Remerciements

Je tiens tout d'abord à remercier les membres de mon Jury, et notamment mes rapporteurs et examinateurs qui ont tous accepté avec enthousiasme de «décrypter» ce volumineux ouvrage. C'est non sans une certaine fierté que je me vois ainsi «examiné» par certains des plus grands noms dans le domaine des systèmes distribués, de la réflexion (au sens informatique - et très précis - du terme) et des langages concurrents.

Dans des circonstances qui restent encore aujourd'hui assez mystérieuses de mon point de vue (sans évoquer les troubles administratifs résultants), j'ai fini par «hériter» de l'encadrement de deux directeurs de thèse. Il y eut tout d'abord Jean-Pierre Briot dont les intuitions forment le fond de ce travail de thèse, qu'il faut voir comme la continuation (explicite) de ses travaux très réputés sur la programmation orientée objet en environnement concurrent. Je ne remercierai jamais assez Jean-Pierre de m'avoir également «poussé» à découvrir le monde, tant scientifique (un carnet d'adresses bien rempli !) que géographique : de l'extrême ouest à l'extrême est du globe. Jean-Pierre m'a transmis ce virus «double» de la recherche scientifique et de la culture japonaise, amalgame conduisant de façon inexorable à la recherche au japon : doomo arigatoo gozaymashita, Jean-Pierre sensei !

En travaillant avec Christian Queinnec, mon deuxième Directeur de thèse, j'ai ressenti ce qui probablement est primordial dans tout exercice doctoral : une nécessaire remise en question. J'ai découvert grâce à notre fructueuse collaboration les implications linguistiques (au sens informatique du terme) de mon travail - je découvrais ainsi, grâce à Christian et sa fantastique expérience dans ce domaine, la face cachée des langages informatiques. Mais je perdis, par la même occasion, quelques nuits de sommeil, ce qui ne m'empêchera pas de remercier Christian très chaleureusement.

Je ne remercierai jamais assez le professeur Gul Agha qui, en plus de m'accueillir à l'Université d'Illinois à Urbana-Champaign pour un été studieux, m'a fait découvrir «de l'intérieur» ces formidables langages d'acteurs. Je remercie également le Professeur Toru Ishida (et le ministère Japonais de la recherche) de m'avoir invité pour un autre été studieux à l'Université de Kyoto. Mes remerciements vont aussi au professeur Akinori Yonezawa qui, s'il ne m'invita point pour un autre de ces étés studieux, me propose plutôt de le rejoindre pour deux années d'études post-doctorales, évidemment studieuses, à l'Université de Tokyo; projet que j'envisage avec grand enthousiasme !

Je remercie aussi vivement mes collègues et amis du Laboratoire d'Informatique de Paris 6 pour leur aide préciseuse et leur sympathie inaltérable. Je remercie tout spécialement mon plus proche collègue (géographiquement au moins) Thomas Meurisse, mon premier stagiaire Olivier Vernin, mon collaborateur «Z» David Julien et Fabrice Kocik pour leur soutien scientifique précieux. Mais je n'oublierais pas non plus de remercier vivement Zahia Guessoum, Mikal Ziane, Jean-François Perrot, Nicolas Revault, Reza Razavi, Diane Vanbergue, Christophe Gonzales ainsi que les équipes administratives - Ghislaine Mary, Jacqueline Le Baquer et David Massot en tête - et techniques - notamment Christophe Bouder et Jean-Pierre Arranz - du laboratoire.

Mais si nous remontons quelques vingt-deux ans en arrière, il faut voir dans tout «cela» l'influence profonde de mon père qui m'a fait découvrir à la fois le monde de la recherche scientifique mais également ma passion informatique ! Tout commença par une séance de «pianotage» sur un petit ordinateur Prime (quelques dizaines de mètres carrés pour quelques centaines de kilo-octets de mémoire) pour écrire, sur les genous de papa, un premier programme Basic, à 7 ans environ...

Le problème, après, fut de s'intéresser à autre chose que ces ordinateurs, passionnants mais «dévorants» : une autre histoire ! Ma mère Dany, mon père Robi (encore lui !), Claude, mon frère Nicolas et sa compagne Isabelle, Flora, Yann et Nils ; tout spécialement ma grand-mère Dora et toute ma famille en général ont eu une influence déterminante sur ce point...

J'aurai, pour terminer, une pensée toute particulière pour ma tendre Aya qui m'a supporté pendant cette période assez «égocentrique» de la rédaction de thèse. Ima wa Tokyo e ikitai !

Table des matières

« Où devrais-je commencer, s'il vous plaît votre Majesté ? Demanda-t-il.
Commencez au commencement, dit le roi, gravement,
et allez ainsi jusqu'à rejoindre la fin: puis arrêtez-vous. »

Alice au pays des merveilles (Lewis Carroll)

Introduction

Les systèmes informatiques d'aujourd'hui sont, en toute généralité, de nature fort complexe. Mais plus encore que cette complexité «de fait», c'est surtout leur tendance à se complexifier tout au long de leur durée de vie qui pose problème. Rares en effet sont les sytèmes modernes dont la conception s'arrête au moment du déploiement. Ces systèmes - des simples services de commerce électronique aux larges infrastructures informatiques de grandes institutions ou entreprises - *évoluent* et ne cessent d'ainsi *évoluer* jusqu'à ce que d'autres systèmes - eux-mêmes évolutifs - ne prennent le relais. C'est sur cet aspect d'*évolution logicielle* que se focalise une large part des recherches actuelles.

Il existe une profusion de pistes pour aborder les questions relatives à l'évolution des logiciels informatiques. Dans cette thèse, nous avons choisi de nous concentrer sur une vision assez opérationnelle de ce principe d'évolution dans le cadre de systèmes distribués conçus et réalisés dans une approche de type interstitiel (ou intergiciel, voir *middleware* pour nos amis anglophones). Notre étude porte plus précisément sur des systèmes pouvant être *composés* et *adaptés*, de façon *dynamique* et *incrémentielle*.

La composition est un principe général de construction incrémentiel de système qui permet, à partir de composants indépendants et clairement identifiés, de produire des systèmes de nature *intrinsèquement* évolutive. Dans la perspective holiste, l'essence de ces systèmes composés (dynamiquement) est bien plus que la somme de leurs parties : la composition est selon nous une problématique en soi et qui dépasse la simple notion de composant.

Les démarches de composition et d'adaptation représentent toutes deux des principes *systématiques*. Mais si la composition est avant tout centrée sur la problématique de construction des système, l'adaptation se focalise elle sur leur modification. Il ne s'agit bien sûr aucunement de modification arbitraire ; l'adaptation se doit d'être à la fois *dirigée* et *contrôlée*. Nous distinguons dans notre approche deux catégories principales d'adaptation :

- l'adaptation *structurelle*, et
- l'adaptation *comportementale*.

Le premier principe est de nature connexe à la démarche de composition évoquée précédemment. De façon quelque peu schématique, les adaptations structurelles sont celles que l'on peut obtenir par utilisation «simple» de mécanismes de composition. En revanche, les adaptations comportementales interviennent à un niveau de granularité plus fin que les entités mises en jeu dans le cadre de la composition. Ces deux catégories reflètent fidèlement une dichotomie - de notre point de vue fondamentale - et que nous étayerons tout au long de notre exposé entre la structure et le comportement. Mais il est important de poser dès maintenant l'imbrication intime de ces deux points de vue. Nous verrons, par exemple, comment intervenir sur le comportement par des procédés purement structurels.

Au niveau des applications pratiques de ces constructions et modifications contrôlés de systèmes distribués, nous ne considérerons pratiquement que des aspects fortement liés à l'infrastructure distribuée sous-jacente. Les modifications «arbitraires» de systèmes nous semblent encore hors de portée pour un certain temps...

Les approches interstitielles «classiques» reposent sur les principes de la programmation orientée-objet. Contrairement à une idée souvent reçue, l'immersion de ces principes dans un univers concurrent et réparti ne se fait pas sans heurt. De nombreux chercheurs ont montré que si cette immersion était possible, elle était en tout cas loin d'être naturelle, même dans le cadre le plus fréquent des architectures client/serveur. D'importants travaux ont ainsi conduit à l'élaboration de modèles et langages rompant, de façon parfois assez franche, avec cette tradition «objet». En nous resituant dans le discours de la composition et de l'adaptation dynamique, les problèmes prennent selon nous une nouvelle dimension et l'appel pour des modèles d'alternance se fait ressentir de façon encore plus forte.

Pour supporter leur composition et adaptation dynamiques, nous pensons que les entités mises en jeu dans les systèmes doivent réunir deux propriétés fondamentales :

– une propriété d'*indépendance comportementale*, et
– une propriété d'*indépendance structurelle*.

Selon nous, le paradigme objet n'offre pas, de façon intrinsèque, ces deux propriétés nécessaires, notamment parce qu'en termes comportementaux, les objets sont de nature *synchrone* et qu'en termes structurels, leur relation de couplage est de nature *implicite*. Les approches à base d'acteur, qui mettent en œuvre des entités asynchrones, et les composants logiciels - au couplage explicite - proposent deux gammes de solutions intéressantes et, de notre point de vue, complémentaires à ces problèmes.

Cette complémentarité nourrit notre intuition forte concernant la possibilité de refondation d'un modèle de développement de logiciel composable et adaptatif ; c'est l'étude d'un tel modèle que nous proposons dans cette thèse.

COMME nous l'a fait très justement remarquer Doug Lea pendant le workshop DOPP à ECOOP'2001, nombreux sont les chercheurs ayant tenté et, selon ses propres termes, échoué en essayant de concilier *asynchronisme*, si naturel dans le cadre des systèmes répartis, et *principes modernes de génie logiciel*. Alors, nous demanda-t-il, qu'est-ce-que pourrait apporter notre proposition sur cette problématique connue ?

Notre réponse forme le fil conducteur de cette thèse, dont le point de départ est représenté par ce constat de l'étonnante complémentarité entre les approches asynchrones - focalisées sur le comportement - et la récente émergence du concept de composant logiciel - concept avant tout structurel. Mais plutôt que de reconstruire les composants à base d'acteurs (cf. [Ast99]), nous proposons de prendre les composants tels quels et de refonder, en quelque sorte, un modèle de comportement asynchrone, forcément éloigné des acteurs eux-mêmes.

Le principal problème réside dans l'ambiguïté du concept de composant qui répond, selon les propositions, à de nombreux problèmes différents. Resitués dans notre problématique de composition et d'adaptation, les composants offrent avant tout l'indépendance structurelle, passant par l'extraction de leur relation de couplage. De notre point de vue, les composants sont donc avant tout intéressants s'ils permettent la composition structurelle, tout comme les acteurs sont intéressants à nos yeux car il permettent - du fait de leur mode de fonctionnement asynchrone - la composition comportementale...

Notre proposition s'articule donc autour de la (re)construction d'un modèle complet mettant en œuvre des composants typés élémentaires et proposant un mode de fonctionnement asynchrone. L'introduction des types est fondamentale car c'est selon nous ce qui caractérise proprement dans notre approche la notion de structure. Une structure devient structure lorsque son type est connu et l'on peut dès lors combiner ces structures en combinant leurs propriétés de type. Le comportement associé à ces structures, puisque asynchrone, permet de supporter ces combinaisons et recombinaisons en termes comportementaux.

La principale innovation concerne, au dela d'une simple alchimie, la fondation d'un modèle formel disposant de caractéristiques lui confèrant une identité propre. Il existe un principe fondamental de la programmation par acteurs : le mode de fonctionnement asynchrone des entités mises en œuvre. De même, l'approche que nous privilégions des composants logiciels érige en principe premier la réification de la relation de couplage. Comme nous le découvrirons au chapitre 7, notre modèle dispose également de son propre trait caractéristique. Ce principe fondamental y est introduit sous la dénomination de mode de communication *à diffusion multi-point implicite en fonction du type*. Il s'agit selon nous d'une avancée notable puisque ce modèle permet de reconstruire, de façon quasi-algébrique (et nous le montrons), la plupart des modèles d'interactions utilisés pratiquement dans les systèmes distribués actuels...

La très importante validation de notre travail est construite selon une démarche descendante. Tout d'abord, en partant de notre problématique «orientée composant» et plutôt structurelle, nous montrons dans la deuxième partie de cet exposé que l'ensemble des concepts que nous proposons peuvent être formalisés relativement simplement, dans un formalisme qui ne semble pourtant pas forcément le plus adapté de prime abord (i.e. la notation Z). La cohérence de cette formalisation est également constamment vérifiée, au fur et à mesure de la descente vers la concrétisation des concepts introduits. Plus important encore, la transition de ces concepts structurels vers leur réalisation opérationnelle - dite comportementale - est obtenue par raffinement, au sens formel du terme. Cette transition est montrée (sous la forme d'un schéma Z) et c'est à notre avis ce point d'inflexion explicite (au chapitre 7) qui justifie en grande partie nos choix méthodologiques.

Dans un deuxième temps, notre démarche descendante doit, à un moment ou à un autre, se concrétiser sous la forme d'une réalisation pratique du modèle. De fait, deux réalisations complémentaires sont proposées. L'une (au chapitre 6) concrétise les concepts structurels fondamentaux de notre proposition (composants, événements, connexions, types). Il s'agit d'un *langage spécifique*, de nom Scope, qui délègue tout ce qui n'est pas du ressort purement structurel à un environnement sous-jacent (de fait, nous supportons deux environnements : Java et Scheme/Common Lisp). L'autre réalisation (décrite au chapitre 10), plus ambitieuse, plonge ce langage Scope dans un environnement concurrent et distribué, sous la forme d'une plateforme instersticielle : Comet. Il est intéressant de noter que malgré le statut expérimental de notre plateforme, l'ensemble des exemples proposés - et parfois décrits à un niveau purement formel - ont été implantés et expérimentés sur un véritable système distribué physique...

Première partie

Problématique

« - Science ? Le Sauvage se renfrogna. Il connaissait le mot mais ce qu'il signifiait exactement il n'en avait aucune idée.
- Oui (...) il n'y a pas que l'Art d'incompatible avec le bonheur. C'est également la Science.
(...) C'est pourquoi nous ne l'autorisons pas à s'occuper d'autre chose que des problèmes immédiats (...)»

Brave New World (Aldous Huxley)

1
Domaine d'étude

Nous allons poser dans ce premier chapitre quelques jalons préliminaires pour pouvoir cerner dans les grandes lignes notre sujet de réflexion.

1.1 Systèmes distribués adaptatifs

Pour entamer cette discussion, établissons la catégorie de systèmes informatiques sur laquelle porte cette étude. Nous parlerons principalement de :

Systèmes distribués composables et adaptables

Il s'agit bien sûr d'une restriction encore bien trop large de l'ensemble des environnements informatiques possibles mais cela nous permet d'ores et déjà d'orienter notre parcours.

Par exemple, nous tenterons d'éviter certains détours : notre propos est bien celui du *système* et des systèmes distribués, disons de «haut niveau», en particulier. C'est-à-dire que nous parlerons principalement de flexibilité et d'adaptation en terme de système et non au sens global et notamment pour tel ou tel domaine d'application. Certes, cette frontière est relativement floue mais nous essaierons quoiqu'il advienne de conserver quelques distances avec les domaines applicatifs précis.

Nous essaierons aussi, autant que possible, de ne pas prendre de raccourcis trop directs. Notre but à travers cette thèse est certes plus constructif que critique mais il faut bien évidemment s'inspirer du passé pour construire le futur.

Donc, dans cette thèse, nous parlerons sans détour et sans raccourci de :

- *systèmes distribués*, de
- leur développement et leur opérationnalisation par *composition*, et de
- leur possible *adaptation* dynamique.

S'attaquer au vaste domaine du *système* et de la distribution est un exercice assez proche de l'équilibrisme en ceci qu'il *traverse*, en quelque sorte, notre paysage informatique et qu'il est difficile de le considérer en occultant les nombreux domaines connexes.

Si l'on analyse la morphologie d'un système distribué, comme celui présenté en figure 1.1 page suivante, nous pouvons dégager un grand nombre de problématiques interdépendantes mais pour autant bien spécialisées.

Au plus bas niveau de la relation entre deux sites distribués (un site local à gauche et un site distant à droite), nous trouvons la couche physique du *réseau* qui possède son propre et très vaste domaine de recherche.

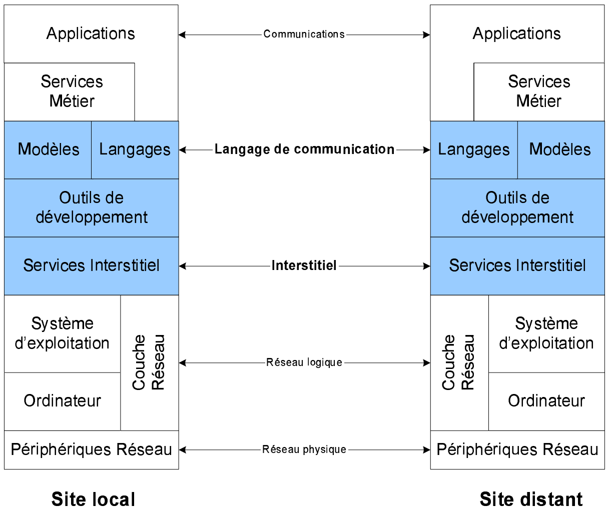


FIG. 1.1 – *Systèmes distribués et domaines de recherche*

L'*architecture des machines*, lorsque ces dernières sont prévues pour fonctionner en réseau, est un autre domaine important et fourni. Dans cette thèse, nous écartons cependant ces deux voies très riches et nous n'aurons que d'assez lointaines considérations avec le matériel physique. Les *systèmes d'exploitation* et leur importante *couche réseau* sont plus proches de nos préoccupations essentiellement logicielles mais nous considérerons plutôt comme acquises les très importantes avancées scientifiques et technologiques dans ces domaines.

De l'autre côté du spectre, les *applications* représentent bien sûr la finalité à laquelle tout informaticien, chercheur ou ingénieur, aspire. Ces applications ne sont plus, aujourd'hui et dans leur grande majorité, développées au cas par cas mais reposent sur d'innombrables *services métiers* (sous formes de bibliothèques, de boîtes à outils, de compilateurs spécifiques, etc.). Cependant, ce n'est pas non plus vers cette problématique de *génie logiciel* que nous nous orientons même s'il est encore une fois fondamental de ne pas remettre en question les grands progrès observés ces dernières années dans cette discipline extrêmement riche et primordiale.

Notre point de focalisation - représenté en grisé sur la figure 1.1 - se situe exactement à l'interface entre les considérations purement liées au réseau et à la distribution, disons de «bas niveau», et à la problématique de description des applications, de plus «haut niveau».

Selon nous, cette interface se décompose en trois grandes problématiques :

- Les *modèles* et/ou *langages* dans lesquels sont exprimées les applications distribuées,
- les *outils* afférents qui permettent l'expression la plus simple possible des systèmes considérés,
- les *technologies* sous-jacentes, dites *interstitielles*, qui effectuent la liaison «mécanique» avec les éléments de plus bas niveau supportant l'opérationnalisation de ces systèmes en environnement distribué.

Ces trois catégories sont bien sûr hautement imbriquées, de sorte qu'il est important de les considérer non seulement comme des parties bien spécifiques mais aussi comme éléments constitutifs d'un «tout» d'ordre supérieur. Notre proposition est de maintenir notre discours au niveau des aspects systèmes couvrant ces trois axes.

Parmi ces aspects systèmes forts nombreux, certains ont été étudiés de façon très approfondie : modèles de communication, gestion des ressources, etc. Certaines des solutions afférentes sont entrées dans les modèles et outils industriels pour le développement de logiciels distribués de grande envergure.

La répartition de systèmes objets préexistants et la conception de nouveaux systèmes sont donc des problématiques aujourd'hui bien prises en compte par les approches industrielles. Cependant, le saut qualitatif vers des systèmes plus ouverts semble encore en devenir.

Aujourd'hui, la communauté scientifique se penche ainsi sur une propriété très attendue mais également très sensible des systèmes à venir : la *dynamicité*. Le cycle «à usage unique» analyse/conception/développement/déploiement/maintenance des systèmes informatiques ne possède plus aujourd'hui le statut de *principe absolu* mais est au contraire bien souvent remis en question et tout particulièrement dans le cadre de la distribution. Selon le postulat de l'*évolution logicielle* [LY00], les systèmes du futur devront être conçus dans l'optique de leur constante modification, extension, adaptation (etc.). Et l'on parle bien d'évolution durant toute le *cycle de vie* applicatif ; c'est-à-dire en particulier au moment même de l'opérationnalisation des systèmes. Du cycle unique de développement, nous passons donc à un mode itératif dans lequel les phases d'analyse/conception/développement/déploiement sont renouvelées au besoin et/ou à l'envie.

Mais ce monde évolutif est un objectif difficile à atteindre car les problèmes posés sont d'une très grande complexité. De fait, nous ne savons pas aujourd'hui créer, dans des conditions de grand confort et de grande simplicité, des logiciels sûrs ; et ce, même en environnement séquentiel et centralisé. Certes, les récents développements dans le domaine sont prometteurs [Abr96], [HKPM97] mais le chemin restant à parcourir semble encore bien long, surtout s'il s'agit de transposer le problème dans l'environnement infiniment plus complexe de la distribution et de la concurrence. En conséquence, les propriétés dynamiques des systèmes distribués sont largement étudiées mais l'approche expérimentale semble pour l'instant privilégiée ; et nous nous inscrivons naturellement dans cette perspective même si nous mettrons un point d'honneur à être aussi précis et formels que possible.

Dans notre travail, nous nous sommes donc concentrés sur les phases de développement et de déploiement des systèmes distribués, versant pragmatique de la problématique logicielle mais qui sert généralement de pivot entre le monde conceptuel (la pensée des systèmes) et le monde opérationnel (leur réalisation effective). Selon nous, il s'agit précisément de l'endroit où l'on peut «attaquer» les problèmes sans passer dans un mode purement descendant (le modèle conceptuel est le tout [RSBP95]) ou au contraire totalement ascendant (le modèle opérationnel est le tout [Mil99]), approches dont nous ne remettons bien sûr pas en question l'intérêt ! Mais la vision «hybride» nous semble également digne d'étude en tant que telle et non comme résultante de l'une ou l'autre des autres approches connues [Hoa99].

1.2 Enoncé du problème

De notre point de vue, une étape préliminaire pour l'obtention - souhaitable - de systèmes distribués réellement évolutifs (au sens de l'évolution logicielle) passe par la conception et la mise en œuvre de modèles et langages ayant pour propriétés intrinsèques :

- Le support d'un mode de développement *itératif* et *incrémentiel*, «cassant» le cycle implantation/compilation/déploiement,
- La possibilité d'intervenir sur le fonctionnement du système, de façon *dynamique* et *contrôlée*.

Pour pousser ces considérations à l'extrême, nous poserons comme postulat qu'à l'origine, le système n'est rien et doit être *composé* dynamiquement puis modifié ou plus précisément *adapté*[1] ensuite, toujours dynamiquement, durant toute la durée de son exécution. D'un point de vue système, un corollaire sera de replacer en objectif prioritaire la phase généralement considérée comme finale du *déploiement*.

La question qui nous intéresse principalement ici sera donc :
Comment convevoir et implanter des systèmes distribués composables et adaptables dynamiquement ?
Pour ceci, nous devons donc nous intéresser à l'étude des :
Modèles (au moins en partie formels) pour la composition et l'adaptation dynamique de systèmes distribués.

Gardons cependant à l'esprit notre focalisation sur les aspects système, notamment pour ce qui concerne l'adaptation.

Pour aborder ces questions, nous avons décidé de séparer la problématique en deux sous-parties à la fois distinctes et interdépendantes :

- Le point de vue structurel
- Le point de vue comportemental

1. Nous marquons par l'emploi du terme *adaptation* le fait que les modifications effectuées sur les systèmes participent d'un objectif clairement établi en amont ; il ne s'agit pas de modifications arbitraires.

La première vision orient notre réflexion sur l'élaboration de structures, essentiellement *conceptuelles* et *linguistiques*, claires pour décrire les systèmes qui nous intéressent ainsi que les adaptations - forcément structurelles à ce niveau - que ces derniers pourront supporter.

Le point de vue comportemental permet de passer de ces concepts structurels à une sorte de *réalité* opérationnelle conséquente.

Cette dichotomie se retrouve tout au long de notre discours, que ce soit dans les chapitres consacrés à l'étude précise des problèmes qui nous concernent ou dans l'exposé de cette proposition. Cependant, le fil conducteur de notre discours est la démonstration constante du caractère inextricable de ces concepts ; ces derniers doivent donc être pensés l'un en fonction de l'autre et *vice et versa*.

De notre point de vue qui érige le concept de réflexion (auto-représentation/modification) au rang d'outil incontournable et fondamental, la validation principalement expérimentale d'un tel modèle concerne la création de services permettant - dans des limites assez strictes - aux applications qui les utilisent de s'auto-composer et s'auto-adapter, et que nous nommerons *services distribués adaptatifs*.

1.3 Un exemple

Pour illustrer notre problématique, nous allons prendre l'exemple d'une application - virtuelle mais réaliste - dans le domaine du *multimédia distribué*. L'intérêt de ce domaine d'application est sa proximité marquée avec les aspects système (et même souvent matériels) ; de sorte qu'il est pratiquement impossible ici de distinguer clairement ce qui serait du ressort d'une problématique purement système ou bien de l'autre côté du spectre essentiellement fonctionnel. Il ne s'agit pas, pour autant, d'observer cette exemple sous l'angle du domaine du multimédia et de son rapport avec l'environnement distribué (on consultera plutôt, pour cela, l'intéressante thèse de Tom Fitzpatrick [Fit99]).

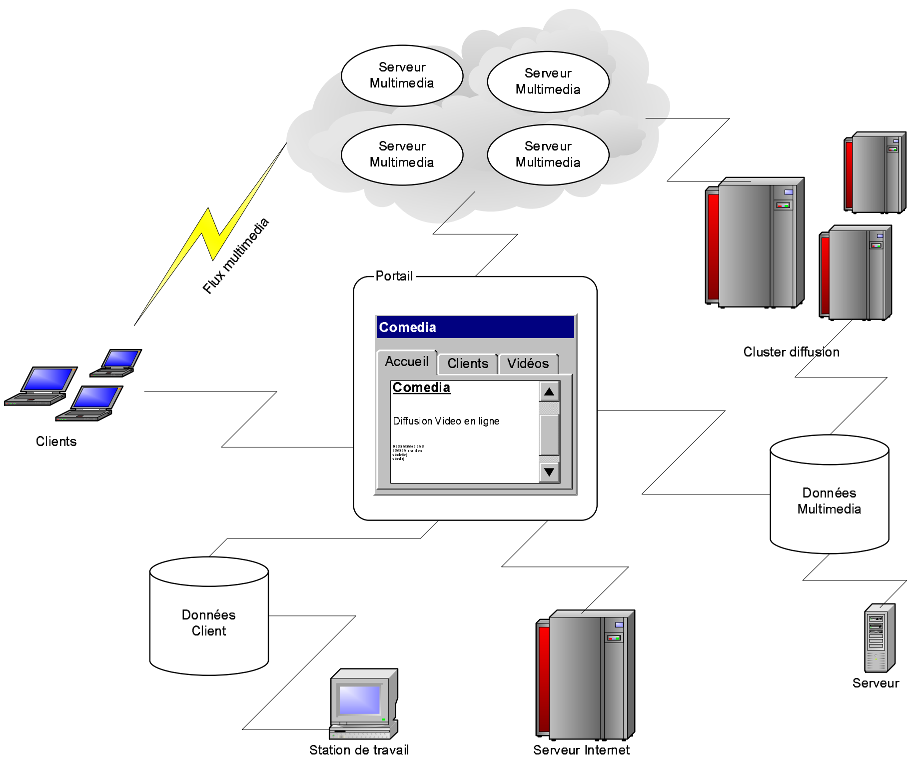


FIG. 1.2 – *Le système de diffusion vidéo en ligne Comedia*

Cette application prototype, Comedia, est un service en ligne de diffusion vidéo à la demande, fortement inspiré de l'exemple servant de fil conducteur à la thèse de David Sturmann sur son *Distributed Interaction Language* [Stu96]. Comme le montre la figure 1.2, ce service se compose de quatre sous-systèmes principaux :

- un portail en ligne s'occupant de la présentation du service et des interactions avec les utilisateurs,
- un service d'enregistrement et de gestion de clientèle,

- une base de donnée distribuée contenant le fond multimédia disponible pour la diffusion,
- un sous-système de diffusion multimédia.

1.3.1 Architecture du système

En suivant la «philosophie» du domaine des *architectures logicielles* [GS93], nous pouvons décomposer la structure logique d'un tel service en éléments constitutifs clairement caractérisés et que l'on nomme *composants logiciels*. En suivant une interaction entre un client et le système sur le scénario présenté en figure 1.3, tentons d'extraire quelques «familles» de composant qu'il sera utile de développer pour supporter notre application.

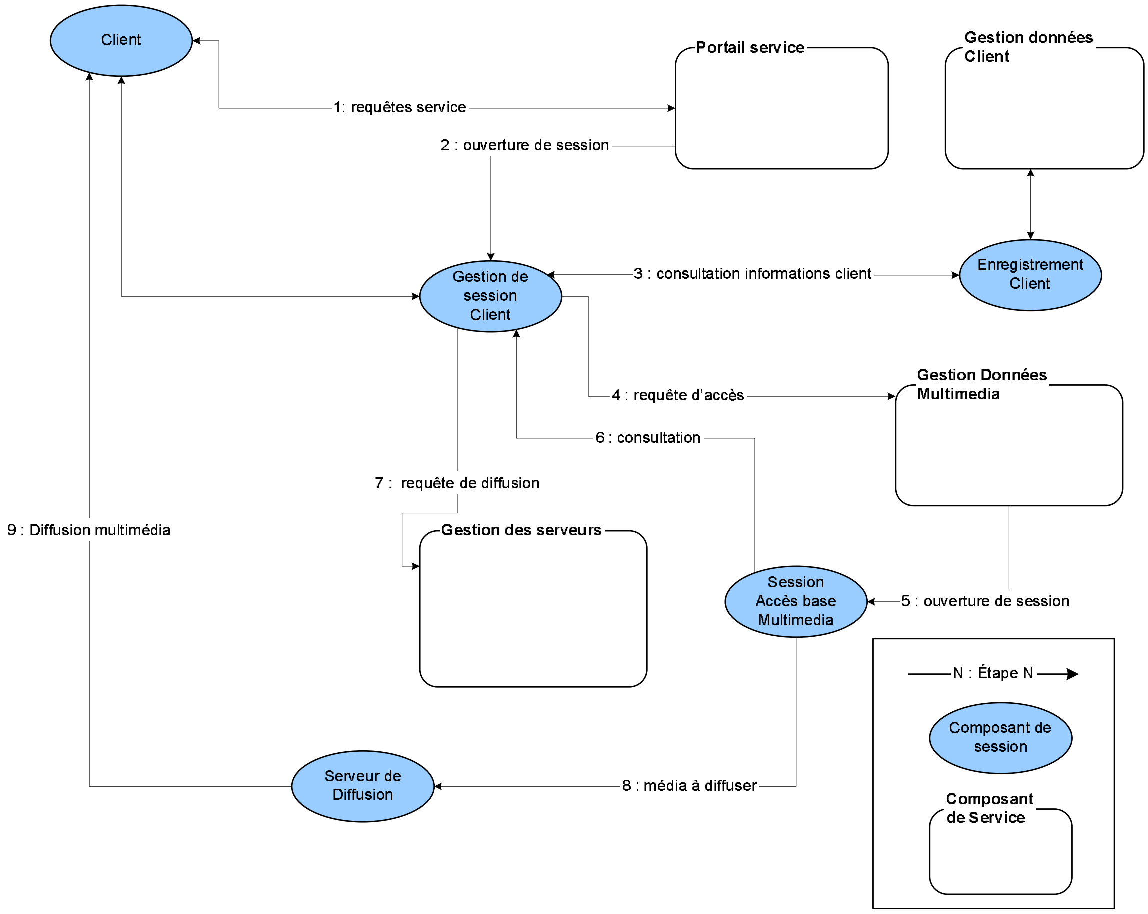


FIG. 1.3 – *Scénario d'interaction avec le système Comedia*

Lorsqu'un client accède au système, il doit tout d'abord s'enregistrer auprès de ce dernier, via la façade d'administration ou portail (étape 1). Ceci permet d'ouvrir une session spécifique pour ce client (étape 2), gérée par un composant logiciel dédié. Ce composant est en relation étroite avec le système de gestion de clientèle (étape 3) qui conserve dans une base de données les informations du client considéré, comme par exemple son niveau d'enregistrement : droits d'accès à la base de donnée multimédia, qualité de la diffusion, etc.

D'autres services de bases de données référencent les documents multimédia disponibles pour la consultation. Ce fond documentaire peut-être consulté au travers d'une session également spécifique (étapes 4, 5 et 6).

Lorsque le client choisit un média à diffuser, le gestionnaire de session client envoie cette requête de diffusion à la couche système concernée. Ceci met le client en relation directe avec un serveur de diffusion qui génère alors le flux d'informations multimédia (étape 7, 8 et 9).

Bien sûr ce scénario est fictif mais reflète assez fidèlement les techniques de développement d'applications en ligne avec en particulier l'importante dichotomie entre les services partagées par les différentes sessions (rectangles arrondis sur la figure) - de nature relativement statique - et les autres services dédiés ; et pour leur part hautement dynamiques (formes ovales grisées).

Nous pensons qu'il est primordial d'approcher ce genre de service dynamique sous l'angle de la *composition* au moment de l'exécution du système à partir des briques logicielles de base que représentent les *composants*.

En dehors de cet aspect de composition dynamique, il n'est point dans notre intention ici d'expliquer la méthode la plus adéquate pour définir le cœur fonctionnel de ce genre de service multimédia. Nous allons plutôt tenter d'en extraire quelques sous-problèmes qui exposent clairement notre seconde problématique d'adaptation dynamique au sens du système.

1.3.2 Adaptations structurelles

Une caractéristique très importante de notre scénario concerne sa nature dynamique. Les composants de session sont créés à la demande en fonction des requêtes client. Et cette caractéristique de dynamicité se double d'une nécessité d'adaptabilité : il est très difficile de prévoir avec exactitude combien de clients auront recours à ce service à un moment donné, pendant combien de temps ils en accapareront les ressources, etc. Or, le but premier d'un tel service concerne généralement l'optimisation du fameux critère de qualité de service (QoS). Selon le niveau de service souscrit par un client donné, le fond accessible sera clairement délimité et les propriétés de diffusion également précisées : du *best effort* pour le «commun des internautes» au *temps réel* des meilleurs «payeurs» [BZB+97] [CN02].

Si nous nous focalisons sur ce sous-système de diffusion vidéo, nous «tombons» d'ores et déjà sur un vaste spectre de problèmes. Concentrons nous sur la problématique d'adaptation structurelle et de son intérêt dans ce cas précis.

Considérons par exemple la simplification du système de diffusion à seulement deux composants : le client et le serveur de la figure 1.4.

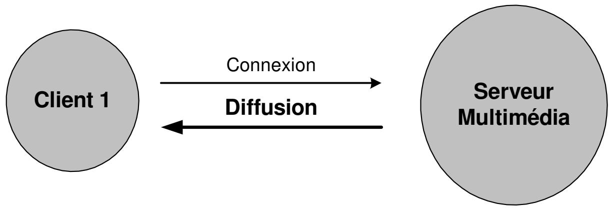


FIG. 1.4 – *Architecture de base du sous-système de diffusion*

Après connexion du client au service (suivie de quelques autres interactions étudiées précédemment), le serveur concerné commencera la diffusion des documents multimédia désirés. Nous pouvons voir cette architecture fort simple comme la *composition* entre un composant client et un composant serveur pour un service de diffusion multimédia.

Cette architecture point-à-point est très simplificatrice et n'est que peu révélatrice des architectures client/serveur en général qui favorisent les interactions de type N clients pour 1 serveur. De sorte que notre serveur multimédia ne s'occupera qu'exceptionnellement d'un client en particulier (à moins que ce dernier soit extrêmement privilégié) mais sera plutôt largement partagé. On dira d'un tel serveur partagé par un ensemble de clients qu'il propose un mode de comportement *multi-session*.

La figure 1.5 présente ce qu'à priori tout serveur digne de ce nom est susceptible de fournir ; c'est-à-dire le service simultané de deux clients.

Si la plupart des serveurs supportent ce mode de fonctionnement, il est encore assez rare de l'observer sous l'angle de la *composition*. Dans cette perspective, le *couplage* entre les composants, ainsi que les règles associées, devraient être définis *en dehors* de l'expression même des composants. De façon informelle, ces règles de couplage pourraient ressembler à :

> *Si un client correctement connecté demande une diffusion multimédia autorisée,*
> *alors le serveur peut fournir ce service*

Il est très important de mettre l'accent sur la caractéristique d'indépendance de ces règles, pour pouvoir entre autre moduler clients et serveurs à souhait sans «incruster» de contraintes trop fortes en terme de couplage dans le fonctionnement même des composants mis en jeu.

De notre point de vue, le système peut alors utiliser ces règles pour s'*adapter* soit par décision externe, soit de façon interne, c'est-à-dire en fonction de son propre état. En se basant sur de telles règles, on pourra par

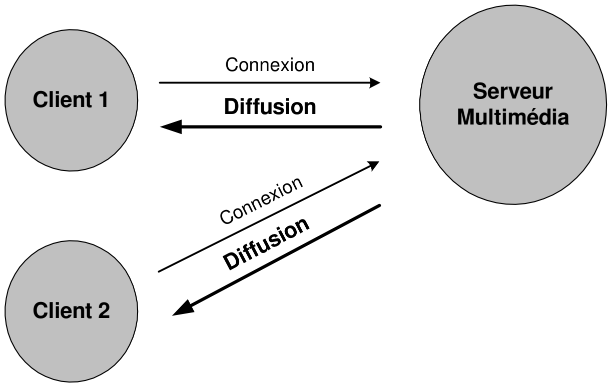


FIG. 1.5 – *Service simultané de plusieurs clients*

exemple exprimer ce que représente dans notre service multimédia ce que représente l'ajout ou le retrait d'un client. Signalons que ces règles de couplage ne sont pas forcément simples et ne concernent pas uniquement des aspects structurels : le serveur de notre exemple proposera ainsi un comportement bien différent de celui des clients ; et ce comportement aura forcément un impact sur ces derniers. Par exemple, la règle informelle proposée ci-dessus pourrait être *raffinée* en :

> Si un client correctement connecté demande une diffusion multimédia autorisée,
> *et s'il est capable de recevoir le flux correspondant*,
> alors le serveur peut fournir ce service

Pour l'instant, considérons ces règles qui se basent sur le point de vue du système (vue comme une interconnexion de composants) comme *structurelles*. Nous essaierons tout au long de notre thèse de donner un sens précis à ces énoncés pour l'instant très informels (et donc très flous). Mais anticipons déjà sur notre approche en remarquant que ces descriptions liées au couplage sous-entendent une phase fondamentale de négociation entre les composants concernés par ces règles.

Des règles d'adaptations structurelles plus complexes peuvent être dès lors décrites. Considérons toujours sur notre exemple le cas où un «client de trop» viendrait à se connecter au serveur. Un réaction intéressante du système serait de pouvoir réagir à cette surcharge, par exemple en créant un nouveau serveur dédié au plus récent des clients ou alors en restructurant le système complet pour offrir la meilleure qualité de service globale possible. La figure 1.6 montre les étapes mises en jeu par un exemple d'un tel *protocole* d'adaptation.

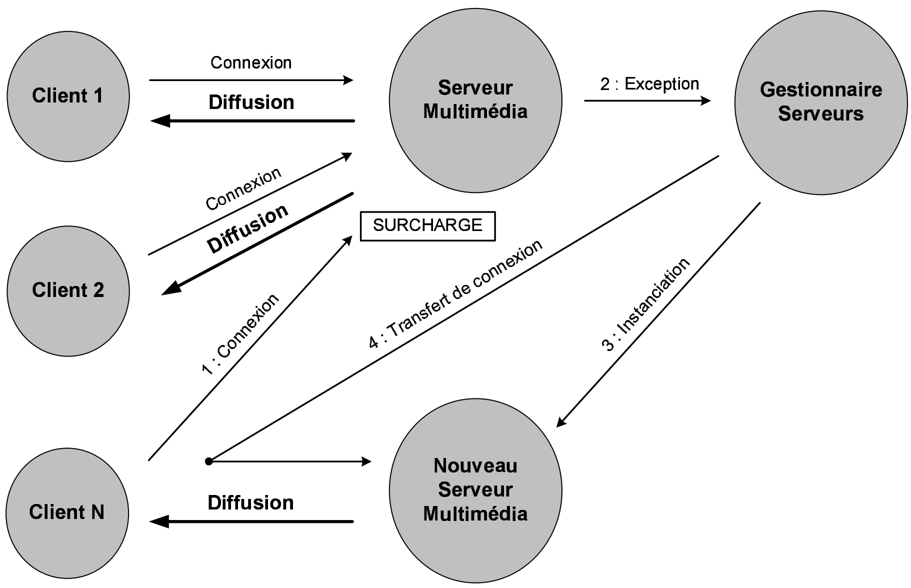


FIG. 1.6 – *Adaptation structurelle en cas de surcharge*

Lorsque le client Client N se connecte, le serveur détecte une éventuelle surcharge et propage cette exception vers un gestionnaire de serveurs (étapes 1 et 2). Ce gestionnaire prend alors la décision de créer un nouveau serveur (étape 3) et de rediriger le client vers ce dernier (étape 4). Dès lors, en supposant que les composants serveurs sont instanciés sur des machines différentes, la charge globale du système s'en trouve allégée.

Bien sûr, il est possible de concevoir un tel système dans une optique spécifique mais encore une fois, notre idée est d'approcher ce genre de problème de façon plus autonome ; tout particulièrement de façon la plus indépendante possible des autres (nombreux) problèmes posés par l'implémentation des clients et serveurs. Nous pourrions ainsi décrire ce *protocole* [2] de la façon suivante :

> *En cas de surcharge détectée sur un serveur au moment de la connexion d'un client, demander au gestionnaire l'instanciation d'un nouveau serveur et rediriger le client concerné vers ce nouveau serveur*

L'intérêt principal de la définition externe de telles règles est qu'il est dès lors possible de les réutiliser dans d'autres contextes. On peut imaginer la description de protocoles permettant l'allocation dynamique de serveurs en fonction de la charge, et ce, sans poser de présomptions trop fortes sur leur architecture interne et leur comportement fin.

1.3.3 Adaptations comportementales

Nous pouvons remarquer sur l'exemple précédent que l'adaptation mise en lumière était structurelle, c'est-à-dire agissant à un niveau *global* (ou tout du moins *non-local*). Cependant, les règles permettant de décrire ce type d'adaptation ne sont pas indépendantes du comportement des composants concernés, sauf cas trivial.

L'obligation de définir proprement les aspects comportementaux de telles adaptations nous obligent à considérer l'intérieur des composants et non seulement leur «interface». Nous proposons de transformer ce qui ressemble à une forte limitation (car l'analyse du comportement est un problème complexe) en une force pour autoriser des adaptations de grain beaucoup plus fin que le niveau structurel.

Ainsi, dans le cadre de notre application multimédia, il s'agira par exemple d'adapter le comportement du couple serveur/client concernant par exemple la diffusion du flot multimédia et sa nécessaire *synchronisation*. Nous aimerions encore une fois disposer de règles permettant de décrire les contraintes de synchronisation entre le client et le serveur et, pourquoi pas, de modifier ces contraintes en fonction de la charge.

Diverses heuristiques seront possibles, par exemple :

- le client met en œuvre une mémoire tampon de taille variable et réalise une synchronisation partielle sur cette mémoire locale du flot émis.
- Le serveur effectue une synchronisation de grain plus gros, basé sur des segments du flux et non sur ces éléments les plus primitifs.

Mais en cas de surcharge de la couche de communication, nous pourrions également autoriser le rejet d'information, des deux côtés de la communication : rejet d'un segment par le serveur ou rejet d'un élément primitif (comme une image) par le client. De fait, les possibilités sont quasiment infinies [KBH+01].

Même si nous n'envisageons pas l'autorisation de modifications arbitraires à l'intérieur des composants, les adaptations concernant des aspects systèmes importants comme la synchronisation du flux, si elles sont convenablement dirigées et contrôlées, doivent être rendues possibles. Ces adaptations, pour ne pas déroger à la règle, devraient être exprimées encore une fois indépendamment des entités sur lesquelles elles opèrent.

1.4 l'approche interstitielle

D'après les exemples donnés ci-dessus, nous voulons donc faire le lien entre :

1. La description de composants logiciels et de règles permettant de les composer et de les adapter durant l'exécution du système, et

2. Le choix de ce terme n'est encore une fois pas choisi au hasard et reflète notre volonté de fournir une «direction» aux descriptions de couplage, orientant les possibilités en terme d'adaptations futures.

2. l'opérationnalisation de ces descriptions et notamment, pour ce qui concerne la composition et l'adaptation dynamique, le contrôle précis de ces adaptations.

Selon nous, la réalisation de modèles et de mécanismes permettant de faire le liens entre ces considérations linguistiques d'une part, et opérationnelles d'autre part, se nommait autre fois *interprétation* ou *compilation*. Cependant, dans le cadre beaucoup plus large de l'informatique distribuée, ce rôle échoit de nos jours au très important concept d'*interstitiel* (*middleware* en anglais).

Il est donc normal, selon nous, d'approcher notre problématique sous l'angle de cette notion.

Les approches interstitielles les plus répandues aujourd'hui sont des lointains descendants des modèles et langages à objets. Les bus d'objets CORBA ORB [Obj01] et JAVA RMI [Sun01] sont les exemples les plus représentatifs de cette famille. Clairement, il s'agit dans ces deux approches de faire le lien entre le mode de développement sous-jacent du paradigme objet et l'opérationnalisation de ce modèle dans un environnement complexe, à la fois concurrent et distribué. Bien sûr, cette vision est plutôt restrictive mais reflète de façon adéquate notre focalisation sur les aspects purement système. Cependant, les problèmes posés à ce niveau sont d'ores et déjà assez nombreux, comme nous allons le voir.

1.4.1 Les objets : un paradigme éprouvé

La programmation orientée objet aura mis plusieurs années à s'imposer devant l'approche procédurale mais cette «prise de pouvoir» est aujourd'hui reconnue. L'idée de favoriser les structures de données sur les traitements que l'on souhaite effectuer a permis, il est vrai, d'obtenir des programmes plus simples et plus modulaires. Les objets représentent ainsi un outil de modélisation très puissant, si bien que très peu de domaines ont «résisté» au pouvoir d'expression de ces entités fort versatiles. De notre point de vue volontairement restrictif, les objets opèrent sur deux niveaux principaux :

1. La conception des applications Les objets sont en premier lieu des outils d'abstraction, au travers des classes qui regroupent les objets par propriétés, fonctionnalités, comportements et relations. On les utilise aussi bien pour représenter des objets «physiques» (du monde), des concepts abstraits ou bien encore des entités purement informatiques. D'un point de vue plus pragmatique, la plupart des techniques modernes et standardisées de génie logiciel sont basées sur le paradigme objet.

2. L'opérationnalisation En plus d'être un outil de conception puissant, l'objet se double d'une étonnante souplesse au niveau opérationnel. La simplicité du modèle favorise ainsi la construction de langages et d'environnements d'exécution pratiques et efficaces.

En corollaire de ce lien conceptuel/opérationnel, nous pouvons poser que les modèles objets (et en particulier les langages à classes, les plus nombreux) représentent un ciment fort intéressant entre le point de vue *structurel* (la classe en cours de définition) et le point de vue *comportemental* (l'instance en cours d'utilisation). Nous l'avons maintes fois répété et nous le répétons encore, c'est cette dichotomie structure/comportement qui conduit notre démarche. Il est donc intéressant de considérer l'existant sous cet angle.

1.4.2 De la difficulté des objets distribués

De la discussion précédente, nous pouvons dire que les objets représentent une sorte de lien entre les aspects de haut niveau, liés au problème à résoudre, et la mise en œuvre informatique de la solution. Il n'est donc pas étonnant de les retrouver à la base du développement de la plupart des systèmes distribués d'aujourd'hui.

Considéré du point de vue des applications concurrentes et réparties, le paradigme objet semble à première vue adapté ; en tout premier lieu parce qu'il reflète assez fidèlement le mode d'interaction privilégié (mais pas unique) des systèmes distribués : le client/serveur. Sans remettre en question ces intérêts de façon globale, des chercheurs ont rapidement montré les nombreuses difficultés liées à l'immersion du paradigme objet (originellement à forte connotation centralisée et séquentielle - ou quasiment séquentielle si l'on songe aux coroutines de Simula67 [DMN68]) dans un environnement réparti et concurrent (cf. par exemple [BGL98], [Gue99] ou encore [MY93]).

a) Objets synchrones versus besoins asynchrones

Le contrôle des objets est géré de façon implicite par le contenu des méthodes qui en définissent le comportement. Sur la figure 1.7, la méthode *m()* de l'objet *A* envoie un message à un autre objet *B*. *A* se met alors en attente de la fin du traitement de ce message. Pendant ce temps, *B* récupère le contrôle en même temps que la requête, sélectionne la méthode *m()* correspondant et en exécute le code. *B* rend ensuite la main à *A* qui peut continuer son traitement. Ce mécanisme se nomme l'*invocation synchrone de méthode* ou parfois l'*émission de message* dont il ne faut pas oublier la nature synchrone (puisque le message envoyé dénote également une valeur de retour).

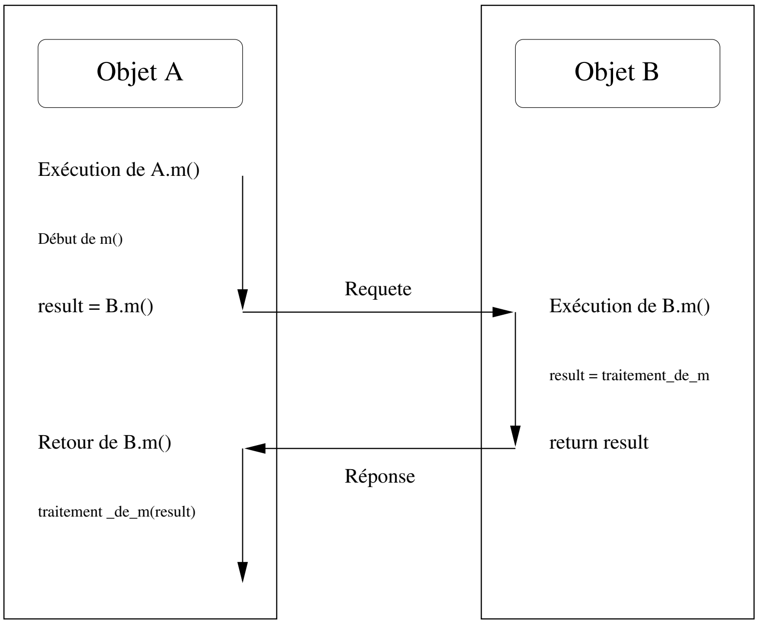


FIG. 1.7 – *Invocation synchrone de méthode dans le paradigme objet*

Selon le point de vue de nombreux chercheurs, cette nature synchrone entre en contradiction avec la vision privilégiée des systèmes distribués : l'asynchronisme. Ce constat est un des éléments fondateurs du *paradigme acteur* [HB77, Agh86] que nous étudierons plus en détail au chapitre 2. Une application répartie à base d'objets doit donc «simuler» le mécanisme synchrone d'invocation de méthode qui leur est implicite par combinaison de communications asynchrones, unique medium réellement disponible dans les systèmes répartis. Cette reconstruction synchrone, quoique très utile, s'avère parfois pénalisante pour répondre aux besoins réellement asynchrones des applications. Retrouver l'asynchronisme «naturel» dans un paradigme fondamentalement synchrone est un objectif complexe, si l'on en croit les propositions actuelles. Les spécifications Corba AMI (*asynchronous method invocations*) pour Corba, par exemple, reflètent bien cette complexité.

Cependant, il n'est pas question dans cette thèse de fournir une critique générale des approches à base d'objets distribués qui, de toute façon, sont désormais du domaine de l'industrie et résolvent de nombreux problèmes complexes et variés.

En revanche, si l'on se replace dans notre problématique de composition et d'adaptation dynamique de système, alors cette nature synchrone représente selon nous un véritable frein à l'élaboration de modèles de haut-niveau pour prendre en charge ces problèmes. De ce point de vue, notre argumentation nous conduira vers l'idée que seules des entités de nature intrinsèquement asynchrones permettraient l'abord de tels problèmes dans de bonnes conditions. Il s'agit, volontairement, de notre principal parti-pris :

La composition et l'adaptation dynamique doivent être approchées sous l'angle de l'asynchronisme

b) Objets statiques versus besoins dynamiques

Si l'invocation synchrone de méthode, sous-jacente des modèles objet, représente un point de départ intéressant pour les applications distribuées, la répartition des ressources (et donc des objets) assimilées nécessite un contrôle dynamique, lié à l'état réparti du système. Le contrôle statique des objets n'est valable que sur des architectures connues à l'avance, ce qui est rarement le cas des infrastructures pour applications réparties, surtout si l'on songe à leur composition dynamique. Cette absence de dynamicité dans les environnements d'exécution classiques des modèles objet pose plusieurs types de problèmes, et notamment :

Problème de nommage Pour s'assurer de l'unicité des objets dans un système centralisé, il suffit de garantir l'unicité de leur nom pour l'environnement d'exécution. Le nom choisi par l'implémenteur pour un objet (son nom de classe) et le nom d'instance associé à une adresse mémoire conviennent dans ce cas parfaitement. Lorsque les objets sont répartis, cette unicité ne peut être garantie que si l'application est gérée de façon centralisée, ce qui n'est ni réaliste ni souhaitable. La seule solution envisageable est la mise en œuvre d'un mécanisme de nommage dynamique garantissant l'unicité de l'objet pour le système, ainsi bien sûr que leur possible «découverte» au moment de l'exécution.

