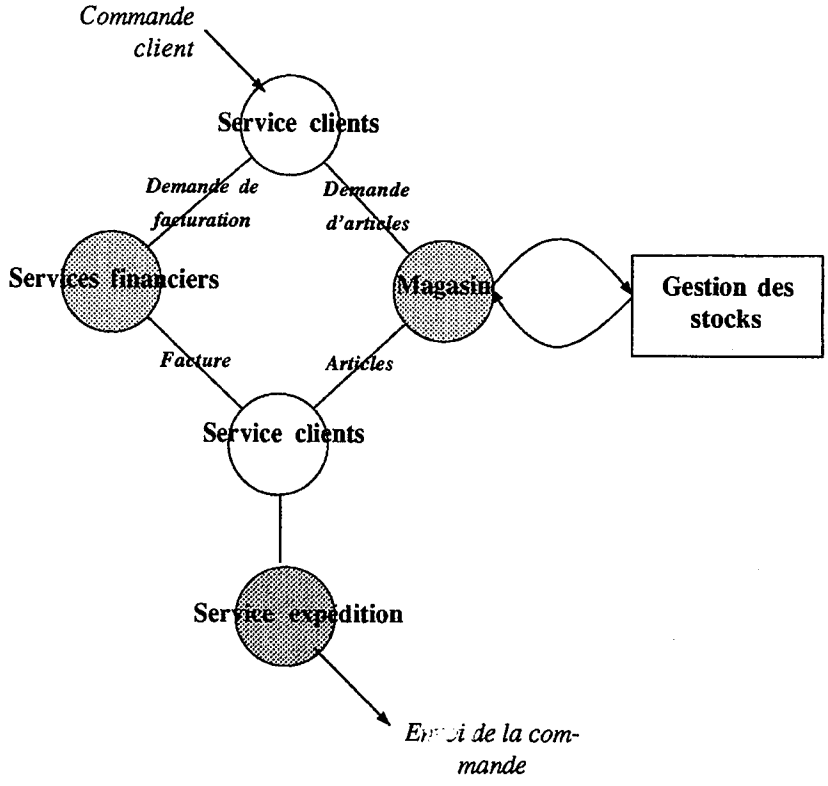


*Fig. 5.2 : Organisation du service de gestion des commandes*

Le service de la clientèle reçoit les commandes et les diffuse en parallèle aux services financiers et au magasin. Il récupère ensuite simultanément les réponses de ces deux services pour la commande en question et la fait suivre au service expédition. En cas de défaut de stock, il est immédiatement notifié par le magasin afin d'avertir le client.

Les services financiers gèrent une liste de produits et de prix leur correspondant. À partir de ces données et des commandes qu'ils reçoivent, ils éditent des factures qu'ils retransmettent au service de la clientèle, en même temps que les produits commandés.

Le magasin gère les stocks de produits. Lorsqu'une commande lui parvient, il examine l'état du stock pour le produit considéré. Si la commande peut être honorée, il prépare les produits concernés et les transmet au service de la clientèle en se synchronisant avec l'envoi de la facture par les services financiers. Sinon, il notifie le service de la clientèle d'un défaut de stock.

Enfin le service expédition reçoit les commandes avec factures et articles et envoie le tout au client.

Chaque service peut fonctionner selon trois modes.

Le premier est le mode déconnecté : le service n'est tenu par personne et est donc inactif. Dans ce cas, les commandes sont mises en file au niveau de ce service en attendant son activation et aucun traitement n'est effectué. Cette mise en file s'accompagne d'une éventuelle notification aux autres services.

Le deuxième mode est un mode manuel : le service est actif, les commandes sont reçues et traitées, mais l'opérateur du service doit réaliser ce traitement à la main.

Le troisième mode est un mode automatique : le service est actif, les commandes sont reçues et traitées automatiquement. L'opérateur n'est qu'averti de la progression du traitement.

Afin d'illustrer ces trois modes, nous considérons le traitement appliqué par le service financier à une commande. Il consiste simplement à calculer le montant de la facture en fonction du nombre d'articles commandé et du prix unitaire de ces articles. Dans le mode déconnecté, la commande n'est pas modifiée. Dans le mode manuel, l'opérateur de service détermine de manière externe à l'application le montant. Il affecte ensuite le champ correspondant à ce montant dans l'objet qui représente la commande. Dans le mode automatique, le montant est calculé par l'application (multiplication de la quantité par le prix unitaire) et affecté au champ correspondant sans intervention de l'opérateur.

#### V.3.1.3 Organisation logicielle

Chacune de ces deux applications met en jeu des classes et objets distribués.

Dans l'agenda, les rendez–vous sont des objets distribués qui contiennent la description complète du rendez–vous : date, heure, lieu, participants.

Les utilisateurs (participants potentiels aux réunions), les groupes d'utilisateurs et les lieux de réunion sont également représentés par des objets distribués, de types différents. Ces types possèdent en commun une liste de rendez–vous qui caractérisent l'agenda de chaque entité : liste de rendez–vous pour un individu, liste de réunions pour un groupe, liste des plages horaires occupées pour une salle. Ainsi, ces classes héritent d'une même classe qui décrit des objets possédant une liste de rendez–vous.

De même, chacun des différents services du système de gestion est représenté par une classe distribuée. Au moment de l'exécution, un employé de l'entreprise, représenté par un objet, exécute une méthode sur l'objet représentant le service à lancer. Si l'employé est habilité, le service fonctionne dans une tâche. Sinon, le service n'est pas activé.

Ces différentes classes sont présentées à l'annexe <Annexe>.

### V.3.2 Utilisation des événements

Les événements servent à réaliser les interactions découplées entre composants et les notifications de rendez–vous. Le mécanisme d'abonnement et de désabonnement permet également aux composants de l'application flot de travail de passer d'un mode de fonctionnement à l'autre sans avoir à avertir leurs interlocuteurs, et donc de manière transparente.

Pour chacune des deux parties de l'application, nous montrons comment apparaît au niveau du code effectivement écrit l'utilisation des événements et comment les mêmes fonctions pourraient être réalisées sans événements avec OC++.

#### V.3.2.1 Agenda

**Programmation événementielle**

L'agenda emploie les événements pour deux tâches. La première est la notification d'une confirmation de rendez–vous aux différents invités. Après la phase de négociation et en cas de succès, un événement est envoyé pour signifier la validation de la négociation de rendez–vous. Cet événement est nommé `ack_RdvEvt` et a pour paramètre la référence de l'objet représentant le rendez–vous. D'autre part,

son émission comporte un filtre qui indique que seuls les invités de la réunion doivent recevoir cet événement. La définition de ce filtre est faite sur la valeur d'un attribut dynamique que ces invités enregistrent auprès du service de désignation associative. À la réception, la validation du rendez–vous est réalisée par les objets dans deux types de contextes différents :

- dans un contexte quelconque si l'objet représente un groupe, un lieu de réunion ou un utilisateur dont l'application « Agenda » ne fonctionne pas : aucun programme (contexte) n'est associé à ces objets qui sont « passifs » et seul leur état doit être modifié.
- dans le contexte où s'exécute l'agenda si l'objet représente un utilisateur dont l'application « Agenda » est ouverte : outre la modification de la liste de rendez–vous, l'apparence de l'agenda sur l'écran de l'utilisateur doit prendre en compte le nouveau rendez–vous.

La seconde utilisation est pour la notification des rendez–vous. Lorsque l'échéance d'un rendez–vous est atteinte, un événement `time_RdvEvt` est adressé à l'ensemble des invités répertoriés dans l'objet correspondant, avec la référence de cet objet en paramètre. La réaction est exécutée dans le contexte de l'application « Agenda » des utilisateurs cibles.

La communication événementielle est donc uniquement utilisée dans ce cadre pour réaliser une diffusion sélective non bloquante.

Cette utilisation a nécessité les seules adjonctions suivantes au code des classes :

- Opérations d'émission des événements `ack_RdvEvt` et `time_RdvEvt`.
- Définition d'une clause réactive pour la validation de rendez–vous dans les classes disposant d'une liste de rendez–vous et opérations d'abonnements correspondantes.

  L'utilisation de l'héritage a permis de ne déclarer qu'une seule clause réactive dans une super–classe dont les trois classes concernées (« Utilisateur », « Groupe » et « Lieu de réunion ») héritent.
- Définition d'une clause réactive pour la notification de rendez–vous pour les classes « Utilisateur » et « Groupe » et opérations d'abonnement correspondantes.

### Programmation sans événements

Pour réaliser ce type de fonctionnement avec OC++, les problèmes rencontrés auraient été les suivants :

- *Comment notifier une validation de rendez–vous à l'ensemble des objets concernés ?*

  Il existe deux solutions.

  La première consiste à exécuter une méthode adéquate sur chacun des objets correspondants. Ceci oblige donc à gérer pour chaque négociation de rendez–vous une **liste de couples objet–méthode** caractérisant les objets à contacter avec la méthode à appeler. La gestion peut être simplifiée si la méthode est identique pour tous les objets. Le parallélisme des appels ne peut être réalisé que par l'intermédiaire de la création **explicite** d'activités OODE, avec toute la charge de gestion que cela comporte.

  La seconde solution consiste à utiliser un **espace partagé** dans lequel la notification est déposée et où des activités associées aux différents objets viennent lire les messages qui les concernent. Ce fonctionnement « à la Linda » nécessite la mise en place d'une gestion associative de cet espace.

- *Comment prendre en compte dynamiquement les changements au niveau de l'interface de l'application ?*

  Les changements dans certains objets, notamment ceux qui représentent les utilisateurs, doivent être répercutés au niveau de l'interface de l'application. La solution consiste à **scruter** régulièrement les objets concernés pour détecter d'éventuels changements. Cette vérification doit être dévolue à une activité de la tâche représentant l'application. Cette activité peut alors agir sur l'interface. L'activité en question peut être soit l'activité principale de la tâche (celle qui correspond à l'exécution de la fonction `main`) soit une activité parallèle à ce flot d'exécution principal. Dans l'un et l'autre cas, le programmeur doit gérer la vérification régulière des objets et le traitement consécutif à un changement.

#### V.3.2.2 Système de gestion de commandes

**Programmation événementielle**

Les différents services de cette partie de l'application possèdent comme nous l'avons indiqué plusieurs types de fonctionnement possibles. Chaque type de service est représenté par une classe et chaque classe comprend la définition de clauses réactives qui caractérisent chacune un type de fonctionnement.

L'arrivée des commandes dans un service est caractérisée par la réception d'un événement. Le nom de l'événement est fonction du service d'où provient la commande, mais il a toujours pour paramètre la référence de l'objet commande correspondant.

En fonction du mode de fonctionnement du service, la réaction correspondante est exécutée. Si le mode est le mode déconnecté, l'exécution a lieu dans un contexte quelconque. Dans le cas contraire, une tâche représente le service actif et l'exécution de la réaction a lieu dans cette tâche : elle agit sur l'état de l'objet représentant le service et sur l'interface de l'application.

Comme pour l'agenda, le code comprend l'adjonction d'opérations d'émission et d'abonnement/désabonnement. La définition des clauses réactives tire également parti de l'héritage : certains services réalisent les mêmes actions en mode inactif.

La gestion du passage d'un mode de fonctionnement à un autre est réalisé par un désabonnement de la clause réactive représentant le mode quitté et par un abonnement à la clause réactive représentant le nouveau mode.

Le cas de la synchronisation entre le service financier et le magasin pour finaliser le traitement de la commande par le service de la clientèle a également été l'occasion de mettre en œuvre la reconnaissance d'événements composites. En effet, chacun des deux services (service financier et magasin) représenté par un objet de classe `Store` et `FinancialService` respectivement émet son propre événement lorsqu'il a achevé avec succès le traitement d'une commande. Ces événements sont nommés `store_FullfilEvt` et `financial_BillEvt`. Afin d'éviter un double traitement au niveau du service de la clientèle, l'idée est d'émettre un unique événement composite lorsque les deux événements concernant une même commande ont été émis.

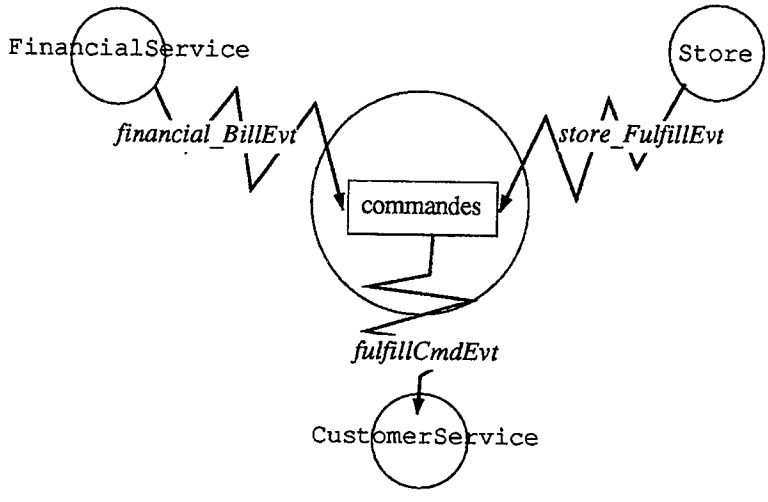


*Fig. 5.3 : Principe de la détection des événements composites*

Le principe de la réalisation est illustré à la figure Fig. 5.3. Il consiste en l'utilisation d'un objet intermédiaire qui recevra les événements `store_FullfilEvt`

et `financial_BillEvt` et gérera une liste de commandes pour lesquelles un de ces événements et un seul est déjà arrivé. À l'occurrence du second, la commande correspondante est enlevée de la file et l'événement `fulfillCmdEvt` auquel est abonné le service de la clientèle est émis. L'événement composite détecté est ici une conjonction de deux événements, mais la détection d'autres combinaisons est possible. Il suffit pour cela de programmer une classe d'objets de reconnaissance adaptée.

**Programmation sans événements**

Par rapport à l'agenda que nous avons discuté en V.3.2.1, la réalisation en OC++ de cette partie de l'application pose des problèmes nouveaux qui sont relatifs à la représentation de l'organisation.

L'organisation est caractérisée par ses lignes de communication entre services. Si on n'utilise que OC++, ces lignes sont soient des objets partagés entre plusieurs services et à travers lesquels ils communiquent, soit des références vers les objets qui représentent les services. Ainsi, le service de la clientèle pour communiquer une commande au magasin peut soit la déposer dans une boîte aux lettres partagée où le magasin pourra la consommer, soit appeler une méthode sur l'objet représentant le magasin pour lui déposer la commande.

Outre l'inconvénient que constitue le caractère synchrone des interactions, ces deux approches posent un problème dès lors que l'on souhaite faire évoluer l'organisation en modifiant les lignes de communication. Ceci suppose de modifier les références des objets utilisés pour communiquer. Une communication anonyme pour le programmeur résout ce type de problèmes.

Enfin, la gestion de l'exécution des traitements doit être décrite explicitement par le programmeur. Si le parallélisme des traitements fait partie des choix de conception, l'exécution de plusieurs traitements en parallèle implique la création et la gestion explicite et à la charge du programmeur des activités correspondantes.

### V.3.3 Conclusion

La réalisation de cette application simple a permis de mettre en évidence les points suivants :

- La communication mise en œuvre est effectivement asynchrone et réalise un vrai découplage entre émetteur et récepteur(s).
- L'évolution dynamique des réactions permet d'adapter rapidement le fonctionnement d'un objet sans nuire au fonctionnement de l'ensemble.

- La conception des interactions à base d'événements est rapide et relativement simple. En particulier, il nous a semblé que l'évolution à partir d'un code existant, non pensé en terme d'événements, était relativement aisée et ne nécessitait pas de bouleversements radicaux du code des classes.
- Les performances du mécanisme souffrent de sa faible intégration avec la plate–forme support (compilateur et machine d'exécution). Ce coût se retrouve à la fois dans la gestion des structures du courtier, dans le coût CPU de son fonctionnement et dans l'utilisation du service de désignation associative.

Nous avons également mis en évidence l'apport de ce mécanisme par rapport à l'utilisation de OC++ seul. Cet apport se traduit dans les points précédents sur l'asynchronisme et la conception rapide, mais le point crucial demeure la gestion des structures d'exécution. L'utilisation de la communication événementielle affranchit le programmeur d'une partie de la gestion des activités nécessaires au fonctionnement de son programme.

## V.4 Synthèse

Les conclusions que nous pouvons tirer de ces réalisations sont de deux ordres : elles concernent d'une part l'utilité et l'efficacité du mécanisme en tant que tel, et d'autre part sa réalisation en plus forte intégration avec la plate–forme support.