Problème de localisation Les environnements d'exécution objet (code compilé, machine virtuelle ou interpréteur) effectuent généralement une liaison statique entre objet et représentation mémoire. Même pour certains langages évolués et utilisant une liaison dynamique (DSM à objets persistants [S+98]), la localisation de l'objet reste relativement statique[3]. Une application répartie déploie généralement les objets qu'elle manipule en fonction de la topologie du système sous-jacent. Une localisation dynamique est donc nécessaire. La prise en compte de la mobilité des objets rend de plus ce problème de localisation très sensible.

Invocation synchrone de méthode (suite) Outre l'inadéquation pour répondre aux besoins asynchrones, un second problème est associé au mécanisme d'invocation synchrone de méthode. Certaines fonctionnalités de systèmes distribuées comme la réplication ou la migration nécessitent l'emploi de modèles de communication très particuliers. La communication multi-point, la diffusion causale, l'absence de duplication de requêtes sont autant de fonctionnalités importantes et difficilement conciliables avec le mécanisme de haut niveau proposé dans le modèle objet. Ce point est plus largement discuté dans [Gue99].

Dans le même ordre d'idée, le siège d'une invocation de méthode, qui correspondra à un *objet serveur* dans le cadre d'un interstitiel objet, accapare les ressources liées au contrôle de l'objet appelant (le client dans notre cas). Or, les infrastructures client/serveur ne sont réellement intéressantes que lorsque les clients se trouvent en beaucoup plus grand nombre que les serveurs. Ces derniers n'ont donc pas d'autre choix que de traiter les requêtes de clients multiples et de ne pas bloquer ce flux de requêtes lorsque d'autres traitements sont déjà en cours. Il est en conséquence nécessaire de fournir une gamme de contrôle beaucoup plus large que ce que propose les modèles objets au niveau conceptuel.

Bien sûr, tous ces problèmes trouvent leur solution dans les interstitiels disponibles ; ce sont par exemple les *services communs* qui prennent en charge ces questions dans le cadre de la norme CORBA. Cependant, s'ils sont parfois très précisément définis au travers de spécifications, la nature informelle de ces spécifications pose problème. De fait, cela fait reposer sur les «épaules» de l'implémenteur du bus objet et des services associés la question de confiance concernant l'adéquation entre les effets observables à l'exécution des applications et les intentions exposées dans les spécifications[4]. De plus comme ces spécifications sont difficilement formalisables, de nombreuses questions sont laissées en suspens et chaque implémenteur sera dès lors libre d'effectuer ses propres choix. Les exemples d'incompatibilité *de fait* entre implantations différentes d'une même spécification sont assez nombreuses (ex.: service de concurrence, de persistance, etc.).

De façon pragmatique, cette approche est pleine de sens et - de fait - fonctionne dans la plupart des cas (et notamment en ce qui concerne les problèmes exposés dans cette section). Mais encore une fois, nous nous intéressons ici à des propriétés dynamiques que l'on voudrait intrinsèques au modèle ou au paradigme car il semble très difficile de les aborder sous l'angle de la provision de services dédiés *en toute généralité.*

> Ceci nous conduit tout naturellement à notre deuxième prise de position fondamentale :
> ***Nous désirons, autant que possible, travailler au niveau des modèles, dans un souci de formalisation omniprésent.***

3. Dans le cas de PerDIS, l'objet est «quelque part dans la mémoire partagée répartie».
4. L'implémenteur aura d'ailleurs généralement anticipé ce fait en faisant signer un sorte de décharge de responsabilité à son client ; formulaire que l'on nomme licence et qui - même dans le cas d'un logiciel payant - n'offre aucune garantie concernant le fonctionnement correct du produit.

c) Des limites structurelles

Les problèmes évoqués dans les sections précédentes se posent principalement en termes de comportement ou d'opérationnalisation. En l'occurrence, il ne s'agit pas de remettre en doute les propriétés structurelles des objets. En clair, ils ne semblent pas incompatibles avec l'idée même de classe et des propriétés afférentes.

Cependant, un des principaux «reproches» formulé à l'encontre du paradigme objet concerne justement la relative inadéquation entre la structure figée de la classe héritée avec les contraintes de synchronisation liées à l'opérationnalisation concurrente des applications. Des travaux comme [MY93] montrent bien cette inadéquation structurelle pour laquelle de nombreuses solutions (encore discutées aujourd'hui [Han99]) ont été proposées.

Nous allons maintenant nous intéresser à deux autres problèmes importants et d'ordre structurel concernant les objets distribués.

Le problème de granularité Les systèmes objets sont généralement optimisés pour la manipulation d'objets de grain fin. Ce choix semble raisonnable puisque ce sont les plus nombreux. Dans un univers réparti, la nécessité de manipuler des entités de plus gros grain apparaît, surtout dans le cas où les objets deviennent unités de répartition (un site = un objet). Certes, les objets de gros grains peuvent être réalisés dans le paradigme objet mais il n'existe pas de différenciation structurelle entre les objets de différentes granularités alors que leur utilisation est clairement distinguée.

Couplage implicite Comme nous l'avons annoncé précédemment, notre intention concerne principalement la possibilité de décrire, sous forme de règles autonomes, la façon dont les éléments constitutifs des applications pourront être reliés les uns aux autres au moment de l'opérationnalisation du système ; et ce qu'il sera alors possible de faire en terme d'adaptation structurelle ou comportementale.

Nous voulons en quelque sorte séparer l'expression du couplage entre les composants du système de l'expression des composants eux-mêmes. Ceci bien sûr afin de pouvoir «jongler» avec ces règles de couplage durant l'exécution du système. Dans cette perspective, nous rencontrons un problème majeur en considérant ce problème sous l'angle des interactions entre objets.

Si nous revenons à notre exemple d'invocation synchrone de méthode (cf. figure 1.7 page 22), nous pouvons déceler un caractère fondamentalement antinomique de l'expression séparée du couplage : les références externes explicites. Par exemple, il est facile de constater le couplage fort entre l'objet A et l'objet B par la manipulation explicite, du côté de A de la référence à B ; et notamment lors de l'invocation :

```
result = B.m()
```

Le problème avec cette règle de couplage, qui ne lie pas uniquement A à B comme on pourrait le croire mais également B à A (car une valeur est retournée), est qu'elle est *intrinsèque* à l'objet A. En effet, cette relation est exprimée à l'intérieur même de l'objet, dans l'expression de son code fonctionnel, de sorte qu'il devient très difficile de l'observer de l'extérieur, au niveau que nous avons nommé structurel. Bien sûr, de nombreuses solutions existent à ces problèmes, ce qui a conduit, selon notre point de vue et au moins partiellement, à «découvrir» les célèbres *design patterns* [5] [GHJV94].

Ces problèmes (parmi d'autres) ont également conduit selon nous à l'introduction de la notion, pour l'instant peu consensuelle, de composant qui se trouve au centre de notre proposition et que nous retrouverons tout au long de notre discours.

> Nous pouvons en conclure notre troisième fondement :
> ***La composition dynamique nécessite l'indépendance structurelle, via l'extraction et la réfication de la relation couplage.***

5. A titre anecdotqie, lors d'un séminaire de l'auteur à l'Université de l'Illinois à Urbana-Champaign, l'un des pères fondateurs des *design patterns* nous a concédé ce fait.

« Non, les braves gens n'aiment pas que l'on suive une autre route qu'eux. »

Georges Brassens

2 Approches alternatives

Pour entamer l'établissement de ce qui sera le cahier des charges de notre proposition pour la description et la mise en œuvre de systèmes distribués composables et adaptables dynamiquement, nous allons effectuer un tour d'horizon des travaux (essentiellement scientifiques) développant ce qui représente selon nous deux conditions primordiales pour les objectifs que nous avons fixés.

Tout d'abord, nous pensons que les entités qui composent (dynamiquement, donc) nos systèmes adaptatifs doivent proposer une importante condition d'*indépendance structurelle*. Comme nous l'avons d'ores et déjà annoncé, il nous semble important à ce niveau d'analyser sous cet angle le récent avènement des *composants logiciels*.

Mais cette propriété d'autonomie structurelle intrinsèque n'est pas concevable independemment d'une «loi» similaire mais à un niveau plus opérationnel, ou plus comportemental selon notre terminologie désormais usuelle. Or, de ce point de vue, nous pensons que notre «meilleur» point de départ réside dans l'étude de *modèles asynchrones*, certes plutôt alternatifs, mais néanmoins fort intéressants.

2.1 Point de vue comportemental

Le document [BGL98] sert de référence concernant les problèmes associés à l'immersion du paradigme objet en environnement concurrent et distribué; sous l'angle privilégié du comportement (les fondements objet n'y sont pas remis en question). Ses auteurs établissent trois approches principales pour aborder cette problématique. Nous allons pour l'instant nous limiter aux deux premières :

Approches applicatives

La première solution envisageable consiste à proposer des bibliothèques d'objets «utilitaires» dont le domaine d'application est la programmation concurrente et répartie. Cette approche permet de ne pas remettre en question les systèmes et langages objets «traditionnels». Il s'agit du point de vue des *services communs* de la norme Corba dont nous avons évoqué certaines limitations dans les principes mêmes.

Approches intégrées

La deuxième voie proposée - celle qui nous intéresse plus particulièrement - concerne une plus large remise en question des modèles et langages objets tels qu'ils sont proposés dans les environnements «standards». Une de ces remises en question concerne la similarité (déjà observable dans Simula67) entre objets et processus. Cette analogie conduit naturellement à la notion d'*objet actif* qui peut être obtenue par *intégration* de constructions linguistiques spécifiques dans les langages objet traditionnels.

Dans notre problématique, l'intérêt primordial de ces approches intégrées concerne la mise en œuvre d'un modèle de comportement reflétant de façon plus précise les besoins en terme de distribution et de concurrence. Or, il est clair que le saut qualitatif du monde synchrone au monde asynchrone semble inhérent à ce type d'approche.

Nous proposons donc de poursuivre notre discussion sur le terrain des approches asynchrones les plus «marquantes» et de les analyser sous l'angle des propriétés dynamiques que l'on aimerait fournir dans le cadre d'un interstitiel pour le développement et le support des systèmes distribués adaptatifs qui nous intéressent ici.

2.1.1 Les modèles d'acteur

Poursuivie à l'extrême, l'approche intégrée conduit aux modèles alternatifs comme celui des acteurs qui se détachent quasiment des principes objets. De fait, les premières approches à base d'acteur sont contemporaines des premiers modèles objets, d'où le relatif cloisonnement des deux mondes.

Le modèle de calcul

Une particularité majeure dans l'histoire des acteurs, commencée à la fin des années 70 sous l'impulsion de Carl Hewitt, concerne le caractère ascendant de l'approche avec la mise en place précoce de modèles de calcul, dans un souci de formalisation omniprésent.

Gul Agha distingue, dans le modèle considéré aujourd'hui comme référence [Agh86], trois types différents d'éléments calculatoires :

1. Les processus séquentiels.
2. Les fonctions transformant des données.
3. Les acteurs, qu'il introduit.

Si les processus, sous-jacents de la programmation impérative, sont de nature séquentielle, leur exécution en parallèle est bien sûr possible. Le modèle de calcul *CSP* (*Communicating Sequential Processes*) de Hoare [Hoa85] et le modèle des variables partagées permettent d'étendre les processus séquentiels à la programmation concurrente et répartie.

L'approche fonctionnelle et son formalisme sous-jacent, le lambda calcul, autorisent également la réalisation de calculs concurrents, par exemple par évaluation parallèle des paramètres des fonctions.

Gul Agha décide de formaliser son modèle de calcul par acteurs à un même niveau d'abstraction que les deux exemples précédents; c'est-à-dire à un niveau complètement opérationnel.

La définition qu'il donne d'un acteur est la suivante :

Un acteur est un agent calculatoire qui associe chaque communication entrante à un triplet constitué de la façon suivante :

- un ensemble fini de communications envoyées à d'autres acteurs;
- un nouveau comportement (lequel va influencer la réponse à la prochaine communication traitée);
- un ensemble fini de nouveaux acteurs créés.

Il est montré que ce modèle d'acteur permet de couvrir un sur-ensemble du modèle des processus séquentiels et de celui des systèmes fonctionnels. Bien sûr, et c'est la principale force de l'approche, le modèle de communication sous-jacent revêt une nature purement asynchrone, permettant de prendre en considération au sein même du modèle opérationnel les problèmes liés à l'environnement concurrent et réparti sous-jacent[1].

D'un point de vue plus structurel, un acteur est composé des éléments suivants :

- Une file de messages associée à une adresse unique,
- Une fonction de traitement de la communication acceptée, implémentant le comportement de l'acteur.

La gestion de la file d'attente doit préserver l'équité entre les différentes sources de communications entrantes. La fonction de traitement spécifie pour sa part une transition de l'état du calcul. Les différentes responsabilités de cette fonction de transition se divisent en trois familles (*cf. figure 2.1*):

1. Carl Hewitt identifie cet environnement à la notion plus générale de système ouvert.

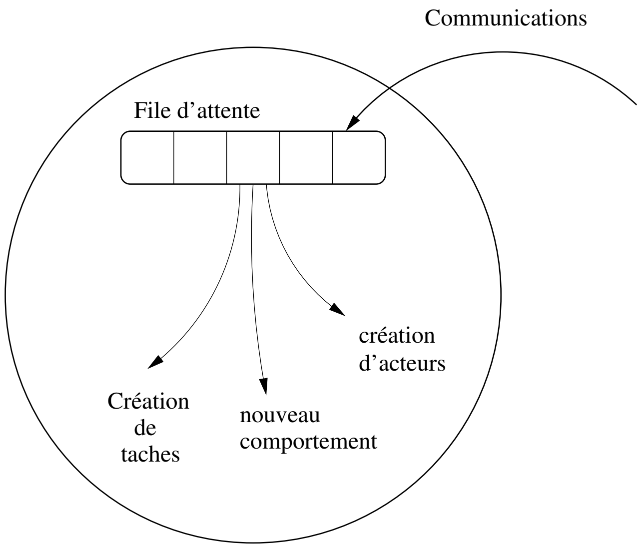


FIG. 2.1 – *Acteurs : responsabilités de la fonction de transition*

- Création de tâches : communications identifiées aux acteurs de l'accointance,
- Spécification d'un comportement de remplacement : génération de la prochaine fonction de transition,
- Création d'acteurs

La destruction d'acteurs est donc implicite dans ce modèle puisque la durée de vie d'un acteur correspond au traitement d'une unique communication.

Du modèle aux langages

Contrairement aux objets qui reposent sur les modèles de type processus séquentiels (ou plus rarement sur des systèmes fonctionnels), le paradigme de la programmation par acteurs dispose d'un modèle de calcul qui lui est propre.

Plusieurs langages d'acteurs ont été créés pour illustrer le grand pouvoir d'expression du modèle proposé. *Sal*, par exemple, permet de créer des acteurs sur le principe de la sélection d'actions à partir du contenu des messages. Voici un exemple de programme dans ce langage :

```
def stack-node (content, link)
  [ case operation of
    pop: (customer)
    push: (new-content)
  end case ]

if operation = pop ĉontent != NIL then
  become forwarder(link)
  send content to customer
fi
if operation = push then
  let P = new stack-node (content, link)
  { become stack-node (new-content, P) }
fi end def
```

Le rôle d'un acteur noeud-pile est d'enregister ou de diffuser un contenu selon le principe du Last In, First Out (*Lifo*). La pile correspond alors à un chaînage (par l'identificateur link) d'acteurs. La définition ci-dessus est une définition de comportement et non d'acteur proprement dite. L'attachement de ce comportement à un acteur se fait par la primitive new. Les messages envoyés à un acteur stack-node peuvent être de deux types : empiler (push) ou dépiler (pop) un contenu. Dans le premier cas, le message doit contenir les informations à empiler et dans le deuxième cas, l'acteur ayant effectué la demande. La construction [case .. end case] permet de caractériser le message en fonction de l'opération invoquée. L'opération pop (en cas de contenu

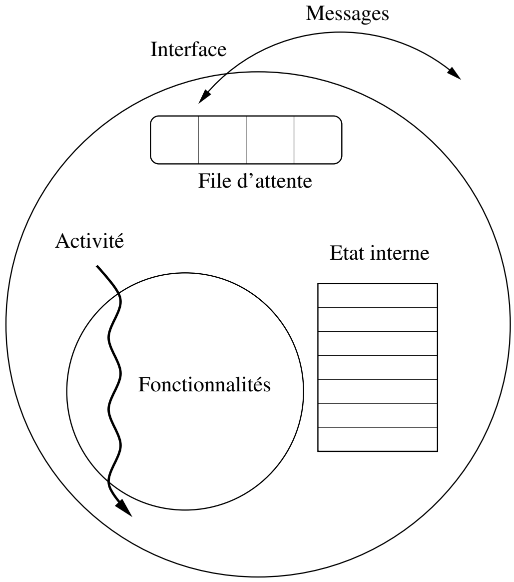


FIG. 2.2 – *Structure d'un acteur Broadway*

non vide) transforme l'acteur stack-node en un acteur forwarder (il redirige les messages suivants au prochain acteur de link) et envoie le contenu au client. L'opération push crée un nouvel acteur P dont le comportement est stack-node sur link et remplace le comportement actuel par un comportement stack-node sur P (maintien du chaînage).

Nous pouvons voir sur cet exemple que l'approche de la programmation par acteurs est assez différente de ce que propose le paradigme objet. La prise en compte du remplacement systématique de comportement est à la fois novatrice et déroutante. Il faut cependant faire la différence entre expression des programmes et modèle sous-jacent. Le framework d'acteur *Broadway* propose une vision plus accessible des acteurs. Un acteur *Broadway* est un objet *C++* composés des éléments suivants (*cf. figure 2.2*) :

- Une file d'attente
- Une interface
- Un état interne
- Une activité
- Des fonctionnalités implémentées par des méthodes

Ainsi, il est possible d'écrire l'acteur noeud-pile dans une syntaxe plus facile d'accès et d'en dégager une sémantique d'acteurs. *Broadway* autoriserait[2] par exemple la définition suivante :

```
Actor stack-node
  contents := NIL;
  link := NIL;

method link (alink)
  link := alink;
end link
method pop
  if(contents!=NIL)
    self := link;
  return contents; end pop
method push (new_contents)
  Actor anew = new Actor(self);
  self.content = new_contents;
  self.link = anew;
end push
```

2. Pas tout à fait, la syntaxe proposée par Broadway est similaire mais plus proche de C++.

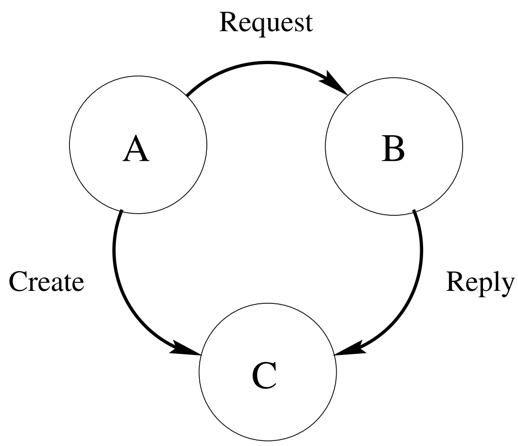


FIG. 2.3 – *Acteurs : invocation pseudo-asynchrone*

Nous utiliserons ce pseudo-langage proche des langages orientés objets pour construire nos prochains exemples d'acteurs.

Modes de communication

Le mode de communication mis en jeu par les acteurs est de type point-à-point asynchrone; c'est-à-dire qu'il reflète précisément le fonctionnement privilégié des systèmes répartis. L'introduction d'abstractions de communication de plus haut niveau est cependant nécessaire pour assurer la généralité du modèle proposé [A+93]. En guise d'exemple, nous allons doter les acteurs de la principale «fonctionnalité» des objets : l'invocation synchrone de méthodes. Nous verrons ensuite une approche possible pour les communications de groupe.

Des objets aux acteurs Le processus d'invocation synchrone des objets est transformé dans la programmation par acteurs en une communication de type requête-réponse. Ce type d'interaction met en jeu un acteur émetteur et un récepteur [3]. La première opération réalisée est l'émission d'un message asynchrone (par la primitive `send`) de l'émetteur vers le récepteur. Deux cas peuvent alors se présenter :

1. l'émetteur a besoin du résultat pour continuer son traitement immédiat
2. le résultat n'est pas attendu de façon synchrone et pourra être traité plus tard

Cette distinction permet de systématiquement proposer le degré de concurrence le plus élevé. Le deuxième cas, pseudo-asynchrone, est traité de façon très simple par les acteurs. Après émission de la requête asynchrone, un acteur de jonction est créé en attente de la réponse (asynchrone également) et l'acteur émetteur peut, après recouvrement de son comportement, continuer à traiter les messages entrants. Une fois la réponse reçue par l'acteur de jonction, ce dernier réalise l'opération souhaitée (*cf. figure 2.3*).

Le cas de l'invocation réellement synchrone est plus complexe. Il est inutile de créer un acteur de jonction puisque l'émetteur sera de toute façon bloqué en attente de la réponse. Le mode de fonctionnement de l'émetteur subit un changement assez notable. Ses entrées asynchrones ne sont pas traitées, à l'exception de la réponse attendue. L'opération qui réalise l'invocation est séparée en deux parties :

1. Les traitements préliminaires à l'envoi de la requête
2. La continuation après réception de la réponse

La réalisation du blocage de l'acteur entre les deux parties de l'opération nécessite l'introduction d'un mécanisme de synchronisation interne. Nous reviendrons sur cet aspect dans la section 2.1.1.

3. Ce modèle n'impose cependant pas l'unicité de l'émetteur et du récepteur.

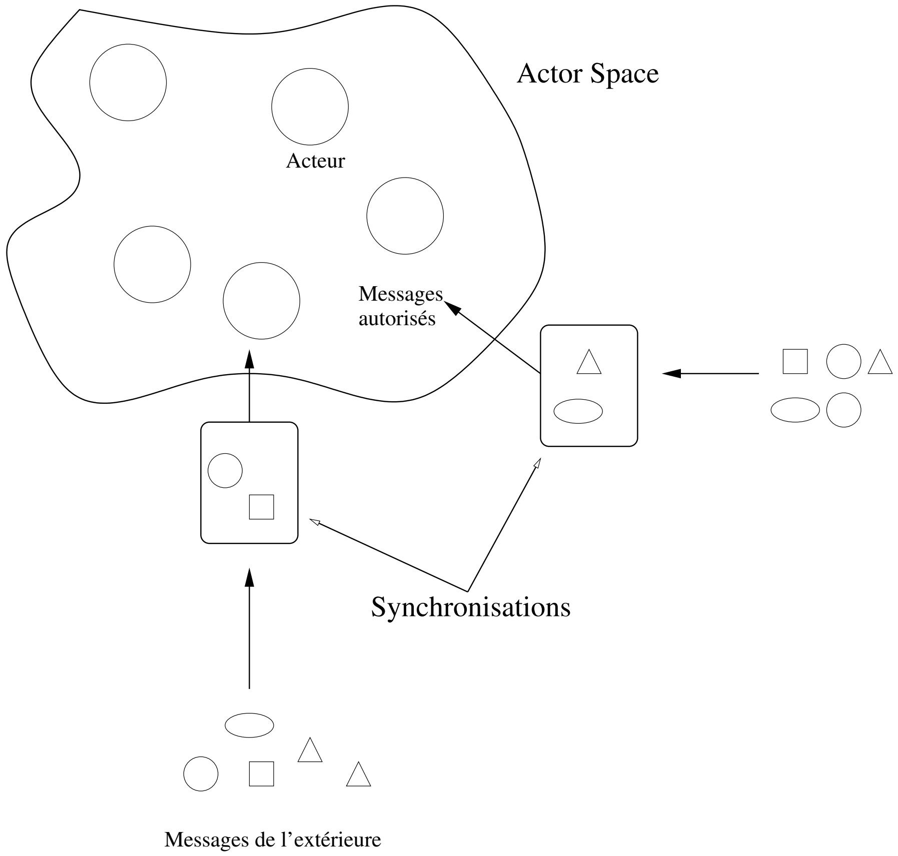


Fig. 2.4 – *ActorSpace et synchronisations associées*

Communications de groupe Les communications point-à-point permettent d'exprimer clairement la localité. Pour de nombreuses applications, il est cependant nécessaire de proposer une vision plus large. L'expression de communications mettant en jeu un groupe d'acteurs est un élément indispensable du modèle de programmation par acteurs. Plutôt que d'introduire une gestion de groupe explicite, les notions d'*ActorSpace* et d'interactions dirigées par des motifs de communication sont présentées.

- Un espace d'acteur (ou *ActorSpace*) est un conteneur passif d'acteurs qui sert de contexte à un ensemble de motifs de communication.
- Un motif de communication est une entité décrivant un ensemble de messages qui peuvent être reçus par un (*unicast*), plusieurs (*multicast*) ou l'ensemble (*broadcast*) des acteurs de l'*ActorSpace*.

Pour réaliser ces communications de groupe, il est encore une fois important d'introduire un mécanisme de synchronisation dont le rôle est de définir l'ensemble des messages acceptés par un *ActorSpace* (*cf. figure 2.4*). Des contraintes d'atomicité et de causalité doivent également être introduites pour garantir la généralité de cette communication de groupe. Ces mécanismes de synchronisation inter-acteurs sont présentés en section suivante.

Mécanismes de synchronisation

Nous avons vu dans la section précédente que deux mécanismes de synchronisation devaient être introduits pour permettre l'expression de modes de communications de haut niveau. D'un point de vue plus général, les besoins en terme de synchronisation proviennent de la nature concurrente des applications basées sur les acteurs. Deux types de synchronisation sont mises en jeu :

1. synchronisation intra-acteur liée au comportement local,
2. synchronisation inter-acteur liée à un comportement de groupe.

Les mécanismes décrits dans [Frø96] répondent à un cahier des charges très précis. Les caractéristiques génériques de ces mécanismes sont les suivantes :

- Expressions localisées : la synchronisation doit pouvoir être exprimable sans intrusion dans le système d'acteurs global
- Réutilisabilité : les primitives et constructions mises en jeu doivent pouvoir être réutilisées dans de nombreuses situations différentes.
- Transformabilité : la sémantique de la synchronisation doit être exprimable dans le modèle de programmation par acteurs.

La notion de réutilisabilité est ici assez déroutante puisqu'introduite au niveau des constructions de synchronisation et non des descriptions fonctionnelles.

Synchronization constraints Lors de l'invocation synchrone d'opération, l'acteur émetteur doit être bloqué en attendant la réception de la réponse à sa requête. Ce blocage ne doit pas mettre en doute le caractère autonome des acteurs. Ainsi, les messages reçus (autres que la réponse attendue) doivent être enregistrés dans la file de messages. Mais aucun traitement ne doit être déclenché. Le mécanisme de synchronisation mis en jeu doit donc servir de filtre à messages au niveau de l'extraction depuis la file de messages. Certains de ces messages (comme la réponse) doivent être traités mais l'acteur doit rester inactif pour les autres messages. Un moyen simple pour exprimer ce genre de fonctionnalité est de désactiver certaines opérations en fonction d'un état abstrait. Chaque message est en effet associé à une opération, il est donc plus simple de gérer la synchronisation des traitements (les méthodes) que de gérer les causes de ces traitements (les messages). En interne, le mécanisme effectuera automatiquement la transition.

Les *synchronization constraints* introduisent de nouvelles primitives dans les langages et frameworks d'acteurs. L'unique construction mise en oeuvre peut être représentée de la façon suivante :

```
disable
  method1(x1,...,xN) if expr(x1,...,xN)
  method2(y1,...,yN) if expr(y1,...,yN)
```

Cette construction indique simplement que la méthode method1 ayant pour paramètres x1, ..., xn est désactivée si l'expression expr, fonction de x1, ..., xn est évaluée à vrai. Une autre méthode, method2, est également contrainte. Il est de plus possible de tout désactiver sauf une méthode avec le constructeur all except. Ceci nous permet de réaliser le blocage d'un acteur émetteur pour une invocation synchrone :

```
Actor requestor
  waiting:=false;
disable
  all except continuation if waiting=true
method invoke(request, receiver)
  send(request, continuation) to receiver;
  waiting:=true;
end invoke
method continuation(reply)
  print("the result is: ", reply);
end continuation
```

Un autre critère important relatif à ces contraintes de synchronisation concerne leur possible *réutilisation*. Le mécanisme proposé ici apporte également une solution aux problèmes évoqués dans [MY93]. L'héritage est donc autorisé sur les constructions de synchronisation. Pour illustrer cette propriété, nous allons reprendre de [MY93] l'exemple célèbre du buffer borné et de sa spécialisation. Voici un extrait de définition pour un acteur BoundedBuffer :

```
Actor BoundedBuffer
  size:=0;
disable
  put if size=MAX
  get if size=0
method put(item) ... end put
method get(client) ... end get
```

Nous nous concentrons ici sur l'aspect de synchronisation lié à la mise en œuvre d'un tel acteur. Les méthodes put et get permettent respectivement l'ajout et le retrait de données dans le buffer. Ces méthodes sont synchronisées sur la taille actuelle du buffer (variable size). Il est alors très simple dans le framework Broadway de réutiliser cette définition pour construire par héritage une solution au problème du buffer avec possibilité de retrait d'éléments 2 par 2 :

```
Actor BoundedBuffer2 extends BoundedBuffer
  size:=0;
disable
  get2 if size<=1
method get2(client) ... end get2
```

L'intérêt de ce mécanisme réside à la fois dans sa simplicité et son autonomie :

- Simplicité car l'unique fonctionnalité proposée est la désactivation de méthode en fonction d'une condition. Il n'est pas possible de mélanger différents mécanismes (gardes, enabled sets, etc.).
- Autonomie par séparation du code des méthodes et du code de synchronisation. Notons toutefois que l'héritage est global car lié à l'acteur tout entier. Il est donc difficile d'établir des modèles de synchronisation génériques et indépendants des fonctionnalités.

Certaines extensions sont prévues dans le mécanisme de *synchonizations constraints* :

- Priorité des messages : pour un ensemble de messages de même type en attente de traitement, il est possible de déterminer une relation d'ordre sur la diffusion de ces messages.
- Récupération de messages : après un certain temps, des messages en attente peuvent perdre de leur intérêt, l'ajout d'une construction permettant de les enlever de la file d'attente sans effectuer les traitements correspondants peut se révéler utile.
- Désynchronisation : l'héritage est ici asymétrique, il serait intéressant de pouvoir ré-autoriser une méthode par héritage (mot-clé reenable). Cette extension n'est toutefois que proposée et non mise en œuvre dans [Frø96].

Synchronizers L'utilisation des *synchronization constraints* permet d'exprimer et de réutiliser très facilement la synchronisation interne des acteurs. Nous avons cependant vu que des besoins de synchronisation inter-acteurs émergeaient des modes de communication de groupe. Les *synchronizers*, également présentés dans [Frø96], permettent cette synchronisation partagée.

Leur conception provient du constat que la synchronisation inter-acteurs pouvait être exprimée de façon indépendante. Les *synchronizers* sont donc des entités logicielles autonomes, servant de points d'entrée dans des *ActorSpaces*. Un message ne peut être émis à un ou plusieurs membres d'un groupe que s'il peut passer «à travers» un *synchronizer*. Prenons l'exemple d'un ensemble de boutons de commande. Chaque bouton est un acteur défini de la façon suivante :

```
Actor Button
  isOn:=false;
disable
  on if isOn
  off if not isOn
method on() isOn:=true; ... end on
method off() isOn:=false; ... end off
```

La sémantique du groupe impose qu'un seul de ces boutons soit actif à un moment donné. Ceci conduit à l'élaboration du *synchronizer* RadioButtons agissant sur un groupe ou *ActorSpaces* buttons. Ce *synchronizer* est associé à un état interne qui indique si un bouton est activé dans le groupe. Si c'est le cas, alors tout message d'activation de l'extérieur doit être refusé (ou plutôt temporisé). L'état interne du *synchronizer* doit de plus être maintenu de façon consistante avec l'état du groupe. Voici la définition de ce synchronizer :

```
Synchronizer RadioButtons(buttons)
 activated:=false;
 for b in buttons :
   trigger
     b.on -> { activated!=true; }
     b.off -> { activated!=false; }
 for b in buttons :
   disable
     b.on if activated
```

Le constructeur for .. in permet d'effectuer des itérations sur les membres d'un *ActorSpace*. Le mot-clé trigger introduit un filtre de messages composé de motifs et d'actions à exécuter si le motif est associable à un message en entrée. Ceci permet dans notre exemple de modifier l'état interne du *synchronizer* pour le maintenir cohérent. Le mot-clé disable permet de constuire des contraintes de synchronisation au niveau du *synchronizer*.

Une autre fonctionnalité intéressante des *synchronizers* est la possibilité de rendre atomique la diffusion de messages dans un *ActorSpace*. Prenons l'exemple célèbre du dîner de philosophes : n philosophes sont assis autour d'une table et disposent de n baguettes. Un philosophe ne peut manger que s'il dispose de deux baguettes. En prenant le problème du point de vue des baguettes, la condition à satisfaire pour éviter les interblocages est l'atomicité dans la prise des deux baguettes. Chaque baguette est définie de la façon suivante :

```
Actor ChopStick
 isPicked:=false;
disable pickup if isPicked
method pickup(phil) isPicked:=true; ... end pickup
method drop() isPicked:=false; ... end drop
```

Nous pouvons maintenant définir le *synchronizer* qui relie deux baguettes et un philosophe :

```
Synchronizer IndivisiblePickup(c1,c2,phil)
 atomic
   ( c1.pickup(p1) if p1=phil,
     c2.pickup(p2) if p2=phil)
```

La construction atomic exprime ici que l'action de prendre c1 et c2 par un même philosophe phil est indivisible : on prend les deux baguettes d'un coup ou on ne prend rien.

DIL : une vision protocolaire des intéractions entre acteurs

L'introduction de mécanismes de synchronisation évolués offre un certain confort dans l'implémentation d'applications réparties complexes dans le modèle d'acteur. Les *synchronization constraints* permettent ainsi très aisément de décrire la synchronisation interne des acteurs. Le cas des *synchronizers* est un peu différent car si leur conception même ne pose pas problème, leur utilisation n'est pas évidente. La gestion explicite de groupe est effectivement assez complexe. Une vision de plus haut niveau consiste à définir des protocoles d'intéraction introduisant une gestion implicite de groupe (liée à la sémantique du protocole). Passer de protocoles d'intéraction à des créations/destructions d'*ActorSpaces* et de *synchronizers* bien localisés n'est cependant pas chose aisée. Le langage d'intéraction *DIL* [Stu96], élaboré au dessus du framework *Broadway*, permet de faire le lien entre vision protocolaire des applications et expression de la synchronisation inter-acteurs émergeant de cette vision.

DIL est un langage d'interactions qui permet de décrire par des constructions syntaxiques appropriées des protocoles mettant en jeu des groupes d'acteurs.

Définition de protocoles La spécification d'un protocole passe par la définition de rôles attribués aux membres du groupe et de fonctionnalités globales. Prenons l'exemple d'un protocole de validation atomique de transactions à deux phases. Deux rôles sont mis en jeu dans ce protocole :

1. L'initiateur qui réalise la transaction et demande la validation ou l'annulation
2. Les serveurs qui valident ou annulent une transaction.

Ce protocole est décrit [4] de la façon suivante en DIL :

```
protocol Two-Phase {

  initialize(actor init)
  addServer(actor server)

  role Initiator {
    MsgList currentAction;
    event beginAction();
    event endAction();
    event transmit(Msg m);
    votePrepared(actor resp);
    voteAbort();
    committed(actor resp);
  }

  role Server {
    MsgList currentAction;
    prepare(MsgList ml);
    commit();
    doAbort();
  }
}
```

Les opérations globales sont ici initialize() et addServer() ; leur but est d'enregistrer les participants à la transaction. La gestion du groupe est donc implicite car liée au protocole et non à des primitives de gestion de groupe.

Le premier rôle décrit est celui de l'initiateur. Ce dernier gère une liste qui enregistre les différents messages émis lors de la transaction pour restauration en cas de panne franche éventuelle. Trois événements sont gérés à ce niveau : beginAction() qui est tiré au démarrage d'une transaction, endAction() qui en indique la terminaison et transmit() qui représente tout simplement l'envoi de message. Un événement est une opération particulière qui est réalisée par l'acteur (indépendamment du protocole) et captée par le protocole. beginAction() et endAction() doivent être définies au niveau de l'acteur pour satisfaire le rôle du protocole [5]. En revanche, transmit() est un événement canonique que tout acteur propose (événement généré par la primitive send). Les opérations du rôle Initiator sont la préparation du vote et l'annulation ou la validation de la transaction au niveau du client.

Le rôle serveur n'est composé que d'opérations permettant de restaurer un vote avant une panne (prepare()), de voter oui (commit()) ou non (doAbort()) pour une transaction donnée.

Les protocoles *DIL* sont instanciables au moment de l'exécution dans le framework *Broadway*. Nous pouvons voir dans notre exemple que le protocole de validation à deux phases ne fait que peu de présomptions sur le contenu des acteurs qu'il attribue aux rôles. Cette caractéristique permet une grande généricité dans la description des algorithmes. Deux autres caractéristiques très intéressantes sont la modularité des protocoles grâce à l'utilisation d'instances composables et la réutilisabilité offerte dans *DIL* par un mécanisme d'héritage. En ce qui concerne la synchronisation, chaque rôle du protocole peut désactiver ou rendre atomique la diffusion de messages. Il s'agit donc d'un compromis entre synchronisation localisée (dans chaque rôle) et indépendante (au niveau de l'instance du protocole). Les *synchronizers* ne sont donc pas utilisés tels quels mais de nombreux résultats les concernant sont exploités au sein de *DIL*.

4. Par souci de consision, nous n'avons indiqué ici que les déclarations des rôles, données et opérations.
5. Il s'agit d'une contrainte de couplage entre niveau acteur et niveau protocole.

Un premier bilan

De l'analyse du paradigme acteur proposée ci-dessus, nous pouvons extraire un certain nombre de remarques nous permettant de resituer cette analyse dans le débat qui nous occupe aujourd'hui.

Tout d'abord, les acteurs représentent - à notre connaissance - le seul formalisme asynchrone dans lequel la composition dynamique est érigée eu principe absolu. En effet, comme nous pouvons le constater sur les exemples donnés précédemment, les acteurs se créent, se recréent et disparaissent tout au long du cycle de vie des systèmes qu'ils opérationnalisent. Nous pouvons trouver dans l'approche DIL un cadre très intéressant pour l'expression à un haut niveau d'abstraction de cette adaptation perpétuelle des systèmes d'acteurs.

Le modèle de communication fait également une distinction (certes, quelque peu floue) entre *acteurs* et *messages* échangés par ces derniers. Nous retrouverons cette dichotomie dans une version plus strict dans le monde des composants logiciels (cf. section 2.2.2). Mais cette intuition se retrouve également dans le monde des acteurs, il est important de le noter.

Malgré toutes les qualités de ce formalisme, incroyablement en avance sur son temps si l'on s'en réfère aux dates de parutions des premiers travaux sur les acteurs, l'approche semble avoir eu du mal à s'imposer dans le monde de la concurrence et de la distribution où ils exhibent pourtant de nombreux avantages sur la «machinerie» objet. Le principal reproche effectué à l'encontre de ce modèle est son relatif «exotisme». L'adoption des acteurs semble effectivement passer par une étape préliminaire de remise en question très profonde qui n'est apparemment plus possible de nos jours.

Durant les années 80/90, d'autres approches ont tenté de circonvenir aux problèmes semble-t-il inhérent au passage d'un monde objet à un monde acteur. Tout en se basant sur des modèles de comportement et de communication asynchrone, certaines de ces approches ont réussi le tour de force de concilier *pureté* des principes et pragmatisme opérationnel. Ces approches se fondent généralement sur la notion d'objet actif, nous allons nous en remémorer deux qui ont marqué, à notre avis, les langages informatiques de leur empreinte.

2.1.2 Le projet ABCL

Le but du projet *ABCL*, largement détaillé dans [Y^+89], est de faciliter la conception et l'implémentation d'applications parallèles. En d'autres termes, il s'agit d'une tentative de rapprochement entre parallélisme et conception objet. Les fondements de ce projet proviennent du modèle de calcul *ABCM/1* et de son langage associé *ABCL/1* qui introduisent la notion d'objet concurrent. Sur ces fondations, des évolutions réflexives ont été conçues : *ABCL/R* [Ish91], *Rbcl* [IMY92] et *ABCL/R2* [MW91]. Ces approches réflexives sont très intéressantes car accompagnées de considérations «formelles».

Le modèle de calcul ABCM/1

Le modèle de calcul *ABCM/1* est une variante plutôt pragmatique des acteurs. Contrairement au modèle de calcul par acteurs, *ABCM/1* ne propose pas une nouvelle sémantique au niveau des processus séquentiels ou des systèmes fonctionnels. Il s'agit plutôt d'une évolution du paradigme objet et des modèles associés. *ABCM/1* effectue en quelque sorte un retour aux sources, au risque de perturber la justification sémantique du modèle. Le recouvrement de comportement, sous-jacent du modèle de calcul par acteurs, est remplacé dans *ABCM/1* par la caractérisation par états plus proche des objets.

Les objets dans ABCM/1 Les entités logicielles manipulées dans *ABCM/1* sont des objets, au sens traditionnel du terme, c'est à dire composés des informations suivantes :

- un état interne
- des fonctionnalités, ici appelées scripts

L'état interne est défini par un ensemble de couples associant nom d'attribut et valeur. Les scripts sont des constructions composées par :

- un motif de message
- une contrainte sur l'état interne (optionnelle)
- une ou plusieurs actions

Les actions d'un script sont exécutées lorsqu'un message en réception correspond au motif acceptant et si l'expression de contrainte sur l'état interne est vérifiée (ou absente).

Les actions réalisables dans un script sont catégorisées de la façon suivante :

- envoi de message (*cf. paragraphe suivant*).
- création d'objets
- référencement et modification d'attributs de l'état interne
- opérations diverses (arithmétiques, logiques, etc.) sur les valeurs et variables de l'état interne et les contenus des messages.

Les objets *ABCM/1* ne diffèrent donc à ce niveau que peu des objets traditionnels. Le langage *ABCL/1* propose un cadre syntaxique pour la réalisation d'objets *ABCM/1*. Cette syntaxe est très proche de *Common Lisp* mais propose certaines formes spéciales de nature impérative comme l'émission de messages ou l'affectation de variable ([variable := valeur]).

Voici la forme générique d'un objet *ABCL/1* :

```
[object Nom
 (state attribut1 attribut2 ... attributN)
 (script
  ( => [:motif_message1] where contrainte1
    ;; actions
  )
  ( => (:motif_message2] where contrainte2
    ;; actions
  ))]
```

L'état de l'objet Nom est défini par la construction state, sous la forme très «lispienne» d'une liste de noms d'attributs (sans information de type). Le script de l'objet décrit un ensemble de motifs susceptibles d'activer l'objet et de déclencher des actions comme la modification de son état interne ou la communication avec d'autres objets. Nous pouvons voir qu'au-delà des quelques subtilités du langage (comme les contraintes sur les messages), nous nous sommes ici rapprochés - si l'on compare aux acteurs - d'un modèle objet assez traditionnel. Notons qu'*ABCL/1* est également un environnement d'exécution très complet, essentiellement destiné aux machines parallèles.

Modes de communication L'introduction de la notion de concurrence se fait, comme pour les acteurs, essentiellement au niveau du processus de communication. La modification majeure concerne le passage d'un mode de contrôle implicitement synchrone à un fonctionnement asynchrone. L'envoi de message par défaut est donc asynchrone : un objet peut envoyer un message à n'importe quel moment et sans interruption dans son exécution. Ce type de transmission est appelé *past* dans *ABCM/1*.

Dans le langage *ABCL/1*, ce type d'envoi est réalisé par la construction suivante :

```
[ destination <= message ]
```

Deux autres modes de transmission sont proposés dans le modèle de calcul. Le premier est le mode *now* qui incarne simplement l'émission synchrone. La construction correspondante est :

```
[ reponse := [ destination <== message ] ]
```

Une variante plus originale et très intéressante car augmentant potentiellement le degré de concurrence est proposée par le mode *future*. Dans ce cas, la source de l'émission reçoit un objet représentant le futur de la réponse (*future object*). Cet objet dispose d'un attribut indiquant si la réponse est disponible. Contrairement au mode *now*, l'objet source peut continuer son exécution et vérifier quand le besoin s'en fait sentir si la réponse est disponible. *ABCL/1* propose la syntaxe suivante pour l'émission future :

```
[ destination <= message $ future ]
```

Le prédicat (ready? future) permet de tester si la réponse future est disponible[6].

La disponibilité des deux modes *now* et *future* est justifiée par la possibilité de les réaliser simplement par transformation de programme. Ainsi, l'unique mode de communication nécessaire au niveau de la couche de communication est l'émission asynchrone.

Certaines propriétés importantes de système de communication sont établies :

- pas de diffusion globale (*broadcast*), les envois sont soit point-à-point (*unicast*), soit multi-points (*multicast*). L'envoi *multicast* est réalisable par simple listage des destinations : [[dest1 dest2 dest3] <= message]
- la topologie de connexions entre objets est dynamique : seules des informations locales sur les autres objets sont utilisées pour l'émission de messages.
- livraison garanti et ordonné (pas de réceptions simultanées)
- livraison dans l'ordre d'émission : hypothèse forte mais irrévocable
- pas d'horloge globale

Ces propriétés sont assez proches de ce que propose intrinsèquement le modèle de calcul par acteurs. Cependant, nous sommes ici dans une vision plus «réaliste» mais moins formelle du problème.

Aspects comportementaux La sémantique complexe du système de messagerie associé aux objets dans *ABCM/1* implique une modification de certains aspects comportementaux dans le modèle de base. Dans le paradigme objet, les objets sont passifs par défaut et activés lors de la réception de message. Ce mécanisme se révèle insuffisant dans le cadre d'*ABCM/1*. Prenons l'exemple d'un objet BoundedBuffer décrit de la façon suivante en *ABCL/1* :

```
[object BoundedBuffer
  (state contents count max full empty)
  (script
    (=> [:put anItem]
      (when full
        (wait-for (=> [:get] )))
      (put-item contents anItem)
      (++ count)
      [full:=(= count max)]
      (if empty [empty:=false]))
    (=> [:get]
      (when empty
        (wait-for (=> [:put anItem]))
      (– count)
      [empty:=(= count 0)]
      (if full [full:=false])
      (! (get-item contents)))))]
```

Cet objet contient deux scripts (ou méthodes) : put et get. Le script put ajoute un élément dans le buffer si ce dernier n'est pas plein. Dans le cas contraire, une attente sur un message get est décrite par la construction (wait-for ...). Ce mode d'attente est indispensable pour la réalisation de synchronisations.

Les états comportementaux possibles d'un objet ABCM/1 sont donc au nombre de trois :

1. endormi (*dormant*) : attente de n'importe quel message permettant l'activation d'un script,
2. actif (*active*) : un traitement est en cours, les messages entrants sont enregistrés,
3. en attente (*waiting*) : attente d'un certain type de message pour réactivation (construction wait-for).

Le graphe d'états/transitions représentés sur la figure 2.5 montre les relations entre états et comportement des objets.

6. En fait, un système de file de réponses futures est proposé.

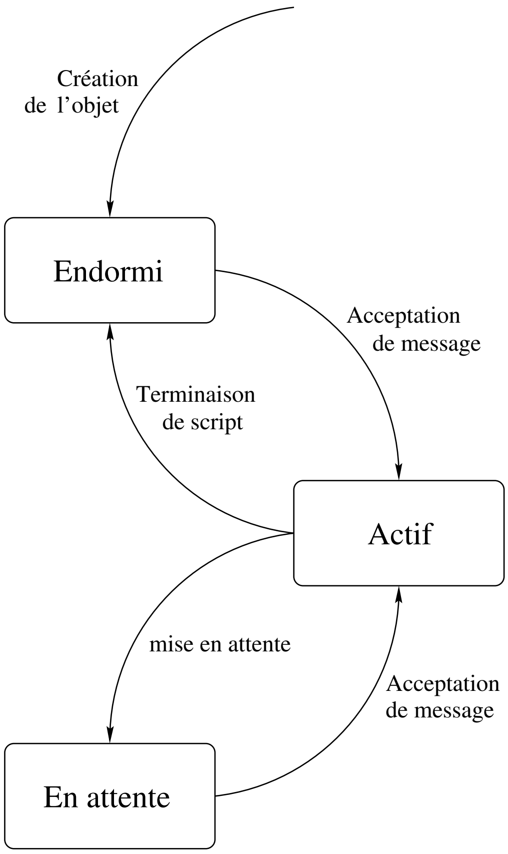


FIG. 2.5 – *Etats et transitions d'un objet ABCM/1*

Aspects structurels Le comportement des objets ABCM/1 est très proche de celui des acteurs. Notons principalement les points communs suivants :

– une unique activité est associée à chaque objet
– enregistrement des messages non traitables instantanément (modes *active* ou *waiting*)

Ce fonctionnement impose aux objets *ABCM/1* une structure relativement proche de celle des acteurs (de haut niveau). Cette structure se compose des éléments suivants :

– un système de communication local
– une boîte à lettres pour les messages reçus et en attente de traitement
– un mécanisme d'activation lors de la réception d'un message acceptable par un script dans l'état *active* ou *waiting* (si le message est celui attendu).

La structure d'un objet *ABCM/1* est représentée sur la figure 2.6 page suivante.

Une hybridation réussie

En faisant certes quelques entorses à l'approche purement ascendante des acteurs[7], le projet ABCL nous propose un modèle consensuel qui semble réunir les avantages des deux mondes : acteurs et objets.

Certes, les langages issus de ce projet ressemblent plus fortement à des langages à prototypes qu'à des langages à classes[8] mais plus que les résultats eux-mêmes, c'est la portée de cette contribution qui nous semble importante : il est désormais possible de concilier la structure objet et le comportement asynchrone que tout semblait séparer.

7. Le modèle de calcul proposé est plus pragmatique, à la sémantique moins opérationnelle que celle des acteurs.
8. Par exemple, la délégation est proposée comme alternative nécessaire à l'héritage [BY89].

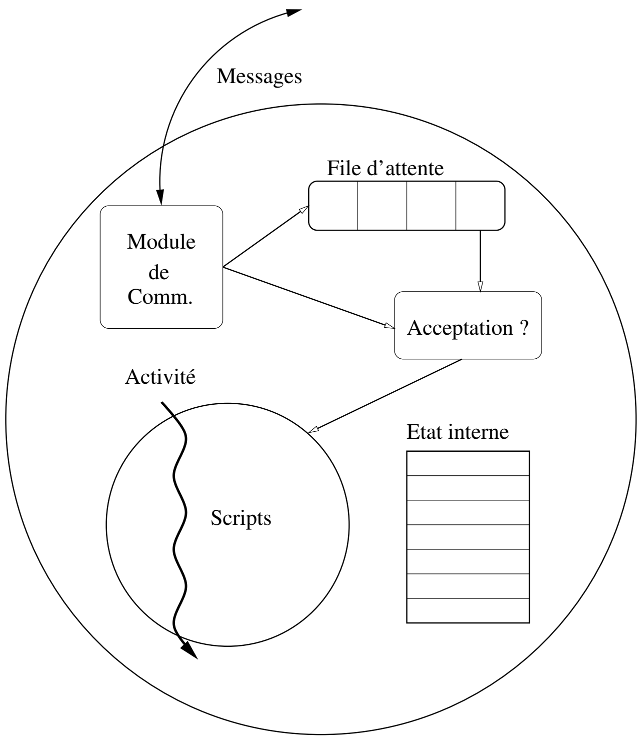


Fig. 2.6 – *Structure d'un objet ABCM/1*

En particulier, les modes de communication de haut niveau du langage ABCL/1 montrent qu'il est possible de proposer des interactions synchrones de type client/serveur à partir de la composition d'interactions de plus bas niveau et elles asynchrones; et ce, de façon intrinsèque au modèle.

2.1.3 Actalk

Pour terminer notre tour d'horizon - non exhaustif - des approches asynchrones, nous allons décrire la proposition de Jean-Pierre Briot, au travers de la boîte à outils Actalk pour immerger le langage Smalltalk/80 dans un monde asynchrone [Bri89].

Des acteurs aux objets actifs

La démarche d'*Actalk* est, en quelque sorte, inverse à celle des acteurs ou d'*ABCL*. Plutôt que de remettre en cause le paradigme objet et la sémantique qui lui est usuellement associée, *Actalk* propose d'étendre l'environnement *Smalltalk/80* et son modèle objet «standard» pour résoudre les nombreux problèmes liés à la programmation concurrente. *Smalltalk* est en effet un environnement hautement extensible par voie applicative. Les extensions ainsi développées peuvent ensuite s'intégrer au langage grâce à des mécanismes réflexifs structurels (protocole de méta-classes) et calculatoires (doesNotUnderstand:).