### V.4.1 Analyse du mécanisme

L'évaluation du mécanisme dans la proposition actuelle s'articule selon nous autour des quatre pôles suivants :

- la satisfaction des objectifs fixés ;
- l'apport du mécanisme pour un modèle tel que Guide ;
- le niveau d'intégration dans le modèle hôte ;
- la généralité du mécanisme ;
- la comparaison avec l'existant.

#### V.4.1.1 Satisfaction des objectifs fixés

Le mécanisme tel qu'il peut être utilisé par le programmeur possède presque toutes les caractéristiques que nous souhaitions mettre en œuvre (cf I.6.2.2).

**Il est asynchrone** : l'émission d'un événement est une opération qui ne peut pas être bloquée à cause des récepteurs. Seules les limitations du système (du courtier) peuvent bloquer l'émission. À l'autre extrémité de la communication, aucune

opération explicite ne doit être réalisée pour concrétiser la réception : dès lors qu'un attachement est actif, la réception est automatique.

**Il est anonyme** : le programmeur n'est pas contraint de spécifier dans son code de destinataires pour les événements qu'il envoie.

**Il réalise une diffusion** : tout événement émis est diffusé indifféremment à tous ceux qui peuvent le recevoir. La portée de cette diffusion peut être limitée par les frontières d'un domaine, mais il est possible de passer outre, par le biais des abonnements entre courtiers. La désignation associative constitue un moyen d'affiner encore la portée.

**Il fonctionne dans un environnement réparti** : cette caractéristique lui est conférée par la gestion transparente de la répartition dans OODE.

**Il permet la définition de comportements réactifs** : la réception d'un événement se traduit toujours par l'exécution d'une réaction. De plus, les comportements réactifs peuvent évoluer par le biais des opérations d'abonnement et de désabonnement. La définition de clauses réactives qui regroupent plusieurs attachements permet enfin de décrire un comportement événementiel global pour chaque objet.

L'intégration avec le langage est le point le moins satisfaisant de la réalisation actuelle. Les points positifs sur ce plan résident à notre sens dans la conservation de la classe comme seul conteneur de déclaration et dans la simplicité de déclaration des familles d'événements. Cependant, si l'utilisation d'artifices syntaxiques permet de masquer la plus grande partie des structures et opérations sous–jacentes, elle ne cache cependant pas le manque d'homogénéité avec les autres aspects du langage. Ce manque se traduit dans le style des constructions mises en place et surtout dans la gestion des paramètres. Le fait de ne pas pouvoir utiliser l'ensemble des types fournis par le langage pour typer les paramètres des événements nuit sans conteste à l'utilisation du mécanisme.

Nous avons déjà justifié ce défaut d'intégration par la non–maîtrise de l'outil de compilation. La modification de la grammaire et du compilateur du langage OC++ pour prendre en compte ce nouveau mécanisme permettrait de remédier à cet état de fait. Elle permettrait également de mettre en place des contrôles statiques plus stricts et de définir et utiliser pleinement des interfaces d'événements telles que nous les avons discutées en III.6.2.

#### V.4.1.2 Apports à Guide

Nous avons montré en V.3.2 comment les événements avaient été utilisés et ce qu'ils permettaient dans un environnement à la Guide tel que OODE avec OC++.

Nous avons en parallèle montré comment ces applications auraient pu être réalisées sans l'appoint de la communication événementielle dans cet environnement.

L'apport principal réside dans la gestion transparente de schémas d'exécution complexes. En effet, l'exécution de réactions à l'occurrence d'événements permet de réaliser des traitements en parallèle dans des tâches sans que le programmeur ait à se soucier de la gestion des activités correspondantes. De même la prise en compte dynamique de changements dans des objets partagés ne nécessite plus la mise en place d'activités scrutatrices par le programmeur.

Le caractère anonyme de la communication est également un plus puisqu'il permet de rendre indépendantes les deux extrémités de la communication. La gestion de références vers d'autres objets est allégée, ce qui limite les facteurs de dépendance.

La diffusion que permet de réaliser la communication événementielle offre ensuite un moyen simple de réaliser en parallèle des traitements sur un grand nombre d'objets, sans avoir à gérer explicitement de listes de références et d'appels de méthodes sur ces objets.

### V.4.1.3 Intégration

Nous avons vu au chapitre III combien est large la gamme de mécanismes de communication événementielle qui peuvent être intégrés dans un modèle à objets. Les principales différences qui distinguent ces mécanismes sont leur degré d'intégration dans le modèle hôte et la complexité de leur modèle d'exécution.

L'intégration dans le modèle hôte se mesure aux indices suivants :

- **la nature des événements et des familles ;**

  Un mécanisme très intégré définira les événements comme des objets et les familles comme des classes. L'héritage pourra alors être appliqué aux familles. Nous avons vu en III.2.4 quelles contraintes cela imposait.

  À l'opposé, on peut définir les familles indépendamment des classes en leur donnant une existence presque fictive par la déclaration implicite. Les événements ne sont alors pas des objets.

- **la présence ou l'absence des événements dans les interfaces des classes ;**

  Un mécanisme très intégré présentera dans l'interface des classes des éléments de description du comportement événementiel des classes : familles des événements émis, familles des événements reçus, familles des événements définis. Ces éléments constituent une

base pour la réutilisation et la description du comportement des classes.

À l'opposé, ces éléments peuvent être totalement absents de l'interface.

- **l'impact des événements sur les contrôles statiques appliqués au langage.**

  Les éléments intégrés au langage peuvent faire l'objet de contrôles statiques. En particulier, la présence dans l'interface d'éléments de description du mécanisme peut entraîner la vérification de l'adéquation de la réalisation de la classe avec son interface. Cette présence peut également être prise en compte dans l'évaluation de la conformité entre classes. Lorsqu'un événement est défini, on peut également contrôler que cette définition a bien lieu au sein de la portée de définition de la famille utilisée.

  À l'opposé, les contrôles peuvent être réduits aux seuls éléments syntaxiques.

Toutes les variations sont possibles le long de ces trois échelles. Pour notre part, le critère déterminant a été la simplicité, dans la limite des objectifs que nous nous étions fixés en I.6.2.2. Ce critère nous a conduit à choisir un faible degré d'intégration. La raison principale de ce choix est d'ordre pratique. L'effort de mise en œuvre nécessaire à la mise en place d'un mécanisme fortement intégré nous a semblé inutile pour atteindre nos objectifs. De plus, cette intégration et ce qu'elle suppose de structures supplémentaires alourdit considérablement le modèle.

La complexité du modèle d'exécution se traduit par une gamme de modes d'exécution pour les réactions. Cette complexité est fortement tributaire des possibilités du modèle d'exécution du modèle hôte ou de son support d'exécution. Les principaux indices de complexité sont les suivants :

- **le support de la concurrence**

  L'exécution concurrente au sein d'un même objet permet des accès en parallèle à un même objet.

- **la détermination du contexte d'exécution**

  Le choix du contexte d'exécution pour une réaction permet de mettre en place des schémas qui respectent la protection et la confidentialité, et permettent de contrôler l'exécution d'applications.

- **la création dynamique de flots de contrôle**

  La création dynamique de flot permet de créer de nouveaux flots pour l'exécution de réactions.

OODE (Guide) offre sur chacun de ces trois points des possibilités étendues que nous avons utilisées afin d'offrir un modèle d'exécution le plus complet possible.

Nos choix sont donc principalement guidés par une assistance au programmeur, caractérisée par une syntaxe supplémentaire minimale et par la gestion transparente de modes d'exécution complexes à mettre en œuvre dans un cadre tel que OODE.

### V.4.1.4 Généralité du mécanisme

La question se pose maintenant de savoir si le mécanisme tel que nous l'avons défini est généralisable et peut être appliqué dans d'autres environnements. Nous avons donné des éléments de réponse en IV.6.1. Les éléments à prendre en compte sont d'une part les aspects déclaratifs et d'autre part le modèle d'exécution.

Les structures de déclaration introduites ne présentent aucune particularité propre à Guide ou OODE. La seule difficulté réside dans leur traduction au niveau de la plate–forme d'exécution. Ce problème se pose plus particulièrement pour les clauses réactives. Nous avons adopté dans notre réalisation le choix de les représenter comme des objets persistants, donc en dehors du code des classes. Cette solution n'est pas envisageable dans certains cas, pour lesquels l'intégration de ces clauses dans le code des classes devrait être plus forte. Les performances d'accès au service n'en seraient qu'accrues.

Le modèle d'exécution constitue le facteur le plus limitant pour la portabilité du mécanisme. Notre proposition utilise en effet pour l'exécution des réactions des caractéristiques de Guide telles que la gestion de la concurrence ou les notions d'activité et de tâche et des fonctions de gestion associées que beaucoup de modèles n'offrent pas. Dans de telles circonstances, notre proposition doit être profondément modifiée.

Il apparaît donc que le facteur le plus limitant à la portabilité d'un mécanisme de communication événementielle conçu pour un modèle à objets réside dans le modèle d'exécution des réactions. En effet, beaucoup plus que les aspects déclaratifs, il est tributaire de la plate–forme d'exécution et de ses possibilités.

### V.4.1.5 Comparaison avec l'existant

Si nous nous référons aux différents travaux autour des modèles à objets que nous avons présentés en II.4, les points de comparaison sont les suivants :

- **Intégration au modèle à objets ;**

  Les principales différences avec les travaux étudiés résident dans la déclaration des événements et dans la gestion des attachements.

  Dans [Menon 93b], les événements ne sont que des signaux nommés qui n'ont pas de paramètres et enregistrés auprès du système par chaque application qui les définit. Leur intégration au modèle est donc faible.

  Dans [Bacon 95], les événements sont des objets qui, instances d'une classe, doivent être instanciés. La portée de la définition d'une classe d'événements est limitée à son serveur de déclaration. L'instanciation équivaut comme dans notre cas à l'émission. L'intégration du mécanisme avec le modèle à objets est donc forte du point de vue de la nature des événements et des familles.

  Les événements décrits dans [Starovic 95] sont instances de familles qui sont déclarées globalement et au même niveau que les classes. Ces événements apparaissent dans les interfaces des classes – événements émis ou reçus – et leur émission est subordonnée à leur présence dans cette interface. À un niveau différent de celui proposé dans [Bacon 95], le mécanisme est intégré avec le modèle hôte : les événements apparaissent dans l'interface des classes et la correspondance entre interface et réalisation de la classe est vérifiée.

  Les aspects déclaratifs relatifs à l'invocation implicite sont longuement discutés dans [Notkin 93] et les questions que nous nous sommes posées au chapitre III y sont abordées. Dans l'application à C++, les événements sont instances d'une classe et déclarés statiquement dans l'interface des classes qui peuvent les émettre. Les autres classes peuvent alors déclarer leur intérêt pour ces événements « publiés ».

- **Modèle d'exécution.**

  Grâce à la richesse des structures d'exécution de OODE, le modèle d'exécution associé à notre mécanisme offre des possibilité élargies par rapport à celle décrites dans les travaux cités.

  Nous avons insisté sur le fait que la description du modèle d'exécution constitue le point clé du mécanisme, puisqu'il en détermine les possibilités, mais que les caractéristiques de la plate–forme peuvent le limiter.

Dans [Menon 93b], le modèle d'exécution est très simple : le traitant est toujours exécuté par un nouveau flot. Néanmoins, la proposition permet d'associer des traitants à des entités actives (les « threads ») et donc d'agir sur leur environnement, ce que ne permet pas notre réalisation.

Le modèle client–serveur est de mise dans [Bacon 95] : les structures d'exécution sont des processus qui définissent eux–mêmes le mode d'exécution de leur réactions. Une grosse part de la gestion de l'exécution est donc laissée au programmeur.

Le peu de description du modèle d'exécution des propositions faites dans [Starovic 95] ne nous permet pas d'y comparer notre approche. Cependant, aucune mention n'est faite de plusieurs modes différents pour l'exécution des réactions. On peut penser que là encore le programmeur a la charge de gérer explicitement les structures d'exécution s'il souhaite mettre en place des schémas complexes.

De même, la description faite dans [Notkin 93] ne s'attache pas à ces aspects.

### V.4.2 Réalisation

Le prototype décrit dans ce chapitre offre les fonctions désirées telles que nous les avons décrites au chapitre IV. Il constitue donc une base valide d'évaluation de l'utilité du mécanisme. Cependant cette évaluation ne peut être que qualitative, car l'implémentation actuelle souffre de défauts rédhibitoires. Les raisons de ces défauts sont à chercher dans la faible intégration du mécanisme avec les structures d'exécution de la plate–forme OODE. Comme pour les aspects relatifs à la compilation, cette faible intégration nuit aux performances.

L'exemple le plus frappant de cette nuisance se trouve dans la gestion de la synchronisation au niveau du courtier. Le courtier n'effectue des traitements que lorsque des événements lui sont adressés. Dans le cas contraire, il est bloqué. Pour éviter une attente active, nous avons utilisé les sémaphores fournis dans OODE. Mais la réalisation de ces sémaphores est actuellement inefficace et affecte le fonctionnement du courtier : ce dernier est toujours actif même quand il ne réalise pas de traitement, ce qui induit de la charge sur le processeur et nuit aux performances.

L'intégration des fonctions du courtier dans les structures d'exécution de OODE permettrait de réduire ces nuisances. Ces structures d'exécution comprennent en particulier un processus démon, dont le rôle est de gérer les communications entre sites distants. C'est la seule entité active propre au système OODE. Ces attributions pourraient être étendues pour prendre en compte les événements.

Ceci permettrait notamment d'offrir au niveau de la machine d'exécution une primitive d'émission d'événement non réalisée par un appel de méthode.

# Conclusion

## 1 Rappel du cadre et des objectifs de travail

Le travail présenté dans ce document a trait aux mécanismes nécessaires au support des interactions entre composants dans les applications coopératives. Cette étude a été menée dans le cadre plus particulier des modèles à objets, ce qui nous a amené à nous concentrer sur un mode d'interaction particulier, celui de la communication asynchrone et anonyme.

Les motivations de ce travail sont à l'origine pragmatiques : il s'agissait d'obtenir rapidement un cadre adapté à la programmation des applications coopératives. Ce cadre devait être construit sur la base offerte par Guide, un modèle à objets répartis, persistants et partageables. L'intégration d'un mode de communication asynchrone et anonyme à cette base de travail constituait donc notre objectif premier.

Cet objectif nous a conduit à articuler notre travail autour de deux axes.

Dans un premier temps, nous avons élargi le contexte de l'étude, en étudiant un tel modèle de communication dans le cadre des modèles à objets en général. Cette étude comportait trois aspects :

1. définir les éléments du mécanisme et le principe de son fonctionnement,
2. examiner comment ces éléments pouvaient être exprimés et intégrés au sein d'un modèle à objets et
3. étudier le modèle d'exécution qui régissait le fonctionnement du mécanisme.

Le second axe concernait donc l'application et la réalisation de cette proposition pour le modèle Guide. La réalisation de cet objectif passait par l'adaptation à un environnement particulier des propositions faites auparavant dans le cas général et par la mise en œuvre d'un prototype d'évaluation.

## 2 Rappel des résultats obtenus

Les résultats que nous avons obtenus concernent deux points : l'étude d'un mode de communication asynchrone et anonyme reposant sur les événements dans le cadre des modèles à objets d'une part, et l'application de cette étude dans le cadre d'un modèle à objets répartis, persistants et partageables.

## 2.1 Communication événementielle

Nous avons précisé dans un premier temps les caractéristiques que devait posséder le mécanisme de communication asynchrone et anonyme dont nous avions besoin. Ces caractéristiques sont au nombre de six : le mécanisme devait être **asynchrone** et **anonyme**, il devait permettre la **diffusion** d'information, fonctionner dans un environnement **réparti**, reposer sur la définition et l'évolution de **réactions** des récepteurs et enfin s'**intégrer** à un modèle à objets.

L'analyse de travaux existants a montré que l'utilisation d'événements couplée à un mécanisme d'attachement offre des possibilités conformes aux besoins énoncés : des entités s'abonnent à certains types d'événements en y attachant des réactions à exécuter et l'occurrence de ces événements déclenche l'exécution des réactions attachées.

Nous avons alors défini les différents éléments de la communication événementielle dans le cadre des modèles à objets passifs en général. Les éléments et caractéristiques mis en évidence sont les suivants :

- les **événements** sont émis à l'occasion de changements dans le système. Ce sont des messages paramétrés construits sur le modèle d'une **famille**.
- les **réactions** sont des programmes exécutés à l'occurrence d'événements auxquels elles ont été attachées.
- les **attachements** décrivent des associations entre un événement et une réaction à exécuter à l'occurrence de l'événement.
- le **modèle d'exécution** définit, outre l'**asynchronisme** de l'émission, le **contexte** dans lequel les réactions sont exécutées. Ce contexte comprend notamment la définition du **flot d'exécution** chargé de la réaction et le moment où l'exécution aura lieu.