Nous avons vu que de nombreux problèmes relatifs à la concurrence étaient naturellement résolus par l'introduction d'entités logicielles dont le mode de fonctionnement est asynchrone. Contrairement aux acteurs, les objets *Actalk* sont des objets «traditionnels» mais disposant d'un comportement modifié. L'asynchronisme est réalisé par génération d'activités associées aux objets qui deviennent ainsi objets actifs.

Architecture du Framework

Pour atteindre son statut de plateforme d'expérimentation, *Actalk* propose un certain nombre de propriétés fondamentales :

1. La simplicité : l'utilisation des concepts et outils proposés se fait de façon intuitive et cohérente avec la «philosophie» Smalltalk.

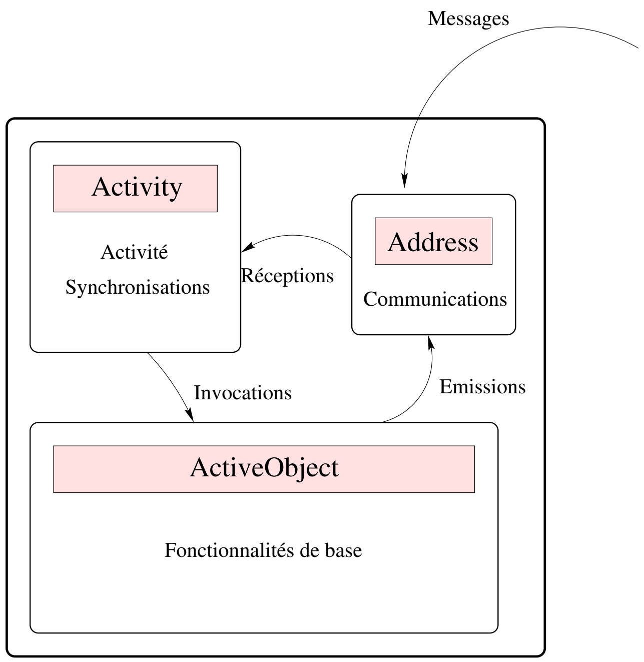


Fig. 2.7 – *Structure d'un objet Actalk*

2. L'évolutivité : le framework lui-même propose une structure simple et évolutive.
3. La flexibilité : les abstractions proposées peuvent être composées pour résoudre des problèmes de natures variées (synchronisation, ordonnancement, etc.).
4. La réutilisabilité : les applications et spécialisations du frameworks doivent être réutilisables dans d'autres contextes.

Ce cahier des charges, assez contraint, était partiellement satisfait dans la version préliminaire d'*Actalk*. En particulier, le modèle était tellement simple que l'on peut parler de minimalisme. Les versions plus récentes [Bri97] augmentent le caractère flexible et réutilisable d'*Actalk*, au prix d'une complexité plus importante.

Pour prendre en compte la modification des interactions entre objets (du modèle synchrone/passif original au modèle asynchrone/actif), le principal outil employé est la réflexion. Il est cependant nécessaire de distinguer deux points de vue à ce niveau :

1. Le point de vue conceptuel : les aspects relatifs à la modification du comportement sont réifiés de façon ad-hoc par le raffinement de la structure des objets.
2. Le point de vue langage : l'intégration dans l'environnement *Smalltalk* est réalisée par utilisation judicieuse des mécanismes réflexifs disponibles. Nous ne reviendrons pas sur cet aspect.

Les aspects considérés dans *Actalk* pour la réification sont les suivants :

– Les fonctionnalités de base des objets et la façon dont elles sont exprimées.
– Leur caractère actif (manipulation de l'activité).
– le contrôle de la transmission des messages entre objets actifs émetteurs et récepteurs.

Le noyau d'un objet *Actalk* est donc défini par un assemblage de trois composantes structurelles implémentant ces divers aspects. Trois classes *Smalltalk* sont ici mises en jeu (*cf. figure 2.7*) :

– La classe ActiveObject qui, une fois spécialisée, sert de cadre à la réalisation du programme de base associé à l'objet.
– La classe Activity permettant la spécialisation du caractère actif (et autonome) de l'objet. La gestion de la sémantique d'invocation des méthodes (sélection, ordonnancement, exécution) est également sous la responsabilité de cette classe.
– La class Address qui implémente la sémantique de transmission des messages.

Composant	Spécialisation possible
ActiveObject	ActorObject (remplacement de comportement), Abcl1Object (wait-for)
Activity	Méthodes gardées, enabled sets
Address	Envois synchrones et futurs, mode de transmission express

TAB. 2.1 – *Spécialisations des composants du noyau d'Actalk*

Ces choix peuvent sembler de prime abord assez déroutants. Il faut cependant garder en mémoire deux motivations principales du framework : réutilisabilité et extensibilité. Ainsi, il est important d'aborder les spécialisations possibles des trois différents composants décrits ci-dessus pour comprendre le caractère générique du modèle. Certaines de ces spécialisations sont présentées dans la table 2.1, elles couvrent trois domaines :

- Constructions et modèles d'objets concurrents
- Modèles de synchronisation
- Modèles de communication

Il existe donc une relation bijective entre les domaine de spécialisation et les composantes structurelles spécialisables par héritage. Un autre intérêt majeur de cette décomposition structurelle concerne l'indépendance relative entre les différents composants réifiés. Il est ainsi possible d'associer un «comportement» de type Acteur (remplacement de comportements) avec un modèle de synchronisation de type méthodes gardées et un modèle de communication «réponse anticipée».

Actalk propose également un mécanisme de méthodes événementielles génériques. Ces méthodes manipulent des événements de trois types : réception, acceptation et terminaison d'une invocation. Ce mécanisme réflexif autorise l'ajout de méta-fonctionnalités comme le traçage des activités, l'exécution pas à pas, les post-traitements, etc.

Hiérarchisation des modèles de synchronisation

Actalk a été utilisé pour constuire un framework générique de mécanismes de synchronisations d'activité [Bri96]. Ce framework est composé d'un ensemble de classes dérivées de la classe Activity. Elles implémentent chacune un modèle (canonique ou composé) de synchronisation. Les modèles de base proposés sont les suivants :

- Méthodes gardées : une condition d'activation est attachée aux méthodes. Cette condition doit être satisfaite pour qu'une méthode soit exécutée. Ce modèle est implémenté par la classe GuardsActivity.
- Enabled sets : Dans ce modèle, un état abstrait est associé à un ensemble de méthodes. Si l'objet synchronisé est dans cet état, alors seules les méthodes de l'ensemble associé sont exécutables. Plusieurs états et ensembles associés peuvent coexister au sein d'un même objet. La classe EnabledSetsActivity est utilisée pour réaliser ce modèle de synchronisation.

La réutilisation/spécialisation d'un modèle se fait simplement par héritage. Par exemple, un raffinement possible de EnabledSetsActivity consiste à toujours autoriser l'exécution de certaines méthodes, indépendamment des états abstraits et ensembles associés. La classe dérivée AlwaysEnabledSetsActivity implémente cette extension.

Les méthodes gardées sont efficaces lorsqu'une seule activité existe dans un objet. Si l'on autorise l'exécution simultanée de plusieurs méthodes au sein d'un même objet, un raffinement du modèle est possible. Il s'agit dans ce cas d'autoriser dans l'expression des gardes l'utilisation de compteurs de synchronisation. Ces compteurs enregistrent le nombre d'invocations reçues, acceptées et terminées. La class ConcurrentActivity autorise les activités multiples au sein d'un objet *Actalk*. SynchroConcurrentActivity ajoute des primitives de synchronisations internes (exclusion mutuelle). La classe CounterActivity qui introduit les compteurs de synchronisation dans le modèle des méthodes gardées hérite à la fois de GuardsActivity et SynchroConcurrentActivity[9]. D'autres raffinements et compositions sont proposés :

- Composition de méthodes gardées/*enabled sets* (classe EnabledSetsCountersActivity)

9. L'héritage multiple n'étant pas proposé dans Smalltalk, la duplication du code d'une des classes parentes est nécessaire.

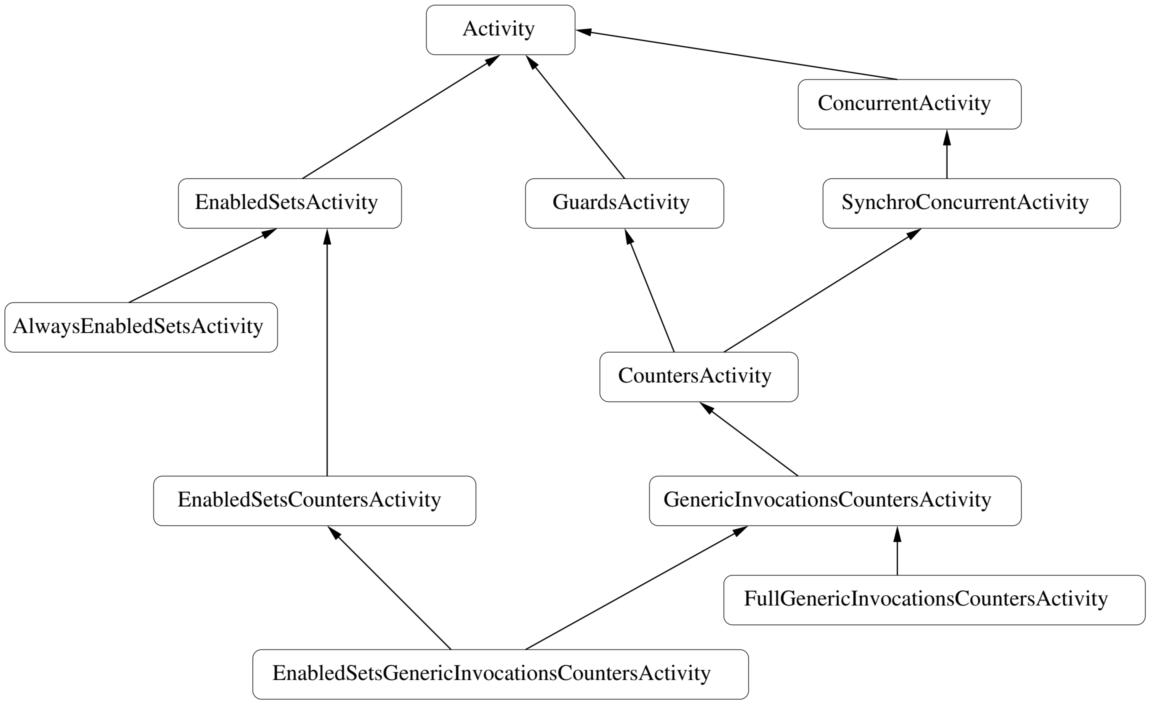


FIG. 2.8 – *Arborescence des modèles de synchronisation proposés dans Actalk*

– implémentation des *synchronization variables* (conditions de haut niveau : ordre d'invocation, priorités, etc.). Le modèle proposé dans ce cas autorise l'ajout d'information aux invocations et l'explicitation de prédicats portant sur ces informations. Ces prédicats sont vérifiés au moment de l'invocation, ce qui explique le nom d'invocation générique donné à ce modèle. La classe GenericInvocationsCountersActivity réunit ce modèle et les méthodes gardées étendues avec des compteurs de synchronisation.

L'ensemble des combinaisons proposées sont présentées en figure 2.8. L'arborescence correspond à la relation d'héritage (multiple) entre classes implémentant les différents mécanismes évoqués ci-dessus.

Vers des approches flexibles

Actalk se distingue nettement des deux approches précédentes par son caractère très versatile. En effet, cette proposition n'effectue aucun choix totalement définitif et laisse toujours son «utilisateur» capable de remettre en question ces choix, dans la plus pure interprétation du concept de boîte-à-outils[10].

Ainsi, la *flexibilité* revient au centre du débat et propose de faire une distinction extrêmement claire entre les concepts (acteurs/objets actifs) et leur mise en œuvre, ici dans un langage qui ne connaît a priori que les objets «traditionnels».

Nous retiendrons de ces approches asynchrones - désormais entrées dans «l'histoire» et malheureusement trop souvent oubliées - qu'elles permettent toutes de décrire, au niveau même des modèles, des comportements qui reflètent parfaitement le cadre opérationnel sous-jacent. Le fait de reposer sur une sémantique asynchrone fait qu'a priori, les entités mises en jeu - et contrairement aux objets - sont indépendantes en terme de contrôle et que le couplage comportemental - ou synchronisation - doit être construit de toute pièce.

2.2 Point de vue structurel

Si le point de vue comportemental permet de caractériser les contraintes liées à l'opérationnalisation de systèmes dynamiques, les progrès en matière par exemple de souplesse d'expression nous apparaissent plus contestables. Toutes les constructions linguistiques et mécanismes associés sont effectivement essentiellement tournés

10. La métaphore n'est peut être pas la plus adéquate puisqu'il est plutôt rare de construire son propre marteau. C'est pourtant ce que propose Actalk via la construction de modèles d'objets actifs variés !

vers les aspects opérationnels. Il nous semble important - et c'est selon nous le noyau de notre contribution - de les reconsidérer sous l'angle des structures ou des architectures logicielles.

Les *composants logiciels* jouent ici, selon nous, un rôle fondamental pour réconcilier cette nécessité de structuration forte avec la problématique du comportement asynchrone.

2.2.1 Les notions de composant logiciel

Le concept de composant logiciel représente aujourd'hui l'un des sujets les plus «à la mode» en matière d'informatique. Dans notre étude, nous considérons principalement deux cheminements différents qui conduisent selon nous à des propositions réellement tangibles de ce concept.

Les architectures logicielles

Le thème de recherche des architectures logicielles [GS93] introduit la notion de composant pour permettre la description conceptuelle et autonome de l'architecture (ou de la structure) d'un système informatique; et ce, indépendamment de ses propriétés fonctionnelles/comportementales. De nombreux styles architecturaux ont ainsi été reconstruits et formalisés par cette approche [DVM95] : tubes et filtres, architectures en couche, types de données abstraits/objets, communications implicites, tableaux noirs, etc. Dans tous les cas, seules deux notions fondamentales sont employées :

- *Les composants* : qui servent à la description autonome d'une sous-partie de l'architecture qu'il n'est pas nécessaire de redécomposer d'un point de vue architecturel.
- *Les connecteurs* : qui décrivent les interactions possibles entre composants.

Il est donc très important de noter que dans cette vision des composants, le connecteur inter-composant est un concept au moins aussi important que le composant lui-même. En d'autres termes, la vision architecturelle réifie la relation de couplage des applications au travers du concept de connecteur.

Les composants binaires

L'autre grande tendance en matière de composant propose une motivation beaucoup plus «terre à terre» pour l'introduction de ce nouveau concept : l'ouverture d'un nouveau marché pour l'industriel du logiciel sous la houlette du géant Microsoft et de sa boîte à outils Com+ [Mic97]. En guise de principe fondamental [Szy98], ces composants doivent être fournis sous forme binaire et peuvent donc être vus comme de (pas si) lointains descendants des bibliothèques partagées. La différence principale réside à la fois dans l'encapsulation d'un état propre et dans la nature dynamique du chargement de ces composants. Notre problématique étant assez éloignée de l'aspect représentation binaire, nous retiendrons surtout de ces approches le fait qu'elles proposent des modèles de composant opérationnels.

Les composants pour la composition

Avec le très précoce *Fabrik* [IWC+88], les *JavaBeans* [Sun98] représentent à nos yeux une sorte de première tentative - relativement réussie - de concilier les deux points de vue évoqués ci-dessus. En effet, ces objets d'un genre un peu particulier abordent simultanément la problématique de composition conceptuelle, reprenant en cela les préceptes des architectures logicielles, et l'opérationnalisation (par transformation en code Java) de cette composition dans l'idée des composants binaires. Les approches plus récentes comme les *Enterprise JavaBeans* [Sun00] ou les *Corba Components* [OMG99] reprennent ces principes mixtes et les appliquent au domaine très en vogue des architecture N-tiers.

Pour synthétiser ces différentes approches hybrides, nous pouvons voir les composants d'une manière qui nous semblent plus adaptée :

> *Les composants sont des entités unitaires de composition de systèmes informatiques*

La *composition*, dans un sens de mécanisme opérationnel, semble être le maître mot ici et c'est bien sûr vers cette tendance que nous nous orientons. Ceci nous rapproche également de piccola components:piccola qui introduit également une vision opérationnelle - et formelle - des composants pour la composition.

2.2.2 Extraction de la relation de couplage

La notion de connecteur tient, dans l'idée des architectures logicielles, un rôle fondamental pour la composition logicielle puisqu'elle permet l'extraction de la relation de couplage entre composants.

Le modèle de composant introduit par le projet Rapide [Luc96] nous semble de ce point de vue particulièrement intéressant puisqu'il s'agit - à notre connaissance - de la première (et une des seules) immersion réussie du comportement asynchrone dans une enveloppe de type componentielle très aboutie où cette notion de connexion opérationnelle est fondamentale.

Les composants Rapide sont décrits au travers de *types* qui peuvent être assimilés à des classes dans la terminologie objet usuelle. Un type Rapide décrit des actions et un comportement permettant de relier ces actions aux *événements* qui lui sont envoyés par d'autres composants.

Le domaine d'application de cette approche concerne la spécification et la vérification expérimentales, par ordonnancement partiel d'événements, d'architectures de logiciels concurrents. Dans cette perspective, décrivons un petit exemple (simplifié) de type producteur/consommateur. Voici la définition du producteur :

```
type Producer(Max : Positive)
  action in Reply(N : Integer);
  action out Send(N : Integer);
behavior
  Start => Send (0);
  Reply(X : Integer) where X < Max => Send(X + 1);
end
```

Les composants de type Producer définissent deux actions :

1. une action entrante, Reply, qui permet de recevoir un entier,
2. une action sortante, Send, qui émet un autre entier en retour.

Le comportement de ce producteur est très simple puisqu'au démarrage il se borne à envoyer l'entier 0. Ensuite, à chaque réception par Reply d'un entier donné X, il émet l'entier successeur $X + 1$ à condition de ne pas dépasser une borne maximale Max.

Le type pour les consommateurs est défini de façon complémentaire :

```
type Consummer()
  action in Receive(N : Integer);
  action out Ack(N : Integer);
behavior
  Receive(X : Integer) => Ack(X);
end
```

Lors de la réception d'un entier par l'action Receive, les consommateurs émettent un événement d'acquittement Ack.

Nous voyons bien ici que malgré l'intention évidente du concepteur de faire communiquer ces deux composants, ces derniers ne font aucune référence explicite les uns envers les autres. Ainsi, lorsqu'un consommateur reçoit un entier, il ne sait pas de qui il l'a reçu et s'il sait qu'il doit envoyer une réponse, il ne sait pas non plus à qui le faire.

Pour décrire notre système producteur/consommateur, il nous reste donc l'étape importante de la spécification de la relation de couplage entre nos composants, au travers de la notion d'architecture :

```
architecture ProdCon()
  Prod : Producer(100);
  Cons : Consummer;
connect
  Prod.Send(n) => Cons.Receive(n);
  Cons.Ack(n) => Prod.Reply(n);
end
```

A partir des types définis ci-dessus, cette architecture ProdCon crée deux instances particulières de composants: Prod et Cons. Les relations entre ces deux composants sont ensuite décrites par l'intermédiaire des connecteurs introduits via la construction connect.

Ainsi, l'action sortante Send du producteur Prod est reliée à l'action entrante Receive du consommateur Cons. De façon réciproque, l'action Ack de Cons est reliée à l'action Reply de Prod. Au moment de l'opérationnalisation d'une telle architecture, il ne reste donc plus aucune ambiguïté concernant l'acheminement des communications entre les deux composants.

L'approche Rapide introduit donc deux différences fondamentales si on la compare aux modèles objets plus courants :

- Les références explicites vers l'extérieur ne sont pas permises. Les objets, au contraire, se basent sur ces références explicites pour communiquer (cf. figure 1.7 page 22).
- Une dichotomie fondamentale est introduite entre les entités qui réalisent les traitements, les composants, et les données qu'ils se communiquent, les événements. L'approche objet privilégie la vision unique ou l'objet est (pratiquement) tout. Cependant, la notion de message est quelque peu ambiguë de ce point de vue puisqu'il s'agit rarement d'un objet à part entière. Les approches de type composant/événement comblent selon nous ce manque de précision.

Ces deux divergences fondamentales et, interdépendantes, qui résultent de la problématique de composition opérationnelle donnent selon notre opinion tout leur sens à l'introduction d'une nouvelle entité nommée composant logiciel; et qui ne serait pas qu'une légère variation sur le thème de la programmation orientée objet.

Il reste cependant quelques problèmes à considérer avant d'entreprendre la mise en œuvre d'un tel modèle pour le domaine d'application qui nous concerne; et qui n'est pas celui de Rapide qui n'aborde pas ou prou les problématiques d'adaptation.

Le premier problème concerne la notion de connecteur qui reste malgré tout assez ambiguë puisque l'on peut la voir selon deux points de vue différents. En effet, on peut soit considérer les connecteurs comme indépendants des composants ou au contraire intégrés à ces derniers. En termes plus pragmatiques, selon les approches considérées, on décrira les connecteurs *plutôt* au niveau des composants ou alors *plutôt* de façon autonome. Mais même observés de façon autonome, les connecteurs ne sont pas indépendants des composants qu'ils se proposent de relier : on ne peut pas totalement découpler les deux points de vue.

Dans le même ordre d'idée, il n'est pas évident de faire le lien entre le concept de connecteur et sa réalisation opérationnelle. Dans le cadre de Rapide, on est clairement du côté de l'opérationnalisation mais dans d'autres approches, comme Unicon [Zel96], la tendance s'inverse et devient plus conceptuelle.

Un troisième problème soulevé par notre étude des principes des architectures logicielles concerne le typage des composants qui semble relativement «sous-estimé», surtout si l'on considère le problème sous l'angle opérationnel. Dans le cadre de Rapide, par exemple, les données qui transitent entre les composants sont assez simples mais dans l'optique de la distribution, les événements que l'on sera amenés à considérer seront de nature beaucoup plus complexes.

« Les contraintes se guérissent par les contraires. »
Maxime lattine

3
Synthèse

Il est temps de rassembler les différents arguments que nous avons développés pour composer notre problématique et détailler les approches qui nous semblent posséder le plus fort potentiel pour pouvoir y répondre de façon satisfaisante.

Tout d'abord, à partir de notre volonté de composition et d'adaptation de systèmes distribués en cours d'opérationnalisation, nous nous sommes orientés vers une approche de type interstitielle.

Nous avons ensuite désigné un certain nombre de limites associées aux approches «classiques» en matière d'interstitiels. Ces limites étaient à la fois d'ordre comportemental - avec notamment le caractère synchrone des interactions mises en œuvre - mais également d'ordre structurel - de par le caractère implicite de la relation de couplage dans le monde objet.

Bien sûr, de nombreuses réponses systématiques existent à ces problèmes, même pour ce qui concerne l'adaptation dynamique [ED99], mais nous pensons que l'approche applicative qui caractérise ces solutions est assez restrictive - surtout si l'on veut donner un caractère formel à ces solutions.

ous avons donc décidé d'aller dans le sens des approches qui vont jusqu'à remettre en question le modèle.

3.1 Retour sur l'exemple

Sous l'angle du comportement, les travaux sur les entités logicielles asynchrones nous semblent les plus appropriés pour aborder notre problématique puisqu'en matière opérationnelle, les solutions existent déjà : les acteurs et objets actifs se composent et s'adaptent dynamiquement.

Si nous reprenons notre exemple du système Comedia (cf. page 16), et tout particulièrement le problème de l'adaptation structurelle du système de diffusion en cas d'affluence de clients, nous pouvons ébaucher des solutions simples en nous basant sur des langages d'acteurs.

En restant à un niveau relativement abstrait, nous pouvons proposer la définition de ce que pourrait représenter le gestionnaire de serveurs dans un monde d'acteurs :

```
Actor ServerManager
  array servers:=[];
  int new_clients:=10;
  int max_new=10;
method connect(Actor client, Media media)
  if(new_clients=max_new) {
    Actor nserver = new Server;
    servers[servers.size]:=nserver;
    send(broadcast,client,media) to nserver;
    new_clients:=1;
  } else {
    send(broadcast,client,media) to servers[servers.size-1];
    new_clients++;
  }
end connect

// etc ...
```

L'acteur ServerManager sert d'allocateur de serveur de diffusion en fonction des requêtes de client. La méthode connect, déclenchée par la réception d'un message depuis un client se charge d'attribuer un serveur à ce client. L'heuristique employée ici pour choisir le serveur est on ne peut plus simple. Si le nombre de nouveaux clients new_clients dépasse un certain seuil fixé à l'avance, ici dans max_new, alors un nouveau serveur est alloué pour ce nouveau client et les prochains qui en effectuerons la demande. Dans le cas contraire, le dernier serveur alloué est utilisé pour la diffusion.

Même si cette définition est abstraite, elle reflète dans une certaine mesure la simplicité d'écriture de tels protocoles adaptatifs dans les langages d'acteur qui ne s'encombrent pas - en termes linguistiques - de considérations relatives à la concurrence ou au parallélisme, prises en compte implicitement dans le modèle.

Au niveau de l'adaptation comportementale, par exemple sur l'adaptation de la politique de diffusion du serveur, on pourrait de la même façon faire appel aux modes d'interaction très développés qui sont proposés tant dans ABCM/1 que dans Actalk. En particulier, avec la variante réflexive ABCL/R, nous pourrions télécharger dynamiquement de nouveaux scripts de synchronisation.

Mais s'il est possible dans ces approches asynchrones de décrire assez simplement des mécanismes complexes et offrant d'intéressantes propriétés dynamiques, le problème de l'indépendance structurelle reste entier. Rien n'empêche acteurs et autres objets actifs de se référencer mutuellement, comme dans le cas des acteurs. Compositions et adaptations dynamiques sont possibles mais seulement au niveau de granularité beaucoup trop fin de la description fonctionnelle des entités mises en œuvre, exactement comme dans le monde objet.

Si l'on prend donc dans nos discussions précédentes le problème dans le sens de la réification structurelle du couplage, les gammes de solutions que nous avons retenues sont bien différentes et s'articulent généralement autour de la notion de composant.

En revanche, et même dans le cas pourtant asynchrone de Rapide, la flexibilité en terme de description de comportement est perdue du fait de cette augmentation dans le niveau de granularité associé à chaque entité du système.

En conséquence de ce relatif paradoxe, il nous faut désormais discuter de ce qui pourrait aider à la fusion des deux points de vue structurel et comportemental. Il nous manque - en quelque sorte - du matériel de construction ...

3.2 Le ciment réflexif

Des trois solutions proposées dans [BGL98] pour circonvenir aux limites des objets une fois plongés dans un cadre concurrent et réparti, nous nous sommes particulièrement intéressés au deux premières : applicatives (disons «à la CORBA») et intégrées (exemple: objets actifs). La troisième approche proposée est présentée comme un moyen de réconcilier les deux premières qui semblent quelque peu antinomiques ; c'est l'approche *réflexive*.

3.2.1 Le concept de réflexion

Donner une définition consensuelle de la réflexion relève de la gageure. Il existe cependant un certain nombre de travaux fondateurs qui permettent de caractériser assez précisément ce terme. Pattie Maes explique dans [Mae87] que «la réflexion calculatoire est l'activité produite par un système calculatoire quand il effectue des calculs sur (et, dans ce sens, probablement affectant) son propre calcul». En d'autres termes, «un programme réflexif est un programme qui raisonne à propos de lui-même» [FR89].

Dans les premiers dialectes réflexifs de *Lisp*, cette idée se traduit en l'implémentation d'une pile d'interpréteurs. Le méta-programme de niveau n est interprété par un interpréteur programmable de niveau n+1, etc. Pour éviter toute régression infinie, l'interpréteur canonique (de niveau 0) est utilisé pour créer une pile méta-circulaire et donc finie.

Réification et réflexion

Il est possible de distinguer deux «étapes» dans le processus de réflexion (*cf. figure 3.1*) :

1. La réification[1] : un concept abstrait d'un programme ou objet est «transformé» en un programme ou objet manipulable.
2. La réflexion : une fois «manipulé», l'objet réifié est «réinjecté» dans le programme.

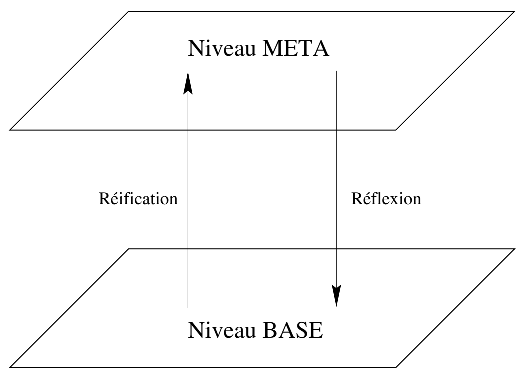


FIG. 3.1 – *Le mécanisme de réflexion*

Smalltalk permet par exemple d'intercepter les messages mal adressés par la méthode doesNotUnderstand: pour ajouter de façon transparente un mécanisme de délégation. L'aspect réifié est ici la transmission de message, la phase de réflexion concerne la modification du routage des messages.

Objets et réflexion

Pattie Maes applique un tel procédé au paradigme objet dans *3-KRS*. L'idée est de cette fois-ci associer un méta-objet à chaque objet manipulé par le système. Il est alors possible de considérer le méta-objet comme un objet, et donc de le manipuler. Ce type de réflexion, dite calculatoire relie un niveau base (ou dénoté) et un niveau méta par un lien de causalité quelconque, indépendant du modèle objet (cf. figure 3.2 page suivante).

Le langage *Smalltalk-80* offre un autre type de réflexion : la réflexion structurelle, associée au protocole de méta-classes [FR89]. Il s'agit en fait d'un cas particulier de la réflexion calculatoire qui identifie lien de causalité base-méta et lien d'instanciation (ou de typage), comme présenté en figure 3.3 page suivante. Chaque objet est instance d'une classe. Chaque classe est également associée à une méta-classe (elle même associée à une méta-méta classe, etc.). Les hiérarchies des niveaux base et méta sont quasiment «parallèles» (un petit tour de passe-passe, la méta-circularité, permet de résoudre le problème des tours méta infinies).

1. Réifier : rendre chose. Faire d'un concept abstrait un objet concret.

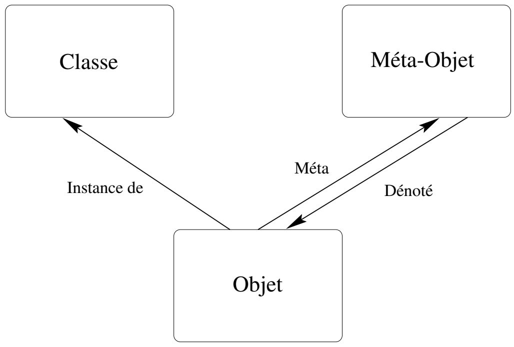


Fig. 3.2 – *Réflexion calculatoire dans 3-KRS*

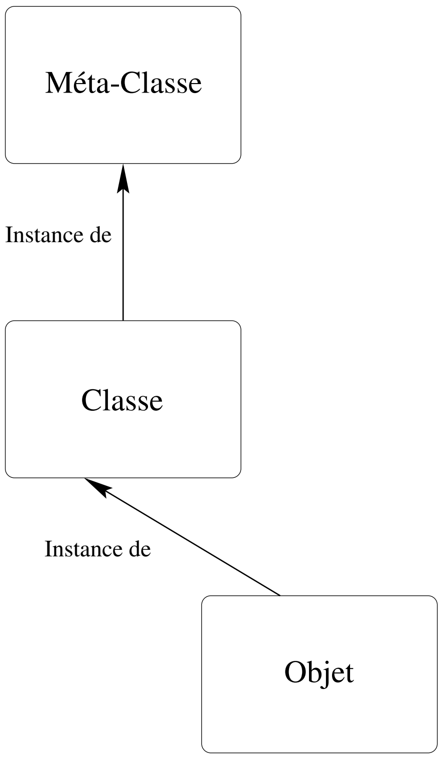


Fig. 3.3 – *Réflexion structurelle dans Smalltalk*

3.2.2 Quelques approches réflexives

Nous considérons dans cette section trois approches qui illustrent notre intuition concernant l'intérêt de la réflexion comme ciment structurel. Ces trois approches correspondent aux modèles asynchrones étudiés au chapitre précédent et nous offrent ainsi trois visions différentes rendant tangible au niveau structurel les propriétés comportementales - et notamment le fonctionnement asynchrone - des entités qu'elles mettent en œuvre.

Exemple 1 : DIL et la réflexion

Pour réaliser la communication entre niveau acteur (ou composant) et niveau protocole, nous avons vu (cf. chapitre 2 page 33) que le mécanisme mis en jeu dans le langage *DIL* était basé sur des événements. Ce fonctionnement suppose la réification de certaines fonctionnalités des acteurs comme la gestion des messages en attente de traitement ou des messages émis. Cet aspect réflexif est donc omniprésent dans DIL mais de façon limitée et proprement justifiée : une sémantique appropriée est présentée.

L'approche choisie par ce framework pour introduire la réflexion est de type calculatoire. Chaque rôle de protocole, une fois associé à un acteur particulier, entraîne la création d'un méta-objet *Communicator* (*cf. figure 3.4*)

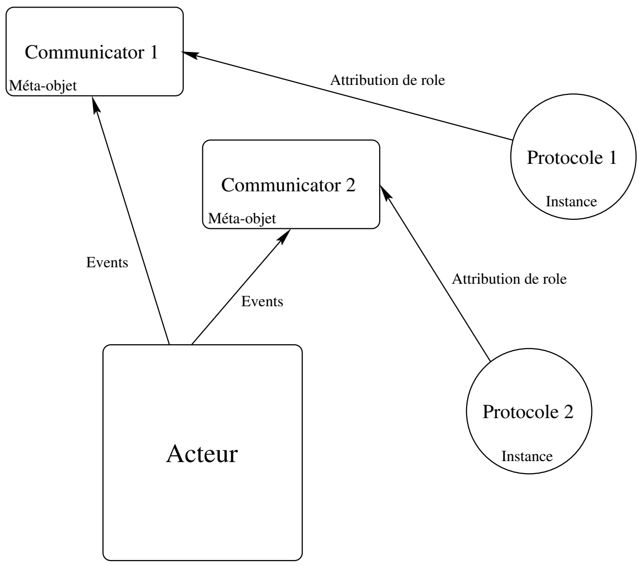


FIG. 3.4 – *Réflexion dans* DIL

Ce méta-objet est ensuite attaché à l'acteur de niveau base et peut dès lors intercepter les événements définis dans le protocole. La composition des méta-objets permet la composition des protocoles.

Une autre possibilité des Méta-objets rôles est de pouvoir introspecter et modifier l'état interne des acteurs. Deux opérations spécifiques : *reifyState()* et *reflectState()* doivent être implémentées dans ce but.

La réflexion au sein de *DIL*, et plus généralement dans l'approche acteur, est considérée comme un outil puissant mais qu'il faut limiter au maximum pour conserver une sémantique claire et formelle.

Exemple 2 : ABLC/R

Tels les acteurs, les objets *ABCM/1* (cf. chapitre 2 page 35) proposent une sémantique de contrôle évoluée et adaptée au problème de la concurrence. La réflexion n'y est donc pas vue comme un outil pour «résoudre» le problème du contrôle implicite des objets traditionnels mais plutôt comme une extension du modèle.

Les motivations pour l'introduction de mécanismes réflexifs dans des langages et systèmes basés sur le modèle de calcul *ABCM/1* étaient initialement les suivantes :

- autoriser l'acquisition dynamique de méthodes,
- tracer le comportement des objets,

– introduire le mécanisme «time warp» en univers concurrent.

Cette approche encore une fois assez pragmatique de la réflexion est principalement intéressante parce qu'elle introduit un ensemble de concepts et outils génériques qui seront repris dans de nombreux projets.

L'approche réflexive dans *ABCL/R* [Ish91] est inspirée des travaux de *P. Maes* sur la réflexion calculatoire. Dans *3-KRS*, chaque objet peut être associé à un méta-objet qui dispose d'une représentation de l'objet dénoté et de moyens pour intervenir sur son comportement. Dans *ABCL/R*, chaque objet x peut être adjoint d'un méta-objet $\uparrow x$. Contrairement à *3-KRS*, un méta-objet $\uparrow x$ est également assimilé à un objet pouvant être à son tour adjoint d'un méta-objet $\uparrow\uparrow x$, et ainsi de suite. Il existe donc conceptuellement une tour infinie de méta-objets pour un objet x donné. Cette régression infinie est réalisée par création paresseuse de méta-objets, en fonction des besoins.

Un méta-objet $\uparrow x$ contient toujours, sous forme de donnée, l'état de son objet dénoté x. Ce lien de causalité est unique : deux méta-objets ne peuvent dénoter le même objet.

Cette généricité conceptuelle est réalisée par réification d'aspects structurels et comportementaux. Au niveau structurel, on trouve dans un méta-objet les informations suivantes :

- une variable *scriptset* qui contient, sous forme de données, les scripts de l'objet dénoté,
- une variable *state* qui implémente l'état interne de l'objet dénoté,
- un objet évaluateur, nommé *evaluator*, lequel implémente les fonctionnalités canoniques des objets,
- un objet *queue* qui représente la boîte à lettre du message et les traitements associés,
- une variable *mode* qui représente l'état comportemental de l'objet (*dormant*,*active*, *waiting*).

Les fonctionnalités comportementales réifiées grâce à cette structure sont :

- L'arrivée d'un message et sa mise en file d'attente,
- La réception d'un message par extraction depuis la file d'attente,
- acceptation d'un message (synchronisation),
- exécution d'un script
- terminaison d'exécution d'un script

L'architecture réflexive d'ABLC/R est représentée sur la figure 3.5. Nous pouvons clairement voir ici l'intérêt de la réification pour capturer, en terme de structure (i.e. la «forme» des méta-objets $\uparrow \ldots \uparrow x$), des propriétés comportementales associées aux entités mises en œuvre par le modèle. Ceci traduit de façon assez évidente le sens de notre idée de *ciment réflexif*.

Le langage *Rbcl* [IMY92], autre évolution réflexive, propose une vision plus radicale de la réflexion. Contrairement à *ABCL/R* qui dispose d'un noyau canonique, toutes les fonctionnalités de *Rbcl* sont modifiables par l'utilisateur, y compris les aspects réflexifs.

Exemple 3 : CodA

La plupart des frameworks étudiés jusqu'à présent ne considèrent la réflexion que comme un outil permettant la méta-programmation. Cette démarche conduit généralement à raffiner structurellement le modèle de programmation afin de permettre la réalisation de méta-fonctionnalités bien choisies et généralement fonction du domaine étudié. *CodA* [McA95a, McA95b] oppose son approche de type descendante (par raffinement structurel) à une approche fonctionnelle et ascendante de la réification. Le principe de *CodA* est d'identifier des méta-fonctionnalités génériques et de les réifier sous forme de méta-objets. L'objectif du framework concerne la réalisation de divers modèles objets composables. Les prototypes d'implémentation, quoique potentiellement neutres, ont intégralement été réalisés en *Smalltalk*.

L'approche ascendante est par nature plus ouverte que l'approche descendante car les méta-fonctionnalités réifiables ne sont pas figées. Cependant, *CodA* définit un certain nombre de fonctionnalités candidates à la réification au sein d'un unique objet. Ces fonctionnalités sont :

- La réception des messages (*Accept*)
- Leur stockage dans une file d'attente (*Queue*)
- Leur extraction depuis la file (*Receive*)

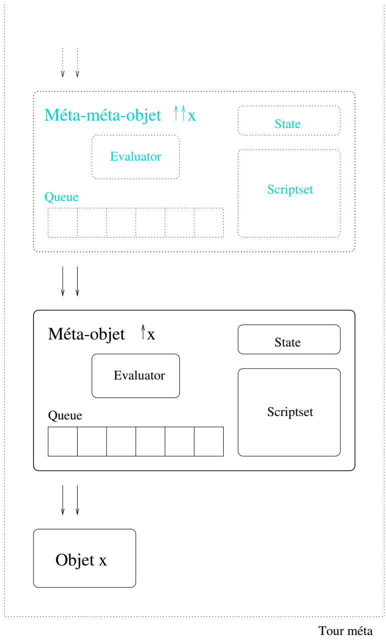


FIG. 3.5 – *L'architecture réflexive ABCL/R*

Méta-objets	Rôle	Par défaut
Accept	Réception des messages	Message accepté et passé à *Queue*
Queue	Mise en file d'attente	court-circuit vers *Receive*
Receive	Extraction depuis la file d'attente	Passage à *Protocol*
Protocol	Sélection de la méthode à exécuter	Sélection de la méthode correspondant au message
Execution	Exécution de la méthode	Exécution simple
State	Accès à l'état interne	Ne fait rien
Send	Envoi de message	Envoi synchrone, asynchrone ou pseudo-asynchrone (future)

TAB. 3.1 – *Fonctionnement par défaut des méta-objets* CodA

- La sélection d'une méthode à exécuter (*Protocol*)
- L'exécution proprement dite (*Execution*)
- Les accès à l'état interne de l'objet (*State*)
- Les envois de messages (*Send*)

Pour réifier ces méta-fonctionnalités, *CodA* définit un ensemble de méta-objets. Même s'il est courant d'associer un méta-objet donné à une méta-fonctionnalité, ce n'est pas une règle immuable. Cependant, dans le cas de base, *CodA* réifie réellement sept méta-objets dont le fonctionnement par défaut est donné en table 3.1. Ce comportement par défaut correspond exactement au comportement des objets passifs de *Smalltalk*. *CodA* se base donc sur une extension de la sémantique du modèle objet et non sur sa remise en question. La sémantique d'émission de message est cependant étendue puisque les trois modes du modèle de calcul *ABCM/1* sont disponibles.

Nous pouvons voir sur la figure 3.6 page suivante que les méta-fonctionnalités réifiées (excepté le cas de *State*) dans *CodA* correspondent au cheminement des messages dans les objets. Il s'agit d'une caractéristique essentielle qui rend «naturelle» la décomposition fonctionnelle choisie. Le choix des noms *Accept* et *Receive* est quelque peu troublant mais se justifie de la façon suivante :

- *Accept* vérifie qu'un message est acceptable par l'objet avant de le transmettre à *Queue*.
- *Receive* rend compte de l'instant correspondant au traitement effectif du message et non de celui de la réception «organique».

Ces termes sont donc choisis par le rôle fonctionnel que les méta-objets correspondants réifient et non par leur réalité structurelle (*Accept* reçoit structurellement les messages, *Receive* les récupère fonctionnellement).

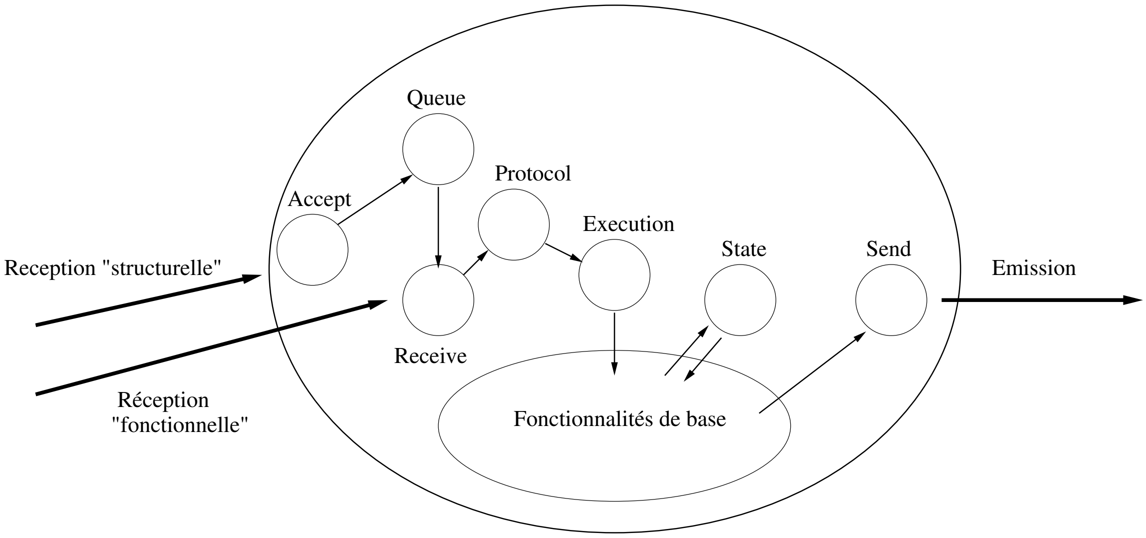


FIG. 3.6 – *CodA : sept méta-objets canoniques*

Toujours dans une optique fonctionnelle, les méta-objets *CodA* ne sont pas eux-mêmes assimilés à des objets. Ils n'ont donc pas de méta-objets associés (le niveau méta-méta n'existe pas). Ce choix qui semble limitatif permet d'éviter les problèmes de tours méta infinies [2]. L'auteur nous fait de plus remarquer que la composition structurelle utilisée dans de nombreux frameworks correspond sémantiquement à une composition fonctionnelle [3]

Spécialisations et compositions de méta-objets Le choix des méta-objets canoniques de *CodA* semble intéressant mais la validation de cette approche ne peut se faire que par des instanciations multiples. Une spécialisation de *CodA* est un modèle objet dont la sémantique diffère de celle de *Smalltalk* (et donc du modèle par défaut dans *CodA*). Trois modèles différents ont été construits sur cette approche :

– Les objets concurrents : ce sont en quelque sorte des acteurs ou objets actifs disposant d'une activité interne les rendant autonomes. Le mode de communication employé est naturellement asynchrone.
– Les objets distribués : il s'agit dans ce cas d'objets distribués dont les composantes mêmes sont distribuables. Par exemple, les méthodes sont situées sur une machine et l'état interne est réparti sur deux machines séparées.
– Les objets «à ports» : ils permettent de modéliser facilement des architectures de type *dataflow*.

Ces trois spécialisations de *CodA* ont été «validées» sur des exemples complets, notamment le système distribué *Tj* et le framework de *dataflow Vibes*.

La grande nouveauté dans le framework réflexif *CodA* est que tout en conservant la généricité du modèle objet, cette boîte-à-outils offre une généricité au niveau de la spécialisation du contrôle de ces objets.

Il est d'ailleurs intéressant de noter que les idées présentées ici ne se limitent pas aux objets. Ainsi, Frédéric Migeon, dans sa thèse très intéressante [Mig99], propose en quelque sorte un «portage» de CodA dans le paradigme acteur.

3.2.3 Structuration du comportement

Dans tous les cas que nous avons étudiés, la réflexion représente un outil privilégié pour structurer le comportement et ainsi le rendre tangible au niveau de la structure des entités qui dénotent ce comportement.

Certes, les approches que nous décrivons ci-dessus se focalisent sur des entités dont la structure est plutôt opérationnelle (acteurs/objets actifs) ; mais elles ne sont pas incompatibles avec l'introduction du concept de composant «à la Rapide».

2. Ainsi que les tours de passe-passe permettant d'éviter plus ou moins proprement, au sens de la sémantique, les régressions infinies.

3. Les niveaux méta ajoutés correspondent à des méta-fonctionnalités supplémentaires et non à des méta-méta-méta(etc...)-fonctionnalités).

Nous entrevoyons ici la possibilité d'ouvrir le comportement de ces composants asynchrones pour l'adapter à leur structure tout à fait particulière qui ne dépend pas que des composants eux-mêmes mais également des connexions qui les relient les uns aux autres.

3.3 Le ciment formel

Le principal reproche formulé à l'encontre des approches réflexives concerne leur nature conceptuellement «difficile» ; qui entraîne selon de nombreux observateurs d'importants obstacles concernant leur formalisation. C'est une des raisons pour lesquelles nous n'étudierons pas ici les *interstitiels réflexifs*. Malgré l'intérêt indéniable des approches de type *OpenCorba* [Led98], leur étude précise sort du cadre de notre propos qui ne concerne pas l'extension des interstitiels disons «classiques» par des considérations réflexives.

Ainsi, l'un des objectifs prioritaires que nous nous fixons est la formalisation complète du versant structurel de notre approche. Il serait dès lors difficile de justifier l'unité de notre contribution si nous ne pouvions pas quelque peu «mener» cette formalisation jusqu'à un niveau - au moins un tant soit peu - comportemental.

3.3.1 Réflexion et formalisation

De ce point de vue, on pourrait être amenés à penser que la piste «réflexive» représente un parcours semé d'obstacles un peu difficiles à franchir, surtout si l'on considère que cela ne couvrirait alors qu'une partie des besoins exprimés dans notre problématique[4]. Cependant, la simplicité particulière de l'approche proposée par CodA nous ouvre une nouvelle perspective.

Il est en effet tout à fait concevable de transposer les propositions de CodA dans un modèle formel, à condition bien sûr de définir proprement ce que représentent les méta-objets de CodA dans cette transposition. Le résultat n'est pas envisagé sous l'angle réflexif, de sorte que la réflexion pour nous est plutôt source d'inspiration - véritable culture - que préceptes à appliquer à la lettre.

3.3.2 Formalismes utilisés

Il existe une incroyable profusion de formalismes mathématiques permettant de modéliser des concepts et outils informatiques.

Mais replaçons dans leur contexte les différents objectifs que nous fixons concernant cette formalisation. Tout d'abord, nous voulons poser un cadre structurel précis en donnant des définitions correctement formalisées des concepts de composants, événements et connexions.

Nous devons aussi nous placer dans l'optique d'une évolution dynamique de cette structure, en modélisant proprement les opérations :

- d'ajout et de retrait de composants,
- de mises en place d'interconnexions entre ces composants.

Le temps occupe donc une place importante dans cette perspective ; nous aimerions ainsi rendre tangible l'évolution temporelle de notre structure. Ceci nous conduit à la piste des systèmes à états/transitions qui peuvent être modélisés dans des formalismes variés.

D'un autre côté, nous voulons décrire le comportement fin des composants en modélisant, par exemple, les opérations liées à la communication des événements :

- Emission d'un événement par un composant source,
- Réception de cet événement par le (ou les) destinataire(s).

Il est également important de rendre tangible la nature asynchrone des de ces communications, par exemple en décrivant formellement l'opération de mise en file d'attente des événements et du traitement effectif de ces derniers.

4. Nous nous plaçons ainsi dans une perspective d'utilisation de la réflexion comme outil important mais outil tout de même. Pour résoudre notre problème ; nous ne voulons pas construire nous-même cet outil.

De toutes les pistes que nous avons explorées, et après l'avoir dans un premier temps écarté du fait de son manque de support utilitaire, la notation Z [Spi01] nous est finalement apparue comme le formalisme le plus à même de répondre à nos besoins.

Tout d'abord, en plus de sa documentation prolixe, le formalisme Z permet la spécification algébrique - dans des conditions selon-nous remarquables - des systèmes basés sur la notion d'états et de transitions entre ces états. De plus, la méthode de raffinement sous-jacente de la notation nous permet - et c'est là nous semble-t-il notre contribution majeure - de réaliser de façon propre et précise la réintégration des deux points de vue comportemental et structurel.

Bien sûr, ce choix n'est pas exempt d'inconvénients, que l'on peut globalement classer en trois catégories, dont les deux premières intéresseront principalement le lecteur.

Tout d'abord, la notation Z n'est pas simple et, pour des raisons diverses mais toutes largement justifiées, s'écarte quelque peu des notations mathématiques usuelles. En revanche, le modèle sous-jacent - celui de la théorie des ensembles dans l'axiomatique de Zermelo-Fænkel - est elle on ne peut plus simple, de sorte que les spécifications Z ne sont finalement pas difficiles à comprendre, à condition de disposer d'une documentation adéquate. Nous avons conçu notre spécification en partant du principe d'une connaissance minimale de la notation Z. Toutes les explications nécessaires se trouvent donc directement dans le texte, au fur et à mesure de l'introduction des concepts nécessaires. Cependant un glossaire de la notation référençant l'ensemble des symboles et constructions (nombreux) que nous utilisons dans cette thèse est proposé (annexe B page 239).

Le deuxième inconvénient de la notation Z est beaucoup plus sensible car lié aux fondements même de la notation. En effet, cette dernière a été conçue dans l'optique de la spécification de processus informatiques séquentiels. Or, notre modèle est bien loin d'être séquentiel puisqu'au sein même d'un composant, la concurrence est de rigueur. Il n'existe pas, d'après nos connaissances, de solution simple à ce problème. Nous proposons donc - et à titre expérimental - notre propre «rustine» formelle, par l'emploi de ce que nous avons nommé des *réseaux d'activation concurrente*. Ces automates, modélisés sous la forme de réseaux de Petri, permettent de rendre compte de façon assez éloquente du parallélisme au sein même du modèle. Encore une fois, les explications nécessaires à la compréhension de ces réseaux d'activation sont proposées au sein même du texte.

Pour valider - au moins partiellement - le modèle, nous avons effectué un certain nombre de preuves qui sont regroupées, du fait de leur relative «lourdeur» dans une annexe séparée (annexe A page 225). Ces preuves soulignent le troisième et dernier inconvénient du choix de la notation Z : son caractère indécidable et la pauvreté de l'outillage qui en résulte. Ainsi, il est pratiquement indispensable d'effectuer ces preuves à la main ; processus long et fastidieux mais bien loin de l'inutilité puisque de nombreuses erreurs de spécifications ont ainsi été mises au jour. Il en subsiste probablement quelques unes mais - tout du moins pour ce qui concerne la partie structurelle - nous pensons qu'elles sont désormais assez bien camouflées !

Une autre validation importante - et même indispensable - de cette formalisation se trouve décrite au chapitre 10, sous la forme d'un prototype opérationnel pour notre interstitiel basé sur la fusion des composants et des modèles comportementaux asynchrones.