Cette étude a mis en lumière des invariants qui demeuraient, quel que soit le modèle à objets hôte. Au delà de ces invariants, le concepteur d'un tel mécanisme possède dans le cadre d'un modèle à objets des alternatives qui permettent d'adapter au mieux le mécanisme à ses besoins. Ces alternatives concernent notamment le modèle d'exécution et les aspects déclaratifs du mécanisme.

Enfin, nous avons montré que l'intégration des événements au sein des modèles à objets n'altère pas les caractéristiques canoniques de ces derniers.

## 2.2 Application

Dans un second temps, nous avons appliqué nos propositions à un modèle à objets particulier. Ce modèle, dénommé Guide, propose des objets passifs, persistants, partageables et répartis et à ce titre, il satisfait une partie des besoins des applications coopératives. En revanche, le seul mode de communication offert

est l'appel de méthode synchrone et le partage d'objets entre activités (flots de contrôle). Guide était par conséquent un cadre adapté à l'extension que nous projetions.

Les caractéristiques de Guide ont fortement influé sur la description du mécanisme. Si la gestion transparente de la répartition permet de s'affranchir de cet aspect dans l'éventualité d'une adaptation à un autre système, il n'en est pas de même pour le partage et la persistance des objets. Ces caractéristiques pèsent en effet sur les choix de conception.

Ces propositions ont été ponctuées par la réalisation d'un prototype et d'une application qui l'utilise. L'intérêt de ces travaux de mise en œuvre était d'évaluer l'utilité du mécanisme et son adéquation aux besoins. Les résultats de cette évaluation fonctionnelle sont satisfaisants, puisque d'une part le mécanisme fournit les fonctions désirées et que d'autre part son utilisation est relativement simple.

Cependant, le prototype actuel souffre de sa faible intégration avec la plate–forme support, tant au niveau du schéma de compilation et du langage qu'au niveau des structures d'exécution.

## 3 Évaluation

Les apports de notre travail se situent sur deux plans.

Tout d'abord, nous avons réalisé une étude complète sur les mécanismes de communication à base d'événements dans les modèles à objets passifs. Cette étude a abordé les problèmes relatifs :

- à la définition des entités manipulées,
- à leur déclaration au sein d'un modèle à objets,
- au modèle d'exécution que respecte le mécanisme, et
- à l'adéquation de ce mode de communication avec les caractéristiques des modèles à objets.

Pour ces différents points nous avons apporté des réponses générales qui peuvent être adaptées dans des contextes variés. Ces adaptations sont commandées par des choix à réaliser et que nous avons détaillés. L'approche a été validée par l'application de cette étude à un cas particulier, ce qui nous a amené à choisir parmi les différentes alternatives laissées.

En second lieu, cette application a permis la définition d'un mécanisme dans le cadre d'un modèle à objets passifs, répartis, partageables et persistants dont les apports sont les suivants :

- **Contrôle du comportement des objets**

  La notion de clause réactive permet de définir et contrôler le comportement des objets vis–à–vis des événements. Ceci permet de synthétiser en une unité un ensemble d'attachements qui seraient autrement disséminés dans le code des classes.

- **Modèle d'exécution**

  Le mécanisme proposé offre une gamme de quatre modes d'exécution différents pour les réactions, ce qui permet une grande variété d'utilisations : lancement asynchrone de nouveaux programmes, maintien en cohérence de données, contrôle d'applications, contrôles d'interface, etc.

  Nous avons à cette occasion montré combien le modèle d'exécution était un point clé du mécanisme et que sa description était contrainte par les possibilité le modèle hôte et/ou de son support d'exécution.

## 4 Perspectives

Les propositions que nous avons faites au cours de cette thèse ouvrent des perspectives multiples et peuvent déboucher sur des prolongements intéressants. Ces perspectives concernent à la fois l'étude générale du chapitre III et la proposition pour Guide des chapitre IV et V.

Concernant l'étude du chapitre III, les problèmes suivants méritent un traitement ultérieur :

- **Événements et interfaces**

  Nous avons indiqué que les événements pouvaient sous différentes formes apparaître dans les interfaces des classes, soit comme événements émis, soit comme événements pouvant être reçus. L'étude mérite d'être complétée notamment pour ce qui concerne l'utilisation d'une telle interface. En particulier, la prise en compte des événements dans l'évaluation de la conformité entre classe mérite un traitement plus ample. La vérification que les opérations événementielles au sein de la classe sont bien conformes à son interface constitue encore un point à éclaircir. Nous reviendrons sur ce point un peu plus loin.

  De même, de telles interfaces permettraient de générer des informations sur les événements qui peuvent circuler dans le système. La définition d'outils d'extraction de ces informations

offrirait un support appréciable au programmeur à la fois pour la réutilisation de classes mettant des événements en jeu et pour la compréhension de ses programmes.

- **Événements et portée**

  Les problèmes de **portée** des événements à l'émission n'ont été que peu abordés au chapitre III. Nous avons donné deux moyens parmi d'autres pour limiter cette portée aux chapitres IV et V (notion de domaine et désignation associative), mais une étude plus exhaustive de ces problèmes permettrait d'adopter des solutions adaptées à chaque cas pour ce problème crucial dans un environnement de grande envergure.

- **Événements composites**

  Nous avons montré en V.3.2.2 comment simuler la détection d'événements composites à l'aide d'objets intermédiaires. Intégrer cette détection dans le mécanisme irait dans le sens d'un meilleur service au programmeur, en l'affranchissant de la programmation de ces objets intermédiaires. Les problèmes que posent les événements composites résident d'une part dans leur description et les aspects déclaratifs qui y sont relatifs, d'autre part dans les structures d'exécution mises en place pour leur détection.

L'évolution de la proposition du chapitre IV et du prototype correspondant pourrait suivre les axes suivants :

- **Intégration à la plate-forme OODE**

  Nous avons maintes fois déploré ce défaut d'intégration qui a conduit à des solutions peu satisfaisantes tant du point de vue du langage que du point de vue de la machine d'exécution. Remédier à ce défaut constituerait donc un prolongement naturel de notre travail.

  Intégrer les structures de déclaration au schéma de compilation de OC++ permettrait plusieurs améliorations du mécanisme. La première consiste en la prise en compte de tous les types pour les paramètres des événements.

  Une deuxième amélioration concerne encore les paramètres et leur traitement ; elle consiste à autoriser la manipulation des paramètres des événements afin de les adapter aux signatures des réactions. Ainsi, il serait possible de modifier l'ordre des paramètres, de leur appliquer des opérations, etc.

Enfin une troisième amélioration renvoie à ce que nous mentionnons plus haut à propos des événements et des interfaces

- **Domaines**

  La notion de domaine constitue une unité naturelle de limitation de la portée des événements. Cependant, si nous l'avons introduite dans nos propositions, son exploitation est encore limitée. Les aspects qui y sont relatifs et qui méritent une étude complémentaire sont les suivants :

  - la manipulation des domaines au niveau du langage : comment cette manipulation peut-elle être exprimée sans surcharger le langage, en particulier au niveau des opérations événementielles.
  - la structuration des applications : les domaines peuvent être des unités de structuration, mais leur utilisation comme tel doit reposer sur une étude plus poussée que celle du chapitre IV.

Deux derniers axes d'étude doivent encore être mentionnés.

Le premier concerne l'adaptation du mécanisme général décrit au chapitre III à d'autres modèles. Nous avons donné quelques éléments de réponse sur les problèmes ou particularités que pourraient survenir en analysant la proposition pour Guide, mais une étude complète reste à faire.

Si nous considérons l'exemple de C++ dans un environnement Unix, en prenant pour base les propositions faites au chapitre IV, les principales adaptations à mettre en œuvre sont relatives à la fugacité des objets et au fait qu'ils ne sont pas partageables (simplement) entre des processus. Ainsi, la gestion des attachements peut être confinée au niveau du processus et ne nécessite pas de gestion globale. Chaque processus est en fait son propre courtier d'événement. La communication événementielle entre processus est possible en utilisant les primitives de communication de Unix, telles que les « sockets », ce qui permet une propagation des événements vers d'autres contextes. Concernant le modèle d'exécution, les possibilités sont limitées à la création de nouveaux processus ou de nouveaux processus légers au sein du processus courant.