Deuxième partie

Modèle structurel

«Les mathématiques peuvent être définies comme une science dans laquelle on ne sait jamais de quoi on parle, ni si ce qu'on dit est vrai»

Bertrand Russel

«Les propositions mathématiques sont reçues comme vraies parce que personne n'a intérêt qu'elles soient fausses»

Montesquieu

4 Système de type

Comme nous allons le voir tout au long de ce chapitre, la notion de *type* se trouve au cœur de notre débat. Si les travaux récents en matière de typage prennent souvent leur source dans la théorie du lambda-calcul typé, nous allons adopter - tout comme Z - un point de vue ensembliste. La principale raison de ce choix est que nous n'allons pas typer des *expressions* mais des *structures*. La *théorie des types* que nous définissons dans cette section possède comme caractéristique fondamentale de supporter une *relation de sous-typage* autorisant la *subsomption*. Cette propriété indique qu'il est possible d'utiliser un *sous-type* dans tout contexte où un *super-type* est attendu. Selon la classification proposée dans [CW85], il s'agit d'un modèle de typage *polymorphe par inclusion*.

4.1 Types et valeurs

Un type est défini, au niveau structurel, comme valeur d'un type générique libre. En Z, l'introduction d'un tel ensemble non contraint se fait de la façon suivante :

$[TYPE]$

Il existe deux approches permettant de donner un sens à la notion de type. La première, la *vision interprétative*, se fixe pour objectif de définir sans ambiguïté le sens du concept. La *vision structurelle*, en revanche, ne s'intéresse qu'aux propriétés «externes» des types, leur interprétation fine étant laissée délibérément ouverte. Dans le modèle Scope, cette vision structurelle est bien sûr privilégiée car l'approche interprétative représente une tâche complexe dépassant le cadre de notre thèse.

Cependant, puisque nous reposons sur le formalisme Z et son interprétation ensembliste des types, nous pouvons considérer dans notre approche qu'un type, élément de $TYPE$, dénote un ensemble de valeurs possibles.

On pourra par exemple définir le type BIT comme l'ensemble de valeurs $\{0,1\}$. D'autres types peuvent dénoter des ensembles infinis; par exemple le type $INTEGER$ qui représente les nombres entiers. L'ensemble des nombres réels $REAL$, pour sa part, est à la fois infini et non-dénombrable. Ce type d'interprétation, forcément *incomplète*[1], pourra cependant se révéler *suffisante* si elle permet de construire les propriétés attendues de notre système de type. Il s'agit donc avant tout d'un problème de point de vue et notre approche nous guide vers un point de vue purement structurel.

1. Ici, on ne définit pas les propriétés discriminantes des entiers et réels, mais seulement un énoncé d'appartenance à un ensemble.

4.2 Relations de typage

Plus qu'à la notion de type elle-même, notre objectif dans cette section est de donner une sémantique claire à la structure du système de type. Ainsi, les éléments du type *TYPE* (c'est-à-dire les types eux-mêmes) peuvent être reliés par un ensemble de relations permettant de les comparer les uns avec les autres.

Tout d'abord, Z fournit les relations d'égalité $=$ et d'inégalité $\neq$ permettant d'identifier ou de distinguer les types deux à deux. En sus de ces relations intrinsèques, nous introduisons deux relations très importante, notées $\succeq$ et $\preceq$, qui définissent un ordre sur les types.

La première de ces relations, $\succeq$, signifie : *est un super-type de* ; on parlera de *sous-type* pour la relation $\preceq$. Dans la suite de notre exposé, nous parlerons invariablement de *relations de sous-typage*.

Nous pouvons poser qu'intuitivement, un *sous-type* (resp. *super-type*) dénote un sous-ensemble (resp. sur-ensemble) de valeurs d'un type donné [2].

Reprenons par exemple nos types *REAL* et *INTEGER* (tous deux de type *TYPE*) dénotant respectivement l'ensemble des nombres réels et l'ensemble des nombres entiers. On pourra alors considérer, toujours dans l'intuition ensembliste, que le type *INTEGER* est sous-type du type *REAL*, où, de façon équivalente, que *REAL* est super-type de *INTEGER*. On notera alors : $REAL \succeq INTEGER$ et $INTEGER \preceq REAL$.

En considérant la propriété de subsomption et de façon toujours intuitivement cohérente, il est possible d'utiliser un élément de type *INTEGER* dans tout contexte ou un élément de type *REAL* est attendu. Bien sûr, la réciproque n'est pas vraie : un réel en général ne peut pas être utilisé dans un contexte où un entier est attendu.

Les propriétés structurelles de la relation super-type $\succeq$ sont les suivantes :

- **Réflexivité** : pour tout x de *TYPE*, alors $x \succeq x$
- **Transitivité** : pour tout x, y et z de *TYPE*, si $x \succeq y$ et $y \succeq z$, alors $x \succeq z$
- **Antisymétrie** : pour tout x et y de *TYPE*, si $x \succeq y$ et $y \succeq x$, alors $x = y$

Ces trois propriétés fournissent à la relation super-type $\succeq$ une structure d'*ensemble partiellement ordonné* (ou *Poset*). Ceci indique que pour un couple de types donné, les éléments de ce couple sont soit reliés par la relation super-type soit non comparables. En corollaire de cette propriété, tout type pourra avoir plus d'un sous-type et plus d'un super-type. Au niveau interprétatif, il faut noter ici que notre volonté est d'imposer une structure la plus flexible possible car, comme nous le verrons plus tard, le modèle Scope n'est pas à usage unique mais peut-être employé dans différents contextes.

Des variantes *strictes* des relations de sous-typage précédentes (dites *larges*), $\succ$ et $\prec$, sont proposées. On indiquera de plus que les types x et y sont reliés d'une façon ou d'une autre (super-types, sous-types larges ou stricts) par la notation : $x \sim y$. De la même façon, deux types non-comparables seront notés : $x \nsim y$.

Le schéma Z suivant résume l'ensemble des relations de typage ainsi que leurs propriétés :

$$
\begin{array}{|l}
\_ \succeq \_ : TYPE \leftrightarrow TYPE \\
\_ \succ \_ : TYPE \leftrightarrow TYPE \\
\\
\_ \preceq \_ : TYPE \leftrightarrow TYPE \\
\_ \prec \_ : TYPE \leftrightarrow TYPE \\
\\
\_ \sim \_ : TYPE \leftrightarrow TYPE \\
\_ \nsim \_ : TYPE \leftrightarrow TYPE \\
\hline
\forall\, x : TYPE \bullet x \succeq x \\
\forall\, x,y,z : TYPE \bullet (x \succeq y) \wedge (y \succeq z) \Rightarrow x \succeq z \\
\forall\, x,y : TYPE \bullet (x \succeq y) \wedge (y \succeq x) \Rightarrow x = y \\
\forall\, x,y : TYPE \bullet x \succ y \Leftrightarrow (x \succeq y) \wedge (x \neq y) \\
\\
\forall\, x,y : TYPE \bullet x \preceq y \Leftrightarrow y \succeq x \\
\forall\, x,y : TYPE \bullet x \prec y \Leftrightarrow (x \preceq y) \wedge (x \neq y) \\
\\
\forall\, x,y : TYPE \bullet x \sim y \Leftrightarrow (x \succeq y) \vee (x \preceq y) \\
\forall\, x,y : TYPE \bullet x \nsim y \Leftrightarrow \neg\, (x \sim y)
\end{array}
$$

2. C'est cette identification de l'inclusion ensembliste et du sous-typage que l'on nomme *polymorphisme d'inclusion*

La partie déclaration du schéma (au dessus du trait horizontal) introduit les différentes relations de typage ainsi que leur notation. En dessous se trouve la partie prédicat qui donne les propriétés de ces relations selon les règles établies précédemment. Désormais, tout schéma Z pourra utiliser ces relations, à condition bien sûr d'en respecter les propriétés.

4.2.1 Super-types et sous-types directs

Dans cette section, nous allons préciser deux relations fondamentales concernant les notions de super et sous-types directs. La nature partielle de la relation d'ordre définie par $\succeq$ indique que plusieurs super-types et sous-types directs sont possibles pour un type donné. Commençons par définir les notions de sous-type et super-type *direct* :

$$\_ \succ_{dir} \_ : TYPE \leftrightarrow TYPE$$
$$\_ \prec_{dir} \_ : TYPE \leftrightarrow TYPE$$

$$\forall x,y : TYPE \bullet x \succ_{dir} y \Leftrightarrow (x \succ y) \wedge [\forall z : TYPE \bullet (z \succ y) \Rightarrow (z \not\succ x) \vee (z \succeq x)]$$
$$\forall x,y : TYPE \bullet x \prec_{dir} y \Leftrightarrow (x \prec y) \wedge [\forall z : TYPE \bullet (z \prec y) \Rightarrow (z \not\prec x) \vee (z \preceq x)]$$

Si x est un super-type direct de y, on notera alors : $x \succ_{dir} y$. La propriété vérifiée par cette relation est que, dans un premier temps, x doit être un super-type strict de y, ou $x \succ y$. De plus, pour tout élément z également super-type strict de y, alors soit z est non relié à x (types incomparables) soit z est un super-type (au sens large) de x. Autrement dit, il ne peut exister un type à la fois sous-type de x et super-type de y qui soit différent de x ou y.

En d'autres termes, la relation $\succ$ forme la *fermeture transitive* de $\succ_{dir}$.

Pour les sous-types directs et la relation $\prec_{dir}$, la propriété est similaire mais porte évidemment sur la relation de sous-typage strict $\prec$.

Les propriétés fondamentales associées aux relations de sous-types et super-types directs sont les suivantes :

$$(1)\, \forall x,y_1,y_2 : TYPE \bullet (y_1 \neq y_2) \wedge (y_1 \succ_{dir} x) \wedge (y_2 \succ_{dir} x) \Rightarrow y_1 \not\succ y_2$$
$$(2)\, \forall x,y_1,y_2 : TYPE \bullet (y_1 \neq y_2) \wedge (y_1 \prec_{dir} x) \wedge (y_2 \prec_{dir} x) \Rightarrow y_1 \not\prec y_2$$

La propriété (1) signifie que pour tout couple donné (y_1,y_2) de super-types directs de x (avec y_1 et y_2 distincts), alors ces deux types y_1 et y_2 ne sont pas comparables entre eux. La seconde propriété (2) impose les mêmes restrictions sur les sous-types directs.

✎Preuve **A.1.1 page 225** : *Propriétés pour les sous-types et super-types directs*

La conséquence principale des propriétés (1) et (2) est que la structure sous-jacente des systèmes de type dérivables de la théorie proposée dans cette section n'est pas exempte d'*ambiguïtés*. Ainsi, lorsque nous «rechercherons» les super-types ou sous-types directs d'un type donné, la multiplicité et le critère de non-comparabilité des types «trouvés» posera des problèmes de choix qu'il nous faudra résoudre précisément.

4.2.2 Bornes maximales

Toujours dans une idée de restriction d'ensembles de super-types ou sous-types, nous définissons ci-dessous deux fonctions utilitaires *tmin* et *tmax* qui extraient d'un ensemble de types les bornes respectivement *minimale* (pour $\preceq$) et *maximale* (pour $\succeq$) :

$$tmin : \mathbb{P}\, TYPE \rightarrow \mathbb{P}\, TYPE$$
$$tmax : \mathbb{P}\, TYPE \rightarrow \mathbb{P}\, TYPE$$

$$\forall T \subseteq \mathbb{P}\, TYPE \bullet tmin(T) = T \setminus \{\, x : T \mid (\exists\, y : T \bullet y \prec x) \,\}$$
$$\forall T \subseteq \mathbb{P}\, TYPE \bullet tmax(T) = T \setminus \{\, x : T \mid (\exists\, y : T \bullet y \succ x) \,\}$$

La première fonction, *tmin*, part d'un ensemble de types T et lui ôte tous les types x dont il existe un sous-type strict y dans T. Il ne reste donc dans l'ensemble final que des types *minimaux* (i.e. sans sous-type strict). La fonction *tmax* effectue le même «travail» mais ne conserve dans l'ensemble résultat que les types *maximaux* (i.e. sans super-types strict).

Contrairement aux définitions usuelles des fonctions d'extraction de bornes (les fameuses *lub* et *glb*), les résultats de ces deux fonctions sont donc des ensemble de types; nous verrons que cette propriété est très importante pour les primitives de connexion définies en section 5.5 page 75.

4.3 Exemple de système et représentation graphique

Résumons la théorie des types que nous avons mis en place dans les sections précédentes au travers d'un exemple que nous nommerons $\mathcal{T}_S$.

L'ordre $(\mathcal{T}_S, \succeq)$ est un ensemble partiellement ordonné. Chaque type de ce système, élément plus généralement de *TYPE* est relié par $\succeq$ à un ensemble (potentiellement vide) de sous-types dans $\mathcal{T}_S$. Certains de ces types sont directs (ils sont reliés par $\succ_{dir}$) et les autres forment la fermeture réflexive et transitive de $\succ_{dir}$.

Pour comprendre la façon dont ces règles fonctionnent, nous allons construire un système de type très simple. Premièrement, le système possède une racine unique que nous appellerons *Event* (cf. section 4.4.1 page ci-contre). Par définition, cette racine unique ne possède pas de super-type et n'est (donc) sous-type d'aucun autre type du système. Nous introduisons ensuite le type *Data*, sous-type direct de *Event* et qui dénote les types de données simples du système. *Number* en est un sous-type pour les nombres en général, son unique sous-types direct étant *Real*. Les entiers peuvent alors être considérés comme sous-types de *Real*, tout comme les nombres rationnels. Les chaînes de caractère sont valeurs du type *String*. Une autre branche de la hiérarchie, *Control* dénote les types ayant un impact au niveau contrôle, dans le cadre du langage Scope décrit au chapitre 6 page 87. Un sous-type *Activate* est introduit pour supporter un protocole d'activation dynamique (qui revêtira par la suite une importance fondamentale). *Initialize* est un autre sous-type de contrôle, ainsi que *InitializeString* qui est à la fois une chaîne de caractère et un indicateur lié au contrôle.

En utilisant la relation prédécesseur (super-type direct), le système peut être décrit de la façon suivante :

- *Event* est la racine unique du système
- $(\mathit{Event} \succ_{dir} \mathit{Data})$ et $(\mathit{Event} \succ_{dir} \mathit{Control})$
- $(\mathit{Data} \succ_{dir} \mathit{String})$ et $(\mathit{Data} \succ_{dir} \mathit{Number})$
- $(\mathit{Number} \succ_{dir} \mathit{Real})$
- $(\mathit{Real} \succ_{dir} \mathit{Integer})$ et $(\mathit{Real} \succ_{dir} \mathit{Rational})$
- $(\mathit{Control} \succ_{dir} \mathit{Activate})$ et $(\mathit{Control} \succ_{dir} \mathit{Initialize})$
- $(\mathit{Initialize} \succ_{dir} \mathit{InitializeString})$ et $(\mathit{String} \succ_{dir} \mathit{InitializeString})$

Cet exemple quoique simple propose quelques caractéristiques intéressantes. Premièrement, la possibilité de distinguer structure du système de type (ce que nous établissons dans cette section) et représentation interne des types mis en œuvre offre une grande liberté concernant la relation de sous-typage. Ainsi, alors que la représentation machine différente des entiers et des réels impose en général de les considérer comme distincts, notre approche permet de définir les entiers comme étant des cas particuliers de réels, ce qui est le cas en mathématique puisque $\mathbb{N} \subset \mathbb{R}$.

Nous pouvons également remarquer la capture, intentionnelle pour l'instant (nous la réaliserons dans les sections suivantes), de propriétés de contrôle au sein même du système de type (types *Activate* et *Initialize*).

Enfin, il est possible dans notre modèle d'exposer des sous-types de plusieurs super-types directs (non comparables entre eux, comme indiqué précédemment), ce qui est par exemple le cas de *InitializeString*, à la fois sous-type de *Initialize* et de *String*.

La figure 4.1 page ci-contre donne une représentation graphique de notre exemple de système de type simple. Nous pouvons vérifier sur cette figure la plupart des propriétés structurelles introduites précédemment :

- multiplicité des sous-types directs : *Data*, par exemple, possède deux sous-types directs distincts : *String* et *Number*
- multiplicité des super-types directs : *InitializeString* est à la fois sous-type direct de *Initialize* et de *String*
- non-comparabilité des sous-types (ou super-types) directs : il n'existe aucune relation graphique entre par exemple les nœuds *String* et *Initialize*.

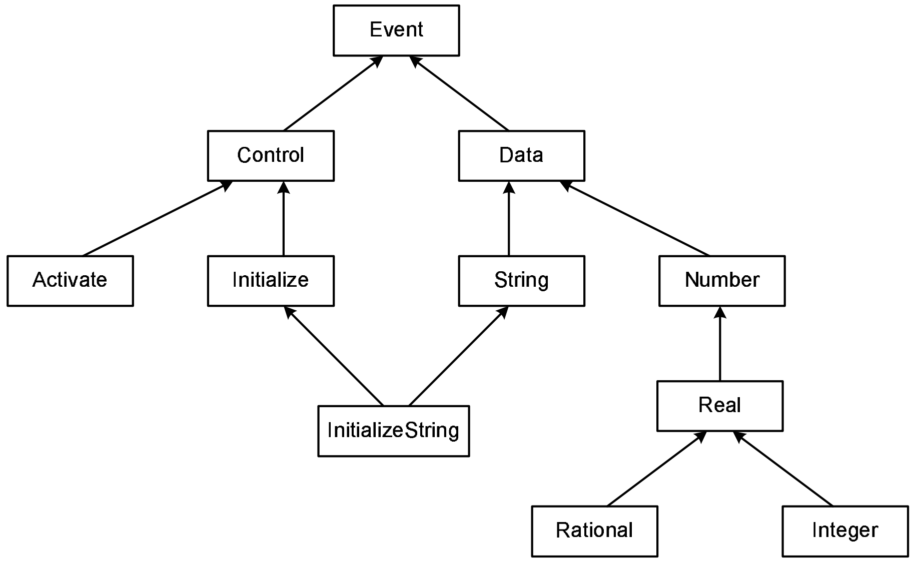


FIG. 4.1 – *Représentation graphique d'un système de type simple*

4.4 Dynamique du système de type

La notion de type dans le modèle Scope est intrinsèquement associée à la notion d'événement. Comme expliqué précédemment, nous mettons en œuvre des composants dont les «conversations» sont essentiellement supportées par l'échange d'événements. L'information de type associée aux événements servira donc également de base au typage des composants eux-mêmes.

Ainsi, en se basant sur notre théorie structurelle des types, définie à la section précédente, nous pouvons proposer le système de type des événements du modèle Scope comme une instance contrainte de cette théorie.

4.4.1 Définition

La propriété fondamentale du système de type (en fait de l'ensemble du modèle formel) proposé dans notre approche, concerne sa nature dynamique. Nous verrons ainsi en section 4.4.2 page suivante que contrairement à la théorie des types elle-même qui dénote l'ensemble des types possibles, le système de type n'est initialement composé que d'un unique type. Toute la puissance du système se révélera lors de son enrichissement au cours même de l'exécution du système informatique qu'il supporte.

La définition Z du système de type *TypeSystem* mis en œuvre dans notre approche est la suivante :

TypeSystem

$$types : \mathbb{P}_1\ TYPE$$
$$super : TYPE \leftrightarrow TYPE$$

$$\forall\, t_1, t_2 : types \bullet \{t_1 \mapsto t_2\} \in super \Leftrightarrow t_1 \succeq t_2$$

$$\mathrm{dom}\ super = \mathrm{ran}\ super = types$$

$$\exists_1\, r : types \bullet \forall\, t : types \bullet r \succeq t$$

La variable *types*, déclarée dans le schéma *TypeSystem*, représente un ensemble non-vide de types. Cette variable représentera l'ensemble des types «connus» par le système à un instant donné. Ainsi, le système ne connaît pas tous les types possibles mais seulement un sous-ensemble fini. La relation *super* est une simple restriction de la relation $\succeq$ qui sera très utile pour définir des contraintes ensemblistes sur la structure de système de type. La première de ces contraintes permet de relier l'ensemble *types* et la relation *super*. Elle indique que cette dernière relation *super* a pour domaine l'ensemble des types valides du système : *types*.

La dernière propriété du schéma pose l'existence et l'unicité du type r du système (et non un type quelconque) tel que tous les autres types «connus» du système à un moment donné sont sous-types de r. On dira que r représente la *racine unique* du système de type.

Pour simplifier les obligations de preuve sur le système de type, nous allons donner l'ensemble des contraintes sur les variables *types* et *super*. Commençons par l'ensemble des types *types* :

$(a) types \in \mathbb{P}_1\ TYPE$ [d'après déclaration]
$(b) types \neq \emptyset$ [idem]

En ce qui concerne la relation *super*, nous obtenons :

(1) $\mathrm{dom}\ super = \mathrm{ran}\ super = types$ [def]
(2) $super \in types \leftrightarrow types$ [(1)]
(3) $\forall\, t_1 : types;\ t_2 : types \bullet \{t_1 \mapsto t_2\} \in super \Leftrightarrow t_1 \succeq t_2$
(4) $\forall\, t_1 : types \bullet \{t_1 \mapsto t_1\} \in super$ [(3) et réflexivité]
(5) $\forall\, t_1 : types;\ t_2 : types;\ t_3 : types \bullet$
$(\{t_1 \mapsto t_2\} \in super \land \{t_2 \mapsto t_3\} \in super) \Rightarrow \{t_1 \mapsto t_3\} \in super$ [(3) et transitivité]
(6) $\forall\, t_1 : types;\ t_2 : types \bullet (\{t_1 \mapsto t_2\} \in super \land \{t_2 \mapsto t_1\} \in super) \Rightarrow t_1 = t_2$ [(3) et antisymétrie]
(7) $\exists_1\, r : types \bullet \forall\, t : types \bullet \{r \mapsto t\} \in super$

Les propriétés (1) et (3) sont reprises du schéma. La propriété (2) qui dit que *super* est une relation de l'ensemble des relations entre *types* et lui-même, est une conséquence logique de (1). Les propriétés (4), (5) et (6) proviennent de l'équivalence, restreinte au sous-ensemble *types*, entre *super* et la relation de type $\succeq$. Ainsi, la relation *super* est elle aussi réflexive, transitive et antisymétrique mais seulement sur son domaine.

4.4.2 État initial

La notion de temps [3] est fondamentale pour notre système de type. Au temps initial, ce dernier n'est composé que d'un unique type racine *Event*. Nous proposerons en section suivante une opération pour enrichir ce système, dynamiquement. Quand un nouveau type a ainsi été ajouté au système, on dira que le temps (virtuel) du système de type a changé.

Du point de vue de la méthode Z, ce temps initial pour le système de type est modélisé sous la forme d'un schéma d'état initial :

InitTypeSystem
TypeSystem'

$types' = \{Event\}$
$super' = \{Event \mapsto Event\}$

Ce schéma introduit le nom du système de type *TypeSystem'* décoré par une apostrophe pour signifier le système de type *après* initialisation. Ce système initial comporte un unique type *Event*, racine du système et donc uniquement relié à lui-même dans la relation de sous-typage ($Event \succeq Event$).

Pour vérifier la cohérence de cet état initial, nous devons montrer qu'il existe bien un état du système *TypeSystem'* qui correspond à *InitTypeSystem*. Cette obligation de preuve se nomme *théorème d'initialisation*.

✎Preuve **A.1.2 page 225** : *Théorème d'initialisation du schéma InitTypeSystem*

4.4.3 Ajout de type

Après avoir correctement formalisé le système de type que nous allons employer et lui avoir adjoint un état initial cohérent, il nous reste à montrer comment ce système peut être modifié dynamiquement. Encore une fois, cette propriété de modification dynamique est essentielle dans notre approche. Notons cependant qu'il

3. Nous considérons dans notre discussion, en suivant les principes de la méthode Z, que le temps est une information implicite, liée aux transitions du système.

s'agit d'un système uniquement constructif. En effet, la seule opération permise concerne l'ajout d'un nouveau type dans le système[4].

Pour illustrer l'ajout dynamique d'un type dans le système, nous allons considérer un système de type abstrait, représenté en figure 4.2.

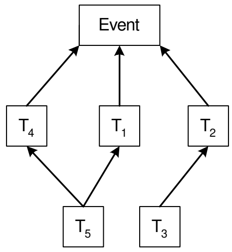


FIG. 4.2 – *Ajout de type : système ts de départ*

L'état du système correspondant à ce graphe peut-être décrit en décrivant une instance *ts* du schéma *TypeSystem* pour l'exemple considéré :

$$\begin{aligned}
ts = \langle & types \Rrightarrow \{Event, T_1, T_2, T_3, T_4, T_5\}, \\
& super \Rrightarrow \{T_1 \mapsto Event, T_2 \mapsto Event, T_4 \mapsto Event, T_5 \mapsto T_1, T_5 \mapsto T_4, T_3 \mapsto T_2, \\
& \quad Event \mapsto Event, T_1 \mapsto T_1, T_2 \mapsto T_2, T_3 \mapsto T_3, T_4 \mapsto T_4, T_5 \mapsto T_5, \\
& \quad T_5 \mapsto Event, T_3 \mapsto Event\}\rangle
\end{aligned}$$

Nous pouvons observer qu'en plus des relations directes représentées graphiquement, nous ne devons pas omettre dans la définition de *super* que les types sont reliés à eux-mêmes (réflexivité) et à leur super-types au sens large (transitivité). Ainsi, par exemple, T_5 est relié à lui-même, à T_4 et T_1 ses deux super-types directs, et à *Event*, son super-type au sens large.

Revenons maintenant à la définition de notre opération d'ajout. Pour paramétrer cette opération, nous aurons besoin d'un type (nouveau pour le système) à ajouter, ainsi que l'ensemble (non-vide) des types qui deviendront ses super-types directs lorsque l'ajout sera réalisé. Le schéma suivant décrit en Z cette opération *AddType* sur l'état *TypeSystem* :

$$\begin{array}{|l}
\hline
\textit{AddType} \\
\Delta\, \textit{TypeSystem} \\
\\
type? : TYPE \\
super? : \mathbb{P}_1\ TYPE \\
\hline
type? \notin types \\
super? \subseteq types \\
\forall s_1, s_2 : super? \bullet (s_1 \sim s_2) \Rightarrow (s_1 = s_2) \\
\\
types' = types \cup \{type?\} \\
super' = (super \cup \{s : super? \bullet s \mapsto type?\})^{+} \cup \{type? \mapsto type?\} \\
\hline
\end{array}$$

La déclaration Δ *TypeSystem* indique que notre opération concerne l'état défini par le schéma *TypeSystem*. Elle introduit les variables *types* et *super* de l'état du système de type *avant* l'opération d'ajout ainsi que les variantes *types'* et *super'* correspondant à l'état du système *après* réalisation de l'opération. Pour Z, l'opération *AddType* représente donc une transition entre un *état avant*, *TypeSystem*, et un *état après*, *TypeSystem'*.

Les variables *type*? et *super*? correspondent aux paramètres de l'opération *AddType*. La première de ces variables, *type*?, donne le type a ajouter[5] au système. La deuxième indique la liste des super-types directs pour le type à ajouter, ce qui permet de le «placer» correctement dans la structure du système de type.

Dans notre exemple, nous allons considérer l'ajout d'un type T_6, avec pour super-types directs : T_5 et T_3. Ainsi, une instance *at* de l'opération *AddType* pourra être, dans notre exemple :

4. Le retrait de type serait possible au prix d'une réorganisation structurelle globale, ce qui ne sera pas facile lorsque le système de type sera distribué.

5. Pour définir un type, un simple nom *unique* suffit au niveau structurel.

$$at = \langle type? \Rrightarrow T_6,\ super? \Rrightarrow \{ T_5, T_3 \} \rangle$$

Les préconditions à satisfaire pour que l'opération soit réalisable sont au nombre de trois. La première impose que le nouveau type à ajouter ne préexiste pas dans le système de type considéré. Il suffit de vérifier que le nouveau type *type?* n'est pas élément des types du système avant ajout : *types*. Dans notre exemple, cette précondition est vérifiée puisque T_6 n'est pas un élément de *types* dans notre système d'exemple *ts*.

Dans le même ordre d'idée, les types indiqués comme super-types directs doivent, eux, être des types valides pour le système. Or, T_3 et T_5 sont bien valides, ils sont élément de l'ensemble *types* dans l'instance *ts* du schéma *TypeSystem*.

La troisième précondition impose que les super-types directs ne soient pas comparables entre eux[6]. Certes, il ne s'agit pas d'une précondition fondamentale (comme nous le verrons par la suite, elle n'a pas d'impact sur les postconditions) mais indiquer deux super-types directs comparables constitue cependant une erreur «conceptuellement» importante. Dans notre exemple, T_3 et T_5 ne sont pas reliés entre eux dans la relation *super* du système d'exemple *ts*.

Les postconditions, dans la troisième partie du schéma de l'opération *AddType*, relient les variables du système de type avant et après l'opération. Tout d'abord, nous indiquons simplement que le nouveau type ajouté est bien un type valide du système résultant : il fait partie de l'ensemble des types *types* après l'opération. On écrira pour notre exemple :

$$\begin{aligned} ts.types' &= ts.types \cup \{ T_6 \} \\ &= \{ Event, T_1, T_2, T_3, T_4, T_5, T_6 \} \end{aligned}$$

Au niveau de la structure du système, les choses se compliquent quelque peu. Nous devons tout d'abord ajouter dans la relation *super* l'ensemble des associations entre le nouveau type ajouté, ici T_6 et ses super-types directs (cf. figure 4.3) :

$$ts.super' = ts.super \cup \{ T_6 \mapsto T_3, T_6 \mapsto T_5 \}$$

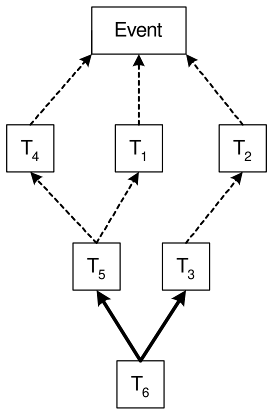


FIG. 4.3 – *Ajout de type : système ts après ajout*

Cependant, cela ne suffit pas puisque, comme indiqué précédemment, la relation *super* (par assimilation à $\succeq$) est réflexive et transitive. Dans un premier temps, nous effectuons (par l'opérateur suffixe $^+$) la *fermeture transitive* de la relation modifiée, on obtient alors (cf. figure 4.4 page suivante) :

$$\begin{aligned} ts.super' = ts.super \cup \{ T_6 \mapsto T_3, T_6 \mapsto T_5 \} \\ \cup \{ T_6 \mapsto T_2, T_6 \mapsto T_4, T_6 \mapsto T_1, T_6 \mapsto Event \} \end{aligned}$$

Finalement, par ajout de l'information $T_6 \succeq T_6$, nous obtenons le système final par fermeture transitive et réflexive[7]. Le résultat pour notre exemple est donc le suivant :

6. Il revient au même, comme précisé dans le schéma de l'opération, de vérifier que deux types comparables sont forcément égaux

7. Nous n'avons pas utilisé l'opérateur * qui réalise, en Z, la fermeture transitive et réflexive pour la simple raison que la fermeture ne se produit pas sur l'ensemble de tous les types (éléments de $TYPE$) mais uniquement sur sa restriction à *types*.

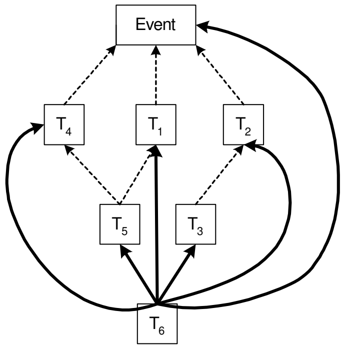


FIG. 4.4 – *Ajout de type : femeture transitive du nouveau système*

$$
\begin{aligned}
ts.super' = ts.super &\cup \{ T_6 \mapsto T_3, T_6 \mapsto T_5 \} \\
&\cup \{ T_6 \mapsto T_2, T_6 \mapsto T_4, T_6 \mapsto T_1, T_6 \mapsto Event \} \\
&\cup \{ T_6 \mapsto T_6 \}
\end{aligned}
$$

Si la définition de l'opération *AddType* ne pose pas d'important problème en elle même, il nous reste à prouver que cette opération est cohérente, c'est-à-dire qu'elle préserve la structure de notre système de type. Pour se faire, il nous suffit de prouver qu'il existe toujours un état d'arrivée cohérent par application de l'opération sur l'état de départ. Pour Z, il s'agit de prouver la validité du schéma pre *Operation* où *Operation* est le schéma décrivant l'opération considérée. Pour notre exemple, notre objectif sera donc de prouver pre *AddType*.

✎Preuve **A.1.3 page 226** : *Preuve de cohérence de l'opération AddType*

«Je composerai jusqu'à la décomposition.»
Serge Gainsbourg

«Les vices entrent dans la composition des vertus, comme les poisons entrent dans la composition des remèdes.»
La Rochefoucauld

5
Modèle de composition

Nous appelons *composition* tout assemblage de composants reliés entre eux par le biais de connexions. Dans un premier temps, nous allons expliquer ce que les composants représentent d'un point de vue structurel. Nous définirons ensuite les opérations qui leur sont associées (ajout et retrait dynamique de composants). Nous verrons par la suite comment se réalise l'interconnexion entre les composants d'un système.

5.1 Composants

Au niveau structurel, un composant Scope est une entité abstraite définie par un identificateur unique et un type. Posons dans un premier temps un type Z abstrait dénotant l'ensemble des identificateurs de composants :

$[IDENTIFIER]$

Un composant abstrait (ou structurel) est alors défini de la façon suivante :

AbstractComponent
$id : IDENTIFIER$

$in\_type : \mathbb{P}\ TYPE$
$out\_type : \mathbb{P}\ TYPE$

En plus de l'identificateur unique *id*, ce schéma introduit deux informations relatives au type des composants.

La variable *in_type* représente le *type entrant* d'un composant, c'est-à-dire l'ensemble des types d'événements que le composant peut recevoir. Par exemple, si le type entrant d'un composant donné contient un type t, alors tout événement de type t reçu pourra être pris en charge par le composant. Naturellement, par subsomption, tout sous-type de t pourra également être accepté (mais il sera probablement considéré comme un t et non plus un sous-type).

Le *type sortant* est pour sa part représenté par la variable *out_type*. Il s'agit de l'ensemble des types d'événements que le composant est susceptible d'émettre (encore une fois, il est toujours possible d'émettre un sous-type).

Nous pouvons remarquer que le schéma d'état *AbstractComponent* est fort peu contraint puisqu'il ne propose pas de partie prédicat. La partie déclaration fixe uniquement les contraintes de type sur les variables.

Une conséquence importante est que les types entrant et sortant ne sont reliés en aucune façon (cf. encart).

En outre, les types entrant *in_type* et sortant *out_type* sont définis comme étant des ensembles de types. Ceci implique l'impossibilité de déclarer plusieurs fois le même type en entrée ou en sortie.

Types et variance

Dans les modèles objets typés usuels, les interprétations ensemblistes ou fonctionnelles imposent des contraintes de variance différentes pour le sous-typage des types de retour et des types des arguments des méthodes. Par exemple, dans Eiffel (d'interprétation plutôt ensembliste), le type de retour est contravariant alors que les types des arguments sont eux covariants [Mey97]. Dans Scope, le découplage entre type entrant et type sortant [a] «contourne» ces problèmes de variance, ce qui représente une propriété intéressante concernant la cohérence de notre approche. Ainsi il sera, selon notre point de vue et de façon toute relative, plus aisé de discuter sur l'extension du sous-typage aux composants eux-mêmes.

[a] De façon un peu raccourcie, nous pouvons utiliser l'analogie: type entrant scope équivalent à type d'argument objet et type sortant équivalent à type de retour.

Par exemple, les composants

$$c_1 = \langle id \Rrightarrow Id_1;\ in\_type \Rrightarrow \{T_1, T_1\};\ out\_type \Rrightarrow \{T_2\}\rangle$$

et

$$c_2 = \langle id \Rrightarrow Id_2;\ in\_type \Rrightarrow \{T_1\};\ out\_type \Rrightarrow \{T_2, T_2\}\rangle$$

ne sont pas valides du point de vue typage. En revanche,

$$c_3 = \langle id \Rrightarrow Id_3;\ in\_type \Rrightarrow \{T_1, T_2\};\ out\_type \Rrightarrow \{T_2\}\rangle$$

est un composant tout à fait acceptable au niveau structurel.

Il est cependant possible d'indiquer des types entrant ou sortant faisant partie de la même branche de sous-typage. Ainsi, en prenant un système de type ts tel que $ts.types = \{Event, T_1, T_2, T_3\}$ et $T_1 \succeq T_2$, $T_2 \succeq T_3$, alors le composant c tel que $c.in\_type = \{T_1, T_2\}$ et $c.out\_type = \{T_3, T_4\}$ est valide.

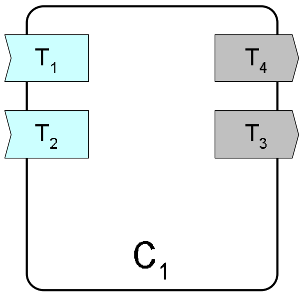


FIG. 5.1 – *Représentation graphique d'un composant abstrait*

Nous donnons en figure 5.1 une représentation graphique possible pour ce dernier composant c. Remarquons sur cette figure la représentation différente des types entrant et sortant.

5.2 Connexions effectives

Dans notre approche, les connexions qui relient les composants entre eux sont de nature abstraite. Cependant, nous les modélisons au niveau opérationnel en tant qu'entités à part entière dénommées *connexions effectives*.

Une telle connexion effective est définie selon le schéma Z suivant :

EffectiveConnection

$source : AbstractComponent$
$dest : AbstractComponent$

$type : \mathbb{P}_1\ TYPE$

$source \neq dest$

Connexions unidirectionnelles

Il est également important de noter que le découplage entre types entrant et types sortant confère aux connexions une caractéristique d'*unidirectionalité*, d'autres approches préférant reposer sur des liens bidirectionnels [IWC+88] [Sun98]. Bien sûr, nous pourrons composer les connexions pour modéliser de tels liens bidirectionnels.

De façon prévisible, ce schéma relie un composant *source* et une destination *dest*. La partie prédicat nous informe que les composants source et destination ne peuvent être égaux; les cycles directs ne sont donc pas autorisés dans Scope.

La variable *type*, fondamentale, dénote un ensemble non-vide de *types effectifs* de connexion. Cet ensemble désigne quels types d'événements peuvent être transmis depuis le composant source jusqu'au composant destination via la connexion considérée.

Prenons l'exemple d'une connexion cx telle que :

$$cx = \langle source \Rrightarrow c_1;\ dest \Rrightarrow c_2;\ type \Rrightarrow \mathcal{T} \rangle$$

où c_1 et c_2 sont deux composants du système et $\mathcal{T}$ est l'ensemble des types effectifs de la connexion [1].

Un événement de type t_1 peut être émis depuis c_1 et reçu par c_2 en utilisant la connexion cx si, et seulement si il en existe un super-type (au sens large) dans le type effectif de connexion, ou, plus formellement :

$$\exists\, t_2 : \mathcal{T} \bullet t_2 \succeq t_1$$

Remarquons que sans investigation plus approfondie, cette condition est ambiguë puisque plusieurs types de $\mathcal{T}$ sont susceptibles de la satisfaire.

5.3 Systèmes opérationnels

Un *système opérationnel abstrait*, que nous nommerons parfois *architecture* pour en souligner le caractère structurel, correspond à l'interconnexion d'un ensemble de composants.

Tout comme le système de type (qu'il intègre), ce constituant très important de notre modèle se représente en Z sous la forme d'un triplet : schéma d'état / état initial / opérations associées.

5.3.1 Etat et invariant

D'un point de vue formel, un système opérationnel consiste en l'agglomération d'un ensemble de composants et d'un ensemble de connexions effectives qui les référencent.

Ceci se traduit en Z de la façon suivante :

$$
\begin{array}{l}
\textit{AbstractOperationalSystemPart} \\
\hline
comps : \mathbb{P}\, AbstractComponent \\
conxs : \mathbb{P}\, EffectiveConnexion \\
\hline
\forall\, c_1, c_2 : comps \bullet c_1.id = c_2.id \Rightarrow c_1 = c_2 \\[1ex]
\forall\, cx : conxs \bullet \\
\quad cx.source \in comps \land \\
\quad cx.dest \in comps \land \\[1ex]
\quad \forall\, t : cx.type \bullet \\
\qquad t \in types \land \\
\qquad \exists\, t_1 : cx.source.out\_type \bullet (t_1 \sim t) \land \\
\qquad \exists\, t_2 : cx.dest.in\_type \bullet (t_2 \succeq t)
\end{array}
$$

1. Par abus de langage, nous désignerons parfois un tel ensemble par «le type effectif de connexion».

Avant de décrire ce schéma, nous pouvons remarquer des références aux relations $\sim$ et $\succeq$, ainsi qu'à la variable *types*, non déclarées mais qui nous font immanquablement penser au système de type défini précédemment. Le suffixe *Part* utilisé dans le nom de ce schéma indique une définition partielle qui doit être complétée par composition, ici avec le schéma *TypeSystem* de la façon suivante :

$$AbstractOperationalSystem \mathrel{\widehat=} TypeSystem \wedge AbstractOperationalSystemPart$$

Les composants gérés par le système sont regroupés dans l'ensemble *comps* et les connexions effectives le sont dans *conxs*. Cela peut paraître anodin mais ces définitions ensemblistes nous informent que composants et connexions sont tous distincts pour un système donné.

Nous retrouvons cette discrimination dans le premier prédicat de l'invariant associé à tout système opérationnel : les identificateurs de composants sont tous différents et permettent donc de désigner de façon unique un composant donné.

Le partie restante de cet invariant concerne les connexions et leurs relations avec l'ensemble des composants et le système de type. Dans un premier temps, il est essentiel de s'assurer que pour toute connexion, les composants source et destination référencés appartiennent bien à l'ensemble des composants gérés par le système : *comps*.

Les trois dernières conditions, plus complexes mais fondamentales, portent sur le système de type. La première indique simplement que le type effectif de connexion (lui-même un ensemble de types) doit être un sous-ensemble des types connus du système. De cette façon, nous établissons le lien entre le système opérationnel et le système de type qu'il gère.

Pour qu'une connexion effective soit cohérente, il est nécessaire de respecter d'autres propriétés importantes au niveau typage.

De façon intuitive, tout événement émis peut être réceptionné par un composant si au moins un type entrant du composant est relié, par sous-typage, avec le type de cet événement. En effet, par subsomption, un composant attendant un type donné peut accepter un sous-type de façon transparente. Comme les types des événements «circulant» via la connexion considérée sont au moins sous-types des éléments t du type effectif $cx.type$, nous pouvons résumer formellement notre propriété de la façon suivante :

$$\forall\, t : cx.type \bullet \exists\, t_2 : cx.dest.in\_type \bullet t_2 \succeq t$$

La contrainte est qu'il doit exister au moins un type t_2 dans les «entrées» du composant destination tel que ce type est super-type des types effectifs de la connexion.

Au niveau du composant source, la contrainte est similaire quoique sensiblement relâchée. En effet, si seulement des sous-types pourront réellement «circuler» via la connexion considérée, il est tout de même possible de l'établir pour un super-type dans le but de permettre les émissions multipoints sélectionnées implicitement en fonction du type, mécanisme très intéressant en environnement distribué.

Considérons en guise d'exemple l'architecture d'un système opérationnel *os* représentée en figure 5.2. Ce système utilise le système de type exemple que nous avons décrit précédemment, dans sa version «après ajout» (cf fig. 4.3 page 66).

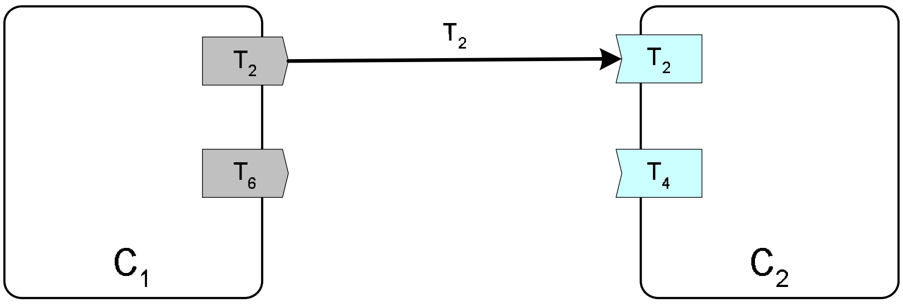


FIG. 5.2 – *Représentation d'un système opérationnel abstrait*

Nous définissons dans ce système exemple deux composants c_1 et c_2 tels que $c_1.out\_type = \{T_6, T_2\}$ et $c_2.in\_type = \{T_4, T_2\}$. L'unique connexion cx_1 représentée est définie par :

$$cx_1 = \langle source \Rrightarrow c_1;\ dest \Rrightarrow c_2;\ type \Rrightarrow \{T_2\}\rangle$$

Cet exemple simple ne pose pas de problème puisque les types entrant, sortant et de connexion effective mis en jeu sont tous trois T_2. Selon l'invariant proposé, des connexions de nature plus complexe, à la fois conceptuellement et à représenter, sont envisageables.

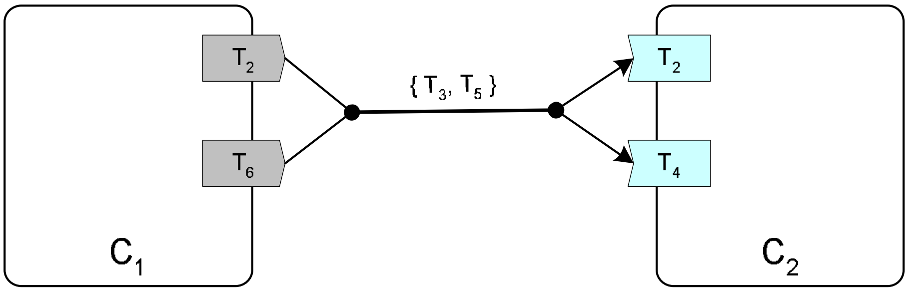


FIG. 5.3 – *Une connexion effective plus complexe*

L'architecture représentée en figure 5.3 introduit une connexion cx_2 dont le type effectif ne se limite pas à un unique type et qui exploite les possibilités de sous-typage :

$$cx_2 = \langle source \Rrightarrow c_1;\ dest \Rrightarrow c_2;\ type \Rrightarrow \{T_5, T_3\}\rangle$$

Ici, nos contraintes de types sont encore une fois respectées à la source puisque l'on a $T_5 \sim T_6$ et $T_3 \sim T_2$. Au niveau de la destination, nous avons bien $T_4 \succeq T_5$ et $T_2 \succeq T_3$. Notons cependant qu'en pratique de tels exemples - à la représentation relativement complexe - ne se présenteront que rarement. Il est cependant important d'explorer toutes les possibilités offertes par notre modèle.

5.3.2 Etat initial

Notre schéma de système opérationnel étant établi, nous pouvons lui adjoindre un état initial cohérent. Au «démarrage» du système, il n'existe aucun composant ni aucune connexion dans le système, ce qui implique le schéma d'initialisation suivant :

InitAbstractOperationalSystemPart

$AbstractOperationalSystem'$

$comps' = \emptyset$
$conxs' = \emptyset$

Pour obtenir le schéma complet, il nous faut le composer avec l'initialisation du système de type, ce qui nous donne :

$$InitAbstractOperationalSystem \mathrel{\widehat{=}} InitTypeSystem \wedge InitAbstractOperationalSystemPart$$

✎Preuve **A.2.1 page 230** : *Théorème d'initialisation pour InitAbstractOperationalSystem*

5.4 Opérations sur les composants

En plus de l'ajout dynamique d'un type dans le système, un premier groupe d'opérations disponibles sur les systèmes opérationnels concerne la gestion des composants.

Ces opérations n'ayant aucune incidence sur le système de type, nous omettrons dorénavant toute référence à l'invariant de *TypeSystem*[2].

2. Nous aurons cependant besoin des variables définies par ce même schéma.

5.4.1 Ajout de composant

La première primitive que nous allons modéliser concerne l'ajout dynamique d'un nouveau composant dans le système. Le schéma qui décrit cette opération est le suivant :

$$
\begin{array}{l}
\hline
\textit{NewComponent} \\
\Delta \textit{AbstractOperationalSystemPart} \\
\Xi\, \textit{TypeSystem} \\
c? : \textit{AbstractComponent} \\
\hline
c?.in\_type \subseteq types \\
c?.out\_type \subseteq types \\
\forall\, c : comps \bullet c.id \neq c?.id \\
conxs' = conxs \\
comps' = comps \cup \{c?\} \\
\hline
\end{array}
$$

Cette opération porte sur le schéma d'état *AbstractOperationalSystem*. Cependant, nous référençons séparément la partie système de type dont l'état est préservé (utilisation de la notation $\Xi Schema$).

La partie *AbstractOperationalSystemPart*, elle, est modifiée et est donc déclarée séparément (notation $\Delta Schema$).

Les préconditions de l'opération *NewComponent* sont simples : il s'agit tout d'abord de vérifier que les types indiqués par le composant à ajouter $c?$ sont valides pour le système considéré. De plus, l'identificateur de ce nouveau composant, $c?.id$ doit être unique au niveau du système.

L'impact au niveau du système correspond simplement à l'ajout du composant à l'ensemble des composants après réalisation de l'opération : $comps'$. L'ensemble des connexions, lui, n'est bien sûr pas modifié.

✎Preuve **A.2.2 page 230** : *Cohérence de l'opération NewComponent*

5.4.2 Retrait de composant

Le schéma suivant donne la définition de la primitive de retrait de composant :

$$
\begin{array}{l}
\hline
\textit{DeleteComponent} \\
\Delta \textit{AbstractOperationalSystemPart} \\
\Xi\, \textit{TypeSystem} \\
id? : \textit{IDENTIFIER} \\
\hline
\forall\, cx : conxs \bullet cx.source.id \neq id? \wedge cx.dest.id \neq id? \\
conxs' = conxs \\
comps' = comps \setminus \{\mu\, c : comps \mid c.id = id?\} \\
\hline
\end{array}
$$

Le paramètre $id?$ est nécessaire pour savoir quel composant retirer du système. Nous avons vu, en effet, que cet identificateur référençait un composant de façon non ambiguë pour un système opérationnel donné.

Le schéma *DeleteComponent* nous donne de plus deux informations importantes concernant la validité de l'opération de retrait. Tout d'abord, en guise de première précondition, le composant à retirer ne doit être référencé par aucune connexion comme composant source ou destination. Ceci signifie que le composant que nous désirons enlever du système ne doit être préalablement connecté à aucun autre composant. Nous verrons en section 5.6.3 page 86 qu'il est possible d'obtenir une opération de retrait opérant sur les composants connectés. Mais il s'agira d'une *opération composée* alors que le retrait simple est défini en tant que *primitive*.

Cette opération n'est également valide (d'après définition de la notation μ qui dénote une précondition d'existence et d'unicité de l'élément considéré) que dans le cas où un composant possède bien l'identificateur $id?$ passé en paramètre.

Ces deux conditions limitent grandement l'impact du retrait sur le système et son invariant, de sorte que la cohérence de l'opération devient «évidente» (et ne sera donc pas montrée).

Types intentionnels et types effectifs de connexion

Il existe une distinction fondamentale entre le type déclaré pour la connexion, que nous nommons *type intentionnel* et l'ensemble des types effectifs (nommé *type effectif de connexion*) qui lui correspond, en termes opérationnels. Le type intentionnel, comme son nom l'indique, est considéré comme la capture de l'intention du «programmeur» [a] demandant la réalisation de la connexion. On pourrait rapprocher cette notion de celle de *contrat* du langage Eiffel [Mey97] même si nous restons dans Scope à un niveau de typage simple. En comparaison, le *type effectif* décrit l'ensemble des types valides pour l'émission des événements, au niveau opérationnel. Cette distinction entre intention et conséquence d'un ordre de connexion sur le système de type induit la mise en œuvre de mécanismes de synthèse ou d'inférence de types (cf. figure 5.4).

[a] Le modèle Scope définit une catégorie de langage que l'on pourrait appeler «langages de composition de système informatique», le terme «programmeur» pourrait donc être remplacé par «compositeur», choix maladroit nous en convenons; ou peut-être, de façon plus adéquate, par «architecte», «arrangeur», etc.

5.5 Primitives de connexion

Après l'ajout et le retrait dynamique de composants dans un système opérationnel donné, la seconde catégorie d'opérations primitives concerne l'établissement des connexions ainsi que d'éventuelles déconnexions entre composants. Ces primitives nous permettent de comprendre le passage du concept de connexion à leur réalisation effective, sous la forme du schéma *EffectiveConnection* vu précédemment (cf. page 70). Ce passage s'effectue par synthèse de type, selon le processus décrit en encart et sur la figure 5.4.

Les primitives de connexion nécessitent trois informations :

- L'identificateur du composant source considéré pour la connexion,
- l'identificateur du composant destination correspondant, et
- le *type intentionnel* de connexion.

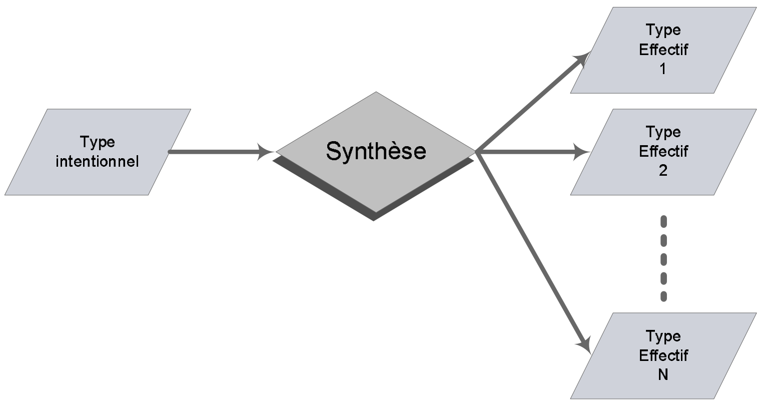


FIG. 5.4 – *Synthèse (inférence) des types effectifs à partir du type intentionnel de connexion*

A partir de la définition du système opérationnel *AbstractOperationalSystem*, nous avons identifié trois façons différentes de connecter des composants entre eux qui, tout en préservant l'invariant principal du système, correspondent à des intentions spécifiques et prégnantes du point de vue du programmeur. Ces trois catégories d'établissements de connexion sont :

1. *Connexion par spécialisation* : les types choisis au niveau des composants considérés par la connexion (et utilisés pour «calculer» les types effectifs résultants) sont choisis par spécialisation du type intentionnel.
2. *Connexion par généralisation à la source* : l'algorithme de sélection peut choisir un super-type à la source.
3. *Connexion par généralisation à la destination* : un super-type peut être considéré à la destination.

Il en résulte la nécessité de décrire trois primitives différentes pour l'établissement de connexions entre composants. Nous décrivons ces primitives dans les sections qui suivent en indiquant à chaque fois leur contexte spécifique d'utilisation.

La principale propriété que nous désirons préserver (en plus de l'invariant du système) est que dans chaque cas, le type effectif synthétisé ne doit contenir que des sous-types (au sens large) du type déclaré. En d'autres termes, il faudra s'assurer que le contrat passé entre le programmeur et le système est respecté. Bien sûr, il existera des cas où ce contrat ne peut être respecté, ce qui devrait conduire le système à interdire la connexion. Pour simplifier notre modèle formel, nous dirons que les connexions échouent si les préconditions qu'elles décrivent ne sont pas respectées [3].

5.5.1 Création de connexion effective

Nos primitives de connexion disposent de sections identiques (mêmes pré et postconditions) que nous allons factoriser en guise de préliminaire. Dans un premier temps, nous pouvons remarquer qu'en cas de succès d'une primitive, une nouvelle connexion devra être ajoutée au système, quel que soit l'algorithme choisi pour en sélectionner le type effectif. Mais avant de pouvoir ajouter cette connexion effective au système, encore faut-il en donner le contenu.

Nous devons donc créer une opération d'initialisation de connexion effective, portant sur le schéma *EffectiveConnection* décrit en section 5.2 page 70 :

InitEffectiveConnection

$$
\begin{array}{l}
\mathit{EffectiveConnection}' \\[4pt]
\mathit{source}? : \mathit{AbstractComponent} \\
\mathit{dest}? : \mathit{AbstractComponent} \\
T? : \mathbb{P}_1\ \mathit{TYPE} \\
\hline
\mathit{source}? \neq \mathit{dest}? \\[4pt]
\mathit{source}' = \mathit{source}? \\
\mathit{dest}' = \mathit{dest}? \\
\mathit{type}' = T?
\end{array}
$$

Le schéma ci-dessus est une opération déclarant trois entrées : le composant source *source*?, le composant destination *dest*? et le type effectif de connexion $T?$. Mais il s'agit également d'un schéma d'état initial pour le schéma d'état *EffectiveConnection*. Cependant , la cohérence de ce schéma «double» est trivialement validée grâce à l'unique précondition du schéma qui permet de s'assurer que les cycles directs sont absents des connexions.

L'opération *InitEffectiveConnection* est une opération locale qui ne concerne qu'une seule connexion. Cependant, nos primitives porteront sur l'ajout d'une connexion au niveau global du point de vue du système opérationnel. En Z, ce changement d'un point de vue local à une vision plus globale consiste en la définition d'un *schéma de promotion*. Ici, nous allons - dans le cadre de l'ajout d'une connexion au système - «promouvoir» l'état décrit par le schéma *EffectiveConnection* au schéma global *AbstractOperationalSystem* de la façon suivante :

AddConnection

$$
\begin{array}{l}
\Delta \mathit{AbstractOperationalSystem} \\[4pt]
\mathit{source}? : \mathit{AbstractComponent} \\
\mathit{dest}? : \mathit{AbstractComponent} \\
T? : \mathbb{P}_1\ \mathit{TYPE} \\
\hline
\mathit{source}? \in \mathit{comps} \\
\mathit{dest}? \in \mathit{comps} \\[4pt]
\forall\, cx : \mathit{conxs} \mid (cx.\mathit{source}? = \mathit{source}? \land cx.\mathit{dest}? = \mathit{dest}?) \bullet cx.\mathit{type} \neq T? \\[4pt]
\mathit{comps}' = \mathit{comps} \\
\mathit{types}' = \mathit{types} \\
\mathit{super}' = \mathit{super} \\[4pt]
\exists\, \mathit{InitEffectiveConnection} \bullet \mathit{conxs}' = \mathit{conxs} \cup \{\Theta \mathit{EffectiveConnection}'\}
\end{array}
$$

3. La *complétude* du modèle nécessiterait la description d'opérations complètes, décrivant ce qui se produit lorsque les préconditions ne sont pas satisfaites. Ce travail quoique relativement simple n'est nécessaire qu'au moment de l'implantation sûre du modèle spécifié, ce qui n'est pas notre objectif ici.

Cette opération, tout comme la précédente, attend les informations nécessaires à la création de la connexion. Seule la variable $conxs'$ dénotant l'ensemble des connexions gérées par le système est modifiée par la condition décrite en dernière ligne du schéma. Littéralement, cette condition impose de s'assurer, pour que la promotion de l'ajout de connexion soit considérée comme valide, qu'il existe un création locale de connexion telle que cette connexion crée soit ajoutée dans le système, uniquement dans la mesure où elle n'en fait pas déjà partie. Bien que d'apparence anodine, cette condition impose de s'assurer que l'invariant global du système est bien préservé par ajout de la connexion, c'est-à-dire que pre *AddConnection* n'est pas d'un calcul aussi facile qu'il en a l'air[4] :

✎Preuve **A.2.3 page 231** : *Preuve de cohérence de l'opération AddConnection*

5.5.2 Identification de composant

Les primitives de connexion que nous allons construire proposent une «interface»[5] basée sur le référencement des composants à partir de leur identificateur unique associé. Cependant, au niveau des conséquences des requêtes de connexion, les composants eux-mêmes doivent être référencés. Il nous faut donc fournir un outil de «translation» entre identifications et références directes. Notre solution consiste en la définition d'un schéma simple, donné ci-dessous :

ConnectComponentPart

$AbstractOperationalSystem$

$id\_source? : IDENTIFIER$
$id\_dest? : IDENTIFIER$

$c_1! : AbstractComponent$
$c_2! : AbstractComponent$

$c_1! = (\mu\, c : comps \mid c.id = id\_source?)$
$c_2! = (\mu\, c : comps \mid c.id = id\_dest?)$

À partir des identificateurs pour les composants source $id\_source?$ et destination $id\_dest?$, cette opération «récupère» les composants correspondant dans l'ensemble des composants gérés par le système $comps$. D'après la sémantique de la notation μ de Z, cette opération n'est valide que si les composants existent et sont uniques.

Le procédé pour décrire une opération complète de connexion sera tout d'abord de «translater» les identificateurs de composants en références valides puis de les «passer» à la primitive de connexion proprement dite. Les sorties $c_1!$ et $c_2!$ du schéma précédent seront pour l'occasion renommées en $c_1?$ et $c_2?$ pour la primitive de connexion. Le rôle de cette dernière sera de sélectionner le type effectif de connexion, à partir des composants en entrées ainsi que du type intentionnel - que nous nommerons $type?$ - puis de «passer» la main à l'opération de création de connexion détaillée dans la section précédente.

Pour l'instant, considérons une primitive de connexion «abstraite» (qui n'est donc pas suffisante pour satisfaire l'invariant du système) :

IncompleteConnectPart

$AbstractOperationalSystem$

$c1? : AbstractComponent$
$c2? : AbstractComponent$
$type? : TYPE$

$source! : AbstractComponent$
$dest! : AbstractComponent$
$T? : \mathbb{P}_1\ TYPE$

$source! = c_1?$
$dest! = c_2?$

$T! = connect\_algorithm(type?)$

4. Dans cette preuve, nous enlevons la détection de conflit avec une connexion déjà établie et possédant les mêmes constituants. En effet, cette condition, bien qu'importante, n'est pas nécessaire pour effectuer les obligations de preuve du schéma de cohérence.

5. Si elle n'existe pas vraiment au niveau formel, la notion d'interface pour un type abstrait (état, état initial, opérations) semble se justifier d'elle même à un niveau plus conceptuel.

Vues de l'extérieur, les primitives de connexions proposent donc les entrées $c_1?$, $c_2?$ et $type?$ et en déduit les sorties $source!$, $dest!$ et $T!$ qui formeront (au renommage près), les entrées de l'opération de connexion.

En Z, nous écrirons finalement (toujours pour notre primitive de connexion abstraite) :

$$\begin{array}{l} AbsFullConnect \mathrel{\widehat=} (ConnectComponentPart \gg IncompleteConnectPart)[source!/source?, dest!/dest?] \\ \quad \gg AddConnection \; (source', dest', source?, dest?) \end{array}$$

L'opérateur $\gg$ de Z propose une sémantique proche de celle de l'opérateur de composition $\fatsemi$ mais avec association préalable entre les sorties de l'opération à gauche de l'opérateur avec les entrées (du même nom) à droite de l'opérateur.

La «traduction» du schéma de connexion complet *AbsFullConnect* correspond à la sélection des composants considérés pour la connexion (schéma *ConnectComponentPart*) suivi du «calcul» des informations de connexion, et notamment son type effectif (schéma *IncompleteConnectPart*). Les sorties de cette première composition de schémas sont ensuite renommées pour être utilisables en entrées de *AddConnection* qui réalise la création effective de la connexion. La dernière étape consiste finalement à cacher les variables non nécessaires au schéma composé.

Sans entrer dans les détails de la méthode Z, le schéma *AbsFullConnect* aura une signature (type) équivalente au schéma suivant :

AbsFullConnectSignature

$\Delta AbstractOperationalSystem$

$id\_source? : IDENTIFIER$
$id\_dest? : IDENTIFIER$
$type? : TYPE$

$\ldots$
$conxs' = \ldots$

Ceci nous montre que l'opération finale correspondra bien à l'établissement d'une connexion à partir des identificateurs de ses composants source et destination, ainsi que de son type intentionnel.

5.5.3 Connexion par spécialisation

Dans la plupart des cas de figure, les composants seront connectés dans une perspective de *spécialisation*. L'idée qui sous-tend ce concept est qu'en règle générale, nous voulons décrire les connexions à un niveau d'abstraction plus élevé que la description des composants eux-mêmes. Par exemple, nous pourrions penser à deux composants opérant sur les entiers et que l'on voudrait interconnecter plus généralement pour des nombres.

En restant pour l'instant à un niveau assez formel[6], considérons les composants :

$$c_1 = \langle id \Rrightarrow Id_1, in\_type \Rrightarrow \emptyset, out\_type \Rrightarrow \{T_6\} \rangle$$

et

$$c_2 = \langle id \Rrightarrow Id_2, in\_type \Rrightarrow \{T_3\}, out\_type \Rrightarrow \emptyset \rangle$$

toujours dans notre système de type exemple ts (cf. fig. 4.3 page 66).

Si nous utilisons T_2 comme type intentionnel de connexion entre c_1 et c_2, et en remarquant que $T_2 \succeq T_3 \succeq T_6$ dans ts, le système doit inférer T_6 comme unique type effectif de connexion par l'algorithme de spécialisation. En effet, le type sortant choisi à la source doit être, par spécialisation, un sous-type au sens large du type intentionnel, ce qui est bien le cas de T_6 (seul type sortant déclaré par c_1). Ce type est bien un sous-type au sens large de l'unique type entrant déclaré à la destination : T_3. Ceci résulte en la connexion effective suivante :

$$cx = \langle source \Rrightarrow c_1, dest \Rrightarrow c_2, type \Rrightarrow \{T_6\} \rangle$$

Nous nous assurons ainsi que les événements de type T_6 (ou de tout sous-type compatible par subsomption) émis par la source peuvent circuler depuis c_1 jusque c_2 via la connexion cx (cf. figure 5.5 page ci-contre). Nous pouvons remarquer, de plus, que le type effectif T_6 est bien un sous-type du type intentionnel T_2.

6. Des exemples plus concrets sont proposés au chapitre suivant.

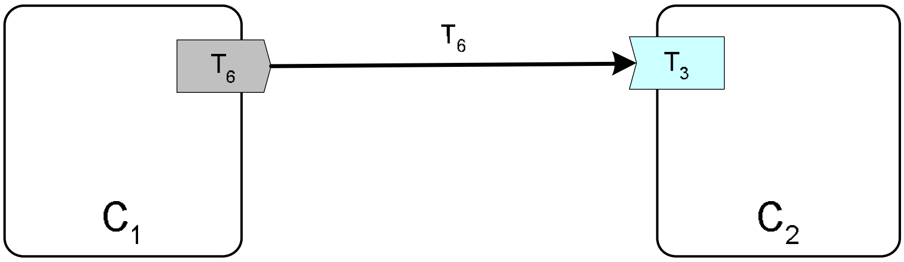


FIG. 5.5 – *Connexion par spécialisation ($T_6 \preceq T_3$)*

Ce cas simple, qui correspond donc à l'intention la plus commune, peut conduire à des conséquences plus ambiguës. Supposons par exemple que nous ajoutions à notre système *ts* un type T_7 tel que $T_3 \succeq T_7$ et $T_6 \not\succeq T_7$ (cf. figure 5.6) et que nous utilisions ce nouveau type comme autre type sortant pour c_1 (i.e. $c_1.out\_type = \{T_6, T_7\}$).

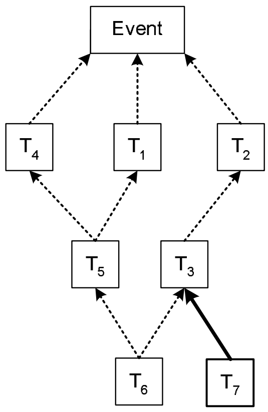


FIG. 5.6 – *Ajout d'un type T_7 dans le système de type ts*

Sous ces conditions, le type effectif inféré par spécialisation à partir de T_2 doit toujours contenir T_6 mais il faut également lui ajouter T_7 qui répond de même aux conditions de connexion (type spécialisé du type intentionnel, compatible à la source et à la destination). Cette fois-ci, la connexion s'écrira donc :

$$cx = \langle source \Rrightarrow c_1, dest \Rrightarrow c_2, type \Rrightarrow \{T_6, T_7\}\rangle$$

D'un autre côté, nous ne pouvons inférer de type effectif si nous essayons de connecter les mêmes composants pour un type trop spécialisé (par exemple, T_7 directement qui ne peut pas être spécialisé à la destination).

La propriété spécifique de la primitive de spécialisation est que les types entrant et sortant choisis à la source et à la destination sont tous sous-types du type intentionnel. Cette discussion permet d'entrer dans le cœur du problème, c'est-à-dire la définition de la primitive de connexion par spécialisation proprement dite :

$$\begin{array}{l}
\textit{ConnectSpecializePart} \\
\hline
\textit{AbstractOperationalSystem} \\[4pt]
c_1? : \textit{AbstractComponent} \\
c_2? : \textit{AbstractComponent} \\
\textit{type}? : \textit{TYPE} \\[4pt]
\textit{source}! : \textit{AbstractComponent} \\
\textit{dest}! : \textit{AbstractComponent} \\[4pt]
T? : \mathbb{P}_1\ \textit{TYPE} \\
\hline
\textit{source}! = c_1? \\
\textit{dest}! = c_2? \\[4pt]
(\mathbf{let}\ T_1 == \textit{tmax}(\{t_1 : c_1?.\textit{out_type} \mid t_1 \preceq \textit{type}?\}); \\
\qquad T_2 == \textit{tmax}(\{t_2 : c_2?.\textit{in_type} \mid t_2 \preceq \textit{type}?\}) \bullet \\
\qquad\quad T_1 \neq \emptyset \land \\
\qquad\quad T_2 \neq \emptyset \land \\
\qquad\quad T! = \{t : T_1 \mid (\exists\, t' : T_2 \bullet t \preceq t')\})
\end{array}$$

Comme vu précédemment, ce schéma sert à «transformer» les arguments de l'opération de connexion. Pour ce qui concerne les composants, il s'agit simplement de transformer les entrées $c_1?$ et $c_2?$ en sorties $\textit{source}!$ et $\textit{dest}!$.

Au niveau du type de connexion, les conditions sont plus contraignantes puisqu'il s'agit de passer du type intentionnel $\textit{type}?$ (type simple) au type effectif de connexion $T!$ (ensemble de types) par synthèse.

Le procédé de calcul est cependant fort simple. Nous sélectionnons tout d'abord l'ensemble des types sortant du composant source (c_1) qui sont sous-types (au sens large) du type intentionnel de connexion $\textit{type}?$. L'utilisation de la fonction $\textit{tmax}$ permet de ne conserver dans cet ensemble que les sous-types maximaux (i.e. qui n'ont pas eux-mêmes de sous-types dans l'ensemble). Informellement, ceci nous construit, dans T_1, l'ensemble des «meilleurs» candidats pour l'émission d'événements du type intentionnel au niveau de la source. Il est évident que l'algorithme employé est linéaire en fonction du nombre de types entrant déclarés au niveau de la source[7].

Du côté de la destination, nous procédons de même par la «création» de l'ensemble T_2 des sous-types entrant maximaux de $\textit{type}?$ dans c_2.

Il ne nous reste plus qu'à «fusionner» ces deux ensembles pour générer notre type effectif de connexion $T!$. Pour se faire, nous sélectionnons dans T_1 (types sortant candidats) tous les types qui peuvent être reçus à la destination, c'est-à-dire ceux qui ont au moins un sous-type dans les «candidats» entrant à la destination (membres de T_2).

Avant de détailler la primitive de connexion suivante, résumons le schéma de connexion par spécialisation :

$$\begin{array}{l}
\textit{AbsConnectSpecialize} \mathrel{\widehat{=}} (\textit{ConnectComponentPart} \gg \textit{ConnectSpecializePart})[\textit{source}!/\textit{source}?, \textit{dest}!/\textit{dest}?] \\
\quad \gg \textit{AddConnection} \setminus (\textit{source}', \textit{dest}', \textit{source}?, \textit{dest}?)
\end{array}$$

✎ Preuve **A.2.4 page 233** : *Preuve de cohérence de l'opération AbsConnectSpecialize*

5.5.4 Connexion par généralisation à la source

Si la spécialisation correspond à l'approche la plus fréquente pour connecter des composants entre eux, elle impose que les types choisis soient tous sous-types du type intentionnel de connexion. Dans certains cas, nous voudrions connecter un composant générique comme source d'événement pour des composants destinations plus spécialisés.

Nous donnerons de nombreux exemples de telles constructions dans la suite de notre discussion. De premiers exemples concrets sont proposés au chapitre suivant. Un autre exemple important concerne la mise en œuvre du routage implicite dans le cadre d'un système de type *publish/subscribe*, au chapitre 7.

Nous pouvons interpréter ce cas de figure à un niveau plus formel comme l'intention d'utiliser un type sortant à la source en tant que sélecteur (ou *dispatcher*) pour différents sous-types entrant dans diverses destinations.

7. Il s'agit donc d'un algorithme très efficace puisque le nombre de types déclarés au niveau d'un composant sera toujours très raisonnable - de l'ordre de la dizaine au maximum - pour tous nos exemples.

Considérons les trois composants suivants (toujours dans le système de type exemple figuré en page 79) :

$$c_1 = \langle id \Rrightarrow id_1, in\_type \Rrightarrow \emptyset, out\_type \Rrightarrow \{Event\}\rangle$$
$$c_2 = \langle id \Rrightarrow id_2, in\_type \Rrightarrow \{T_1\}, out\_type \Rrightarrow \emptyset\rangle$$
$$c_3 = \langle id \Rrightarrow id_3, in\_type \Rrightarrow \{T_2\}, out\_type \Rrightarrow \emptyset\rangle$$

en n'oubliant pas que : $Event \succeq T_1$ et $Event \succeq T_2$.

Nous voudrions, dans une optique de généralisation à la source, connecter c_1 à c_2 pour le type intentionnel T_1 et c_1 à c_3 pour T_2, comme décrit en figure 5.7. Avec l'approche par spécialisation, ceci conduirait à l'échec puisque *Event* est à la fois super-type de T_1 et T_2 et ne peut donc pas être considéré comme candidat pour la spécialisation à partir des types intentionnels proposés.

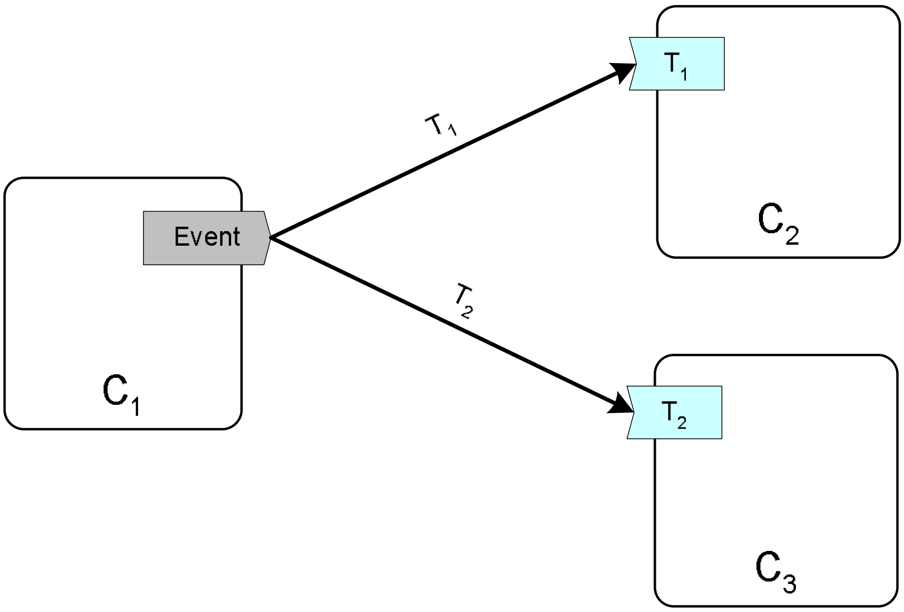


FIG. 5.7 – *Connexion par généralisation à la source : c_1 est* type dispatcher *pour le type Event*

Dans ce type de configuration où un composant (ici c_1) est considéré comme sélecteur de type (configuration *type dispatcher*), il nous faut considérer une nouvelle primitive de connexion. Cette primitive doit nous permettre, dans notre exemple, d'établir - à partir des types intentionnels respectifs T_1 et T_2 - les connexions :

$$cx_1 = \langle source \Rrightarrow c_1, dest \Rrightarrow c_2, type \Rrightarrow \{T_1\}\rangle$$
$$cx_2 = \langle source \Rrightarrow c_1, dest \Rrightarrow c_3, type \Rrightarrow \{T_2\}\rangle$$

La propriété spécifique de cette variante sera que des super-types pourront désormais être considérés à la source, d'où le schéma suivant :

$$
\begin{array}{|l}
\hline
\textit{ConnectGenSourcePart} \\
\textit{AbstractOperationalSystem} \\
c_1? : \textit{AbstractComponent} \\
c_2? : \textit{AbstractComponent} \\
type? : \textit{TYPE} \\
source! : \textit{AbstractComponent} \\
dest! : \textit{AbstractComponent} \\
T? : \mathbb{P}_1\ \textit{TYPE} \\
\hline
source! = c_1? \\
dest! = c_2? \\
(\textbf{let}\ T_1 == tmax(\{t_1 : c_1?.out\_type \mid t_1 \preceq type?\}) \cup tmin(\{t_1 : c_1?.out\_type \mid t_1 \succ type?\}); \\
\quad T_2 == tmax(\{t_2 : c_2?.in\_type \mid t_2 \preceq type?\}) \bullet \\
\qquad T_1 \neq \emptyset \wedge \\
\qquad T_2 \neq \emptyset \wedge \\
\qquad T! = \{t : T_2 \mid (\exists\, t' : T_1 \bullet t \preceq t')\}) \\
\hline
\end{array}
$$

La principale différence avec le schéma de connexion *ConnectSpecializePart* vu précédemment réside dans la construction de l'ensemble T_1 qui accepte désormais pour *type*?, en plus de ses sous-types sortant maximaux, ses super-types sortant minimaux au niveau du composant source.

Ensuite, sont sélectionnés les types de T_2 qui ont un sous-type dans T_1 pour former le type effectif de connexion résultant : $T!$.

Nous pouvons maintenant définir la primitive complète de connexion par généralisation à la source :

$$
\begin{aligned}
&AbsConnectGenSource \mathrel{\widehat=} (ConnectComponentPart \gg ConnectGenSourcePart)[source!/source?, dest!/dest?] \\
&\quad \gg AddConnection \setminus (source', dest', source?, dest?)
\end{aligned}
$$

✎Preuve **A.2.5 page 233** : *Preuve de cohérence de l'opération AbsConnectGenSource*

5.5.5 Connexion par généralisation à la destination

La catégorie de connexion en quelque sorte complémentaire à la précédente concerne la *généralisation à la destination*. Dans ce cas de figure, nous considérons une destination «générique» qui autorise, avec perte possible d'information, des connexions pour des types intentionnels plus spécialisés.

Par exemple, supposons que nous désirions connecter les composants :

$$c_1 = \langle id \Rrightarrow Id_1, in\_type \Rrightarrow \emptyset, out\_type \Rrightarrow \{T_1\}\rangle$$

et

$$c_2 = \langle id \Rrightarrow Id_2, in\_type \Rrightarrow \emptyset, out\_type \Rrightarrow \{T_2\}\rangle$$

à un composant destination c_3 défini comme suit :

$$c_3 = \langle id \Rrightarrow Id_3, in\_type \Rrightarrow \{Event\}, out\_type \Rrightarrow \emptyset\rangle$$

En utilisant les types intentionnels T_1 et T_2 (rappel : $Event \succeq T_1$ et $Event \succeq T_2$), nous voudrions obtenir :

$$cx_1 = \langle source \Rrightarrow c_1, dest \Rrightarrow c_3, type \Rrightarrow \{T_1\}\rangle$$

et

$$cx_2 = \langle source \Rrightarrow c_2, dest \Rrightarrow c_3, type \Rrightarrow \{T_2\}\rangle$$

Nous appelons la destination dans une telle configuration, un «concentrateur de type» (ou *type-concentrator*) comme indiqué en figure 5.8 page suivante.

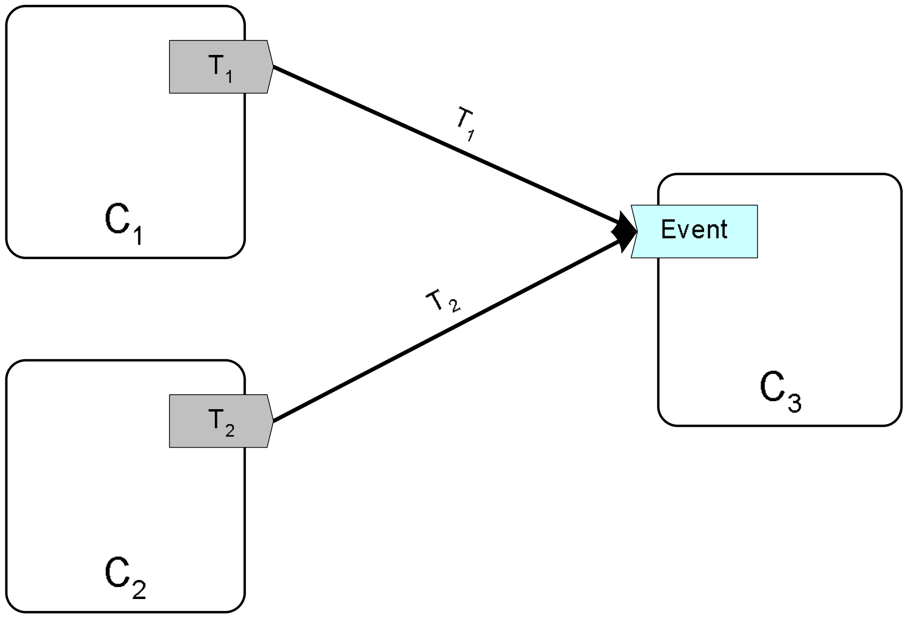


FIG. 5.8 – *Connexion par généralisation à la destination : c_3 est* type concentrator *pour le type Event*

Pour obtenir cette configuration à partir des types intentionnels T_1 et T_2, il nous faut introduire une nouvelle opération spécifique de connexion :

ConnectGenDestPart

$$
\begin{array}{l}
AbstractOperationalSystem \\
c_1? : AbstractComponent \\
c_2? : AbstractComponent \\
type? : TYPE \\
source! : AbstractComponent \\
dest! : AbstractComponent \\
T? : \mathbb{P}_1\ TYPE \\
\hline
source! = c_1? \\
dest! = c_2? \\
(\mathbf{let}\ T_1 == tmax(\{t_1 : c_1?.out\_type \mid t_1 \preceq type?\}); \\
\quad T_2 == tmax(\{t_2 : c_2?.in\_type \mid t_2 \preceq type?\}) \cup tmin(\{t_2 : c_2?.in\_type \mid t_2 \succ type?\}) \bullet \\
\qquad T_1 \neq \emptyset \land \\
\qquad T_2 \neq \emptyset \land \\
\qquad T! = \{t : T_1 \mid (\exists t' : T_2 \bullet t \preceq t')\})
\end{array}
$$

Cette fois-ci, nous «ouvrons» la sélection des types effectifs aux super-types du type intentionnel à la destination.

Le schéma global de l'opération est le suivant :

$$
\begin{array}{l}
AbsConnectGenDest \mathrel{\widehat=} (ConnectComponentPart \gg ConnectGenDestPart)[source!/source?, dest!/dest?] \\
\quad \gg AddConnection \setminus (source', dest', source?, dest?)
\end{array}
$$

✎Preuve **A.2.6 page 234** : *Preuve de cohérence de l'opération AbsConnectGenDest*

5.5.6 Retrait de connexion

Etant donnée la séparation conceptuelle entre type intentionnel et types effectifs inférés par les primitives de connexion définies dans les sections précédentes, il n'est pas simple de designer une connexion unique.

Les informations qu'il faudrait communiquer au système devraient en effet être composées des références aux composants source et destination concernés ainsi que l'ensemble des types effectifs établis pour la connexion.

Pour simplifier la primitive de retrait, nous la définirons selon le même principe consistant à discriminer intention et opérationnalisation du retrait. Ainsi, nous ne diffuserons qu'une seule information concernant le type du retrait, sous la forme d'une entrée *type*? et le retrait signifiera alors que nous souhaitons déconnecter les composants pour qu'ils ne puissent plus effectivement s'échanger d'événements de ce type ou de tout sous-type. La conséquence principale est qu'un unique «appel» à la primitive pourra conduire au retrait de plusieurs connexions.

Le schéma Z correspondant au retrait de connexion est cependant relativement simple et peut donc être défini directement sur le schéma d'état global *AbstractOperationalSystem* de la façon suivante :

$$
\begin{array}{|l}
\hline
\textit{DisconnectPart} \\
\Delta \textit{AbstractOperationalSystem} \\
c_1? : \textit{AbstractComponent} \\
c_2? : \textit{AbstractComponent} \\
\textit{type}? : \textit{TYPE} \\
\hline
\textit{types}' = \textit{types} \\
\textit{super}' = \textit{super} \\
\textit{comps}' = \textit{comps} \\
(\textbf{let}\ \textit{remove} == \{cx : \textit{conxs} \mid (\forall\, t : cx.\textit{type} \bullet \textit{type}? \succeq t) \wedge cx.\textit{source} = c_1? \wedge cx.\textit{dest} = c_2?\} \bullet \\
\qquad \textit{remove} \neq \emptyset \wedge \\
\qquad \textit{conxs}' = \textit{conxs} \setminus \textit{remove}\ ) \\
\hline
\end{array}
$$

Dans un premier temps, nous plaçons dans la variable locale *remove* l'ensemble des connexions répondant aux critères de retrait sur leurs constituants :

- Le composant source référencé doit être valide ($c_1? \in \textit{comps}$),
- le composant destination également ($c_2? \in \textit{comps}$), et
- le type intentionnel de déconnexion doit être un super-type de tous les types effectifs de connexion ($\forall\, t : cx.\textit{type} \bullet \textit{type}? \succeq t$).

Une fois «calculé», cet ensemble est retiré des connexions effectives avant changement d'état *conxs* pour obtenir les connexions après transition $\textit{conxs}'$. Nous ajoutons enfin une condition qui vérifie que cet ensemble de connexions effectives à ôter du système est non vide. Ceci marque, en quelque sorte, la nécessité de produire un message d'erreur si aucune connexion effective ne correspond à la demande.

Le retrait de connexion est trivialement une opération cohérente vis à vis de l'invariant principal du système (inv *AbstractOperationalSystem*) car ce dernier ne définit aucune condition entre les connexions (elles sont indépendantes). De plus, toute connexion effective initialement présente satisfait déjà cet invariant de sorte que le retrait n'a aucun impact (contrairement à l'ajout).

Par souci «d'interface», nous ajoutons comme de coutume un schéma représentant la primitive de déconnexion vue de «l'extérieur», c'est-à-dire se basant sur des identificateurs plutôt que les composants eux-mêmes :

$$\textit{Disconnect} \mathrel{\widehat{=}} \textit{ConnectComponentPart} \gg \textit{DisconnectPart} \setminus (c_1?, c_2?)$$

Ce dernier schéma clôt la définition des primitives liées aux connexions/déconnexions.

5.6 Opérations composées

À partir des primitives définies dans les deux sections précédentes, nous pouvons construire des opérations composées en les combinant et en ajoutant des contraintes sur ces combinaisons si nécessaire.

En règle générale, les opérations que nous allons définir n'ajouteront pas de conditions ou «d'effets de bord» sur l'état global du système, ce qui permet de limiter (voir, dans notre cas, d'éliminer) les obligations de preuve.

5.6.1 Connexions généralisées

La première opération que nous allons détailler est très importante puisqu'il s'agit de la principale opération de connexion proposée dans notre modèle. Nous avons pour l'instant proposé trois catégories différentes de connexion : par spécialisation, par généralisation à la source ou par généralisation à la destination. Cette décomposition était dans le même temps motivée par une identification à trois intentions discriminantes lors de la composition d'architectures (composant «standard», *type dispatcher* et *type concentrator*) et également du fait de leur complémentarité formelle dans le modèle que nous proposons.

D'un point de vue plus pragmatique, ce choix entre trois types de connexion semble alourdir quelque peu le modèle proposé. Heureusement, un choix adéquat d'une catégorie de connexion peut généralement être inféré à partir d'une simple analyse des types entrant et sortant des composants concernés.

Dans le cas général, il peut exister des ambiguïtés concernant ce choix mais, à force d'expérimentations, nous avons pu confirmer l'intuition que dans la plupart des cas, un composant donné «oriente» son interface (couple type entrant/type sortant) vers un rôle bien distinct au sens des primitives de connexion. En d'autres termes, il est rare qu'un composant soit créé, pour une même branche de la hiérarchie des types, dans l'optique de servir à la fois en tant que composant spécialisé, *type dispatcher* ou *type concentrator*. De plus, nous pouvons remarquer que la spécialisation est le cas le plus fréquent et devra donc être traité en priorité. En revanche, le choix entre la généralisation à la source et à la destination semble quelque peu arbitraire mais la discrimination de type sortant nous apparaît comme relativement plus fréquent d'où notre choix de favoriser la généralisation à la source.

En résumé, la primitive de connexion généralisée tentera, en premier lieu et si les préconditions associées sont satisfaites, de connecter par spécialisation. Sinon, des connexions par généralisation respective à la source puis à la destination seront considérées.

Ainsi, cette simple heuristique nous permet de proposer le schéma suivant pour la primitive de connexion généralisée :

$$\begin{aligned}&\mathit{Connect} \mathrel{\widehat{=}} \textbf{if } \text{pre } \mathit{ConnectSpecialize} \textbf{ then } \mathit{ConnectSpecialize}\\ &\quad\quad \textbf{else if } \text{pre } \mathit{ConnectGenSource} \textbf{ then } \mathit{ConnectGenSource}\\ &\quad\quad \textbf{else } \mathit{ConnectGenDest}\end{aligned}$$

Nous pouvons facilement vérifier que si :

$$\neg\,(\text{pre } \mathit{ConnectSpecialize}) \wedge \neg\,(\text{pre } \mathit{ConnectGenSource}) \wedge \neg\,(\text{pre } \mathit{ConnectGenDest})$$

alors la connexion généralisée échoue.

Il est intéressant de noter que sur nos trois exemples (figures 5.5 page 79, 5.7 page 81 et 5.8 page 83), la primitive proposée «fonctionne» correctement. De fait, sur tous les exemples proposés dans cette thèse, la primitive de connexion généralisée est adéquat et, dans nos investigations, seuls des exemples «théoriques» l'ont mise en doute (et dans ce cas, on peut se rabattre sur les opérations primitives).

5.6.2 Connexions inférées et déconnexions globales

En employant le type le plus général dans la hiérarchie des types - *Event* - nous pouvons rendre l'opération de connexion indépendante de toute intention spécifique. Même si de notre point de vue, une information importante est par la même occasion écartée[8] et que cela peut conduire à des ambiguïtés en pratique (et pas qu'en théorie), cette variante se révélera très intéressante pour tout ce qui concerne la «découverte» de connexions possibles entre deux composants donnés.

Pour la définir, nous devons simplement, au moment de la connexion (généralisée), associer le type intentionnel *type*? avec *Event*, ce qui nous donne :

$$\mathit{ConnectInfer} \mathrel{\widehat{=}} \mathit{Connect} \wedge (\mathit{type}? = \mathit{Event})$$

Ceci réalise une connexion unique, intuitivement la plus générale possible entre les deux composants concernés.

8. La «philosophie» de notre approche est que pour obtenir une composition «saine», il faut se concentrer sur le couplage lui-même plutôt que sur les caractéristiques des composants que l'on veut interconnecter.

Toute connexion entre deux composants possède des types effectifs sous-type du type le plus général *Event*. Cette propriété est évidente puisque tous les types du système sont forcément sous-types de *Event*. Ainsi, pour déconnecter deux composants «complètement», il suffit de les déconnecter «simplement» pour le type le plus général.

En suivant le même principe que pour les connexions inférées, nous pouvons donc construire une opération de déconnexion généralisée de la façon suivante :

$$DisconnectAll \mathrel{\widehat=} Disconnect \land (type? = Event)$$

5.6.3 Retrait généralisé de composant

La primitive de retrait de composant présuppose l'absence de connexion effective faisant référence à ce composant. En d'autres termes, il ne doit être connecté à aucun autre composant avant de pouvoir être retiré du système. Pour retirer un composant, indépendamment de son graphe d'interconnexion, il nous suffit donc dans un premier temps de le déconnecter complètement comme source, ensuite de le déconnecter complètement comme destination, puis de le retirer effectivement.

Pour traduire ceci en Z, nous devons tout d'abord définir un schéma qui sélectionne le composant dans le système :

```
── RemoveSelectComponentPart ──────────────
│ AbstractOperationalSystem
│ rid? : IDENTIFIER
├──────────
│ ∃₁ c : comps | c.id = rid?
└───────────────────────────────────────────
```

Ici, *rid?* correspond à l'identificateur du composant à «enlever». Nous pouvons ensuite construire simplement le schéma final pour l'opération de retrait généralisée :

$$RemoveComponentGen \mathrel{\widehat=} RemoveSelectComponentPart \land (DisconnectAll \land (id_s ource? = rid?) \fatsemi$$
$$DisconnectAll \land (id_d est? = rid?) \fatsemi RemoveComponent \land (Id? = rid?)$$

Ce schéma sélectionne donc dans un premier temps le composant à retirer, c'est à dire qu'il crée une variable *rid?* et vérifie qu'elle correspond bien à un composant existant dans le système. Ensuite, Cet identificateur est associé au composant source pour une déconnexion totale, puis l'opération est renouvelée en considérant *rid?* comme identificateur de destination. Finalement, nous pouvons appliquer la primitive de retrait de composant puisque nous savons qu'aucune connexion ne peut plus référencer le composant considéré.

« [Le langage] C est imprédictible, défectueux et un succès énorme.»

Dennis Ritchie

«On parle tant du langage alors qu'il n'y personne pour entendre.»

William Carlos Williams

6
Le langage Scope

Le modèle structurel présenté dans les sections précédentes fournit une sémantique pour la composition dynamique de composants logiciels. Dans cette section, nous décrivons un *langage spécifique* [vDKV00] implantant ce modèle.

Le langage Scope établit une distinction précise entre les aspects structurels, qu'il prend en charge directement, et les aspects comportementaux (ou non structurels [1]), gérés pour leur part dans un environnement informatique sous-jacent. Par exemple, la structure des composants sera décrite par des constructions spécifiques alors que l'implantation de chaque fonctionnalité associée se fera dans le langage sous-jacent.

Au cours de ce travail, nous avons construit deux versions du langage Scope, l'une basée sur Scheme et l'autre sur notre plateforme Comet, elle-même implémentée en Java. Nous ne décrivons ici que la deuxième implantation, la version Scheme étant décrite de son côté dans [PQB02]. Il est important de signaler que du point de vue structurel, les différences entre ces deux versions sont de nature purement syntaxique.

6.1 Constructions syntaxiques

Le langage Scope considère comme entités de premier niveau les concepts introduits par le modèle sous-jacent : types d'événements, composants, connexions, etc. Ce seront les *abstractions structurantes* du langage [2]. D'autre part, chaque primitive et opération décrite dans le modèle Scope est proposée au niveau de l'environnement sous-jacent, sous la forme par exemple de fonctions en Scheme ou de classes/méthodes en Comet/Java.

Il existe donc une séparation forte entre *spécification* et *utilisation* des concepts du modèle. Par exemple, spécifier un composant Scope et l'utiliser se font dans deux environnements séparés. La spécification est établie au niveau du langage Scope mais en utilisant les propriétés du langage sous-jacent (ex. : choix de la structure du composant et implantation de ses méthodes). L'utilisation correspond, elle, à l'invocation de «services» du langage Scope depuis l'environnement sous-jacent (ex. : instantiation du composant, connexion à d'autres composants, etc.).

Les constructions syntaxiques introduites par le langage sont donc séparées en deux ensembles : les *définitions* (au niveau du langage Scope) et les *commandes* (au niveau de l'environnement sous-jacent).

Une définition Scope est composée d'une séquence de déclarations qui sont analysées et transformées pour l'environnement sous-jacent par le *traducteur Scope*. Nous décrivons ci-dessous la syntaxe globale (dans une version quelque peu étendue de la notation *Backus-Naur*) de ces définitions :

1. Tout ce qui sort du cadre du modèle Scope sera considéré comme aspect comportemental.
2. Par analogie, les classes, interfaces et méthodes seront considérées comme les abstractions structurantes du langage Java.

<definition>	→	*<construct>+*
<construct>	→	*<event>*
	|	*<component>*
	|	*<java-global>*

Ainsi, il est possible de définir en séquence des catégories d'événements et de composants. Le non-terminal *<java-global>* correspond à une déclaration de l'environnement sous-jacent (par exemple, une classe java ou un import, etc.).

6.2 Gestion des événements

La primitive event permet d'ajouter un nouveau type d'événement au système. L'interprétation d'une telle construction correspond à l'opérationnalisation de la primitive *AddType* du modèle formel.

En plus de la donnée du type et de ses relations avec les autres types (liste des super-types), le langage Scope autorise la définition de variables (nommées *slots*) et de méthodes sur les événements. Contrairement aux classes des langages objet, ces méthodes n'implantent aucun *comportement* mais uniquement des fonctionnalités *passives* d'accès aux variables.

Le système lui-même définit le type Event comme racine de la hiérarchie des types. Les événements de ce type (i.e. tous les événements) peuvent par exemple tous lire et écrire un *slot* sender_id référençant l'émetteur de l'événement.

Voici en guise d'illustration la définition d'un type RealEvent dénotant l'ensemble des événements de type flottant dans notre langage :

```
event RealEvent is NumberEvent {
  slot _value type float;

  RealEvent() { _value = 0.0; }
  RealEvent(float init_val) { _value = init_val; }

  public float getValue() { return _value; }
  public void setValue(float new_val) { _value = new_val; }

}
```

La déclaration ci-dessus définit un sous-type de NumberEvent comme l'association d'un *slot* public flottant de nom _value (de type sous-jacent float[3]), de constructeurs et de méthodes d'accès en lecture/écriture à la variable _value. Par convention, les noms de types d'événement sont suffixés par Event.

La syntaxe complète de la primitive Event est la suivante :

<event>	→	`event` *<type-name>* *<supertype>** `{` *<event-part>** `}`
<type-name>	→	*<java-ident>*[4]
<supertype>	→	`is` *<type-name>*
<event-part>	→	*<slot>*
		| *<java-part>*
<slot>	→	`slot` *<slot-name>* *<slot-type>;*
<slot-name>	→	*<java-ident>*
<slot-type>	→	`type` *<scope-type-name>*

Dans cette grammaire, le non-terminal *<scope-type-name>* doit être remplacé par la définition d'une déclaration de type pour le *slot* considéré. Ce type peut être un type de l'environnement sous-jacent ou un type défini par le traducteur Scope, dont une liste non-exhaustive est proposée en table 6.1 page ci-contre.

3. Les types sous-jacents sont ceux de Java dans notre exemple : type de classe ou type de base.
4. *<java-ident>* correspond aux règles d'écriture d'un identificateur dans l'environnement sous-jacent. De façon plus générale, les non-terminaux commençant par le préfixe `java` ne seront pas analysés par le traducteur (au niveau syntaxique).

Nom du type Scope	**Equivalent Java**	**Définition**
boolean	boolean	booléens
short	int	entiers courts
int	int	entiers simples
long	long	entiers longs
float	float	réels simple précision
double	double	réels double précision
String	String	chaînes de caractères
Object	Object	type java standard
Array[type]	type[]	type tableau simple
DynArray[type]	ArrayList	type tableau dynamique

TAB. 6.1 – *Types de* slot *disponibles (extrait)*

Au niveau de l'environnement sous-jacent, les événements correspondent à des objets java dont la classe porte le nom du type déclaré. Le code Java suivant montre quelques interactions possibles avec des événements du type RealEvent[5] :

```
event = new RealEvent(3.14159265);
event.getValue();
==> 3.14159265
event.setValue(0.69314718);
==> 0.69314718
event.getTypeName();
==> "RealEvent"
event.getSupertypeNames();
==> [ "NumberEvent" ]
```

Les méthodes standard getTypeName() et getSupertypeNames() retournent respectivement le nom du type de l'événement et la liste des noms de ses super-types. La table 6.2 recense l'ensemble des méthodes standard proposées pour chaque type d'événement (définies par la class Event).

Méthode	**Définition**
String getSenderId()	Nom de l'émetteur de l'événement
String getTypeName()	Nom du type de l'événement
String[] getSupertypeNames()	Noms des super-types
Class[] getSupertypes()	Classes Java des super-types

TAB. 6.2 – *Méthodes canoniques des événements Scope*

L'identification du type de l'événement à une classe Java pose des problèmes concernant le sous-typage. En Scope, héritage et sous-typage sont distincts, de sorte que dans notre exemple, la classe RealEvent n'hérite pas forcément de NumberEvent mais peut-être d'une classe construite pour l'occasion. Nous reviendrons sur ce problème en section 6.6 page 97.

6.3 Gestion des composants

Tout comme pour les événements, nous séparons en Scope la description des composants, abstractions décrites dans le langage dédié, et leurs instances concrètes, objets «physiques» manipulés dans l'environnement sous-jacent. Dans notre terminologie, un *composant* sera assimilé à une instance et une *définition de composant* correspondra à sa description préliminaire.

En complément du modèle formel axé sur la structure des composants, le langage Scope leur associe des propriétés supplémentaires :

5. Les exemples d'interactions présentés dans cette discussion correspondent, à quelques «arrangements syntaxiques» près, à des exécutions de scripts dans l'environnement Jython.

– Un nom de définition (ou nom du type du composant),
– un ensemble de fonctionnalités, prenant en charge le traitement des événements reçus,
– des attributs et méthodes «utilitaires», sous forme de code Java (ou Scheme).

Notons de plus que Scope n'autorise pas la définition de constructeurs pour les composants. Nous verrons en effet par la suite que la création des composants et leur initialisation sont deux phases clairement distinguées dans notre approche.

Pour chaque type entrant du composant, il existe au plus une fonctionnalité expliquant la réaction du composant à une telle réception d'événement[6]. S'il existe plusieurs fonctionnalités compatibles (au sens du sous-typage) avec un événement donné, alors seule la fonctionnalité la «plus» compatible (super-type le plus proche) sera déclenchée. En d'autre terme, la réception d'un événement ne peut conduire à l'exécution que d'une unique fonctionnalité.

Voici en guise d'exemple la définition d'un composant réalisant des additions de nombres réels dans un modèle de *flots de données réactifs*[7] :

```
component AddReal {
  receive RealEvent;
  send RealEvent;

  when (RealEvent event) {
    if(op1==null)
      op1 = new Float(event.getValue());
    else {
      RealEvent out_event = new RealEvent(op1.floatValue() + event.getValue());
      send(out_event);
      op1 = null;
    }
  }

  protected Float op1 = null;
}
```

Le nom de cette définition de composant est AddReal; nous en donnons une représentation graphique en figure 6.1. En utilisant les mot-clés receive et send, nous indiquons respectivement les types entrant et sortant du composant. La fonctionnalité de réception des événements de type RealEvent est définie dans une clause when du langage Scope. L'algorithme proposé donne au composant un caractère *réactif*. Pour un opérateur d'addition binaire, deux événements sont attendus. Le premier voit sa valeur (de type Java float) stockée dans la variable op1 et le deuxième est ajouté au premier avant émission du résultat de l'addition via l'invocation de la primitive send. Nous pouvons remarquer à ce propos l'absence de désignation du destinataire de l'envoi, ce qui - nous le rappelons - est un critère déterminant pour la composition dynamique (extraction de la relation de couplage).

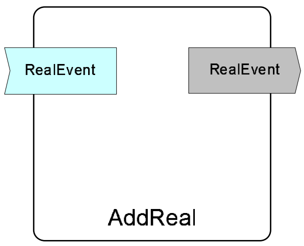


FIG. 6.1 – *Représentation graphique d'un composant d'addition binaire*

Les *générateurs de constantes* représentent une autre catégorie importante de composant dans le cadre des architectures à flots de données. Leur propos est de générer une valeur «constante» en réponse à une activation. La constante sera modifiée par initialisation ou réinitialisation du composant. Ces activations et initialisations

6. Si aucune fonctionnalité n'est déclarée, alors l'événement est rejeté silencieusement ou *dispatché* par héritage.
7. Tous les exemples de cette section s'inspirent du framework Fabrik [IWC+88] et se placent donc dans le domaine de la programmation «visuelle».

mettent en jeu des événements de types standard, ActivateEvent et InitializeEvent mais supportés par des constructions spécifiques[8].

Nous définissons ci-dessous un générateur de constantes réelles :

```
component GenReal {
  send RealEvent;
  send ActivateEvent;

  initialize (InitializeEvent init) {
    constant = (float) init.getContents();
  }

  activate (ActivateEvent event) {
    RealEvent out_event = new RealEvent(constant);
    send(out_event);
    send(event); // propager l'activation
  }

  protected float constant;
}
```

Quand un événement GenReal reçoit un événement de type InitializeEvent, il décode une information spécifique, ici un flottant Java, utilisée pour fixer la valeur de la constante à générer. Si l'événement est de type ActivateEvent, alors le bloc activate est déclenché. Dans notre exemple, nous émettons un événement contenant la constante réelle et nous propageons l'activation en réémettant l'événement «activateur».

La syntaxe proposée pour la définition des composants Scope est la suivante :

```
<component>          →  component <type-name> <java-type> {
                           <component-type>*
                           <component-part>*
                         }
<component-type>     →  <input-type>
                         | <output-type>
<input-type>         →  receive <event-type> ;
<output-type>        →  send <event-type> ;
<component-part>     →  <scope-part>
                         | <java-part>
<scope-part>         →  <initialize-clause>? <activate-clause>? <when-clause>*
<when-clause>        →  when <when-label>? ( <type-name> <ident> ) { <java-code> }
<activate-clause>    →  activate ( <type-name> <ident> ) { <java-code> }
<initialize-clause>  →  initialize ( <type-name> <ident> ) { <java-code> }
```

Signalons qu'il s'agit d'une syntaxe simplifiée car le traducteur sera en fait moins contraignant concernant l'agencement des différentes clauses[9]. Cependant, dans tous nos exemples, la grammaire donnée ci-dessus sera respectée. Notons également la possibilité de déclarer qu'un composant étend une classe (non-terminal *java-type*). Il sera par exemple possible de «sous-classer» une définition de composant, avec une sémantique tout à fait comparable à l'héritage[10].

Concernant l'utilisation des composants, notons l'adjonction de commandes spécifiques, présentes dans l'environnement sous-jacent, et permettant de :

- instancier des composants à partir de leur définition (commande instantiate),
- initialiser un composant en lui envoyant un événement d'initialisation (initialize),
- activer un composant par l'émission d'un événement de type ActivateEvent (activate),
- connecter/déconnecter les composants (cf. section suivante),
- retirer un composant du système (commande delete).