Enfin, le point le plus important demeure l'utilisation de nos propositions pour la réalisation de connecteurs dans OLAN. Dans ce contexte, le premier point clé est l'adaptation à des environnements d'exécution divers, puisqu'OLAN ambitionne de fournir ses services dans un environnement hétérogène. Ce point rejoint le précédent

sur l'adaptation du mécanisme. Il est d'autant plus ardu à réaliser qu'il faudra harmoniser les fonctions du mécanisme entre ces diverses plates–formes, y compris le modèle d'exécution des réactions, afin d'offrir une sémantique homogène.

Les problèmes relatifs aux interfaces et à la conformité constituent le second point clé. En effet, les connecteurs mettent en relation des composants qui ont une certaine interface et le contrôle de la conformité de ces interfaces constitue le fondement pour l'établissement des interactions entre composants. Ces contraintes doivent conduire d'une part à l'introduction des événements dans l'interface des composants et d'autre part à la définition d'une conformité prenant en compte ces événements.

# Annexe A

# Exemples de classes avec clauses réactives

Nous présentons dans cette annexe des exemples de programmes OC++ qui décrivent d'une part l'interface d'accès au mécanisme et d'autre part son utilisation dans les applications présentées au chapitre V.

## A.1 Classe `EventService`

La classe `EventService` définit l'interface d'accès au mécanisme de communication événementielle.

```
distributed
class EventService
    {
public:
    // Courtier d'événements
    EventBroker *myBroker;

    // Constructeur
    // brokerName est le nom du courtier auquel est
    // connecté l'objet créé (myBroker).
    EventService(T_String brokerName);

    // Émission d'un événement
    virtual void Send(T_Event *event);

    // Abonnement
    // subscrType détermine le mode d'exécution des réactions
    virtual void Subscribe(
        T_String clauseName;
        T_SubscrType subscrType);

    // Désabonnement
    virtual void Unsubscribe(T_String clauseName);
    }
```

## A.2 Classes de l'application Agenda

La classe ScheduledEntity décrit l'interface d'un objet qui possède une liste de rendez-vous.

```
distributed
class ScheduledEntity : public EventService
    {
public:
    // Liste des rendez-vous
    DistList *agenda;

    // Constructeur
    ScheduledEntity(T_String brokerName);

    // Ajouter un rendez-vous provisoire à l'agenda
    virtual T_Return newRDV(RDV *rdv);

    // Supprimer un rendez-vous de l'agenda
    virtual T_Return removeRDV(RDV *rdv);

    // Invalider un rendez-vous provisoire
    virtual T_Return rollbackRDV(RDV *rdv);

    // Valider un rendez-vous provisoire
    virtual T_Return commitRDV(RDV *rdv);

    // Aviser l'entité de l'échéance d'un rendez-vous
    virtual T_Return adviceRDV(RDV *rdv);

    // Clauses réactives
    COORDINATIONS
        CLAUSE normal
            // La référence du rendez-vous concerné
            // est passée en paramètre des événements
            ON commitRDV_Evt OBJ DO commitRDV
            ON rollbackRDV_Evt OBJ DO rollbackRDV
            ON adviceRDV_Evt OBJ DO adviceRDV
    END_COORDINATIONS
    }
```

Cette classe est utilisée comme super-classe des classes qui représentent les utilisateurs (classe Person), les groupes (classe Group) et les lieux de réunion (classe MeetingRoom). Ces classes sont identiques dans leur interface à ScheduledEntity, mais surchargent la méthode adviceRDV d'annonce de rendez-vous.

## A.3 Classes de l'application de gestion de commandes

La classe Service est une classe de base pour tous les services.

```
distributed
class Service : public EventService
    {
private:
    // Liste des employés habilités
    DistList *habilitations;

public:
    // Constructeur
    Service(T_String brokerName);

    // Mettre une commande en attente
    virtual T_Return QueueCommand(Command *cmd);

    // Lancement du service
    virtual T_Return activateService(Employee *emp);
    }
```

À titre d'exemple, nous donnons l'interface de la classe CustomerService qui représente le service de la clientèle.

```
distributed
class CustomerService : public Service
private:
    // Liste des commandes à traiter manuellement
    DistList *manualList;

public:
    // Constructeur
    CustomerService(T_String brokerName);

    // Traitement d'une commande arrivée
    virtual T_Return TreatNewCommand(Command *cmd);

    // Traitement d'une commande honorée
    virtual T_Return TreatReadyCommand(Command *cmd);

    // Mise en attente d'une commande à traiter manuellement
    virtual T_Return QueueManualCommand(Command *cmd);

    // Aviser le client (commande prête ou en attente)
    virtual T_Return AdviceCustomer(Commad *cmd);

    // Clauses réactives
    // newCommand_Evt annonce l'arrivée d'une commande
    // fulfillCmdEvt annonce le succès d'une commande
    // store_NotAvailEvt annonce un défaut de stock
    COORDINATIONS
        CLAUSE inactive
            // Quelque soit l'événement, la commande passée
            // en paramètre est mise en file d'attente.
            ON newCommandEvt OBJ DO QueueCommand
            ON fulfillCmdEvt OBJ DO QueueCommand
            ON store_NotAvailEvt OBJ DO QueueCommand
```

```
        CLAUSE manual
            // Quel que soit l'événement, la commande est mise
            // en attente de traitement dans la liste
            // manualList
            ON newCommandEvt OBJ DO QueueManualCommand
            ON fulfillCmdEvt OBJ DO QueueManualCommand
            ON store_NotAvailEvt OBJ DO QueueManualCommand
        CLAUSE auto
            // Une commande nouvelle ou remplie est traitée
            // automatiquement, tandis qu'un défaut de stock
            // fait l'objket d'un avis auprès du client.
            ON newCommandEvt OBJ DO TreatNewCommand
            ON fulfillCmdEvt OBJ DO TreatReadyCommand
            ON store_NotAvailEvt OBJ DO AdviceCustomer
    END_COORDINATIONS
    }
```

La classe `Synchronizer` permet de réaliser la synchronisation entre le service financier et le magasin telle qu'elle a été décrite en V.3.2.2. Son interface et sa réalisation sont les suivantes :

```
distributed
class Synchronizer : public EventService
    {
private:
    // Liste des commandes traitées par au moins un des
    // deux services
    DistList *commands;
public:
    // Constructeur
    Synchronizer(T_String brokerName) :
                            EventService(brokerName)
        {
        commands = NIL;
        SUBSCRIBE normalFucntion
        }

    // Traitement d'un événement entrant
    virtual void TreatCommand(Command *Cmd);

    // Clauses réactives
    COORDINATIONS
        CLAUSE normalFunction
            ON store_FulfillEvt OBJ DO TreatCommand
            ON financial_BillEvt OBJ DO TreatCommand
    }
```

```
// Code de la méthode TreatCommand
void Synchronizer::TreatCommand(Command *cmd)
    {
    DistList *aList;
    // La liste des commandes est vide, on y insère la
    // commande passée en paramètre et on sort
    if (commands == NIL)
        {
        commands = new DistList((dvoid *) cmd);
        return;
        }
    // On recherche dans la liste des commandes si cmd  est
    // déjà présente
    if ((aList = commands->FindItem((dvoid *) cmd)) == NIL)
        {
        // Si non, on l'insère et on sort
        aList = new DistList((dvoid *)cmd);
        commands->InsertBefore(aList);
        commands = aList;
        return;
        }
    // Si oui, on supprimme l'élément de la liste ...
    if (aList == commands)
        commands = commands->next;
    aList->Delete():
    delete aList;
    // et on émet l'événement fulfillCmdEvt
    SEND fulfillCmdEvt OBJ(cmd)
    }
```

# Références bibliographiques

[Agha 86] Agha G., *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Ma., 1986.

[Agha 93] Agha G. et Callsen C.J., "ActorSpace: An Open Distributed Programming Paradigm", *ACM SIGPLAN Notices (Proceedings of the 4th ACM Conference on Principles and Practice of Parallel Programming)*, 28(7), pp. 23–32, 1993.

[Ananda 92] Ananda A.L., Tay B.H. et Koh E.J., "A Survey of Asynchronous Remote Procedure Calls", *Operating Systems Review (ACM SIGOPS)*, 26(2), pp. 92–109, Avril 1992.

[Andreoli 93] Andreoli J.–M., Ciancarini P. et Pareschi R., "Interaction Abstract Machines", *Research Directions in Concurrent Object–Oriented Programming*, édité par G. Agha, P. Wegner et A. Yonezawa, pp. 257–280, MIT Press, Cambridge Mass, 1993.