8. Il n'est donc pas nécessaire de déclarer la réception de tels événements (leur envoi, en revanche, doit être déclaré).
9. Par exemple, les clauses de type peuvent être placées n'importe où dans la définition du composant et non obligatoirement avant les autres sections.
10. La version Scheme du langage Scope autorise même l'héritage multiple pour les définitions de composants.

Par exemple, nous pourrons interagir avec un composant GenReal de la façon suivante :

```
gen_id = instantiate("GenReal");
initialize(gen_id, new InitializeEvent( new Float(10)));
activate(gen_id, new ActivateEvent());
==> Warning : Cannot emit event of type 'RealEvent' : no destination found
```

L'activation du composant est «sanctionnée» par un message d'avertissement puisque l'envoi de la constante ne trouve aucun receveur.

La table 6.3 page 96 référence l'ensemble des commandes disponibles pour la gestion des composants au moment de l'exécution.

6.4 Opérations de connexion

Dans le but de décrire les commandes correspondant aux opérations formelles de connexion (cf. section 5 page 69), nous allons construire un exemple très simple de flot de données : un convertisseur de degrés Fahrenheit en degrés Celsius.

La formule bien connue pour convertir une température $temp_F$ (degrés Fahrenheit) en une température $temp_C$ (degrés Celsius) est la suivante :

$$temp_C = (temp_F + -32) \times \frac{5}{9}$$

Par souci de simplification, nous effectuerons des calculs génériques sur le type RealEvent défini précédemment. De plus, nous utilisons pour l'instant uniquement des opérateurs commutatifs (comme + ou ×), nous aborderons le cas des opérateurs non-commutatifs en section 6.5 page 96. Les types utilisés dans notre exemple ainsi que leur relations au sens du sous-typage sont représentés en figure 6.2.

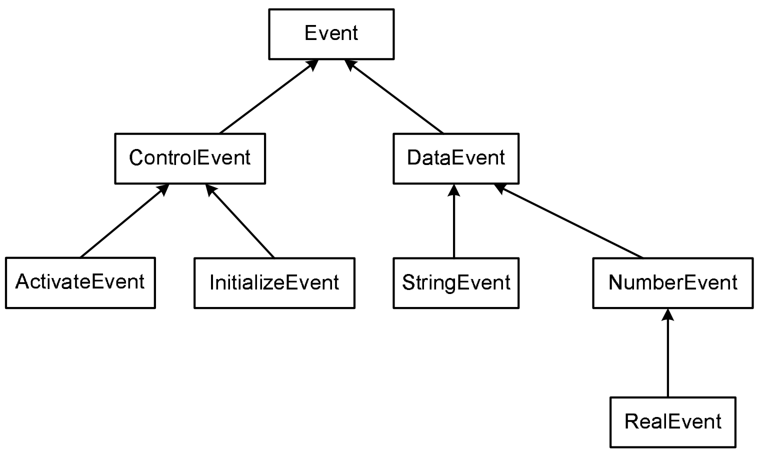


FIG. 6.2 – *Système de type minimal pour les architecture à flots de données*

Pour composer un flot de données réalisant cette opération élémentaire, nous aurons besoin d'un additionneur AddReal et d'un multiplicateur MultReal (dont la définition est pratiquement similaire), ainsi que deux générateurs de constantes de type GenReal.

Le script suivant décrit cette architecture, représentée en figure 6.3 page ci-contre :

```
// instanciations des composants
add_id = instantiate("AddReal");
mul_id = instantiate("MulReal");
const1_id = instantiate("GenReal");
const2_id = instantiate("GenReal");


// initialisations
initialize(const1_id, new InitializeEvent( new Float(-32.0)));
initialize(const2_id, new InitializeEvent( new Float(5.0/9.0)));
```

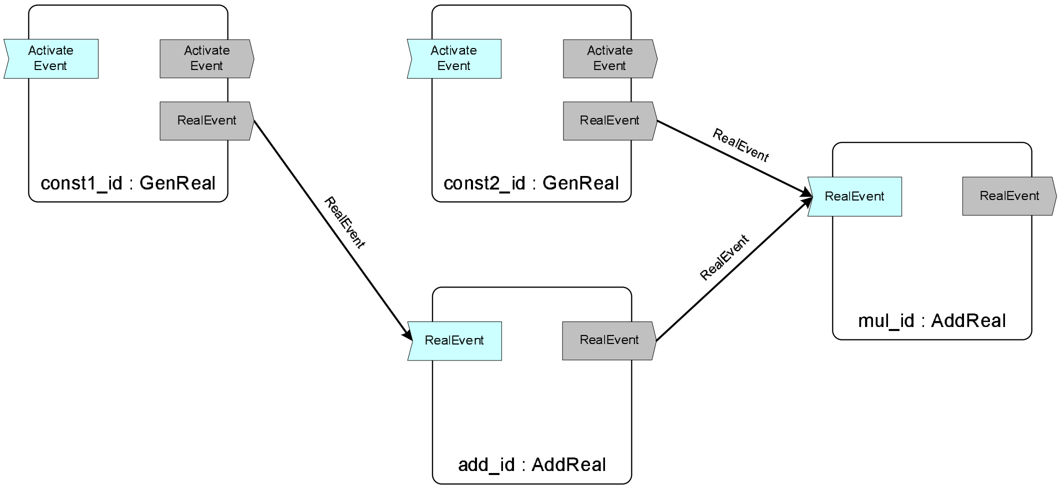


FIG. 6.3 – *Architecture partielle d'un convertisseur de température*

Pour démarrer notre composition, commençons par connecter le composant générateur identifié par const1_id à l'additionneur add_id. Nous pouvons remarquer, d'après la figure 6.2 page précédente, que les réels sont des sous-types de NumberEvent, nous pouvons donc utiliser cette information comme type intentionnel de connexion, dans le cadre d'une spécialisation :

```
connect_specialize(const1_id, add_id, "NumberEvent");
==> [ RealEvent ]
```

En guise d'information, l'interprète nous donne le type effectif inféré, ici le type RealEvent, qui correspond bien aux données échangées par les composants. Puisqu'il n'existe aucune ambiguïté en terme de typage, nous aurions pu simplifier ce script en nous reposant sur les commandes de connexions généralisées : connect et connect-infer. Employons-les pour connecter le deuxième générateur et l'additionneur au multiplicateur pour parachever notre architecture de base :

```
connect(const2_id, mul_id, "NumberEvent");
==> [ RealEvent ]

connect_infer(add_id, mul_id);
==> [ RealEvent ]
```

Dans les deux cas, l'unique type effectif considéré correspond bien à nos attentes (raisonnables, il est vrai). Comme le montre la figure 6.3, notre système n'est pas encore complet; Il nous faut tout d'abord répondre aux questions suivantes :

- Comment fournir la température à convertir ?
- Que faire du résultat de la conversion, envoyé par le multiplicateur sous forme d'événement ?

Pour la deuxième question, nous allons définir un composant générique d'affichage de donnée (très simplifié), de la façon suivante:

```
component DataOutput {
  receive DataEvent;

  when (DataEvent event) {
    System.out.println(event.getValueObject().toString());
  }
}
```

En cas de réception d'un événement de type DataEvent (dont tous les types de base sont sous-types), la valeur de la donnée contenue est affichée[11].

Pour agrémenter notre architecture, connectons une instance de ce composant d'affichage comme destination du multiplicateur. Comme ce dernier génère des sous-types de DataEvent, nous reconnaissons ici une situation où l'afficheur est un concentrateur de type. La commande de connexion par généralisation à la destination, connect_gen_dest peut être utilisée dans ce cadre :

```
// création de l'afficheur
out_id = instantiate("DataOutput");

// première possibilité : connexion spécifique
connect_gen_dest(mul_id, out_id, "RealEvent");
==> [ RealEvent ]

// deuxième possibilité : connexion inférée
connect(mul_id, out_id, "RealEvent");
==> [ RealEvent ]

// troisième possibilité : connexion non contextuelle
connect_infer(mul_id, out_id);
==> [ RealEvent ]
```

Bien sûr, il est encore une fois possible d'utiliser les commandes de connexion plus générales.

Finalement, réalisons un composant «symétrique» permettant la saisie et l'émission d'un réel :

```
component DataInput {
  send DataEvent;
  send ActivateEvent;

  activate (ActivateEvent event) {
    float input = Console.readFloat();
    send( new RealEvent(input));
    send( event ); // propager l'activation
  }
}
```

Lorsqu'un tel composant est activé, une saisie au clavier (sur la console Java) est effectuée et un nombre réel en est récupéré. Ce nombre est ensuite transmis sous forme d'événement de type RealEvent et le signal d'activation est propagé.

Le trait distinctif des composants de type DataInput est qu'ils correspondent à des «dispatchers» pour le type DataEvent. La connexion par généralisation à la source semble donc indiquée si nous voulons connecter un tel composant à notre additionneur :

11. La méthode getValueObject() doit être implémentée par tout sous-type de DataEvent, mais le traducteur Scope la définit automatiquement.

```
// création du composant de saisie
in_id = instantiate("DataInput");

// première possibilité : connexion spécifique
connect_gen_source(in_id, add_id, "RealEvent");
==> [ RealEvent ]

// deuxième possibilité : connexion inférée
connect(in_id, add_id, "RealEvent");
==> [ RealEvent ]

// troisième possibilité : connexion non contextuelle
connect_infer(in_id, add_id);
==> [ RealEvent ]
```

Pour terminer notre architecture, nous devons relier le composant de saisie aux deux constantes pour que toute nouvelle émission de «température» (pour l'instant du type plus générique RealEvent) corresponde à l'activation des destinations et donc à la génération des constantes :

```
connect(in_id, const1_id, "ActivateEvent");
==> [ ActivateEvent ]

connect(in_id, const2_id, "ActivateEvent");
==> [ ActivateEvent ]

connect(const1_id, const2_id, "ActivateEvent");
==> [ ActivateEvent ]
```

Notre architecture est ainsi complétée et supporte un système de conversion opérationnel, représenté en figure 6.4.

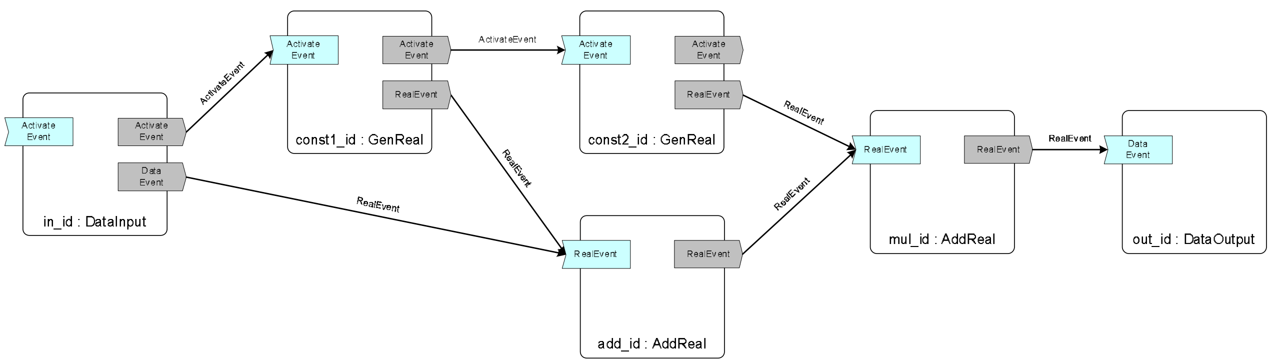


Fig. 6.4 – *Architecture complète du convertisseur de température*

Le script ci-dessous donne un exemple d'interaction d'utilisation de ce système :

```
activate(in_id, new ActivateEvent());
? 68
20
```

Bien que cet exemple soit très primitif, toute architecture de flot de données mettant en jeu des opérateurs et des entrées/sorties beaucoup plus complexes peut être réalisée selon les mêmes principes. Signalons de plus le caractère dynamique et incrémentiel de la méthode de développement sous-jacente du langage Scope et des commandes associées (résumées en table 6.3 page suivante).

Finalement, gardons à l'esprit l'omniprésence du typage et des vérifications associées, qui assurent la cohérence des échanges entre composants du point de vue de leur type.

Commande de gestion de composant	Définition
String **instantiate**(String ctype)	Création d'une instance pour le composantde type ctype
void **initialize**(String id, InitializeEvent event)	Initialisation du composant id par envoi de event
void **activate**(String id, ActivateEvent event)	Activation du composant id par envoi de event
void **delete**(String id)	Déconnexion et retrait du composant id

Commande de connexion
void **connect_specialize**(String source, String dest, String type) **Définition :** Connexion par spécialisation, type est le type intentionnel
void **connect_gen_source**(String source, String dest, String type) **Définition :** Connexion par généralisation à la source
void **connect_gen_dest**(String source, String dest, String type) **Définition :** Connexion par généralisation à la destination
void **connect**(String source, String dest, String type) **Définition :** Connexion non spécifique
void **connect_infer**(String source, String dest) **Définition :** Connexion non contextuelle

TAB. 6.3 – *Langage Scope : commandes de base*

6.5 Connexions paramétrées

Dans certaines situations, il est possible que la vision «type uniquement» favorisée par notre modèle de composition se révèle quelque peu contraignante.

Supposons par exemple que nous désirions rajouter des opérateurs de soustraction et de division à notre «boîte à outils» de composants sur flots de données. Le problème soulevé réside dans la nature non-commutative de ces opérateurs. Le fait que ces derniers puissent recevoir ou émettre des événements de type réel ne suffit plus puisqu'il nous faut un moyen de distinguer les deux opérandes de l'opération.

Une première proposition serait de supposer que deux événements ne peuvent pas arriver exactement au même moment. Ainsi, le premier événement reçu serait l'opérande de gauche. Nous verrons par la suite que cette hypothèse, dans un environnement concurrent, n'est pas raisonnable. De plus, il serait alors très difficile de comprendre le fonctionnement d'une architecture puisque le résultat dépendrait d'un paramètre temporel implicite. D'après la «philosophie» de notre modèle, une autre solution plus réaliste serait de proposer des sous-types «réel à gauche» et «réel à droite» pour distinguer les deux cas. Mais si nous encourageons effectivement dans notre approche la définition de types les plus explicites possibles pour décrire le couplage au niveau d'abstraction souhaité, il est tout aussi important de ne pas crypter la composition par une profusion de types incompréhensibles. Il est évident que de ce point de vue, la séparation entre nombres à gauche et nombres à droite semble peu raisonnable. La troisième solution - celle qui nous semble la plus adaptée - consiste à ajouter au langage Scope le nécessaire pour pouvoir, le cas échéant, «retrouver» la notion de port physique présente dans d'autres modèles de composant. Ici, nous voudrions introduire deux ports physiques «gauche» et «droite», associés tous deux au type RealEvent. Pour se faire, il suffit de «décorer» une fonctionnalité de composant avec un identificateur qui représentera le nom du port. Dans la grammaire des définitions de composant, cette possibilité est introduite par le non-terminal optionnel *<when-label>*.

Ceci nous permet de construire un composant de soustraction :

```
component SubReal {
  receive RealEvent;
  send RealEvent;

  when label left (RealEvent event) {
    op1 = new Float(event.getValue());
    if(op2!=null) // Opérande droit disponible
      do_subtraction();
  }

  when label right (RealEvent event) {
    op2 = new Float(event.getValue());
    if(op1!=null) // Opérande gauche disponible
      do_subtraction();
  }

  protected void do_subtraction() {
    RealEvent out_event = new RealEvent(op1.floatValue() - op2.floatValue());
    send(out_event);
    op1 = null;
    op2 = null;
  }

  protected Float op1 = null;
  protected Float op2 = null;
}
```

Le problème consiste désormais à indiquer, au moment de la connexion entre deux composants, quel port devra être choisi. Le concept de *connection labellée* correspond à une extension des connexions Scope standard permettant la déclaration d'un label au moment de la connexion. Pour connecter un composant source (par exemple un DataInput) au label left d'un composant de soustraction, nous pourrons employer la construction suivante :

```
connect(in_id, sub_id, "NumberEvent", "left");
```

La figure 6.5 donne une représentation graphique pour le composant SubReal. Tout se passe «visuellement» comme si nous définissions en fait deux types distincts.

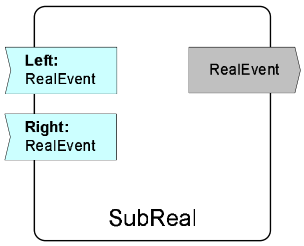


Fig. 6.5 – *Représentation graphique d'un composant à ports labellés*

S'il n'existe pas de port left pour un type compatible avec NumberEvent à la destination, alors une telle requête de connexion devra bien sûr échouer. En revanche, si le composant définit de plus un bloc when sans distinction de port, la connexion sera acceptée.

6.6 Traduction et outils

Nous traitons succinctement dans cette section du problème de l'analyse et de l'interprétation des définitions Scope. Puisque ce langage «repose» sur un environnement sous-jacent (dans notre cas, le kit de développement Java), nous nous orientons vers une solution «source à source».

Nous pouvons remarquer que la syntaxe du langage Java et les constructions spécifiques au langage Scope sont complémentaires. Les mots-clés et leur placement dans le texte source ne peuvent conduire à aucune ambiguïté concernant l'analyse des programmes sources. Ainsi, le mécanisme de transformation proposé consiste en la collection des constructions spécifiques Scope et leur traduction en Java. Le reste des programmes sera vérifié par le compilateur Java.

Il est intéressant de noter que notre système et les vérifications associées peuvent être pratiquement déléguées aux mécanismes (pourtant frustres) du compilateur Java. Cette propriété facilite dans de larges mesures le «portage» de notre système sur cette plateforme.

Le traducteur que nous présentons dans cette section est incomplet puisqu'il repose sur l'existence d'un environnement *runtime* que nous ne pourrons détailler qu'au moment de la description de la plateforme Comet.

6.6.1 Interprétation des définitions d'événements

Le problème de la traduction en code Java des définitions d'événement se divise en deux aspects :

1. la traduction des propriétés structurelles du type : structure cible, nommage, position dans la hiérarchie des types.
2. l'interprétation du cadre d'utilisation des événements associés : création, utilisation, réception, et émission.

De par sa nature de langage à classes, il n'est pas étonnant que notre traduction de la définition d'un événement corresponde à la création d'une nouvelle classe en Java.

Le nom de la classe porte le nom du type (sans oublier un préfixe de paquetage optionnel) et hérite d'une classe dont nous donnerons les spécificités par la suite. Il est cependant important de retenir que ceci impose une distinction très claire entre le type d'événement et la classe correspondant en Java.

Il est cependant très important de pouvoir récupérer les définitions des classes qui définissent les super-types indiqués pour l'événement (mot-clé if). La méthodes getSupertypes(), générée automatiquement à partir de la définition Scope retourne ces classes[12]. Les sous-types de DataEvent définissent également une méthode getValueObject() qui «transforme» leur *slot* _value en un objet Java même s'il s'agit d'un type de base. Le traducteur Scope génère automatiquement cette définition à partir de l'information de type associé au *slot*. Ces transformations implicites seront complétées pour satisfaire les exigences de la plateforme Comet. Nous reviendrons sur ce point par la suite.

Les définitions de *slot* sont elles simplement transformées en variables protégées portant le même nom et associé au type Java défini par correspondance en table 6.1 page 89. Le reste de la définition n'est pas modifiée et sera donc analysé par le compilateur Java.

Prenons l'exemple du type RealEvent défini en section 6.2 page 88. La classe générée porte le nom du type et offre la structure suivante :

```
public class RealEvent extends NumberEvent {
  ... // Définitions générées
}
```

Puisqu'un seul super-type est indiqué, nous pouvons dans ce cas précis «mixer» héritage et sous-typage en indiquant NumberEvent comme super-classe (cf. encart pour le cas du sous-typage multiple).

La définition du *slot* _value était la suivante :

```
slot _value type float;
```

La transformation génère un attribut portant le même nom et associé au même type puisqu'il s'agit d'un type Java :

```
protected float _value;
```

12. Il ne faudra donc pas utiliser les mécanismes réflexifs de Java sur les classes définissant les événements

Puisque le type RealEvent est un sous-type au sens large de DataEvent, le *slot* _value est utilisé pour la génération automatique de la méthode d'accès partagée par tous les sous-types de DataEvent :

```
public Float getValueObject() { return new Float(_value); }
```

Bien sûr, si le type indiqué est différent, la traduction sera elle-même modifiée en conséquence.

Considérons par exemple la définition d'un *slot* de type tableau comme ci-dessous :

```
slot _value type Array[int];
```

Les informations générées seront alors :

```
protected int[] _value;

public int[] getValueObject() { return _value); }
```

Ici, les types Scope et Java associés au *slot* _value diffèrent.

Les méthodes d'accès aux informations de type sont également générées automatiquement. Dans le cadre de notre type RealEvent, le résultat sera :

```
public String getTypeName() return "RealEvent";

public String[] getSupertypeNames() return [ "NumberEvent"];

public Class[] getSupertypes() return [ NumberEvent.class ];
```

Finalement, les autres définitions sont «recopiées» telles quelles dans la définition de classe.

6.6.2 Interprétation des définitions de composant

En ce qui concerne les composants, la génération du code Java associée aux définitions Scope repose sur les mêmes principes que la traduction des événements. Mais contrairement à ces derniers, les composants Scope exposent des comportements (complexes dans le cadre de la plateforme Comet). Nous imposons donc, pour de nombreuses raisons, la déclaration d'au plus une super-classe (ou super-définition) pour un composant donné.

Par défaut, cette définition correspond à la classe Java DefaultComponent. Ainsi, la classe générée pour la définition AddReal vue précédemment revêtira la forme suivante :

```
public class AddReal extends DefaultComponent
  // Type du composant
{

  ... // Définitions générées

}
```

La super-classe DefaultComponent donne la sémantique des opérations spécifiques aux composants concernant par exemple la gestion de l'identificateur unique ou encore les commandes d'émission des événements. Pour l'instant, nous considérerons cette classe comme abstraite. Il est bien sûr possible d'indiquer une autre super-classe dans la définition du composant, à condition que cette dernière ait elle-même pour ancêtre la classe DefaultComponent. Dans le cas contraire, la traduction doit échouer.

Nous remarquons sur la structure de classe générée ci-dessus le commentaire concernant le type du composant. Si le nom du type est AddReal, nous devons lui associer une définition, c'est-à-dire la donnée de son couple (type entrant, type sortant). Les mots-clés send et receive de la définition permettent d'identifier ces informations de typage. Cependant, plusieurs questions se posent à ce niveau :

1. Comment représenter, en Java, les types d'événements que le composant peut recevoir (type entrant) ?

Sous-typage multiple de Scope et héritage de Java

Nous traitons ici de la question du sous-typage «multiple» que notre modèle structurel et le langage associé supportent (sans «trouble sémantique», de par la nature passive des événements). Puisque le langage Java impose la déclaration d'une unique super-classe, il nous faut trouver un moyen pour simuler l'héritage multiple en Java.
La solution retenue consiste en la génération d'une classe «fantôme» effectuant l'aggrégation des propriétés des super-types de l'événement considéré. Le modèle de transformation proposé est très simple : *slots* et méthodes des super-types sont combinées dans une classe unique. Si nous partons de deux super-types Type1Event et Type2Event, la classe générée portera le nom de Type1EventType2Event_SCOPE_SHADOW[a]. Les *slots* et définitions des deux super-types seront combinés au sein de cette nouvelle classe, en considérant les cas suivants :

- les constructeurs sont regénérés pour l'occasion,
- les *slots* et méthodes de noms différents sont recopiés tels quels,
- les *slots* et méthodes portant le même nom, de signature et de définitions identiques dans plusieurs super-types agrégés sont factorisés (un seul exemplaire généré),
- les autres conflits (même nom, signature et définition incompatibles) conduisent au rejet de la traduction.

Cette méthode *défensive* empêche toute mauvaise surprise au moment de la compilation en Java et permet de traiter les problèmes de conflit de nommage très en amont dans la chaîne d'outillage proposée. Sans entrer dans les détails (qui dépassent le cadre de notre étude), l'absence de «comportement propre» associé au caractère disjoint des super-types directs «proches» dans la hiérarchie des types (même branche) permet d'offrir des garanties concernant la sémantique des définitions combinées. Par exemple un sous-type «abscon» indiquant qu'il est à la fois sous-type de RealEvent et IntegerEvent conduira à un conflit.
Cependant, la génération d'une telle classe «fantôme» ne sert qu'à contourner une limite importante (mais justifiable) du langage Java puisque dans notre version Scheme du langage Scope, le traitement du sous-typage multiple est beaucoup plus «propre».

[a] Les noms de classes suffixés par _SCOPE_SHADOW sont donc réservés en Scope.

2. Même question pour les types des événements qu'il peut émettre (type sortant) ?
3. Plus subtil : quels sont les types entrant effectivement gérés par le composant ?

Les deux premières questions passent par la génération d'interfaces spécifiques associées à chaque type d'événement : les *receivers* et les *senders*.

Dans notre exemple, des interfaces devront être générées pour - respectivement - la réception et l'émission de RealEvents. La première, l'interface *receiver* pour le type RealEvent, portera le nom de RealEventReceiver. La seconde - le *sender* - sera nommée, selon le même principe, RealEventSender. Ces interfaces doivent également refléter la relation de sous-typage, ce qui conduit aux définitions suivantes :

```
public interface RealEventReceiver extends NumberEventReceiver {}

public interface RealEventSender extends NumberEventSender {}
```

Il faut de plus, au niveau de la classe générée pour le composant, indiquer que ce dernier implémente ces deux interfaces. Ainsi, nous pourrons utiliser les méthodes standard pour le test de sous-typage en Java. Par exemple, si nous voulons tester si un composant component peut recevoir des RealEvent, nous pourrons utiliser le test suivant :

```
boolean can_receive = RealEventReceiver.class.isAssignableFrom(component.getClass());
```

Une autre question importante concerne la transformation et la prise en compte des blocs de réception d'événement (clause when). Pour pouvoir tester la présence ou non d'un tel bloc, une interface spécifique est générée pour chacun d'entre eux. Cette catégorie d'interface représente les *fetchers* d'événements. Pour le type RealEvent, l'interface générée est la suivante :

```
public interface RealEventFetcher {
  public void whenRealEvent(RealEvent event);
}
```

Ainsi, puisque la classe du composant implémente cette interface (car la définition de AddReal comporte un bloc when pour le type RealEvent), la fonctionnalité de réception sera transformée en une implémentation de la méthode whenRealEvent. Puisque le reste de la définition reste inchangée, nous obtenons donc finalement la définition complète ci-dessous :

```
public class AddReal extends DefaultComponent
  implements RealEventReceiver, RealEventSender, RealEventFetcher {
  public void whenRealEvent(RealEvent event) {
    if(op1==null)
      op1 = new Float(event.getValue());
    else {
      RealEvent out_event = new RealEvent(op1.floatValue() + event.getValue());
      send(out_event);
      op1 = null;
    }
  }

  protected Float op1 = null;
}
```

La traduction en Java du couple (RealEvent, AddReal) est présentée de façon synthétique en figure 6.6 page suivante. Ce diagramme de classes UML développe les différentes relations entre classes et interfaces «standard» (fond blanc) et les classes et interfaces générées (fond grisé).

Les figures 6.7 page suivante et 6.8 page 103 récapitulent pour leur part les différentes générations associées respectivement aux définitions de type d'événement et aux définitions de composants.

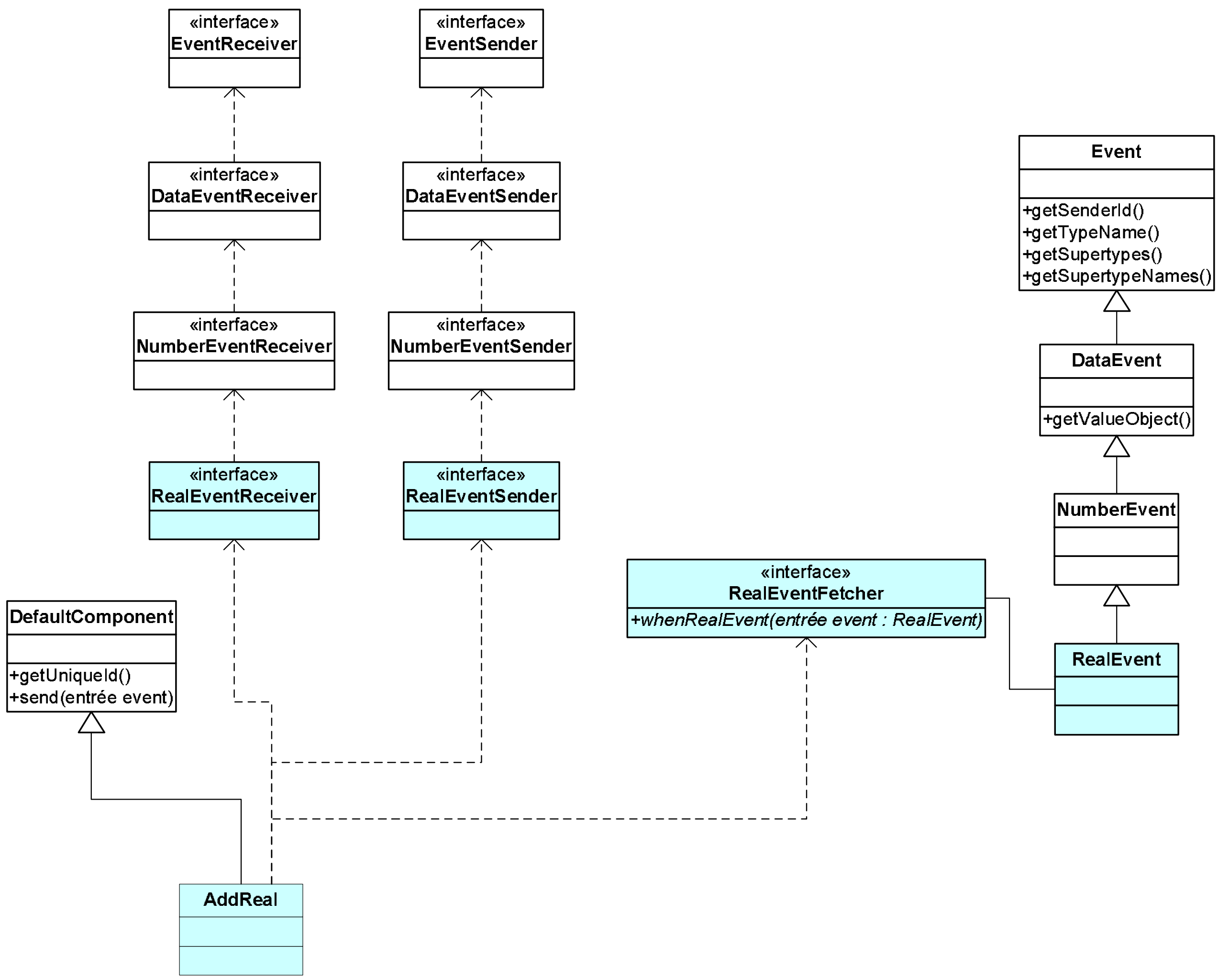


FIG. 6.6 – *Traduction Scope vers Java : Diagramme de classes UML*

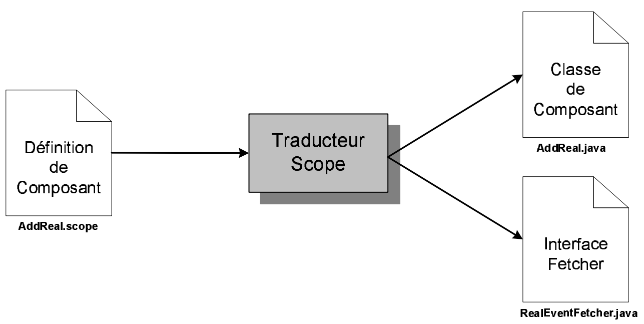


FIG. 6.7 – *Traduction Scope vers Java : Définitions d'événements*

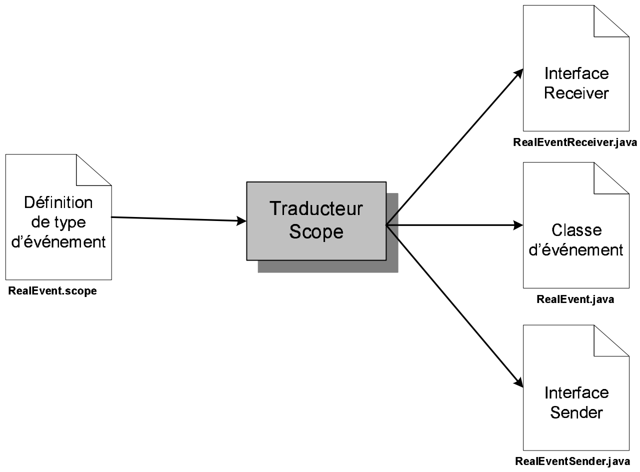


FIG. 6.8 – *Traduction Scope vers Java : Définitions de composants*

6.6.3 Connections avec label

Si la version Scheme du traducteur Scope fournit une solution *intrinsèque* pour les connexion labellées, le parti-pris dans la version Comet s'oriente vers leur «simulation» à partir des éléments standard du système (connexions sans label).

Si d'un point de vue «modélisation», l'introduction de types spécifiques pour les labels de connexion n'est pas souhaitable, leur introduction *automatique* pour le support opérationnel des labels ne pose aucun problème. Dans notre cas, il suffira cependant de générer les interfaces Java indiquant la réception effective d'un événement donné (interfaces *Fetcher*). Ainsi, notre solution au problème des labels passent par la génération automatique de *fetchers* spécifiques, sous-types du *fetcher* sans label correspondant. Par exemple, l'indication des labels Left et Right pour le type RealEvent conduit à la génération des interfaces LeftRealEventFetcher et RightRealEventFetcher.

La subtilité est que si ces deux interfaces changent effectivement le nom de la méthode générée pour la réception des événements en fonction du label, le type de l'événement reçu lui est toujours le même (RealEvent dans notre cas) :

```
public interface LeftRealEventFetcher {
  public void whenLeftRealEvent(RealEvent event);
}

public interface RightRealEventFetcher {
  public void whenRightRealEvent(RealEvent event);
}
```

Le traducteur Scope génère de plus, à partir de la définition du composant avec labels, une réaction standard pour le type sans label (dans notre exemple: RealEvent) en vérifiant au préalable dans une structure de donnée adaptée si la source a été connectée pour le label Left ou Right. Les sources concernées devront être enregistrées au préalable via émission d'un événement standard, de type ConnectWithLabelEvent qui contient un identificateur de source, un type d'événement et un label associé.

La forme du composant généré est donc la suivante :

```
public class SubReal extends DefaultComponent
 implements LeftRealEventReceiver, RightRealEventReceiver, RealEventSender /* etc... */ {

 public void ConnectWithLabel(ConnectWithLabelEvent event) {

   // Enregistrement d'un triplet(type,label,source)
   registerLabel(event.getSenderId(), event.getLabel());

 }

 public void whenRealEvent(RealEvent event) {

   // Réception d'un événement sans label
   // Dispatch en fonction des triplets enregistrés

   String label = fetchLabel(event.getSenderId()) {
   if(label.equals("Left")) {
    whenLeftRealEvent(event);
    return;
   } else if(label.equals("Right")) {
    whenRightRealEvent(event);
    return;
   }

   // Suit le cas sans label (si existant)
 }

 public void whenLeftRealEvent(RealEvent event) {

   // Réception d'un événement pour le label Left

 }

 public void whenRightRealEvent(RealEvent event) {

   // Réception d'un événement pour le label Right

 }

}
```

La réception standard pour les événements de type RealEvent, définie par la méthode whenRealEvent générée automatiquement, sert donc d'aiguillage entre les différentes variantes avec label. La méthode fetchLabel est utilisée pour récupérer le label en fonction de l'émetteur de l'événement. Si une connexion labellée a été enregistrée pour cette source (appel à registerLabel), alors le bloc when adéquat est exécuté.

Si une réponse générique (indépendante du label) est proposée (existence d'un bloc when pour le type sans label), alors le corps de ce bloc est finalement exécuté par la méthode d'aiguillage.

Notons que dans les détails, le mécanisme proposé est un peu plus complexe, notamment pour empêcher les conflits de nom [13].

13. La transparence de la traduction au niveau du langage source est un problème assez complexe. Par exemple, les noms registerLabel et fetchLabel sont trop probables pour être utilisés dans le code généré automatiquement. Nous retrouvons ici les fameux *phénomènes de capture* liés aux macro-expressions de Common Lisp

Troisième partie

La plateforme Comet

«Les idées nouvelles passent par trois phases :
- Cela ne peut être fait.
- Cela peut probablement être fait, mais cela ne vaut pas la peine d'essayer.
- Je le savais, c'était une bonne idée !»

Arthur C. Clarke

7
Modèle comportemental

Le modèle structurel, défini dans la première partie de notre étude, donne un sens à «l'activité» de composition dynamique d'applications à partir de briques logicielles de base : les composants. La plateforme interstitielle Comet que nous introduisons dès ce chapitre met en œuvre ce modèle structurel (et le langage Scope associé) dans un univers *concurrent* et *réparti*. Cette immersion introduit de nombreuses «zones d'ombres» concernant l'opérationnalisation de notre modèle. Des questions simples (au sens de la structure) comme la création d'un nouveau composant ou l'émission d'un événement par un composant et sa réception - éventuellement multiple et distante - se transforment alors en problèmes relativement complexes.

Nous devons donc en quelque sorte «entrer» dans les composants - dont la *structure* nous est désormais connue - pour en décrire le *comportement*.

7.1 Principes de base

Si le sens du mot *comportement*, en fonction du contexte, se trouve hautement connoté et incertain, une terminologie - dans notre cas précis - peut être posée de façon assez rigoureuse. Nous utilisons notre constituant fondamental - le composant - pour distinguer et relier les deux termes : structure et comportement. Informellement, le point de vue *structurel* s'intéresse essentiellement à l'*extérieur* des composants alors que le point de vue *comportemental* se concentre sur leur *intérieur*. Bien sûr, ces deux aspects sont très liés. Par exemple, il est difficile de savoir précisément si un type entrant ou sortant (concepts définis au niveau structurel) est du domaine extérieur ou intérieur. Dans le même ordre d'idée, l'émission effective d'un événement est un aspect essentiellement comportemental et cependant très dépendant de la structure des deux composants en interaction.

Nous définirons un *modèle de comportement* de composant comme un *raffinement* du *modèle structurel* de composition que nous avons «élaboré» au préalable.

Cette définition entraîne deux implications majeures :

1. Le langage Scope, dans une version étendue pour l'occasion et moyennant adaptation des mécanismes de traduction, sera utilisé pour décrire les applications Comet *in extenso*.
2. La formalisation des modèles comportementaux correspondra, de façon toute aussi formelle, à un ensemble de raffinements du modèle structurel initial.

Si la première implication possède essentiellement des «vertus» pragmatiques (nul besoin de réinventer un langage de composition spécifique), la seconde est *théoriquement* fondamentale. Lorsque nous parlons de *raffinement* au niveau formel, il s'agit de raffinement au sens de la notation Z et donc également au niveau de la dénotation mathématique (ensembliste) sous-jacente. Il ne s'agit plus simplement d'un mot (également très «chargé») de la langue française.

Modèle de contrôle asynchrone

Sous l'angle du comportement, la composition dynamique passe selon nous par la satisfaction d'une propriété de *découplage comportemental*.
L'introduction du modèle des connexions typées de Scope nous permettait d'offrir le découplage structurel entre composants - qui ne peuvent se référencer mutuellement dans ce modèle. Au niveau du comportement, il faut pouvoir prolonger cette indépendance structurelle intrinsèque entre composants par la considération de l'environnement distribué et concurrent qui les supportent. Comme largement décrit dans les travaux sur les acteurs, une solution privilégiée consiste à doter les composants d'une sémantique *asynchrone*. Ainsi, un composant qui émet un événement n'est aucunement lié, en terme de contrôle, aux composants qui recevront cet événement. Tout comme pour l'unidirectionalité, l'asynchronisme restreint les possibilités d'intéractions directes (i.e. atomiques). Nous aborderons le cas de la bidirectionnalité et des échanges synchrones sous l'angle - indirecte - de la composition à partir du modèle de base, au chapitre 9.

Puisque défini comme raffinement du modèle structurel, le modèle de comportement canonique qui sous-tend l'interstitiel Comet en hérite les concepts fondamentaux (composants, événements, connexions) et les précise en les dotant de propriétés opérationnelles.

La composition dynamique nous obligeait, dans Scope, au *découplage structurel*. Ce dernier a été obtenu essentiellement par l'introduction des types entrant et sortant associés aux composants et de la spécification puis l'établissement de connexions unidirectionnelles pour les relier. Avec notre nouvelle vision comportementale, il nous faut pouvoir conserver les possibilités de composition dynamique et introduire, en guise de préliminaire fondamental, notre modèle de contrôle *asynchrone* (cf. encart).

Pour introduire notre modèle de comportement, nous allons décrire un service distribué de notifications système, proposé en standard dans l'environnement Comet. Ce service met en œuvre des composants sondes (ou *probes*) permettant de donner des indices de fonctionnements de machines sur le réseau. Plusieurs types de sondes sont disponibles :

- sondes CPU qui délivrent un indice de performance processeur,
- sondes MEM pour les accès mémoires,
- sondes DISK pour les accès disque,
- sondes NET pour les communications réseau,
- etc.

A intervalles réguliers, ces sondes envoient leurs indices sous forme de notifications vers des composants d'analyse ou de visualisation. La figure 7.1 page suivante montre un exemple de configuration dans lequel trois machines sont «sondées» sur diverses critères (icônes en forme de processeur, de disquette ou d'unité centrale). Les résultats sont acheminés vers un serveur d'analyse (ordinateur portable) et comparés à cet endroit.

Les événements mis en jeu par ce service de notification sont tous sous-types de InfoEvent et manipulent des indices *immédiats* et *en moyenne*. La structure du type InfoEvent (et de ses sous-types) est la suivante :

```
event InfoEvent is Event {
  slot _imm_value type int;
  slot _avg_value type int;

  InfoEvent(int imm_value, int avg_value) {
    _imm_value = imm_value;
    _avg_value = avg_value;
  }

  int getImmediateValue() { return _imm_value; }
  int getAverageValue() { return _avg_value; }

}
```

Les indices respectivement immédiat et moyen sont disponibles en lecture seule grâce aux accesseurs getImmediateValue() et getAverageValue().

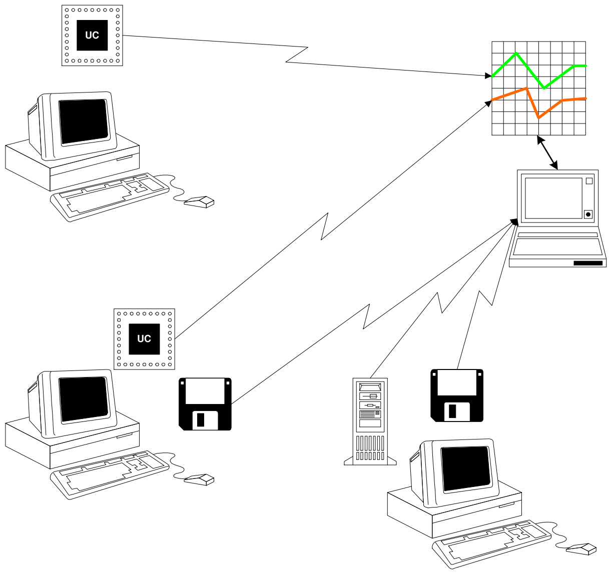


Fig. 7.1 – *Service distribué d'analyse d'indices systèmes*

Nous pouvons maintenant commencer la description des composants de service. Tout d'abord, les composants sondes ont pour simple principe, à intervalles réguliers, de :

1. calculer, par diverses méthodes [1], l'indice immédiat de la machine sondée, pour le type de sonde considéré.
2. mettre à jour l'indice en moyenne,
3. émettre un événement de notification de type InfoEvent ou d'un sous-type.

Par exemple, la forme générale d'une sonde CPU est la suivante :

```
component ProbeCPU {
  send CPUInfoEvent;

  int avg_value, probe_count = 0;

  void probe_routine() {
    int imm_value = update_probe();
    probe_count++;
    avg_value = avg_value + (imm_value/probe_count);

    CPUInfoEvent info = new CPUInfoEvent(imm_value,avg_value);

    send(info);
  }

}
```

Cette définition est bien sûr incomplète puisque rien ne dit quand la méthode probe_routine() sera déclenchée, ni ce que contient la méthode de calcul de l'indice immédiat : update_probe(). Considérons pour l'instant qu'à intervalles réguliers, ces routines sont déclenchées et qu'en conséquence, un événement de type CPUInfoEvent soit effectivement envoyé par le composant sonde.

Ce composant, puisqu'il ne fait que générer des événements, joue le rôle d'*émetteur* dans notre protocole.

Pour «traiter» les événements émis, un composant de visualisation pourra proposer la «forme» suivante :

1. Les algorithmes utilisables pour donner un indice de fonctionnement à une machine peuvent être nombreux. Ainsi, il existera potentiellement plusieurs types de sonde pour le même indice. Par exemple, pour les tests mémoires, on pourra s'intéresser à la taille disponible, à la vitesse d'allocation de grands blocs contigüs ou encore aux allocations/désallocations de petits blocs épars, etc.

```
component Viewer {
  receive InfoEvent;
  when (InfoEvent info) {
    // Visualisation de l'indice reçu
    update_view(info.getImmediateValue(), info.getAverageValue());
  }
}
```

Ce visualisateur, utilisable quelque soit le type de la notification reçue (du moment qu'il s'agit d'un sous-type de InfoEvent), propose un comportement purement «réactif». Le bloc when décrit ci-dessus est déclenché à chaque fois qu'un indice est reçu, son contenu (indices immédiat et moyen) étant traité par la méthode update_view().

De tels composants purement réactifs et isolés (ils n'émettent aucun événement) jouent le rôle de *récepteurs* pour le protocole considéré.

En connectant deux composants ProbeCPU et Viewer, pour le type CPUInfoEvent (ou un super-type qui sera spécialisé automatiquement), nous obtenons l'architecture représentée en figure 7.2.

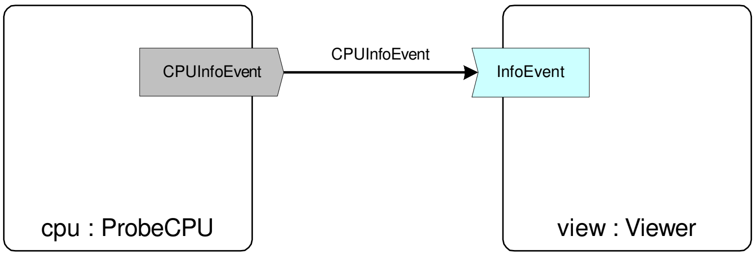


FIG. 7.2 – *Exemple de configuration émetteur/récepteur*

Après connexion, le composant ProbeCPU, si sa méthode probe_routine() est déclenchée, enverra un événement de type CPUInfoEvent au composant Viewer. La figure 7.3 montre ce que cette émission induit en terme de contrôle (les segments fléchés décrivent le flot de contrôle).

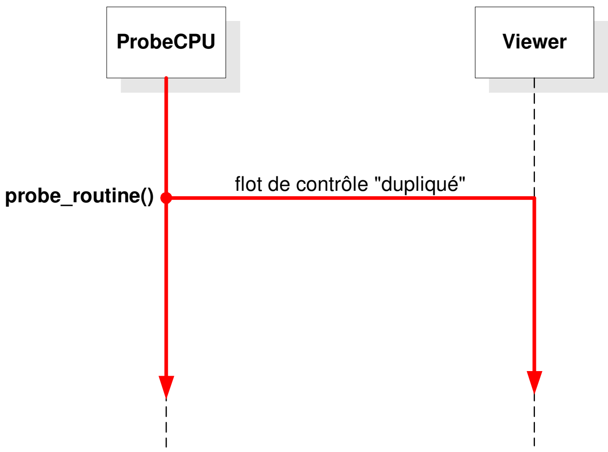


FIG. 7.3 – *Emission d'un événement : contrôle asynchrone*

Si dans le monde des objets, nous sommes habitués à ce que ces derniers se passent le contrôle les uns aux autres (vision synchrone), les composants Comet agissent à ce niveau de façon totalement découplée (vision asynchrone) : le contrôle est «subdivisé» en deux *activités* concurrentes[2], l'une pour l'émetteur (qui continue de travailler comme si de rien n'était) et l'autre pour le récepteur (créée pour l'occasion, par «réactivité»).

Le *routage* de l'événement émis par l'émetteur est *implicite* : c'est l'établissement préalable d'une connexion entre les deux composants qui induit la «destination» de l'événement. Cette nature implicite du protocole de

2. De fait, aucune subdivision n'existe réellement si les deux composants sont distants, les deux activités sont intrinsèquement séparées dans ce cas puisque les composants «tournent» sur des machines différentes.

routage des événements est un aspect fondamental de notre modèle de comportement, beaucoup plus «puissant» que ce que notre exemple suggère. Illustrons notre propos en introduisant un composant ProbeFilter servant d'intermédiaire entre sondes et visualisateurs/analyseurs :

```
component ProbeFilter {
  receive InfoEvent;
  send InfoEvent;

  when (InfoEvent info) {
    send(info);
  }
}
```

Cette définition, quoique minimaliste, n'est pas aussi simple qu'il n'y paraît de prime abord. En effet, on ne fait ici qu'exploiter la puissance «cachée» du modèle de communication proposé, et notamment le couple sous-typage/protocole de routage. Cette définition dénote d'ailleurs toute une catégorie de composants dits *médiateurs* [3] dont nous mesurerons l'importance par la suite. Grâce à de tels médiateurs, nous pouvons construire des architectures totalement découplées, de type émetteurs/médiateur/récepteurs, comme celle décrite en figure 7.4.

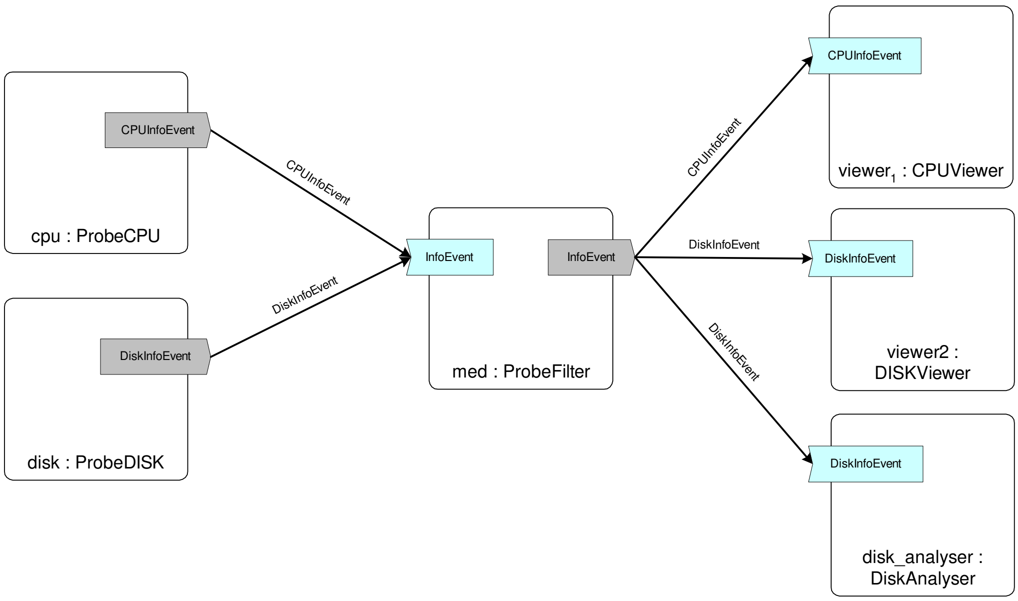


FIG. 7.4 – *Exemple de configuration à médiateur*

Dans cette configuration, deux émetteurs sont connectés à un médiateur, lui-même relié en sortie à trois récepteurs différents. Émetteurs et récepteurs se spécialisent en deux catégories: selon qu'ils émettent ou reçoivent des événements sur les indices de performances processeur ou disque. Rappelons que les types d'événement CPUInfoEvent et DiskInfoEvent sont sous-types de InfoEvent.

Détaillons dans un premier temps le processus d'acheminement d'un événement de type CPUInfoEvent émis par le composant cpu. Cet événement est implicitement routé vers le médiateur ProbeFilter puisque une connexion «du bon type» les relie. Cet événement, instantanément réémis par le médiateur, est acheminé vers le récepteur viewer_1 qui peut alors le traiter.

La figure 7.5 page suivante décrit l'impact au niveau du contrôle d'une telle interaction.

Conformément à nos attentes, les composants restent totalement découplés en terme de contrôle dans cette configuration. Au niveau structurel, nous pouvons remarquer un découplage supplémentaire entre émetteurs et récepteurs, obtenus par l'introduction de notre médiateur. Notre architecture découple donc *totalement* les émetteurs des récepteurs, que ce soit en terme de structure ou de comportement.

3. Le composant mis en œuvre dans cet exemple est plus précisément un *filtre*; le fitrage étant un exemple possible de procédé de médiation.

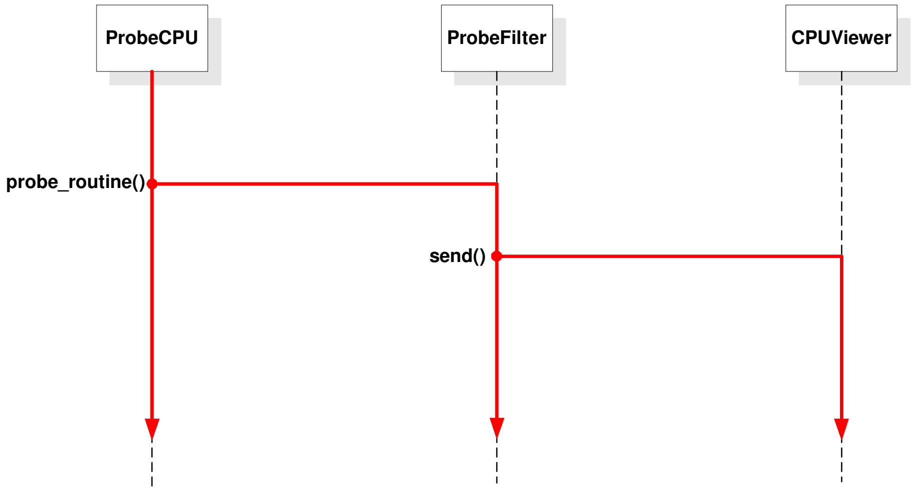


FIG. 7.5 – *Composant médiateur et contrôle asynchrone*

Diffusion multi-point implicite en fonction du type

Le mode de diffusion multi-point représente le deuxième trait caractéristique (et la principale originalité) de modèle de communication proposé par la plateforme Comet.
La motivation première pour l'introduction de ce mode de diffusion multi-point comme cas par défaut est qu'il représente un sur-ensemble du mode de diffusion point-à-point plus classique et qu'il offre, dans ce sens, un modèle plus général, voir plus expressif pour la description d'interactions distribuées.
Dans l'acceptation courante du terme, un type dénote un ensemble de valeurs. Ainsi, dans notre approche, un type effectif de connexion dénote un ensemble d'événements pouvant être acheminés par la connexion considérée.
Du point de vue d'un composant, il semble donc naturel d'accepter - pour émettre un événement - toute connexion dont le type effectif est compatible avec cet événement.
Nous verrons par la suite que ce mode de diffusion possède de nombreuses propriétés intéressantes. Par exemple, le multi-point implicite nous permettra d'offrir au chapitre 8 un service de réplication passive, de façon extrêmement concise. Nous modéliserons également, au chapitre 9, un protocole de vote synchrone pour la validation atomique de transaction dans lequel le mode de diffusion implicite permet de faire varier dynamiquement le nombre de participants au vote.

L'émission d'un événement de type DiskInfoEvent se déroule globalement de la même façon, tout du moins pour ce qui concerne la partie émetteur/médiateur. Le principal changement intervient au niveau du médiateur puisque deux composants récepteurs sont connectés à ce dernier pour le type DiskInfoEvent.

Dans ce cas de figure, trois possibilités sont à considérer :

1. Le médiateur peut vouloir «choisir» entre les deux récepteurs;
2. L'ambiguïté ne peut être résolue simplement, il faut annuler la «diffusion»;
3. Les deux connexions correspondent bien au contrat «support d'un événement de type correct» et l'événement est donc «dupliqué» et transmis aux deux récepteurs.

Dans le cadre de la plateforme Comet, la troisième possibilité est privilégiée : l'événement DiskInfoEvent sera reçu par les deux récepteurs connectés au médiateur pour ce type. Il s'agit d'un mode de diffusion *multi-point* implicite en fonction du type (cf. encart).

Si la diffusion multi-point représente le cadre général - ou implicite - de notre modèle de communication, il est important de pouvoir «retrouver» le cas plus précis de la diffusion point-à-point. Lorsqu'une seule connexion est établie pour un type donné en sortie d'un composant, alors le multi-point devient naturellement un point-à-point implicite.

Il existe cependant des cas où l'explicitation du destinataire d'une émission point-à-point est nécessaire. Ainsi, toutes les primitives d'émission d'événement proposent une variante paramétrée par l'identificateur d'un récepteur (point-à-point explicite) ou même une liste de récepteurs (multi-point explicite). Notons cependant que ceci introduit au niveau structurel un couplage implicite (manipulation d'identificateurs) pouvant poser d'importants problèmes pour la composition dynamique. La table 7.1 résume l'ensemble des primitives de communications disponibles (pour le mode de communication asynchrone).

Primitives d'émission d'événement
send(Event event) **Définition :** Emission en mode multi-point implicite en fonction du type de event
send(Event event, String id) **Définition :** Emission en mode point-à-point explicite vers id
send(Event event, String[] id) **Définition :** Emission en mode multi-point explicite vers tous les id

TAB. 7.1 – *Plateforme Comet : primitives d'émission*

Malgré sa définition relativement simple, notre exemple réalise un protocole de communication découplé, de type *publish/subscribe* [Mey00] avec des propriétés relativement avancées :

- architecture de type «médiateur», permettant le découplage structurel et comportemental entre les émetteurs (*publishers*) et récepteurs (*subscribers*),
- souscription implicite, basée sur l'établissement dynamique de connexions entre les récepteurs et le médiateur d'une part, et entre ce dernier et les récepteurs d'autre part,
- acheminement asynchrone des événements,
- sélection/Routage en fonction du type/sous-type, et
- choix des événements par le serveur (couple émetteur/médiateur) ou mode d'acheminement de type *push*.

De plus, grâce au modèle de composition dynamique, notre service de notification ainsi créé possède de nombreuses propriétés liées au propos de notre thèse, c'est-à-dire *l'adaptation dynamique* :

- Ajout/Retrait à la volée de sondes, visualisateurs, médiateurs, etc.
- Reconfiguration des connexions impliquant une modification des propriétés de filtrage

Cet exemple est donc emblématique de la facilité avec laquelle certains services distribués, généralement considérés comme non-triviaux (pour l'exemple du *publish/subscribe*, lire par exemple [Mey00]) peuvent trouver des solutions exprimables relativement simplement par changement de point de vue : du synchrone point-à-point courant à l'asynchrone multi-point typé de l'interstitiel Comet.

Notre protocole peut bien sûr être raffiné dans de nombreuses directions : mode d'acheminement de type *pull*, sélection en fonction du contenu (cf. chapitre 8), etc.

L'intérêt du modèle de communication décrit dans ce chapitre dépasse bien évidemment le cadre de cet exemple construit pour en illustrer les principes. De par sa nature asynchrone, toutes les interactions possibles dans les modèles d'acteurs ou d'objets actifs sont exploitables directement dans le cadre de Comet. Des exemples de telles interactions ont été décrites au chapitre 2. L'adjonction des règles de typage au niveau de l'interconnexion des composants offre en complément la gestion implicite du routage des événements; ce qui - répétons-le - représente selon nous la principale innovation de notre modèle et, en quelque sorte, son essence.

7.2 Modèle de comportement canonique

Sur les principes décrits dans la section précédente, nous devons maintenant construire formellement notre modèle comportemental puis rattacher ce dernier au modèle structurel défini précédemment. Dans ce but, nous devons proposer une spécification Z composée de schémas d'état et de schémas d'opération pour décrire le comportement des composants. Nous devons donc, dans un premier temps, réfléchir à l'état du comportement d'un composant ainsi qu'aux différentes opérations associées.

7.2.1 Niveau méta-comportement

D'après la discussion en section précédente, nous pouvons nous rappeler qu'un composant est avant tout une entité logicielle asynchrone. Au même titre que la majorité des modèles concurrents (qu'ils soient basés sur les notions d'acteur, d'objet actif, etc.), ceci implique un découplage entre la prise en compte des requêtes de traitement et les traitements proprement dits. Dans notre «métaphore» événementielle, nous distinguons ainsi :

- la réception des événements (requêtes de traitement),
- l'exécution des fonctionnalités associées (traitements), et
- l'émission des événements (effets des traitements).

La réception d'un événement peut, dans un «univers» asynchrone, survenir à tout moment. Au moment *effectif* de la réception, il est possible (voir même fréquent) que le composant récepteur ne puisse traiter l'événement reçu dans l'immédiat. Pour ne pas perdre cette requête, il faut bien sûr la mettre en *file d'attente* et fournir un moyen de récupérer, lorsque le composant est prêt, des événements depuis cette file et lancer les traitements correspondants. Dans le même ordre d'idée, puisque nous voulons une forme de comportement totalement asynchrone, l'émission d'un événement ne sera effective que quand la couche de communication sera prête[4]. Une file d'attente en sortie est donc également mise en œuvre[5].

Pour résumer, nous pouvons extraire les propriétés minimales associées aux composants pour leur donner une sémantique asynchrone et distribuée. Concernant l'état des composants, nous déduisons la nécessité de modéliser formellement :

- une couche de communication (entrées et sorties séparées),
- un système de file d'attente en entrée,
- un système de file d'attente en sortie.

Au niveau des opérations associées, nous pouvons dresser une liste relativement complète :

- réception des événements,
- mise en file d'attente de ces événements en entrée,
- extraction des mêmes événements depuis la file d'attente,
- exécution des traitements (ou fonctionnalités) associées,
- mise en file d'attente des événements émis,
- extraction des événements à émettre,
- emission effective, finalement.

Le fait de rendre *tangible* au niveau du modèle (et de son implémentation, cf. chapitre 10) ces fonctionnalités de contrôle liées au cheminement des événements dans les composants se nomme une *réification*. Dans la plus pure tradition des travaux liés au concept de *réflexion*, nous dirons que nous construisons le modèle de comportement des composants à un niveau *méta*. L'intérêt de cette réification est double puisqu'elle permet dans un premier temps de décrire précisément ces fonctionnalités *nécessaires* pour doter le modèle d'une sémantique adaptée mais également d'offrir un support, tant conceptuel que pratique[6], pour la construction de divers raffinements selon des directions aussi nombreuses que le nombre de fonctionnalités réifiées. Bien sûr, de nouvelles directions peuvent être proposées (et nous le ferons par la suite) mais ceci se fera au prix d'un impact plus important sur le modèle formel puisqu'il faudra modéliser une nouvelle fonctionnalité de contrôle, associée à un raffinement de l'état du comportement.

Le fait de décomposer les étapes nécessaires au cheminement d'un événement ou d'un message dans une entité de type asynchrone rejoint l'idée de *réflexion comportementale* de grain très fin, et notamment les travaux de Jeff Mc Affer dans CodA [McA95b], ainsi que d'autres travaux qui s'en inspirent [Mig99]. Nous dénommons notre modèle réflexif le *niveau méta-comportement* des composants. Ce dernier est représenté «conceptuellement» en figure 7.6 page suivante.

4. Les mécanismes sous-jacents de la communication événementielle ne sont pas triviaux. Il s'agit, d'une part, d'exploiter les ressources réseau - largement partagées - et, d'autre part, de supporter la diffusion multi-point implicite.

5. Il s'agit d'une autre «originalité» de notre modèle qui trouvera justification tout au long de notre discours.

6. Au niveau de l'implémentation, les mêmes fonctionnalités de niveau méta seront également accessibles et spécialisables.

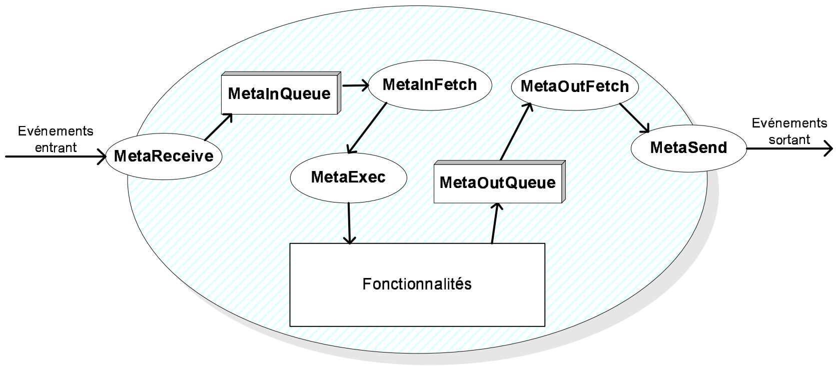


FIG. 7.6 – *Niveau méta-comportement*

Si au niveau formel nous parlerons plutôt d'opération, pour des raisons principalement historiques, les fonctionnalités répertoriées sur la représentation du méta-niveau sont associées à des entités nommées *méta-objets* dont les noms sont très spécifiques. Ainsi, le méta-objet MetaReceive prend en charge la réception des événements depuis la couche de communication. Il les transmet ensuite au méta-objet MetaInQueue dont le but est de les stocker en file d'attente. De façon *concurrente* (nous reviendrons sur ce point essentiel), le méta-objet MetaInFetch, lorsqu'il est actif, extrait les événements depuis la file d'attente et les communique au méta-objet MetaExec qui se charge d'associer un traitement à l'événement. En sortie, toute émission est tout d'abord prise en charge par le méta-objet MetaOutQueue qui le place en file d'attente. Et, encore une fois de façon concurrente, ces mêmes événements sont extraits par MetaOutFetch et envoyés effectivement (routage compris) par MetaSend.

Le choix de ces catégories de fonctionnalités de niveau méta ne s'est bien sûr pas fait au hasard et résulte principalement du potentiel de spécialisations qu'elles offrent. Nous donnons ci-dessous quelques exemples de spécialisations envisageables pour chacun des méta-objets introduits précédemment :

- MetaReceive : adaptation à diverses sémantiques de communication (fiables ou non-fiables, ordonnées ou non, persistantes, synchrones, etc.)
- MetaInQueue : systèmes à files d'attente multiples, files à priorité.
- MetaInFetch : modèles de synchronisation basés sur les événements reçus, séquences atomiques.
- MetaExec : modèles de synchronisation basés sur les fonctionnalités : méthodes gardées, contraintes de synchronisation [Frø96], concurrence interne.
- MetaOutQueue : émissions concurrentes, réordonnancement causal style RSVP [BZB+97].
- MetaOutFetch : files d'attente multiples et heuristiques de choix associées, temporisation pour satisfaire une contrainte de stabilité.
- MetaSend : modèles de communication variés.

Bien sûr, ces spécialisations ne sont obtenues qu'au prix de changements parfois important dans le modèle et son implantation. Cependant, la délimitation claire du «lieu» de la modification (le méta-objet concerné) est une étape préliminaire très importante au niveau méthodologique. Dans le reste de notre discussion, nous lèverons le voile sur certaines de ces spécialisations, ce qui nous permettra d'en dégager une sorte de «philosophie» sous-jacente de l'adaptation comportementale du modèle Comet. Certaines des spécialisations possibles, et notamment celles qui concernent la synchronisation relative à l'état interne des composants, correspondent à des portages «évidents» des variantes proposées dans CodA. Nous ne détaillerons dans ce qui suit que des modèles proposant une réelle «valeur ajoutée», du point de vue de la plateforme Comet.

7.2.2 Schéma de comportement

Pour aborder notre description formelle du modèle de comportement des composants Comet, détaillons tout d'abord la partie spécifique au comportement pour ensuite élaborer la jonction avec la partie structurelle.

Trois informations doivent être modélisées, du moins partiellement, à ce niveau :

1. La couche de communication utilisée, décomposée en *canaux de communication* entrant et sortant,

2. les système de file d'attente, et
3. les fonctionnalités associées aux événements reçus.

Avant d'introduire le schéma correspondant, définissons ce que sera pour nous un événement, au niveau opérationnel :

$$
\begin{array}{|l}
\hline
\textit{Event} \\
\textit{type} : \textit{TYPE} \\
\textit{sender\_id} : \textit{IDENTIFIER} \\
\textit{receiver\_id} : \mathbb{P}\, \textit{IDENTIFIER} \\
\hline
\textit{sender\_id} \notin \textit{receiver\_id} \\
\hline
\end{array}
$$

Il s'agit principalement de la donnée d'un type et de l'identification de l'émetteur de l'événement. Le champ *receiver_id* va nous permettre de réaliser des émissions explicites[7].

Nous pouvons désormais introduire le schéma de comportement par défaut :

$$
\begin{array}{|l}
\hline
\textit{DefaultBehavior} \\
\textit{in\_channels} : \mathbb{P}\, \textit{CommChannel} \\
\textit{out\_channels} : \mathbb{P}\, \textit{CommChannel} \\
\textit{in\_queue} : \operatorname{seq} \textit{Event} \\
\textit{out\_queue} : \operatorname{seq} \textit{Event} \\
\textit{functs} : \mathbb{P}\, \textit{Functionality} \\
\hline
\end{array}
$$

Nous pouvons d'ores et déjà noter que les contraintes associées à ce schéma sont fort peu nombreuses. En particulier, il n'y a pas d'invariant associé. Cependant, plusieurs informations importantes sont proposées. Tout d'abord le modèle de couche de communication, comme souvent [Hoa85] [Mil99], est découplé en canaux entrant et canaux sortant.

Nous donnerons les propriété de ces canaux par la suite, en expliquant le schéma *CommChannel*. Les files d'attente en entrée et en sortie sont définies, dans le modèle par défaut, comme de simples séquences d'événements. Enfin, les fonctionnalités associées à la réception des événements sont représentées par un schéma propre : *Functionality*.

Canaux de communication

La couche de communication, comme nous l'avons vue, est représentée par deux variables : *in_channels* et *out_channels*, toutes deux du type *CommChannel*. De manière relativement «frustre», nous définissons un canal de communication dans Comet comme une version comportementale des connexions effectives structurelles, mais géré du point de vue du composant :

$$
\begin{array}{|l}
\hline
\textit{CommChannel} \\
\textit{id} : \textit{IDENTIFIER} \\
\textit{type} : \mathbb{P}\, \textit{TYPE} \\
\textit{events} : \operatorname{seq} \textit{Event} \\
\hline
\forall\, \textit{event} : (\operatorname{ran} \textit{events}) \bullet (\exists\, t : \textit{type} \bullet \textit{event.type} \preceq t) \\
\hline
\end{array}
$$

L'identificateur *id* correspond soit au composant source (canal entrant), soit au composant destination (canal sortant) qui est relié par le canal de communication considéré. Le type effectif *type* dénote les types (ou super-types) des événements circulant sur ce même canal. Ces événements - en cours de communication - appartiennent à une séquence. L'invariant proposé explique simplement que les événements eux-mêmes (codomaine de la séquence) ne peuvent être que sous-types des types indiqués par la variable *type*.

7. Parce qu'une connexion est extérieure à tout composant, on ne peut - au niveau du modèle formel - émettre un événement à soi même, d'où l'unique condition proposée pour l'invariant du schéma. Cette contrainte peut-être levée, si nécessaire, en proposant des primitives spécifiques aux envois «cycliques».

Pour initialiser un canal de communication, il nous faut lui désigner un identificateur de composant et un type effectif. Le canal sera initialement vide «au démarrage» :

$$
\begin{array}{|l}
\textit{InitCommChannel} \\
\hline
id? : \mathit{IDENTIFIER} \\
type? : \mathbb{P}\, \mathit{TYPE} \\
\mathit{CommChannel}' \\
\hline
id' = id? \\
type' = type? \\
events' = \langle\rangle \\
\hline
\end{array}
$$

Bien sûr, cette initialisation est localisée et doit donc être promue à l'état du comportement, selon deux schémas. Tout d'abord un schéma de promotion pour l'ajout d'un canal entrant :

$$
\begin{array}{|l}
\textit{AddInputChannel} \\
\hline
\Delta \mathit{DefaultBehavior} \\
id? : \mathit{IDENTIFIER} \\
type? : \mathbb{P}\, \mathit{TYPE} \\
\hline
\exists\, \mathit{InitCommChannel} \bullet in\_channels' = in\_channels \cup \{\Theta\, \mathit{CommChannel}'\} \\
out\_channels' = out\_channels \\
in\_queue' = in\_queue \\
out\_queue' = out\_queue \\
funcs' = funcs \\
\hline
\end{array}
$$

Ce schéma de promotion ressemble à l'ajout de connexion effectif, détaillé page 76. De la même façon, l'ajout d'un canal sortant s'exprimera de la façon suivante :

$$
\begin{array}{|l}
\textit{AddInputChannel} \\
\hline
\Delta \mathit{DefaultBehavior} \\
id? : \mathit{IDENTIFIER} \\
type? : \mathbb{P}\, \mathit{TYPE} \\
\hline
\exists\, \mathit{InitCommChannel} \bullet out\_channels' = out\_channels \cup \{\Theta\, \mathit{CommChannel}'\} \\
in\_channels' = in\_channels \\
in\_queue' = in\_queue \\
out\_queue' = out\_queue \\
funcs' = funcs \\
\hline
\end{array}
$$

Les opérations disponibles sur ces canaux sont relativement simples; il s'agira d'ajouter ou d'enlever un événement depuis ou vers le canal. Pour l'ajout, reposons-nous sur une solution simple :

$$
\begin{array}{|l}
\textit{PutCommChannel} \\
\hline
\Delta \mathit{CommChannel} \\
event? : \mathit{Event} \\
\hline
\exists\, t : type \bullet event?.type \preceq t \\
events' = \{event?\} \frown events \\
type' = type \\
id' = id \\
\hline
\end{array}
$$

L'événement *event*? est ajouté en tête de la séquence d'événements, uniquement bien sûr si son type est valide pour le canal considéré.

D'un point de vue global, une telle émission effective d'événement sur un canal de communication doit correspondre à la modification d'un canal sortant pour un composant donné. Il nous faut donc décrire un schéma de promotion pour rattacher cette opération locale à l'état global qui nous intéresse :

$$
\begin{array}{|l}
\hline
\textit{ChannelOutPromote} \\
\Delta \textit{DefaultBehavior} \\
\textit{CommChannel}' \\
\textit{channel}? : \textit{CommChannel} \\
\hline
\textit{channel}? \in \textit{out\_channels} \\
\textit{out\_channels}' = \textit{out\_channels} \setminus \{\textit{channel}?\} \cup \{\Theta \textit{CommChannel}'\} \\
\hline
\end{array}
$$

Cette promotion un peu «technique» décrit simplement le remplacement du canal *channel*? (s'il fait bien partie des canaux sortant) par le nouveau schéma caractéristique «créé» (par exemple) par *PutCommChannel* et nommé $\Theta \textit{CommChannel}'$. Ainsi, pour exprimer qu'un canal sortant *channel*? est «vidé» d'un événement, nous écrirons :

$$\textit{PutCommChannel} \wedge \textit{ChannelOutPromote} \setminus (\textit{event}, \textit{type}, \textit{id}, \textit{event}', \textit{type}', \textit{id}')$$

Pour le retrait (en sortie) d'un événement du canal, plusieurs choix sont possibles selon la sémantique de communication désirée :

- communications avec ou sans ordre : la récupération se fait selon l'ordre d'émission ou un ordre quelconque,
- communications fiables ou non : les événements perdus peuvent être modélisés par un retrait non-déterministe.

Nous verrons que notre prototype opérationnel se base sur le protocole de communication TCP qui - grossièrement - respecte à la fois les contraintes d'ordre et de fiabilité. Ainsi, avec TCP, deux paquets transmis dans un ordre précis sont à la fois *bien* reçus et dans le *même ordre*. Dans ce cas, le retrait d'un événement ordonné et déterministe peut s'écrire :

$$
\begin{array}{|l}
\hline
\textit{GetCommChannel} \\
\Delta \textit{CommChannel} \\
\textit{event}! : \textit{Event} \\
\hline
\textit{events} \neq \langle\rangle \\
\textit{event}! = \mathrm{last}\ \textit{events} \\
\textit{events}' = \mathrm{front}\ \textit{events} \\
\textit{type}' = \textit{type} \\
\textit{id}' = \textit{id} \\
\hline
\end{array}
$$

L'événement «reçu», représenté par la variable de sortie *event*! est donc le dernier élément de la séquence «transportée». Le retrait est rendu effectif en indiquant que la séquence après modification correspond au reste de la séquence de départ (*front events* correspond à la séquence *events* privée de son dernier élément). Rappelons que ceci est une définition de propriété et non la description d'un algorithme. Bien sûr, le retrait d'un événement ne peut se produire que si la séquence *events* est non-vide, d'où la précondition du schéma.

Ce schéma d'opération locale doit également être promu au schéma *DefaultBehavior* de la manière suivante :

$$
\begin{array}{|l}
\hline
\textit{ChannelInPromote} \\
\Delta \textit{DefaultBehavior} \\
\Delta \textit{CommChannel} \\
\hline
\Theta \textit{CommChannel} \in \textit{in\_channels} \\
\textit{in\_channels}' = \textit{in\_channels} \setminus \{\Theta \textit{CommChannel}\} \cup \{\Theta \textit{CommChannel}'\} \\
\hline
\end{array}
$$

La principale différence ici est que le canal «à vider» est choisi de façon non-déterministe et non plus explicite. Du point de vue du comportement d'un composant, la réception effective d'un événement s'écrira donc :

$$GetCommChannel \land ChannelInPromote \setminus (event, type, id, event', type', id')$$

Dans certains cas (par exemple, si l'on se base sur le protocole UDP), il peut être intéressant de définir une propriété de transmission sans ordre. Dans ce cas, nous ne pouvons rien dire sur la position de l'élément retiré. Pour ce faire, nous devons tout d'abord proposer un schéma générique de choix non-déterministe (que nous réutiliserons à de nombreux endroits). Ce schéma correspond à la fonction générique *oneElement* décrite ci-dessous qui permet d'extraire, de façon non-déterministe, un élément depuis un ensemble :

$$\begin{array}{l}
[X] \\
\hline
oneElement : \mathbb{P}\, X \nrightarrow X \\
\hline
\forall\, y : \operatorname{ran} oneElement \bullet y \neq \emptyset \\
oneElement(E) \in E
\end{array}$$

oneElement est une fonction partielle car l'ensemble vide n'est pas une image «possible». En d'autres termes, cette fonction ne «marche» que pour des ensembles non-vides et ne peut retourner qu'un élément différent de l'ensemble vide[8].

Armé de ce «moteur» non-déterministe, la communication sans ordre est modélisable de façon assez simple :

$$\begin{array}{l}
GetCommChannelUnOrdered \\
\hline
\Delta CommChannel \\
event! : Event \\
\hline
events \neq \langle\rangle \\
\textbf{let } selected_index == oneElement(\{n : \mathbb{N} \bullet n \leq \#events\}) \bullet \\
\quad events' = events \setminus \{selected_index \mapsto event!\}
\end{array}$$

Contrairement au cas ordonné, l'événement à retirer est choisi ci-dessus par son indice pris «au hasard» dans le domaine de la séquence *events*[9].

Pour ajouter un caractère non-déterministe à la communication, nous pouvons dans un premier temps ajouter une opération qui «fait disparaître» des événements au hasard dans la séquence :

$$\begin{array}{l}
DisturbCommChannel \\
\hline
\Delta CommChannel \\
\hline
events \neq \langle\rangle \\
\textbf{let } lost_events == oneElement(\mathbb{P}\, events \setminus \{\emptyset\}) \bullet \\
\quad events' = events \setminus lost_events
\end{array}$$

Ainsi, pour obtenir une communication ordonnée non fiable nous utiliserons le schéma qui compose la réception ordonnée avec la possibilité de perdre des événements :

$$UnreliableOrdGetCommChannel \mathrel{\widehat=} GetCommChannel \land DisturbCommChannel$$

De la même façon, la communication non-ordonnée et non-fiable (dont nous ne pourrons donc pas dire grand chose) pourra s'écrire :

$$UnreliableUnOrdGetCommChannel \mathrel{\widehat=} GetCommChannelUnOrdered \land DisturbCommChannel$$

Nous pourrions également modifier les événements en changeant les informations associées, dupliquer des événements, introduire un modèle stochastique, etc. Il existe de nombreuses variantes de communications et au moins autant de façon de les modéliser. Rappelons que dans notre étude, nous resterons dans le cadre (certes simplifié) de la communication fiable et ordonnée. Cependant, lorsque la propriété d'ordre sera déterminante pour une problématique donnée (par exemple dans le cadre d'interactions liées à la résistance aux pannes), il sera possible - voir essentiel - de ne pas se reposer sur la couche de communication bas niveau et donc d'ajouter dans le modèle des outils pour fiabiliser les communications [GGM97].

8. L'ensemble vide possède l'importante propriété d'appartenir à tout produit d'ensembles. Ceci découle du fait que l'ensemble vide est un sous-ensemble de tout ensemble dans l'axiomatique de Zermelo-Frænckel ($\forall X \bullet \emptyset \subseteq X \Rightarrow \forall X \bullet \emptyset \in \mathbb{P}\, X$)

9. Une séquence Z est définie comme une fonction de $\mathbb{Z}$ vers le type des éléments, ici *Event*.

Schéma de fonctionnalité

Dans le langage Scope, nous représentions une fonctionnalité de composant par un bloc when, activé lors de la réception (effective) d'un événement du type correspondant à cette fonctionnalité (ou un sous-type). Le corps de ce bloc «réactif» était composé de code Java standard. En guise d'exemple, considérons le cas d'un filtrage de médiateur quelque peu étendu :

```
when (CPUInfoEvent info) {
  if(info.getImmediateValue()>_threshold) {
    send(new ExceptionEvent("Overload"), info.getSenderId());
    send(info);
  } else send(info);
}
```

Ici, en plus du filtrage en fonction du type, une exception peut être levée pour le composant source si l'indice reçu dépasse un certain seuil _threshold. D'un point de vue formel, seul nous intéresse l'ensemble des effets liés à l'exécution d'une telle fonctionnalité sur le comportement du composant; nous parlerons d'*activation* de fonctionnalité. Pour l'instant, les activations ne concernent que les émissions d'événement. Dans ce cas, notre exemple dénote deux activations différentes : la séquence «émission (avec référence explicite) d'une exception et réémission de l'événement reçu»[10] si le seuil est dépassé. Dans le cas contraire, la séquence est atomique: «réémission simple de l'événement reçu».

Ceci nous conduit à définir le schéma de fonctionnalité suivant :

$$\begin{array}{|l} \hline \textit{Functionality} \\ \textit{type} : \textit{TYPE} \\ \textit{activation} : \mathbb{P}\operatorname{seq}\textit{Action} \\ \hline \end{array}$$

La variable *type* désigne le type d'événement permettant d'activer la fonctionnalité. Les activations associées, définies par la variable *activation*, correspondent à des séquences d'actions associées au schéma *Action*. Ce dernier est pour l'instant défini de la façon suivante :

$$\textit{Action} \mathrel{\widehat{=}} \textit{SendAction} \lor \textit{SendExplicitAction}$$

Du point de vue du modèle de comportement canonique, nous ne nous intéressons donc qu'aux émissions d'événements avec désignation implicite (*SendAction*) ou explicite (*SendExplicitAction*) des récepteurs.

Pour notre exemple, la fonctionnalité *filtre_ex* sera définie ainsi :

$$\textit{filtre\_ex} \mathrel{\widehat{=}} \langle \textit{type} \Rrightarrow \textit{InfoEvent}, \textit{activation} \Rrightarrow \{s_1, s_2\} \rangle$$

où s_1 et s_2 sont des séquences d'opérations, de types :

- $\textit{SendExplicitAction} \fatsemi \textit{SendAction}$ pour s_1,
- *SendAction* pour s_2

Nous détaillerons les schémas *SendExplicitAction* et *SendAction* par la suite.

Initialisation de comportement

Le schéma correspondant à l'état initial du comportement d'un composant est donné ci-dessous :

$$\begin{array}{|l} \hline \textit{InitDefaultBehavior} \\ \textit{DefaultBehavior}' \\ \hline \textit{in\_channels}' = \emptyset \\ \textit{out\_channels}' = \emptyset \\ \textit{in\_queue}' = \langle\rangle \\ \textit{out\_queue}' = \langle\rangle \\ \hline \end{array}$$

10. En environnement asynchrone, deux émissions successives sont - grossièrement - simultanées, d'où l'emploi de la conjonction *et*. Cependant, au niveau formel, il s'agira réellement de composition séquentielle.

Suivant notre philosophie du «100% dynamique», il n'existe aucun canal de communication établi au moment de l'initialisation d'un composant. De la même façon, les systèmes de files d'attente sont également vides. En revanche, seul le compilateur Scope peut donner des informations sur les fonctionnalités associées au composant, ce qui nous empêche de les contraindre au niveau du modèle de comportement (la variable $functs'$ est donc «libre»).

Puisque le schéma d'état du comportement ne comporte pas d'invariant proprement dit, il est clair que notre schéma d'initialisation en respecte les propriétés (théorème d'initialisation implicite).

7.2.3 Opérations comportementales

A chaque méta-objet impliqué dans le «parcours» d'un événement au sein d'un composant correspond une opération formelle sur le schéma d'état du comportement. Ces opérations décrivent un invariant que toute implémentation sous forme de méta-objet doit respecter.

La première de ces opérations concerne la réception des événements acheminés par les canaux de communication entrant. De fait, nous avons déjà modélisé l'extraction d'un événement sur un canal de communication :

$$DefaultMetaReceive \mathrel{\widehat{=}} GetCommChannel \land ChannelInPromote \setminus (event, type, id, event', type', id')$$

La mise en file d'attente d'un événement est une opération également très simple dans le cas par défaut. Il s'agit simplement de placer l'événement en tête de la file d'attente en entrée du composant. Ceci s'écrit simplement :

$$\begin{array}{|l}
\multicolumn{1}{l}{DefaultMetaInQueue} \\
\Delta DefaultBehavior \\
event? : Event \\
\hline
in_queue' = \langle event? \rangle \frown in_queue \\
\hline
\end{array}$$

L'événement à stocker, nommé $event?$, est tout d'abord placé comme élément unique d'une séquence , elle-même concaténée à la séquence des événements en file d'attente in_queue. Ceci construit la nouvelle file d'attente in_queue' après activation de l'opération.

Ce schéma nous permet de définir la «procédure» complète de réception d'événement, formée d'une séquence d'extraction et de mise en file d'attente :

$$DefaultReceive \mathrel{\widehat{=}} DefaultMetaReceive \gg DefaultMetaInQueue$$

L'événement extrait du canal entrant, représenté par la variable de sortie $event!$ du schéma $DefaultMetaReceive$, est renommé en $event?$ pour servir d'entrée au schéma $DefaultMetaInQueue$.

Une fois reçus et mis en file d'attente, les événements en sont extraits par l'opération $DefaultMetaInFetch$ définie de la façon suivante :

$$\begin{array}{|l}
\multicolumn{1}{l}{DefaultMetaInFetch} \\
\Delta DefaultBehavior \\
event! : Event \\
\hline
\# in_queue > 0 \\
event! = \mathrm{last}\ in_queue \\
in_queue' = \mathrm{front}\ in_queue \\
\hline
\end{array}$$

L'événement extrait est représenté par la variable de sortie $event!$. Cet événement correspond au dernier élément de la file d'attente (sémantique *fifo*); élément qui n'existe bien sûr que si la file est non vide (# est l'opérateur de cardinalité en Z). Finalement, la file doit être mise à jour, on y conserve tous les éléments sauf le dernier (opérateur *front*).

Pour décrire la partie exécution du comportement, rappelons-nous les deux opérations modélisées à ce niveau :

- *SendAction* pour l'envoi implicite d'un événement,

– *SendExplicitAction* pour l'envoi explicite.

Dans les deux cas, il s'agit de préparer un événement à mettre en file d'attente (en sortie). Pour l'envoi implicite, le schéma utilisé est le suivant :

$$\begin{array}{|l}
\hline
\textit{SendAction} \\
\textit{DefaultBehavior} \\
\textit{event?} : \textit{Event} \\
\textit{event!} : \textit{Event} \\
\hline
\textit{event!.type} = \textit{event?.type} \\
\textit{event!.sender_id} = \textit{id} \\
\textit{event!.receiver_id} = \emptyset \\
\hline
\end{array}$$

Cette définition «réécrit» donc l'événement à émettre en positionnant convenablement les variables *sender_id* (identificateur du composant émetteur) et *receiver_id* (récepteurs explicites, ici vide).

Pour l'envoi explicite, la modification est la suivante :

$$\begin{array}{|l}
\hline
\textit{SendExplicitAction} \\
\textit{DefaultBehavior} \\
\textit{event?} : \textit{Event} \\
\textit{rids?} : \mathbb{P}\,\textit{IDENTIFIER} \\
\textit{event!} : \textit{Event} \\
\hline
\textit{event!.type} = \textit{event?.type} \\
\textit{event!.sender_id} = \textit{id} \\
\textit{event!.receiver_id} = \textit{rids?} \\
\hline
\end{array}$$

La mise en file d'attente en sortie est ensuite décrite de la façon suivante :

$$\begin{array}{|l}
\hline
\textit{DefaultMetaOutQueue} \\
\Delta \textit{DefaultBehavior} \\
\textit{event?} : \textit{Event} \\
\hline
\textit{out_queue}' = \langle \textit{event?} \rangle \frown \textit{out_queue} \\
\hline
\end{array}$$

Le but de l'opération *DefaultMetaExec* consiste ensuite à sélectionner une suite d'actions résultant de l'activation d'une fonctionnalité correspondant à l'extraction d'un événement depuis le système de file d'attente en entrée.

Dans un premier temps, définissons le schéma de sélection des actions :

$$\begin{array}{|l}
\hline
\textit{SelectActions} \\
\textit{DefaultBehavior} \\
\textit{event?} : \textit{Event} \\
\textit{actions!} : \mathrm{seq}\,\textit{Action} \\
\hline
\mathbf{let}\ \textit{fun_types} == \{\textit{fs} : \textit{functs} \bullet \textit{fs.type}\} \bullet \\
\quad \mathbf{let}\ \textit{best_type} == \mu\, t : \textit{fun_types} \bullet t = \textit{tmin}(\{\textit{ft} : \textit{fun_types} \mid \textit{ft} \succeq \textit{event?.type}\}) \bullet \\
\quad\quad \mathbf{let}\ \textit{all_actions} == (\mu\, \textit{fs} : \textit{functs} \bullet \textit{fs.type} = \textit{best_type}).\textit{activation} \bullet \\
\quad\quad\quad \textit{actions!} \in \textit{all_actions} \\
\hline
\end{array}$$

A partir d'un événement entrant *event?*, l'opération *SelectActions* sélectionne tout d'abord dans la variable *fun_types* tous les types de fonctionnalités définis pour le composant considéré. Nous extrayons ensuite de ces types les super-types minimaux du type de l'événement à traiter. Pour que l'opérateur μ ait un sens, il faut que l'ensemble résultant soit bien défini et qu'il se résume de plus à un seul élément. Ce lemme anticipe le «rattachement» du modèle comportemental au modèle formel[11], décrit par la suite.

Une fois le «meilleur» type de fonctionnalité sélectionné, nous pouvons récupérer dans *all_actions* toutes les séquences d'opérations possibles pour le type d'événement considéré. Les actions finalement sélectionnées correspondent à un choix non-déterministe (au niveau du modèle formel) dans les séquences d'actions possibles.

Nous pouvons maintenant décrire le schéma *DefaultMetaExec* par composition, depuis l'extraction dans le système de file d'attente jusqu'aux mises en file d'attente en sortie :

$$
\begin{aligned}
&\mathit{DefaultMetaExec} \mathrel{\widehat=} (\mathit{DefaultMetaInFetch} \gg \mathit{SelectActions}) \gg \\
&\quad \mathrm{head}\ \mathit{actions}? \mathbin{⨟} \mathrm{head}\ \ \mathrm{tail}\ \mathit{actions}? \mathbin{⨟} \\
&\qquad \mathrm{head}\ \ \mathrm{tail}\ \ \mathrm{tail}\ \mathit{actions} \mathbin{⨟} \\
&\qquad\quad \ldots\,\text{[12]} \mathbin{⨟} \\
&\qquad \mathrm{last}\ \mathit{actions}?
\end{aligned}
$$

L'exécution globale correspond donc à une sorte de «processus séquentiel» commençant par l'extraction d'un événement dans le système de file d'attente. Une séquence d'actions correspondant à l'activation de la fonctionnalité associée au type de l'événement est ensuite «exécutée», toujours séquentiellement.

Rappelons que les actions peuvent être des émissions implicites ou explicites d'événements. Dans chaque cas, un événement est placé en file d'attente par *DefaultMetaOutQueue*. Il nous reste donc à décrire la phase d'extraction en sortie et d'émission effective des événements.

Tout d'abord, l'extraction est prise en charge par l'opération *DefaultMetaOutFetch* définie de la façon suivante :

$$
\begin{array}{|l}
\hline
\textit{DefaultMetaOutFetch} \\
\Delta \mathit{DefaultBehavior} \\
\mathit{event}! : \mathit{Event} \\
\hline
\#\mathit{out\_queue} > 0 \\
\mathit{event}! = \mathrm{last}\ \mathit{out\_queue} \\
\mathit{out\_queue}' = \mathrm{front}\ \mathit{out\_queue} \\
\hline
\end{array}
$$

Cette définition est quasiment identique au schéma *DefaultMetaInFetch*. L'opération associée d'émission effective est cependant plus complexe. Tout d'abord, il faut prendre en compte les deux catégories d'émissions : implicite et explicite. Commençons par l'envoi implicite, favorisé par le modèle :

$$
\begin{array}{|l}
\hline
\textit{ImplicitSendPart} \\
\mathit{DefaultBehavior} \\
\mathit{event}? : \mathit{Event} \\
\mathit{event}! : \mathit{Event} \\
\mathit{channels}! : \mathit{CommChannel} \\
\hline
\mathit{event}?.\mathit{receiver\_id} = \emptyset \\
\mathit{event}! = \mathit{event}? \\
\mathit{channels}! = \{ c : \mathit{out\_channels} \mid (\exists\, t : c.\mathit{type} \bullet t \preceq \mathit{event}?.\mathit{type}) \} \\
\hline
\end{array}
$$

Cette opération prend en entrée l'événement à diffuser *event*?. Pour vérifier qu'il s'agit bien d'une requête de diffusion implicite, la précondition du schéma «teste» si le contenu de la variable *receiver_id* de l'événement est bien vide. La variable *event*! représente une «copie» de l'événement; et *channels*! dénote pour sa part l'ensemble des canaux sur lesquels cet événement doit être transmis. C'est ici que nous trouvons le «cœur» de notre système de diffusion multi-point en fonction du type. Au niveau formel, il s'agit simplement de sélectionner

11. Rappelons que nous avons établi informellement qu'il existe au plus une fonctionnalité pour un type d'événement donné.

12. Nous désirons dans ce schéma réaliser une composition séquentielle sur des schémas contenus dans une séquence. Malheureusement, il n'existe pas en Z d'opérateur permettant, par exemple, d'itérer sur les éléments d'une séquence. Cela paraît logique étant donné la nature «non opérationnelle» des spécifications mais l'opérateur de composition séquentielle ⨟ représente une sorte de «phénomène étrange» et pourtant nécessaire à la notation; il faudrait pouvoir le généraliser.

parmi l'ensemble des canaux de communication sortant (variable *out_channels* du comportement), ceux qui sont «compatibles» avec le type de l'événement à émettre, c'est à dire les canaux définissant au moins un sous-type effectif.

Pour l'émission explicite, la principale modification concerne donc le choix des canaux sortant, comme le montre le schéma correspondant :

$$
\begin{array}{|l}
\hline
\textit{ExplicitSendPart} \\
\textit{DefaultBehavior} \\
\textit{event?} : \textit{Event} \\
\textit{event!} : \textit{Event} \\
\textit{channels!} : \textit{CommChannel} \\
\hline
\textit{event?.reiver\_id} \neq \emptyset \\
\textit{event!} = \textit{event?} \\
\textit{channels!} = \{c : \textit{out\_channels} \mid (c.id \in \textit{event?.receiver\_id}) \wedge (\exists\, t : c.type \bullet t \preceq \textit{event?.type})\} \\
\hline
\end{array}
$$

Dans ce cas, un canal est sortant est sélectionné, non seulement en fonction de son type, mais également en vérifiant que l'identificateur du composant destination référencé est bien membre de l'ensemble des récepteurs désigné par $\textit{event?.receiver\_id}$.

Finalement, l'opération d'émission effective (indépendamment de la nature implicite ou explicite de la requête) consiste donc à émettre une copie de l'événement *event*? sur chaque canal référencé par *channels*?, selon le schéma suivant :

$$
\begin{array}{|l}
\hline
\textit{DefaultMetaSend} \\
\Delta \textit{DefaultBehavior} \\
\textit{event?} : \textit{Event} \\
\textit{channels?} : \textit{CommChannel} \\
\hline
\forall\, \textit{channel?} : \textit{channels?} \bullet (\textit{PutCommChannel} \wedge \textit{ChannelsOutPromote}) \\
\hline
\end{array}
$$

En conclusion, la phase d'extraction en sortie et d'émission sera définie par le schéma composé suivant :

$$
\begin{aligned}
&\textit{DefaultSend} \mathrel{\widehat{=}} \textit{DefaultMetaOutFetch} \gg (\textit{ImplicitSendPart} \vee \textit{ExplicitSendPart}) \gg \\
&\quad \textit{DefaultMetaSend}
\end{aligned}
$$

Réseaux d'activation comportementale

La méthode de spécification sous-jacente de la notation Z est relativement libre concernant l'interprétation des schémas. Le cas le plus simple concerne la spécification de systèmes à base d'états et de transitions entre les états. Ce sont par exemple tous les schémas d'état et d'opération que nous avons construits jusqu'à présent. Cependant, les schémas d'opération, s'ils donnent des informations sur la transition (état avant et après), ne fournissent aucune règle sur l'exécution proprement dite d'une transition.

Sans information supplémentaire, l'unique moyen de décrire une exécution relative à une spécification Z composée de schémas d'états et de schémas d'opérations est d'indiquer que seule une séquence de transitions indépendantes est réalisable.

En d'autres termes, les spécifications Z sous-tendent - sans précision supplémentaire - une opérationnalisation de type séquentiel. Pour le modèle structurel, ceci ne représente pas vraiment une limite puisque même si dans la pratique, nous souhaiterions autoriser des modifications concurrentes, ceci peut être effectué - de façon informelle bien sûr - au niveau de l'implémentation[13].

Au niveau du comportement d'un composant, en revanche, il est très important de pouvoir modéliser le fait que certaines opérations sur l'état du comportement s'effectuent de manière concurrente. Il nous faut donc trouver un moyen pour décrire une activation concurrente d'opérations Z.

13. Rappelons que notre modèle formel n'est pas raffiné jusqu'à l'implémentation, étant donnée la nature colossale (voir impossible) de la tâche.

Nous établissons dans cette section des modèles qui permettent de décrire des activations concurrentes d'opération Z sous la forme de réseaux de Petri. Cette fusion Z/Réseau de Petri est encore expérimentale mais nous présentons ici les premiers résultats (encourageant) de cette approche hybride.

Dans ces réseaux d'activation, nous trouvons des *places*, des *marques* (marquées par des *jetons*), des transitions et des arcs reliant les places aux transitions (et vice-versa).

Les places peuvent représenter des activations d'opérations Z ou des états «utilitaires» du réseau de Petri. Si une place correspondant à un schéma Z (ou *ZPlace*) est marquée, alors la transition correspondante est «opérationnalisable» dans cet état du réseau de Petri (il faut donc être sûr que l'opération Z est cohérente dans cet état). Si deux ZPlaces, par exemple $Operation_1$ et $Operation_2$ sont marquées à un moment donné, il est donc important de s'assurer que leur combinaison est cohérente. Ainsi, dans un réseau où ces deux places peuvent être marquées «en même temps», il faut pouvoir prouver :

$$(\text{pre } Operation_1) \land (\text{pre } Operation_2) \land (\text{pre } Operation_1 \land Operation_2)$$

On s'assure ainsi que les opérations sont cohérentes prises individuellement ou simultanément. La dernière preuve n'est valide que si les deux opérations n'interfèrent pas, c'est-à-dire si la composition séquentielle revient à la composition simple. Sinon, la preuve est plus «lourde» :

$$(\text{pre } Operation_1 \mathbin{⨾} Operation_2) \land (\text{pre } Operation_2 \mathbin{⨾} Operation_1)$$

Que les opérations soient composées dans «un sens» ou dans l'autre, l'activation concurrente doit maintenir la cohérence de l'état sur lequel elles agissent.

Dans le cas général, on décrira cette obligation de preuve de la façon suivante :

$$Operation_1 \parallel Operation_2$$

Si une opération est marquée par plusieurs jetons, alors elle peut être activée plusieurs fois simultanément. Dans ce cas, la preuve à conduire nécessite un petit «truc». Chaque activation doit être renommée (par exemple, par un suffixe numéral) :

Operation devient $Operation_1$ (pour la première activation), $Operation_2$, etc. La preuve est ensuite la même que pour les opérations «normales». Il faut autant de schémas renommés qu'il existe d'activations possibles (place bornée) ou, dans le pire des cas, il faut faire la preuve pour tout entier positif (place non bornée).

Les opérations de preuve sont donc à la fois plus nombreuses et plus complexes dans le cadre de l'activation parallèle. Dans la pratique, les schémas restent suffisamment simples pour laisser certaines preuves «de côté». Nous donnerons cependant quelques exemples de telles preuves dans cette section.

Pour notre modèle de comportement, nous devons donner l'ensemble des réseaux de Petri permettant de décrire les différentes possibilités d'activation en parallèle des méta-objets régissant la «vie» des événements. En particulier, nous devons exprimer formellement que la phase de réception/mise en file d'attente est réalisée en concurrence avec la phase d'extraction/exécution. Dans le même ordre d'idée, les phases de sortie exécution/mise en file d'attente et d'extraction/émission sont «parallélisables». En d'autres termes, nous devons donner un véritable sens au comportement asynchrone des composants.

Commençons par décrire la phase de réception, considérée isolément. Il est clair que les activations respectives des opérations *DefaultMetaReceive* et *DefaultMetaInQueue* sont exclusives; nous ne pouvons mettre en file d'attente un événement qui n'a pas été préalablement récupéré sur le réseau.

Le réseau de Petri comparant les activations de ces deux opérations est donné en figure 7.7 page suivante.

Ce réseau comporte deux états *ZStates* (en grisé sur la figure) correspondant aux deux schémas d'opérations comportementales mis en jeu pour la phase de réception. Nous introduisons de plus un état utilitaire (sur fond blanc) permettant d'«expliquer» que le déclenchement de la réception correspond à la présence d'un événement dans un canal entrant. Cet état n'a pas d'impact sur la formalisation en Z mais permet de mieux comprendre le réseau de Petri.

Pour franchir une transition, il faut que les places *sources* (situées en amont de la transition) contiennent chacune un nombre de jetons égal au poids de l'arc entrant qui les relie à la transition. Sans indication supplémentaire, le poids d'un arc est de 1. La place *EventInChannel* contenant un jeton à l'origine, il est possible de franchir la transition qui permet d'activer *DefaultMetaReceive*. Le nombre de jetons générés en sortie de transition dans les places *destinations* (situées en aval) correspond au poids des arcs sortant. Encore

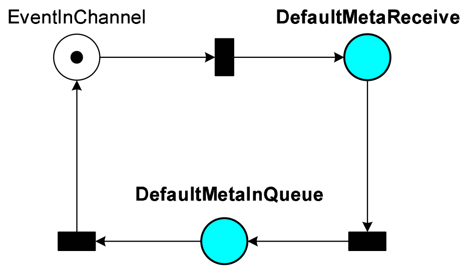


FIG. 7.7 – *Réseau d'activation de la phase de réception*

une fois, le nombre de jeton généré dans *DefaultMetaReceive* par le franchissement est unitaire. Il est évident sur ce réseau que nos deux *ZPlaces* sont 1-bornées (marquage maximum de un jeton) et qu'elles ne peuvent être marquées «en même temps» (ou plutôt dans un état donné du réseau de Petri). Si on pose la fonction de marquage m qui associe à chaque place le nombre de jetons qu'elle contient, les propriétés précédentes correspondent aux énoncés formels :

- places 1-bornées : $m(DefaultMetaReceive) \leq 1 \wedge m(DefaultMetaInQueue) \leq 1$
- exclusion mutuelle : $m(DefaultMetaReceive) + m(DefaultMetaInQueue) \leq 1$

Ces propriétés peuvent être prouvées par analyse de flot ou plus simplement en utilisant l'un des nombreux outils disponibles pour les modèles à base de réseau de Petri[14]. Lorsque les places sont N-bornées, la valeur N est indiquée à côté de la place. Lorsque cette borne est 1, il s'agit du cas par défaut et rien n'est représenté (cas de la figure 7.7).

En conclusion, les deux opérations ne sont pas activables concurremment, les deux obligations de preuve résultantes sont donc simplement :

pre *DefaultMetaReceive*
pre *DefaultMetaInQueue*

✎Preuve **A.3.1 page 234** : *Preuve de cohérence de l'opération DefaultMetaReceive*

✎Preuve **A.3.2 page 236** : *Preuve de cohérence de l'opération DefaultMetaInQueue*

La file d'attente en entrée est une ressource partagée entre la phase de réception et la phase d'exécution. Les deux opérations en conflit potentiel sont *DefaultMetaInQueue* (producteur) et *DefaultMetaInFetch* (consommateur). Le schéma d'activation associé à cette phase de synchronisation est représenté en figure 7.8 page ci-contre.

La place utilitaire *InQueueLock* est utilisée pour synchroniser les opérations Z *DefaultMetaInQueue* et *DefaultMetaInFetch*. Par analyse structurelle simple, il est par exemple trivial de montrer la relation de marquage suivante :

$$m(DefaultMetaInQueue) + m(InQueueLock) + m(DefaultMetaInFetch) \leq 1$$

Ceci implique que les opérations *DefaultMetaInQueue* et *DefaultMetaInFetch* sont en *exclusion mutuelle* et ne peuvent «agir» simultanément sur le comportement du composant.

Ce schéma ajoute une obligation de preuve «standard» :

pre *DefaultMetaInFetch*

Mais également une obligation implicite induite par le fait qu'en joignant nos deux réseaux de Petri, il est possible que les places *DefaultMetaReceive* et *DefaultMetaInFetch* soient marquées simultanément. Il faut donc prouver l'activabilité parallèle des opérations.