[Andreoli 94] Andreoli J.–M., Gallaire H. et Pareschi R., "Objects Meet Rules", *Object World Germany'94*, Frankfurt, Sep. 1994.

[Andreoli 96] Andreoli J.–M., Freeman S. et Pareschi R., "The Coordination Language Facility: coordination of distributed objects", *Theory and Practice of Object Systems (TAPOS)*, à paraître, 1996.

[Anwar 93] Anwar E., Maugis L. et Chakravarthy S., "A New Perspective on Rule Support for Object–Oriented Databases", *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, édité par Buneman P. et Jajodia S., pp. 99–108, Washington, DC, Mai 1993.

[Bacon 95] Bacon J., Bates J., Hayton R. et Moody K., "Using Events to Build Distributed Applications", *Proc. IEEE Services in Distributed and Networked Environments*, pp. 148–155, Whistler, Canada, Juin 1995.

[Baecker 92] Baecker R.M., *Readings in Groupware and Computer–Supported Cooperative Work*, Morgan–Kaufmann Publishers, 1993.

[Balter 91] Balter R., Bernadat J., Decouchant D., Duda A., Freyssinet A., Krakowiak S., Meysembourg M., Le Dot P., Nguyen Van H., Paire E., Riveill M., Roisin C., Rousset de Pina X., Scioville R. et Vandôme G., "Architecture and implementation of Guide, an Object–Oriented Distributed System", *Computing Systems*, 4(1), pp. 31–67, Hiver 1991.

[Balter 94] Balter R., Lacourte S. et Riveill M., "The Guide language: Design and Experience", *The Computer Journal*, 37, pp. 519–530, Décembre 1994.

[Balter 95] Balter R. et Krakowiak S., *Objectifs et plan de travail du projet Sirac*, (1–95), IMAG–INRIA, Projet Sirac, Juin 1995.

[Balter 96] Balter R., Bellissard L., Boyer F. et Riveill M., *Environnement de développement d'applications réparties : objectifs et plan de travail*, (7–96), IMAG–INRIA, Projet Sirac, Février 1996.

[Bellissard 96a] L. Bellissard, S. Ben Atallah, A. Kerbrat et M. Riveill, "Component–based Programming and Application Management with Olan", *Workshop on Object–Based Parallel and Distributed Computation (OBDPC)*, édité par J.P. Briot, J.M. Geib et A. Yonezawa, Lecture Notes in Computer Science (LNCS) 1107, Tokyo, 1996.

[Bellissard 96b] L. Bellissard, S. Ben Atallah, F. Boyer et M. Riveill, "Distributed Application Configuration", *16th International Conference on Distributed Computing Systems*, pp. 579–585, IEEE, Hong–Kong, Mai 1996.

[Beust 93] Beust C. et Nahaboo C., "The Koala Bus Group Communication Software", *Programmer's manual for version 1.28*, pp. 1–35, Document Bull S.A, Février 1993.

[Beust 96] Beust C., *Conception d'outils destinés à assister au développement d'applications distribuées*, Thèse de Doctorat Nouveau Régime, Université de Nice–Sophia Antipolis, Mai 1996.

[Boyer 94] Boyer F., *Coordination entre outils dans un environnement intégré de développement de logiciels*, Thèse de Doctorat, Université Joseph Fourier, Grenoble, France, Février 1994.

[Boyer 95] Boyer F., "Coordinating Software Development Tools with Indra", *7th Conference on Software Engineering Environments (SEE'95)*, pp. 1–13, IEEE Computer Society Press, Noordwijkerhout (Netherlands), Avril 1995.

[Bull 94] Bull Open Software Systems, "OODE : Une plate–forme à objets pour les applications coopératives", *AFCET*, Paris–France, Novembre 1994.

[Cagan 90] Cagan M., "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools", *Hewlett–Packard Journal*, 41(3), pp. 36–47, Juin 1990.

[Callsen 94] Callsen C.J et Agha G., ''Open Heterogeneous Computing in ActorSpace'', *Journal of Parallel and Distributed Computing*, 21, pp. 289–300, 1994.

[Cardelli 85] Cardelli L et Wegner P., ''On understanding types, data abstraction, and polymorphism'', *Computing Surveys*, 17(4), pp. 471–522, 1985.

[Carriero 89a] Carriero N. et Gelernter D., ''Linda in context'', *Communications of the ACM*, 32(4), pp. 444–458, Avril 1989.

[Carriero 89b] Carriero N. et Gelernter D., ''How to Write Parallel Programs: A Guide to the Perplexed'', *ACM Computing Surveys*, 21(3), pp. 323–357, Septembre 1989.

[Carriero 92] Carriero N. et Gelernter D., ''Coordination Languages and their Significance'', *Communications of the ACM*, 35(2), pp. 97–107, Février 1992.

[Casati 95] Casati F., Ceri S., Pernici B. et Pozzi G., ''Conceptual Modeling of WorkFlows'', *Proceedings of 14th International Conference on Object–Oriented and Entity–Relationship Modelling (OOER'95)*, édité par Papazoglou M. P., pp. 341–354, Lecture Notes in Computer Science 1021, Springer–Verlag, Goald Coast, Australia, Décembre 1995.

[Collet 94] Collet C., Coupaye T. et Svensen T., ''NAOS Efficient and modular reactive capabilities in an Object–Oriented Database System'', *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, édité par J. Bocca, M. Jarke et C. Zaniolo, pp. 132–143, Santiago, Chili, Septembre 1994.

[Collet 95] Collet C. et Coupaye T., ''The NAOS System'', *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (Exhibit Program)*, édité par Carey M. et Schneider D., San Jose–USA, Mai 1995.

[Collet 96] Collet C., Habraken P. et Roncancio C., ''Règles actives dans les SGBD'', *Ingénierie des Systèmes d'Information (ISI)*, à paraître, Juin 1996.

[Crowley 90] Crowley T., Milazzo P., Baker E., Forsdick H. et Tomlinson R., ''MMConf: An Infrastructure for Building Shared Multimedia Applications'', *Proceedings of the 3rd Confernec on Computer–Supported Cooperative Work (CSCW'90)*, pp. 329–342, ACM, Los Angeles–Californie, Octobre 1990.

[Decouchant 96] Decouchant D., Quint V. et Romero Salcedo M., "Structured and Distributed Cooperative Editing in a Large Scale Network", *Groupware and Authoring*, édité par R. Rada, pp. 265–295, (chap 13), Academic Press, Mai 1996.

[Ellis 79] Ellis C.A, "Information Control Nets: A Mathematical Model of Office Information Flow", *Proceedings of the 1979 ACM Conference on Simulation, Measurement and Modeling of Computer Systems*, pp. 225–239, Août 1979.

[Ellis 91] Ellis C.A., Gibbs S.J. et Rein G.L., "Groupware: Some Issues and Experiences", *Communications of the ACM*, 34(1), pp. 38–58, Janvier 1991.

[Ellis 93] Ellis C.A. et Nutt G.J., *The Modelling and Analysis of Coordination Systems*, (CU–CS–639–93), University of Colorado, Boulder, Co., 1993.

[Garlan 91] Garlan D. et Notkin D., "Formalizing Design Spaces: Implicit Invocation Mechanisms", *VDM'91 – Formal Software Development Methods*, édité par Prehn S. et Toetenel W.J., pp. 31–44, Lecture Notes in Computer Science 551, Springer–Verlag, Octobre 1991.

[Gatziu 92] Gatziu S. et Dittrich K. R., "SAMOS: an Active, Object–Oriented Database System", *IEEE Quarterly Bulletin on Data Engineering*, Special Issue on Active Database Research, 15, pp. 1–4, Décembre 1992.

[Gatziu 93] Gatziu S. et Dittrich K.R, "Events in an Active Object–Oriented Database System", *Proceedings of the 1st International Workshop on Rules in Database Systems (RIDS)*, édité par Paton N.W et Williams H.W. dans Rules in Database Systems, Workshop in Computing, Springer–Verlag, Edinbourg, Août 1993.

[Gehani 91] Gehani N. H. et Jagadish H. V., "Ode as an Active Database: Constraints and Triggers", *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB'91)*, édité par Lohman G., Sernadas A. et Camps R., pp. 327–336, Barcelone, Espagne, Septembre 1991.

[Gehani 92] Gehani N. H., Jagadish H. V. et Shmueli O., "Event Specification in an Active Object–Oriented Database", *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, édité par Stonebraker M., pp. 81–90, ACM SIGMOD Record, 21(2), San Diego, California, Juin 1992.

[Gelernter 89] Gelernter D., "Multiple Tuple Spaces in Linda", *PARLE'89*, édité par E. Odjik, M. Rem et J.C. Syre, pp. 20–27, Springer–Verlag, 1989.

[Geppert 95] Geppert A., Gatziu S., Dittrich K.R., Fritschi H. et Vaduva A., *Architecture and Implementation of the Active Object–Oriented Database Management System SAMOS*, (95.29), Université de Zürich, Suisse, 1995.

[Lacourte 91a] Lacourte S., ''Exceptions in Guide, an Object–Oriented Language for Distributed Applications'', *Proceedings of European Conference on Object Oriented Programming '91*, édité par P. America, pp. 268–287, Lecture Notes in Computer Science 512, Springer–Verlag, Juillet 1991.

[Lacourte 91b] Lacourte S., *Exceptions dans les langages à objets*, Thèse de Doctorat, Université Joseph Fourier – Grenoble 1, Juillet 1991.

[Malone 90] Malone T.W et Crowston K., *What is Coordination Theory and How Can It Help Design Cooperative Work Systems?*, (112), Center for Coordination Science, MIT, Cambridge, Mass., Avril 1990.

[Marangozov 95] Marangozov V., *Conception et réalisation d'un service de désignation associative pour la construction d'applications coopératives*, Rapport de DEA, I.N.P. Grenoble – Université Joseph Fourier, Grenoble, Juin 1995.

[Matsuoka 88] Matsuoka S. et Kawai S., ''Using Tuple Space Communication in Distributed Object–Oriented Languages'', *Proceedings of the Conference on Object–Oriented Programming Systems, Languages and Applications (OOPSLA'88)*, édité par Meyrowitz N., pp. 276–284, SIGPLAN Notices, 23(11), ACM Press, Novembre 1988.

[Menon 93a] Menon S., *Asynchronous Events: Tools for Distributed Programming in Concurrent Object–Based Environments*, (GT–CC–93–09), Georgia Institute of Technology, Géorgie, USA, 1993.

[Menon 93b] Menon S., Dasgupta P. et LeBlanc J. Jr, ''Asynchronous Event Handling in Distributed Object–Based Systems'', *ICDCS'93*, pp. 383–390, IEEE Computer Society Press, Los Alamitos, CA, Bruxelles, Tokyo, Juin 1993.

[Nierstrasz 89] Nierstrasz O., ''A Survey of Object–Oriented Concepts'', *Object–Oriented Concepts, Databases and Applications*, édité par Kim W. et Lochovsky F., pp. 3–21, ACM Press et Addison–Wesley, 1989.

[Notkin 93] Notkin D., Garlan D., Griswold W.G. et Sullivan K., "Adding Implicit Invocation to Languages: Three Approaches", *Object Technologies for Advanced Software, First JSSST International Symposium*, édité par S. Nishio et A. Yonezawa, pp. 489–591, LNCS 742, Springer–Verlag, Novembre 1993.

[Ohmori 92] Ohmori T., Maeno K., Sakata S., Fukuoka H. et Watabe K., "Distributed Cooperative Control for Sharing Applications Based on Multiparty and Multimedia Desktop Conferencing System: MERMAID", *Proceedings of the 12th International Conference on Distributed Computing Systems*, pp. 538–546, IEEE Computer Society Press, Yokohama–Japon, Juin 1992.

[Oki 93] Oki B., Pfluegl M., Siegel A. et Skeen D., "The Information Bus: An Architecture for Extensible Distributed Systems", *Operating Systems Review*, 27(5), pp. 58–68, Décembre 1993.

[OMG 91] The Object Management Group, *The Common Object Request Broker Architecture and Specification*, Décembre 1991.

[OMG 93] The Object Management Group, *Common Object Services Specification – Chapter 4: Event Service Specification*, Mars 1994.

[OMG 96] The Object Management Group, *The Common Object Request Broker Architecture and Specification*, Mai 1996.

[Osher 92] Osher H., "Object Request Brokers", *Byte*, pp. 172–174, Janvier 1992.

[Ousterhout 94] Ousterhout J. K., *Tcl and the Tk Toolkit*, Addison–Wesley, Juin 1994.

[Reiss 89] Reiss S., *Interacting with the Field Environment*, (CS–89–51), Brown University, Department of Computer Science, Mai 1989.

[Reiss 90] Reiss S., "Connecting Tools Using Message Passing in the Field Environment", *IEEE Software*, 7(4), pp. 57–66, Juillet 1990.

[Starovic 95] Starovic G., Cahill V. et Tangney B., "An Event Based Object Model for Distributed Programming", *Proceedings of of the International Conference on Object–Oriented Information Systems*, pp. 72–86, Dublin City University, Dublin, Décembre 1995.

[Sullivan 92] Sullivan K.J et Notkin D, ''Reconciling Environment Integration and Software Evolution'', *ACM Transactions on Software Engineering and Methodology*, 1(3), pp. 229–268, Juillet 1992.

[SunSoft 91] SunSoft White Paper, *Introduction to the Sunsoft's Tooltalk Service*, Juin 1991.

[Trevor 93] Trevor J., Rodden T. et Blair G., ''COLA: A Lightweight Platform for CSCW'', *Proceedings of the Third European Conference on Computer–Supported Cooperative Work*, édité par De Michelis G., Simone C. et Schmidt K., pp. 15–30, Kluwer, Milan, Italie, Septembre 1993.

[Wegner 90] Wegner P., ''Concepts and Paradigms of object–oriented programming'', *OOPS Messenger*, 1(1), pp. 8–87, Août 1990.

# Table des matières

# Introduction

## Chapitre I

## Coordination et communication asynchrone

## Chapitre II

## État de l'art

## Chapitre III

## Modèles d'événements

## Chapitre IV

## Application : Extension du modèle Guide

## Chapitre V

## Mise en œuvre et évaluation

## Conclusion

## Annexe A

## Exemples de classes avec clauses réactives

# Communication par événements dans les modèles à objets


## Résumé

Les applications coopératives mettent en jeu des interactions complexes entre les différents éléments qui les composent. L'environnement utilisé pour leur programmation doit donc offrir les mécanismes nécessaires à la réalisations de ces interactions. Le choix d'un environnement à objets pour cette mise en œuvre revêt de nombreux avantages, relatifs aux caractéristiques canoniques des modèles à objets (notion d'interface, héritage, encapsulation), mais certains mécanismes nécessaires aux applications coopératives ne sont pas ou peu pris en compte dans ce contexte. Parmi ceux–ci, nous nous intéressons plus particulièrement à un mode de communication asynchrone et anonyme, complément de l'appel de méthode synchrone. Nous proposons d'intégrer un tel mode de communication dans les modèles à objets en utilisant la notion d'événement : un événement peut être émis et déclencher de manière asynchrone des actions dans les objets. Nous décrivons les différents choix possibles pour réaliser cette intégration. Ces choix concernent à la fois les aspects déclaratifs du mécanisme et ceux relatifs au modèle d'exécution associé. Nous examinons également les effets de cette extension sur les caractéristiques canoniques des modèles à objets. Nous appliquons ensuite cette étude au modèle Guide, qui offre des objets répartis, partageables et persistants. Nous montrons comment le mode de communication proposé peut être intégré à ce modèle particulier en insistant sur les critères qui motivent nos choix. Enfin, la réalisation d'un prototype et d'applications tests a permis de valider les propositions faites.



## Abstract

Groupware applications involve complex interactions between their components. Their implementation requires that the environment it uses provides adapted mechanisms for these interactions. Object–oriented models offer many advantages in this framework, because of their canonical characteristics (interface, inheritance and encapsulation). But they do not provide some required mechanisms in the groupware context. Among these mechanisms, we focus on an asynchronous and anonymous communication scheme, an addition to the common synchronous method invocation. We propose the use of events to realize this scheme: an event can be raised and asynchronously trigger reactions within objects. We describe the choices offered for the integration of this event–based communication within object–oriented models. These choices concern declarative aspects of the mechanism as well as its execution model. We also analyze the consequences of this integration on the canonical characteristics of object–oriented models. Then we apply this study to the Guide model, which provides sharable, distributed, persistent objects. We show how our proposed communication scheme can be included within this particular model and insist upon the criteria which induce our choices. Then, a prototype and test applications offer validation data for our proposal.