14. Dans le cadre de notre thèse, nous avons principalement utilisé deux outils : HPSim 1.1(design graphique, simulation empirique) et DaNAMics (calculs d'invariant).

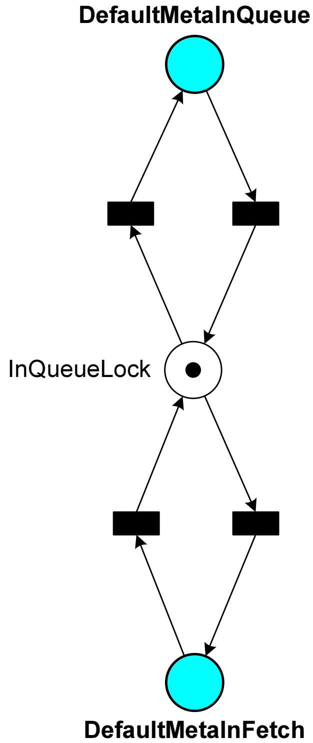


Fig. 7.8 – *Synchronisation sur le système de file d'attente en entrée*

✎Preuve **A.3.3 page 236** : *Preuve d'activabilité concurrente de DefaultMetaReceive et DefaultMetaInFetch*

Pour la partie exécution, après l'extraction d'un événement dans le système de file d'attente (*DefaultMetaInFetch*), toutes les opérations considérées sont séquentielles. Le réseau d'activation associé n'est donc pas important à ce niveau. En revanche, nous pouvons donner le réseau d'activation décrivant la synchronisation de l'accès à la file d'attente en sortie (cf. figure 7.9).

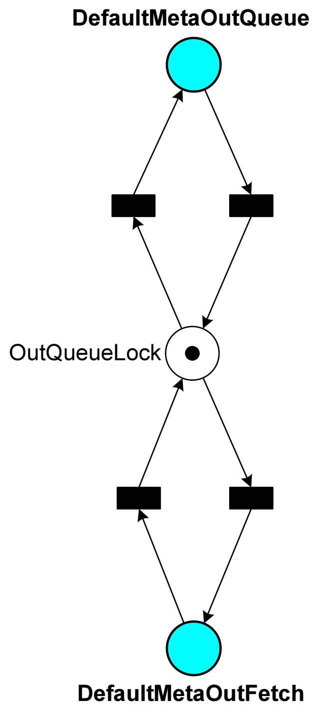


Fig. 7.9 – *Synchronisation sur le système de file d'attente en sortie*

Comme pour l'entrée, les opérations *DefaultMetaOutQueue* et *DefaultMetaOutFetch* ne peuvent se produire simultanément.

A partir de ces réseaux partiels, nous devons construire le réseau complet indiquant l'ensemble des synchronisations mises en jeu dans l'activation de notre modèle de comportement. Comme le montre la figure 7.10, ce réseau représente plus que la somme des réseaux partiels vus précédemment.

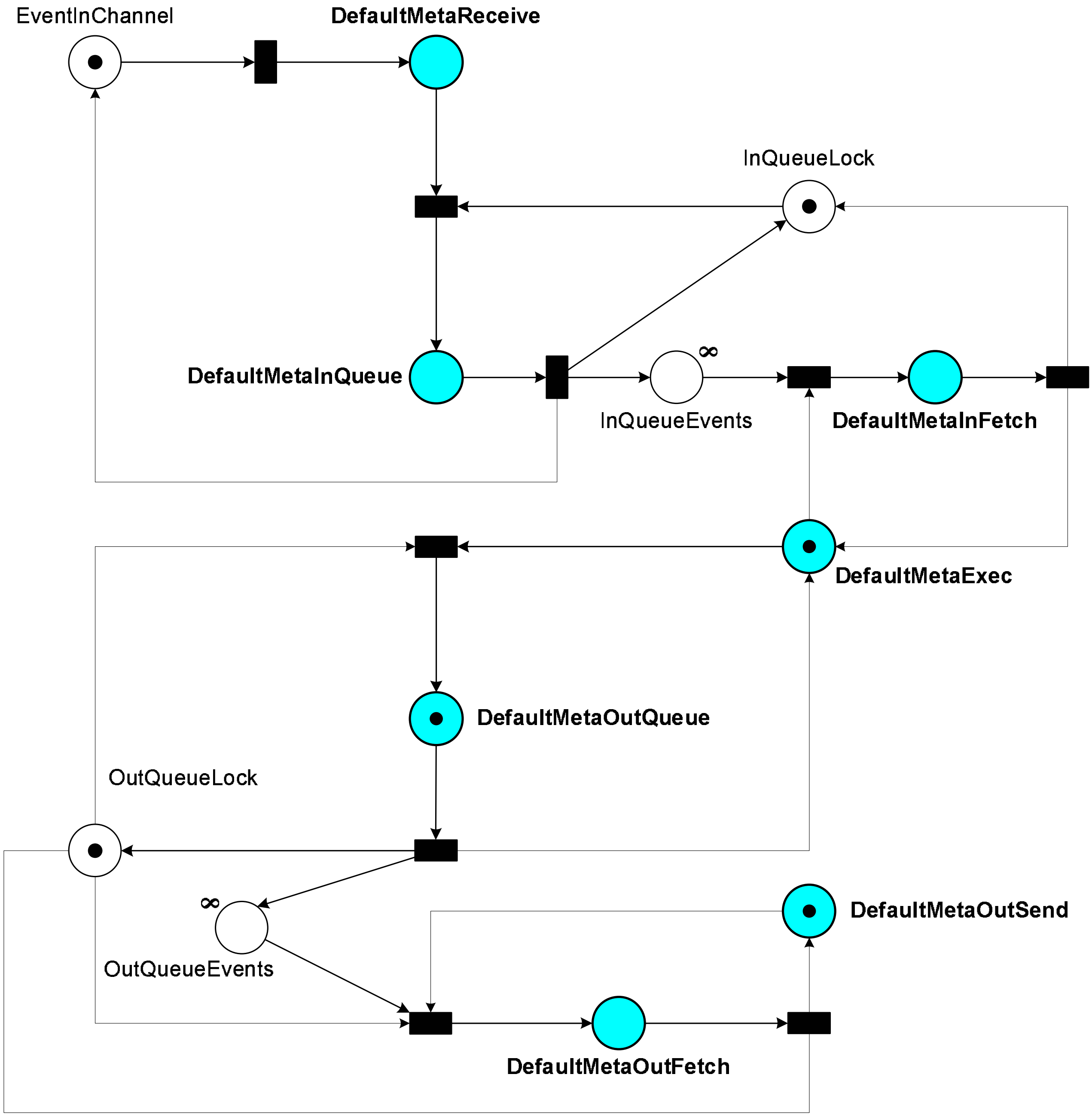


FIG. 7.10 – *Réseau d'activation concurrente du modèle de comportement par défaut*

Une des raisons de la relative complexité du réseau *composé* d'activation est qu'il est important de découpler les différentes synchronisations : ressource partagée (verrou), séquencement (mise en file d'attente uniquement après réception), etc.

Par exemple, l'opération *DefaultMetaInQueue* ne peut être activée que si le verrou associé est libre (*InQueueLock* marquée) et qu'un événement a été reçu (*DefaultMetaReceive* marquée). La transition en amont a donc besoin de deux jetons pour être franchissable. Bien sûr, l'opération *DefaultMetaInQueue* n'est activée qu'une seule fois donc un seul jeton est généré par la transition.

Pour *DefaultMetaInFetch*, nous ajoutons en plus de la prise de verrou un contrainte permettant de mieux

Invariant de réseau	**Obligation de preuve**
$m(DefMReceive) + m(DefMInQueue) \leq 1$	pre $DefMReceive$ $\wedge$ pre $DefMInQueue$
$m(DefMInQueue) + m(DefMInFetch) \leq 1$	pre $DefMInQueue$ $\wedge$ pre $DefMInFetch$
$m(DefMInFetch) + m(DefMExec) + m(DefMOutQueue) \leq 1$	pre $DefMExec$ $\wedge$ pre $DefMOutQueue$
$m(DefMOutQueue) + m(DefMOutFetch) \leq 1$	pre $DefMOutFetch$
$m(DefMOutFetch) + m(DefMetaSend) \leq 1$	pre $DefMetaSend$
$m(DefMReceive) + m(DefMInFetch) \leq 2$	$DefMReceive \parallel DefMInFetch$
$m(DefMReceive) + m(DefMExec) \leq 2$	$DefMReceive \parallel DefMExec$
$m(DefMReceive) + m(DefMOutQueue) \leq 2$	$DefMReceive \parallel DefMOutQueue$
$m(DefMReceive) + m(DefMSend) \leq 2$	$DefMReceive \parallel DefMSend$
$m(DefMInQueue) + m(DefMExec) \leq 2$	$DefMInQueue \parallel DefMExec$
$m(DefMInQueue) + m(DefMOutQueue) \leq 2$	$DefMInQueue \parallel DefMOutQueue$
$m(DefMInQueue) + m(DefMSend) \leq 2$	$DefMInQueue \parallel DefMSend$
$m(DefMInFetch) + m(DefMOutQueue) \leq 2$	$DefMInFetch \parallel DefMOutQueue$
$m(DefMInFetch) + m(DefMSend) \leq 2$	$DefMInFetch \parallel DefMSend$
$m(DefMExec) + m(DefMSend) \leq 2$	$DefMInFetch \parallel DefMSend$
$m(DefMOutQueue) + m(DefMSend) \leq 2$	$DefMOutQueue \parallel DefMSend$

TAB. 7.2 – *Modèle de comportement par défaut : obligations de preuve*

comprendre le réseau : la place *InQueueEvents* doit être marquée. Cette place dénote en fait les événements mis en file d'attente. Nous pouvons également remarquer que les opérations *DefaultMetaExec* et *DefaultMetaSend* sont activées à l'état initial du réseau. Ceci montre bien que ces méta-objets sont responsables des activités de traitement et d'émission. En revanche, *DefaultMetaOutQueue* n'est activée que si l'étage d'exécution lui donne la main (ou plutôt lui passe le jeton). Nous retrouvons ensuite notre contrainte d'accès à la file de sortie.

Par analyse structurelle, nous pouvons vérifier certaines propriétés intéressantes pour ce réseau :

- Propriété de *vivacité* : à tout moment, au moins une transition est franchissable.
- Propriété de *sûreté* : il n'existe pas d'état du réseau dans lequel une transition devient infranchissable de façon permanente.
- Places 1-bornées sauf *InQueueEvents* et *OutQueueEvents* qui sont non-bornées (repérables par leur étiquette ∞).

La dernière propriété indique que ce mode d'activation ne gère pas les capacités de files d'attente. Nous devrons attendre le chapitre 9 pour aborder ce problème.

L'analyse structurelle de ce réseau global permet de calculer l'ensemble des *ZPlaces* pouvant être marquées simultanément et d'en déduire nos obligations de preuve. La table 7.2 montre les calculs d'invariant que nous avons effectués pour en déduire l'ensemble des obligations de preuve permettant de vérifier le modèle de comportement par défaut. La plupart de ces preuves sont bien sûr triviales étant donné l'indépendance relative des méta-objets concurrents[15].

7.2.4 Modèle de composant raffiné

Le but de cette section est de définir le très important lien entre modèle structurel et modèle comportemental. Dans un premier temps, nous devons associer le schéma structurel local de composant à un modèle de comportement adéquat. Cette association correspond au schéma d'état *DefaultComponent* défini ainsi :

DefaultComponent

AbstractComponent
DefaultBehavior

$\forall\, inct : \{c : in\_channels \bullet c.type\} \bullet (\forall\, it : inct \bullet \exists\, t : in\_type \bullet t \succeq it)$
$\forall\, outct : \{c : out\_channels \bullet c.type\} \bullet (\forall\, ot : outct \bullet \exists\, t : out\_type \bullet t \succeq ot)$

$\forall\, t : in\_type \bullet \exists\, fs : functs \bullet t \preceq fs.type$

15. Par exemple, *DefaultMetaReceive* et *DefaultMetaSend* opèrent sur des parties disjointes de l'état du comportement et sont donc activables simultanément.

Le but du schéma de raffinement est de relier, au niveau de l'invariant, les variables introduites dans le schéma de départ (ici, le modèle de composant structurel) et le schéma raffiné (comportement). Les relations «évidentes» concernent en premier lieu les liens entre canaux de communication sur lesquels circulent les événements et les types déclarés au niveau du composant. Ainsi, les deux premières conditions de l'invariant stipulent que tous les types effectifs des canaux entrant et sortant doivent être compatibles avec le type du composant. Dans le même ordre d'idée, la dernière précondition précise qu'il existe au moins une fonctionnalité permettant de «comprendre» un événement reçu.

✎Preuve **A.3.4 page 237** : *Cohérence du raffinement d'état pour DefaultComponent*

Il est facile de montrer que les différentes opérations liées au modèle de comportement (méta-objets) s'étendent naturellement au schéma de raffinement. En effet, ces opérations ne modifient en rien la structure des composants et n'ont donc aucun impact sur les variables de type. En revanche, montrer que les opérations structurelles (ajout/retrait de composant, établissement de connexion, etc.) s'appliquent également au niveau raffiné est beaucoup plus complexe. Tout d'abord, ces opérations agissant au niveau de la structure modifient le système opérationnel entier et non uniquement un composant donné. Il nous faut donc proposer un schéma de système opérationnel prenant en compte l'intégration du comportement.

Le schéma de système opérationnel concret est le suivant :

OperationalSystem

$AbstractOperationalSystem$

$comps : DefaultComponent$

$\forall\, source : comps \bullet$
$\quad \forall\, cout : source.out\_channels \bullet$
$\quad\quad \exists_1\, destination : comps \bullet cout.id = destination.id \land$
$\quad\quad\quad \exists_1\, cin : destination.in\_channels \bullet cin.id = source.id$

Au niveau des définitions, seul le type de la variable *comps* est modifié par le raffinement. Les canaux de communication sont, par définition, partagés entre les composants source et destination des connexions. Plus précisément, il s'agit de l'unique lien «comportemental» avec l'extérieur. Il n'est donc pas étonnant de retrouver, au niveau du système opérationnel global, une forte contrainte à ce niveau. En l'occurrence, cette contrainte pose simplement qu'à tout canal sortant d'un composant source il existe une unique extrémité représentée par un canal entrant au niveau du composant destination pour la connexion considérée.

L'initialisation du schéma d'état *OperationalSystem* est identique à la version abstraite (schéma *InitAbstractOperationalSystem*).

Le passage du système opérationnel abstrait au système raffiné induit une modification relativement importante des primitives de connexion, puisque ces dernières devront désormais établir les canaux de communication reliant les composants, en plus de l'établissement «logique» des connexions structurelles. Pour cela, il faut composer les schémas de connexions structurelles avec l'établissement des canaux de communication entrant et sortant. En premier lieu, nous devons promouvoir les créations de canaux au niveau du système opérationnel entier. Comme les opérations de connexion définissent les composants *source*? et *destination*? concernés ainsi que le type effectif *T*? de connexion nous allons «partir» de ces variables pour construire nos schémas de promotion.

Commençons par définir un schéma de promotion pour la création d'un canal entrant en effectuant le lien entre le composant à modifier localement et sa «place» dans le système opérationnel :

AddInputChannelPromote

$\Delta\, OperationalSystem$

$id? : IDENTIFIER$
$type? : \mathbb{P}_1\, TYPE$

$\exists_1\, c : comps \bullet c.id = id?$

$\exists\, AddInputChannel \bullet comps' = comps \setminus \{\Theta DefaultComponent\} \cup \{\Theta DefaultComponent'\}$

Ici, il s'agit simplement d'associer les paramètres de création de canal entrant (identificateur de source *id*? et type effectif *type*?) avec un composant du système (première précondition). La création du canal correspond, pour ce composant au retrait de son état avant ajout du canal et à l'adjonction du nouveau composant modifié ensuite.

De la même manière, nous pouvons définir la promotion «en sortie» :

$$
\begin{array}{|l}
\textit{AddOutputChannelPromote} \\
\hline
\Delta\, \textit{OperationalSystem} \\
\textit{id}? : \textit{IDENTIFIER} \\
\textit{type}? : \mathbb{P}_1\ \textit{TYPE} \\
\hline
\exists_1\ c : \textit{comps} \bullet c.\textit{id} = \textit{id}? \\
\exists\, \textit{AddOutputChannel} \bullet \textit{comps}' = \textit{comps} \setminus \{\Theta \textit{DefaultComponent}\} \cup \{\Theta \textit{DefaultComponent}'\} \\
\hline
\end{array}
$$

Il est maintenant simple de faire le lien entre la connexion structurelle et la connexion complétée par l'impact comportemental. Par exemple nous obtenons pour la primitive de connexion par spécialisation le schéma suivant :

$$
\begin{aligned}
\textit{ConnectSpecialize} \mathrel{\widehat{=}} & \textit{AbsConnectSpecialize} \fatsemi (\textit{AddInputChannelPromote}[\textit{id}?/\textit{dest}?.\textit{id}, \textit{type}?/T?] \fatsemi \\
& \textit{AddOutputChannelPromote}[\textit{id}?/\textit{source}?.\textit{id}, \textit{type}?/T?])
\end{aligned}
$$

La traduction de ce schéma explique que la primitive de connexion par spécialisation complète correspond à la composition séquentielle de la primitive de connexion structurelle et de l'ajout successif des canaux entrant et sortant associés. Dans chaque cas, nous prenons soin d'associer le paramètre d'identification du composant pour la création des canaux avec les composant source (pour la sortie) et destination (pour l'entrée) considérés par la connexion (paramètres *source*? et *dest*? de *AbsConnectSpecialize*).

Une propriété fondamentale permettant de vérifier la cohérence de ce schéma opérationnel est de satisfaire l'invariant principal du schéma raffiné. Cet invariant impose aux canaux entrant et sortant créés pour une connexion effective donnée de correspondre. Cette propriété est respectée par notre primitive tout d'abord parce que le canal établi à la source l'est pour le type *type*? (renommage de T?), ce qui est également vrai pour la destination. De plus, les composants source et destination introduits sont bien ceux reliés par la connexion structurelle.

7.3 Exemple de spécialisation

L'intérêt de l'approche réflexive pour modéliser et implanter le comportement des composants est double. Tout d'abord, la séparation *de facto* entre les descriptions *fonctionnelles* (au niveau des fonctionnalités) et les descriptions *non-fonctionnelles* (au niveau méta-comportement) permet de se concentrer sur le *propos* d'un composant sans pour autant contraindre de façon trop drastique le mode d'exécution sous-jacent. Cela permet également de faire varier ces deux aspects indépendamment l'un de l'autre.

Il est important de préciser ce que nous entendons par propriété non-fonctionnelle qui n'est pas, selon nous, un concept *universel* mais plutôt la caractérisation d'une sorte de frontière très dépendante du système considéré. Si l'on se limite aux primitives d'émission implicite d'événement, il n'est pas possible d'agir directement *à l'extérieur* du composant considéré. Nous pouvons tout au plus «proposer» une information qu'un *alter ego* pourra interpréter à sa guise. De la même façon, nous ne pouvons pas intervenir (encore une fois directement) sur la façon dont ces événements «circulent» au sein du composant. Au niveau d'une fonctionnalité, il nous est seulement possible de lire les données contenues dans un événement et d'en tirer des conclusions *locales*, puis, éventuellement, de communiquer des informations induites à l'extérieur. La séparation non/fonctionnel se situe donc exactement - dans la plateforme Comet - à la frontière entre le contenu d'un événement et son traitement par le système.

Bien sûr, ceci ne se généralise pas forcément aux applications[16] mais la possibilité de spécialiser le comportement - via réalisation d'un niveau méta-comportement adapté - offre une large flexibilité concernant le positionnement de cette frontière.

16. La sécurité, concept souvent «promu» à l'état de propriété non-fonctionnelle peut, dans certains cas, devenir un aspect hautement fonctionnel. Imaginons par exemple la réalisation d'une application d'accès sécurisé à l'information (genre *PKI*). A notre avis, cette problématique du non-fonctionnel (ou des aspects [K+97]) est très dépendante du point de vue et ne peut - malgré l'intérêt indéniable des travaux afférents - s'étendre au titre de concept universel.

7.3.1 Modèle de comportement multi-session

Pour illustrer cette idée de spécialisation, nous allons créer dans cette section une variation du modèle de comportement canonique. Remarquons que dans ce dernier, seuls comptent les types des composants que l'on relie et des événements qu'ils s'échangent. Il n'est pas possible - en préservant la propriété d'indépendance structurelle - de distinguer les instances des composants interconnectés. En terme de comportement (ce qui nous intéresse ici), nous pouvons caractériser une autre «limite» : l'absence de concurrence interne. Tout comme les acteurs, les composants Comet sont concurrents mais ne gèrent - par défaut - qu'une seule activité en interne[17]. Si le modèle formel sous-jacent du comportement d'un acteur impose cette limite, ce n'est pas le cas de notre modèle, probablement plus pragmatique.

Ainsi, nous allons passer, par l'introduction du modèle *MSessionBehavior*, d'un comportement d'acteur à un comportement permettant de discriminer des sessions[18]. Pour simplifier le modèle, chaque session sera totalement indépendante des autres et se déroulera dans une activité qui lui est propre.

La figure 7.11 décrit la structure d'un composant multi-session. Tout d'abord, observons - et c'est bien sûr une obligation - que le niveau méta-comportement associé propose *au moins* le comportement canonique des composants, partagé par toute les sessions. Mais ces dernières disposent également de méta-objets spécifiques qui gèrent leurs événements spécifiques. Ainsi, une session aura ses propres systèmes de file d'attente en entrée et en sortie.

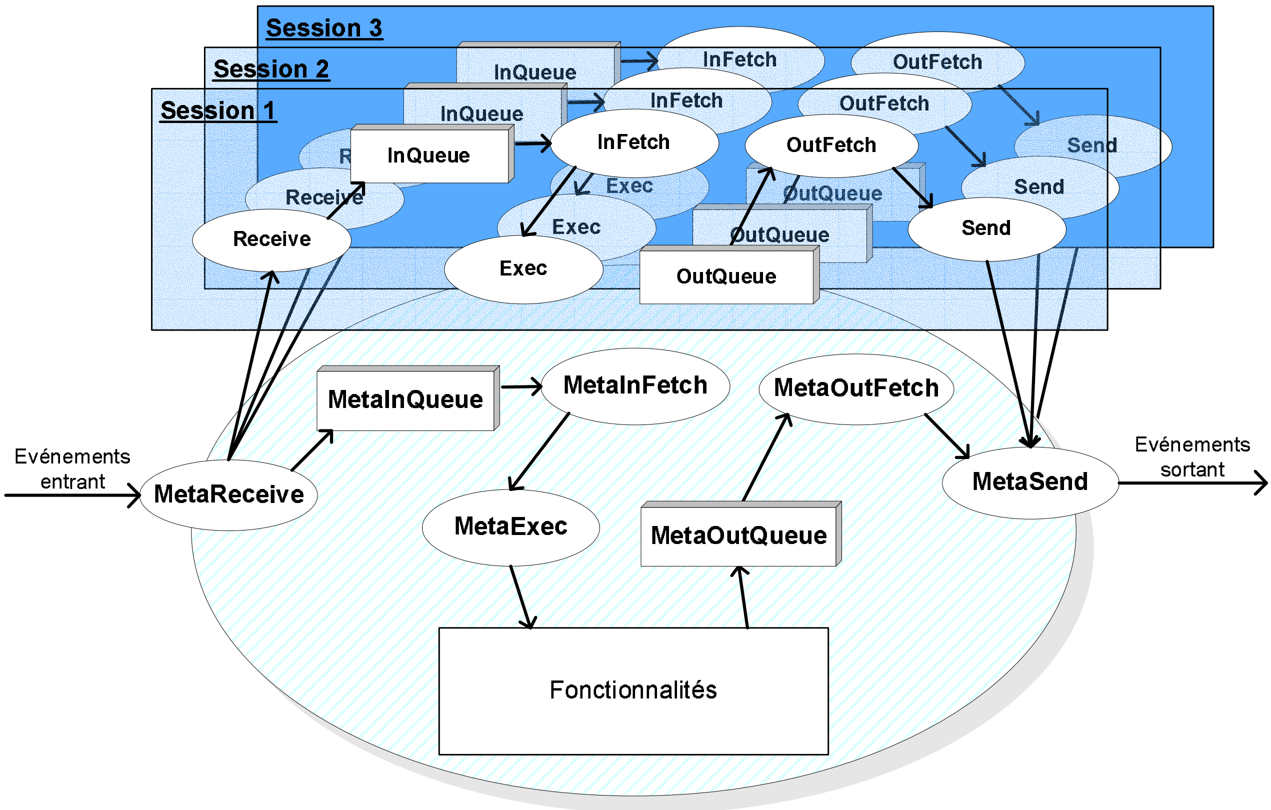


FIG. 7.11 – *Modèle de comportement multi-session*

Une première phase d'enregistrement permet d'ouvrir une session pour un composant donné (repéré par son identificateur unique). En règle générale, une modèle de comportement spécialisé introduit un *langage spécifique* formé par des *primitives* - souvent très simples car très spécialisées - permettant la communication entre le niveau base (ou fonctionnel) du composant et le niveau méta-comportement ainsi spécialisé[19]. L'ouverture et la fermeture de session du modèle *MSessionBehavior* sont proposées par les commandes décrites en table 7.3 page suivante.

Notons que les primitives de gestion de session ne «cassent» pas réellement la règle d'indépendance structurelle puisque l'identificateur du composant source, s'il est indiqué, n'est manipulé qu'au niveau méta-comportement

17. Cette limite n'en est pas vraiment une si l'on considère Comet comme un descendant des langages d'acteurs.
18. Connexions en entrée et en sortie, pour des types identiques mais des instances différentes de composants.
19. Pour le modèle de comportement canonique, un exemple d'une telle primitive est `send`.

Ouverture de session
open_session(String event_type) **Définition :** Ouverture d'une session pour le traitement spécifique des événements sous-types de event_type
open_session(String event_type, String source_id) **Définition :** Ouverture d'une session uniquement pour les événements en provenance de source_id
open_session(String source_id) **Définition :** Ouverture d'une session pour tous les événements en provenance de source_id

Fermeture de session
close_session(String event_type) **Définition :** Fermeture des sessions ouvertes pour les événements sous-types de event_type
close_session(String event_type, String source_id) **Définition :** Fermeture des sessions ouvertes pour le composant id et les sous-types de event_type
close_session(String source_id) **Définition :** Fermeture de toutes les sessions ouvertes pour le composant id

TAB. 7.3 – *Primitives de gestion de session pour MSessionBehavior*

et non au niveau fonctionnel. L'ouverture d'une session correspond à la création d'un cheminement autonome des événements du type de la session (une union de types d'événement) et uniquement pour le composant concerné si ce dernier est indiqué. Les sessions sont donc discriminées par type et par (éventuellement) les composants connectés en entrée.

7.3.2 Modèle formel

Contrairement à l'usage, nous allons modéliser proprement ce modèle de comportement spécialisé avant de l'illustrer sur une application pratique. Si d'un point de vue pédagogique, ce choix est contestable, cet ordre reflète de façon plus adéquate la méthodologie sous-jacente du raffinement comportemental. Puisque les modifications sont très ciblées, le pari est que, même dans le cadre d'un changement assez majeur comme c'est le cas ici, la plupart des propriétés exposées par les schémas spécialisés résultent des schémas qu'ils raffinent.

Schéma d'état des composants multi-session

Pour commencer, nous devons décrire l'état du comportement d'un composant multi-session. La principale différence avec le modèle de comportement canonique *DefaultBehavior* concerne l'introduction de *sessions*.

En premier lieu, du fait du coût relativement élevé des ressources liées à la concurrence, nous allons fixer une limite sur le nombre de sessions qu'un composant peut ouvrir, sous la forme d'une définition axiomatique :

$$MSession\_max : \mathbb{N}_1$$

Bien sûr, cette «constante» est un paramètre de la spécification formelle. De façon pratique, chaque composant multi-session pourra définir sa propre limite [20].

Nous pouvons maintenant définir le principal schéma d'état pour le comportement multi-session :

$$
\begin{array}{l}
\textit{MSessionBehavior} \\
\hline
\textit{DefaultBehavior} \\
\\
\textit{ssessions} : (\textit{TYPE},\textit{IDENTIFIER}) \rightarrowtail\!\!\!\!\!+ \textit{MSession} \\
\textit{gsessions} : \textit{TYPE} \rightarrowtail\!\!\!\!\!+ \textit{MSession} \\
\hline
(\#\,\mathrm{ran}\,\textit{ssessions}) + (\#\,\mathrm{ran}\,\textit{gsessions}) \leq \textit{MSession\_max} \\
\\
\forall (t,id) : \mathrm{dom}\,\textit{ssessions} \bullet \exists\, in : in\_channels \bullet \\
\quad (\exists\, ct : in.type \bullet t \preceq ct) \land \\
\quad id = in.id \\
\hline
\end{array}
$$

20. Il sera même possible de modifier cette limite en cours de fonctionnement, mais on sortira alors quelque peu du modèle formel «strict».

Ce schéma introduit deux variables et deux conditions d'invariant. La première variable, *ssessions*, dénote l'ensemble des sessions ouvertes pour un couple (*type*,*identificateur*); ce sont les *sessions spécifiques* à un composant source donné. En complément, les *gsessions* sont ouvertes pour un type d'événement et sont donc partagées par toutes les sources, d'où leur nom de *sessions génériques.* Ces deux variables sont définies comme des *fonctions partielles* (introduits par $\rightarrow\!\!\!\!\!\!+\;$) dont le codomaine est de type *MSession* qui représente l'état d'une session[21].

La première condition d'invariant «vérifie» que le nombre total de sessions ouvertes ne dépasse pas la limite $MSession\_max$. La deuxième condition ne s'applique qu'aux sessions spécifiques. De telles sessions ne sont valides que lorsqu'il existe un canal entrant ouvert pour le couple (*type*,*id*) associé à la session considérée (au sous-typage près).

L'initialisation du modèle de comportement est simple puisque, par défaut, aucune session n'est ouverte :

$$\begin{array}{|l}
\textit{InitMSessionBehaviorPart} \\
\hline
\textit{MSessionBehavior}' \\
\hline
\textit{ssessions}' = \textit{gsessions}' = \emptyset \\
\hline
\end{array}$$

N'oublions pas l'initialisation de la partie canonique du schéma :

$$\textit{InitMSessionBehavior} \mathrel{\widehat=} \textit{InitDefaultBehavior} \land \textit{InitMSessionBehaviorPart}$$

Nous pouvons maintenant introduire le schéma définissant le modèle de composant multi-session :

$$\begin{array}{|l}
\textit{MSessionComponent} \\
\hline
\textit{DefaultComponent} \\
\textit{MSessionBehavior} \\
\hline
\forall\, st : \mathrm{dom}\, \textit{gsessions} \bullet \exists\, t : \textit{in_type} \bullet st \preceq t \\
\hline
\end{array}$$

En plus d'intégrer les changements comportementaux du schéma *MSessionBehavior*, ce schéma ajoute une condition d'invariant structurel portant sur les sessions génériques. Il est en effet important de vérifier qu'une telle session ouverte pour un type donné peut être active, c'est-à-dire que des événements de type compatible peuvent être reçus par le composant. Il suffit pour cela de vérifier qu'au moins un des types entrant du composant est super-type du type associé à la session.

Schéma d'état des sessions

Le principal vide qu'il nous faut combler pour parachever notre modèle formel de composant multi-session concerne la définition proprement dite des sessions, de type *MSession* :

$$\begin{array}{|l}
\textit{MSession} \\
\hline
\textit{type} : \textit{SESSION_TYPE} \\
\textit{in_queue} : \mathrm{seq}\, \textit{Event} \\
\textit{out_queue} : \mathrm{seq}\, \textit{Event} \\
\hline
\end{array}$$

Comme nous pouvons le constater sur ce schéma, une session dispose de ses propres canaux et files d'attente en entrée et en sortie. Le type de session est à choisir parmi les possibilités suivantes:

$$\textit{SESSION\_TYPE} ::= \textit{specific\_session} \mid \textit{generic\_session}$$

Toutes les opérations sur cet état se concentreront autour de ces files d'attente et de la «communication» avec le niveau méta-comportement *MSessionBehavior*.

L'initialisation de ce schéma est simple :

21. A un instant donné, il n'existe pas de session, par exemple spécifique, pour tout couple (*type*,*id*); la fonction *ssessions* est donc bien partielle.

$$
\begin{array}{|l}
\hline
\textit{InitMSession} \\
\textit{MSession}' \\
\textit{stype}? : \textit{SESSION\_TYPE} \\
\hline
\textit{type} = \textit{stype}? \\
\textit{in\_queue}' = \langle\rangle \\
\textit{out\_queue}' = \langle\rangle \\
\hline
\end{array}
$$

Les files d'attente, comme on pouvait s'y attendre, sont vides initialement. Nous pouvons proposer deux variantes non paramétrées de cette initialisation :

$$
\begin{array}{l}
\textit{InitMSessionSpecific} \mathrel{\widehat{=}} \textit{InitMSession} \land \textit{type} = \textit{specific\_session} \\
\textit{InitMSessionGeneric} \mathrel{\widehat{=}} \textit{InitMSession} \land \textit{type} = \textit{generic\_session}
\end{array}
$$

Enregistrement de session

Dans une deuxième étape, nous devons donner la sémantique des différentes primitives associées au modèle de comportement multi-session : open_session et close_session.

La première opération «sémantique» associée concerne l'ouverture d'une session spécifique, pour un couple $(id?, type?)$:

$$
\begin{array}{|l}
\hline
\textit{OpenSpecificSession} \\
\Delta \textit{MSessionBehavior} \\
\textit{id}? : \textit{IDENTIFIER} \\
\textit{type}? : \mathbb{P}\ \textit{TYPE} \\
\hline
(\textit{id}?, \textit{type}?) \notin \operatorname{dom} \textit{ssessions} \\
(\#\operatorname{ran} \textit{ssessions}) + (\#\operatorname{ran} \textit{gsessions}) < \textit{MSession\_max} \\
\exists\, \textit{in} : \textit{in\_channels} \bullet (\exists\, t : \textit{in.type} \bullet \textit{type}? \preceq t) \land \textit{id}? = \textit{in.id} \\
\exists\, \textit{MSession}' \bullet \textit{InitMSessionSpecific} \land \\
\quad \textit{ssessions}'(\textit{id}?, \textit{type}?) = \Theta \textit{MSession}' \\
\hline
\end{array}
$$

Après avoir vérifié que la session n'existe pas déjà et qu'il reste «de la place» pour l'ouverture, la troisième condition d'invariant «cherche» au moins un canal permettant de fournir des événements à cette session. La quatrième condition stipule pour sa part que l'ouverture de session est valide uniquement s'il existe une initialisation correcte pour cette session. C'est ici que se trouve la relation de promotion entre l'état local d'une session et l'état global du comportement. Ce schéma contient donc également une promotion «déguisée». Finalement, la fonction gérant les sessions spécifiques, $ssessions$, est mise à jour.

De son côté, l'ouverture «générique» est décrite par la variante suivante :

$$
\begin{array}{|l}
\hline
\textit{OpenGenericSession} \\
\Delta \textit{MSessionBehavior} \\
\textit{type}? : \mathbb{P}\ \textit{TYPE} \\
\hline
\textit{type}? \notin \operatorname{dom} \textit{gsessions} \\
(\#\operatorname{ran} \textit{ssessions}) + (\#\operatorname{ran} \textit{gsessions}) < \textit{MSession\_max} \\
\exists\, \textit{MSession}' \bullet \textit{InitMSessionGeneric} \land \\
\quad \textit{gsessions}'(\textit{type}?) = \Theta \textit{MSession}' \\
\hline
\end{array}
$$

Dans la table 7.3 page 133, il est proposé une troisième variante concernant l'ouverture d'une session pour un composant donné mais sans restriction au niveau du type des événements. Au niveau formel, il n'est pas nécessaire de construire une opération propre puisqu'il s'agit simplement d'une restriction de l'ouverture d'une session spécifique pour le type le plus générique $Event$.

Nous pouvons ensuite définir les opérations de fermeture de session, dans un premier temps pour la catégorie spécifique :

$$
\begin{array}{l}
\textit{CloseSpecificSession} \\
\hline
\Delta \mathit{MSessionBehavior} \\
\mathit{MSession} \\
id? : \mathit{IDENTIFIER} \\
type? : \mathbb{P}\ \mathit{TYPE} \\
\hline
(id?,type?) \in \mathrm{dom}\ \mathit{ssessions} \\
\mathit{ssessions}(id?,type?) = \Theta \mathit{MSession} \\
\mathit{ssessions}' = \mathit{ssessions} \setminus \{(id?,type?) \mapsto \Theta \mathit{MSession}\} \\
\hline
\end{array}
$$

Puis pour les sessions génériques :

$$
\begin{array}{l}
\textit{CloseGenericSession} \\
\hline
\Delta \mathit{MSessionBehavior} \\
\mathit{MSession} \\
type? : \mathbb{P}\ \mathit{TYPE} \\
\hline
type? \in \mathrm{dom}\ \mathit{gsessions} \\
\mathit{gsessions}(type?) = \Theta \mathit{MSession} \\
\mathit{gsessions}' = \mathit{gsessions} \setminus \{type? \mapsto \Theta \mathit{MSession}\} \\
\hline
\end{array}
$$

Il est possible d'étendre ces opérations pour la fermeture simultanée de plusieurs sessions, en tirant partie de la relation de sous-typage. De la même façon, l'opérationnalisation de la fermeture doit correspondre à une fermeture potentielle ou à un «raccrochement» des canaux associés à la session vers la session principale (canonique) du comportement. Ces contraintes d'opérationnalisation dépassent le cadre de cette section mais ces questions doivent être soulevées pour obtenir une implantation correcte des gestionnaires de session.

Méta-objets comportementaux et activation

Le principal intérêt d'avoir construit un schéma d'état spécifique pour les sessions est de pouvoir pratiquement réutiliser *tels quels* les schémas comportementaux associés aux méta-objets de la couche réflexive de *DefaultBehavior*.

Cependant, il faudra toujours penser, en cas de modification de l'état d'une session, à promouvoir cette modification au niveau du composant. En supposant que ces modifications sont capturées par les schémas caractéristiques de session $\Theta \mathit{MSession}$ (avant modification) et $\Theta \mathit{MSession}'$ (après modification), nous pouvons proposer un schéma générique de promotion :

$$
\begin{array}{l}
\textit{MSessionPromote} \\
\hline
\Delta \mathit{MSessionBehavior} \\
\Delta \mathit{MSession} \\
\hline
\textbf{if } \Theta \mathit{MSession}.type = \mathit{specific\_session} \textbf{ then} \\
\quad \textbf{let } (id,type) == \mathit{ssessions}^{\sim}(\Theta \mathit{MSession}) \bullet \\
\quad\quad \mathit{ssessions}'(id,type) = \Theta \mathit{MSession}' \\
\textbf{else} \\
\quad \textbf{let } type == \mathit{gsessions}^{\sim}(\Theta \mathit{MSession}) \bullet \\
\quad\quad \mathit{gsessions}'(type) = \Theta \mathit{MSession}' \\
\hline
\end{array}
$$

Nous utilisons ici l'identification du type de session pour connaître la variable à modifier au niveau du schéma de comportement (*ssessions* pour les sessions spécifiques et *gsessions* pour les sessions génériques). Ces modifications ne sont valides que s'il existe bien un antécédent pour le schéma caractéristique $\Theta \mathit{MSession}$ dans ces variables. Nous devrons nous assurer de ce fait à chaque fois que nous composerons ce schéma de promotion avec une opération au niveau d'une session.

Commençons par décrire la phase de réception des événements. Au niveau de la réception effective, rien n'est modifié par l'introduction des sessions; le schéma *DefaultMetaReceive* non modifié est valide. Cependant, la deuxième phase de cette réception (avant mise en file d'attente) est complexifiée dans d'importantes proportions par l'introduction des sessions. En effet, il est important de pouvoir faire la distinction entre trois types possibles de «routes» pour l'événement :

1. via une session spécifique,
2. via une session générique,
3. via la «route» par défaut.

Les deux schémas suivant permettent de choisir cette route, à partir d'un événement entrant représenté par une variable $event?$:

$$
\begin{array}{l}
\textit{MSessionSpecificDispatch} \\
\hline
\Xi \textit{MSessionBehavior} \\
\textit{event?} : \textit{Event} \\
\textit{session!} : \textit{MSessionx} \\
\hline
\textbf{let } \textit{ok\_ssessions} == \{ s : \textit{ssessions} \mid (\exists\, t : s.\textit{type} \bullet \textit{event?}.\textit{type} \preceq t) \land \textit{event?}.\textit{sender\_id} = s.\textit{id} \} \bullet \\
\quad \textbf{if } \textit{ok\_ssessions} \neq \emptyset \textbf{ then} \\
\qquad \textit{ssession!} = \mu\, s : \textit{ok\_ssessions} \bullet s.\textit{type} \subseteq \textit{tmin}(\{ s_2 : \textit{ok\_ssessions} \bullet s_2.\textit{type} \}) \\
\quad \textbf{else} \\
\qquad \textit{session!} = \emptyset
\end{array}
$$

La variable $session!$ contient, en sortie de cette opération de sélection, soit la session *la plus spécifique* permettant de traiter cet événement (à partir de son type et de l'identificateur de l'émetteur), soit - en deuxième choix - l'ensemble vide si aucune session ne correspond.

Une autre opération effectue le même travail avec les sessions génériques; en tentant d'en extraire la plus spécifique (c'est-à-dire pour le super-type minimal de l'événement reçu) :

$$
\begin{array}{l}
\textit{MSessionGenericDispatch} \\
\hline
\textit{MSessionBehavior} \\
\textit{event?} : \textit{Event} \\
\textit{session!} : \textit{MSession} \\
\hline
\textbf{let } \textit{ok\_gsessions} == \{ s : \textit{gsessions} \mid (\exists\, t : s.\textit{type} \bullet \textit{event?}.\textit{type} \preceq t) \bullet \\
\quad \textbf{if } \textit{ok\_gsessions} \neq \emptyset \textbf{ then} \\
\qquad \textit{session!} = \mu\, s : \textit{ok\_gsessions} \bullet s.\textit{type} \subseteq \textit{tmin}(\{ s_2 : \textit{ok\_gsessions} \bullet s_2.\textit{type} \}) \\
\quad \textbf{else} \\
\qquad \textit{session!} = \emptyset
\end{array}
$$

Dans les deux schémas précédents, le critère d'existence et d'unicité (opérateur μ) peut être prouvé de façon relativement triviale.

Après réception et dispatch, la mise en file d'attente correspond à deux cas possibles :

1. prise en charge par une session,
2. prise en charge par les méta-objets canoniques.

Le premier cas est modélisé de la façon suivante :

$$
\begin{array}{|l}
\hline
\textit{MSessionInQueueSession} \\
\Delta \textit{MSession} \\
\textit{event}? : \textit{Event} \\
\textit{session}? : \textit{MSession} \\
\hline
\textit{session}? \neq \emptyset \\
\textit{session}? = \Theta \textit{MSession} \\
\textit{in\_queue}' = \langle \textit{event}? \rangle \frown \textit{in\_queue} \\
\textit{out\_queue}' = \textit{out\_queue} \\
\hline
\end{array}
$$

La principale précondition proposée vérifie que la session passée en paramètre n'est pas l'ensemble vide. En effet, ce dernier cas indique la sélection de la route canonique pour traiter l'événement. En vue de la promotion (par *MSessionPromote*), ce paramètre *session*? est identifié au schéma caractéristique Θ*MSession*. Ensuite, la mise en file d'attente effective est proposée.

La variante pour la route par défaut est la suivante :

$$
\begin{array}{|l}
\hline
\textit{MSessionInQueueDefault} \\
\textit{DefaultMetaReceive} \\
\textit{session}? : \textit{MSession} \\
\hline
\textit{session}? = \emptyset \\
\hline
\end{array}
$$

Il s'agit ici pratiquement d'une copie de *DefaultMetaReceive* mais il est important de vérifier que le choix de routage est cohérent (session sélectionnée identifiée à l'ensemble vide).

Finalement, nous pouvons résumer le modèle formel de réception, dans le cadre du comportement multi-session :

$$
\begin{aligned}
&\textit{MSessionReceive} \mathrel{\widehat=} \textit{DefaultMetaReceive} \gg (\textit{MSessionSpecificDispatch} \land \textit{MSessionGenericDispath}) \gg \\
&\quad (\textit{MSessionInQueueSession} \land \textit{MSessionPromote}) \lor \textit{MSessionInQueueDefault}
\end{aligned}
$$

Ce schéma composé quelque peu complexe décrit une composition séquentielle. Tout d'abord, l'événement est reçu effectivement via *DefaultMetaReceive* puis *dispatché* vers les trois routes possibles. Ceci est bien visible sur le réseau d'activation représenté en figure 7.12 page suivante où les trois transitions possibles sont nommées *DispatchDefault* (route par défaut), *DispatchGeneric* (activation de session générique) et *DispatchSpecific* (activation de session spécifique). Finalement,

En dehors de la phase de réception effective des événements, le principal intérêt d'avoir construit un schéma d'état spécifique pour les sessions est de pouvoir réutiliser *tels quels* les schémas comportementaux associés aux méta-objets de la couche réflexive de *DefaultBehavior*.

Ainsi, on pourra définir *MSessionMetaInFetch*, par exemple, comme une copie pratiquement conforme de *DefaultMetaInFetch* (cf. 121) :

$$
\begin{array}{|l}
\hline
\textit{MSessionInFetch} \\
\Delta \textit{MSession} \\
\textit{event}! : \textit{Event} \\
\hline
\# \textit{in\_queue} > 0 \\
\textit{event}! = \mathrm{last}\ \textit{in\_queue} \\
\textit{in\_queue}' = \mathrm{front}\ \textit{in\_queue} \\
\hline
\end{array}
$$

La différence est qu'au lieu de «travailler» directement sur le modèle de comportement - *MSessionBehavior* dans le cas présent - cette opération ne s'intéresse qu'à la session concernée. En revanche, il faudra toujours combiner les opérations avec le schéma de comportement :

$$\textit{MSessionMetaInFetch} \mathrel{\widehat=} \textit{MSessionInFetch} \land \textit{MSessionPromote}$$

Nous supposons de la même façon disposer des autres opérations méta-comportementales, dans leur version multi-session :

– *MSessionMetaExec* pour l'exécution des fonctionnalités associées
– *MSessionMetaOutQueue* pour la mise en file d'attente en sortie
– *MSessionMetaOutFetch* pour l'extraction en sortie
– *MSessionMetaSend* pour l'émission effective

En terme de contrôle, la nouveauté réside dans l'association d'une activité propre à chaque session créée pour le composant. Ceci modifie dans de larges mesures les réseaux d'activation concurrente associés au modèle de comportement multi-session.

Par exemple, l'opération *DefaultMetaInFetch* peut se dérouler en parallèle d'au plus *MSession_max* activations de

MSessionMetaInFetch (une par session). Le réseau d'activation correspondant est décrit en figure 7.12.

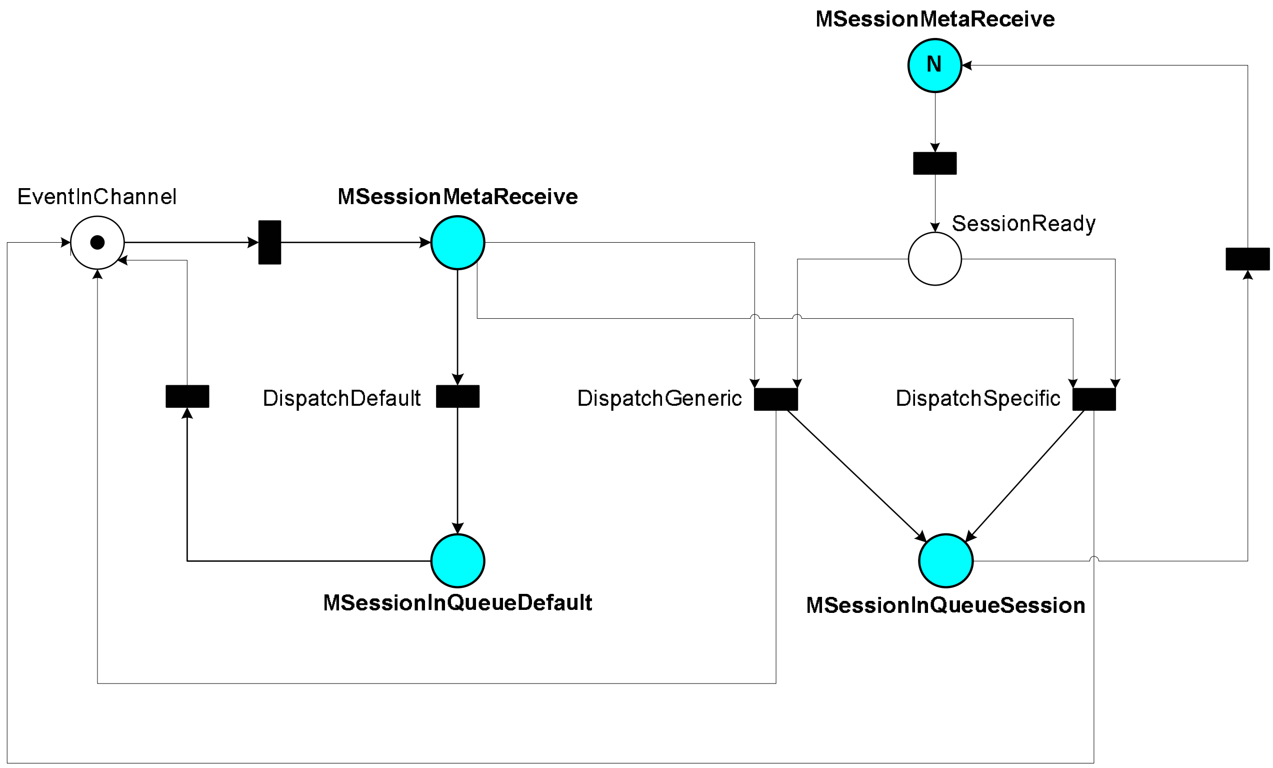


FIG. 7.12 – *Réseau d'activation concurrente du modèle de comportement multi-session*

Nous pouvons remarquer l'emploi d'un nouveau type de place, les *NPlaces*, qui dénotent la correspondance entre un marquage de réseau de Petri et un entier naturel associé à une définition axiomatique de Z. La *NPlace MSession_max* contient en l'occurrence autant de jetons qu'il existe au maximum de session en parallèle au sein d'un composant multi-session. Nous pouvons constater sur ce réseau l'invariant structurel :

$$m(MSessionInQueueSession) \leq m(MSession\_max)$$

Ceci indique que les différentes sessions sont indépendantes en terme de contrôle et peuvent donc «tourner» dans des activités parallèles. Il faut cependant prouver la cohérence de l'activation multiple de ces différentes activités :

$$MSessionInQueueSession_1 \mid\mid MSessionInQueueSession_2 \mid\mid ... \mid\mid MSessionInQueueSession_N$$

De même, chaque méta-objet potentiellement concurrent dans le modèle par défaut peut désormais être activable en même tant qu'un méta-objet de même nature. Si l'on se rappelle que la phase de mise en file d'attente en sortie (opération *MSessionMetaOutQueue*) est indépendante de l'extraction en entrée (opération *MSessionMetaInFetch*), nous aurons potentiellement à prouver :

$$(MSessionMetaInFetch_1 \mid\mid MSessionMetaOutQueue_1) \mid\mid ...$$
$$\mid\mid (MSessionMetaInFetch_N \mid\mid MSessionMetaOutQueue_N)$$

Fort heureusement, nous avons déjà vu que les opérations activables concurremment sont pour la plupart indépendante. De ce point de vue, les obligations de preuve sont pratiquement similaires au cas par défaut. En revanche, si nous introduisons par exemple un méta-objet pour gérer les accès à l'état interne (ce que nous ferons en pratique, cf. chapitre 10), le critère d'indépendance entre instances de ce méta-objet ne sera probablement plus vérifié.

7.3.3 Exemple simple : médiateur concurrent

La mise en œuvre d'un composant multi-session nécessite généralement l'introduction de mécanismes de synchronisation interne permettant de contrôler les interactions entre les différentes sessions. La synchronisation interne, dans Comet, est comme nous l'avons déjà signalé une simple réutilisation des modèles proposés dans CodA et qui, à ce titre, ne font pas l'objet d'une étude précise dans ce document.

Pour illustrer notre modèle de comportement multi-session, nous allons donc prendre un exemple dans lequel les interactions entre les sessions (autres que ceux exposés au niveau du modèle formel) sont inexistantes. Une condition suffisante consiste à considérer le cas de composants sans état interne (et donc sans grand problème de synchronisation à ce niveau). Dans cet objectif, reprenons une dernière fois notre exemple de notification système et considérons en particulier le composant médiateur qui se charge de «router» les événements expédiés par les sondes vers les composants d'analyse et de visualisation. En utilisant le modèle de comportement canonique, ce médiateur représente une sorte de «goulet d'étranglement» du fait de la sérialisation des événements routés. En effet, comme nous l'avons vu, la sémantique d'acteur associée aux composants par défaut ne peut dénoter qu'une seule activité de traitement des événements.

Une première solution pour résoudre ce problème consiste à déployer plus d'un médiateur. Cette solution purement structurelle est représentée en figure 7.13 et à l'avantage d'être réalisable sans modification comportementale.

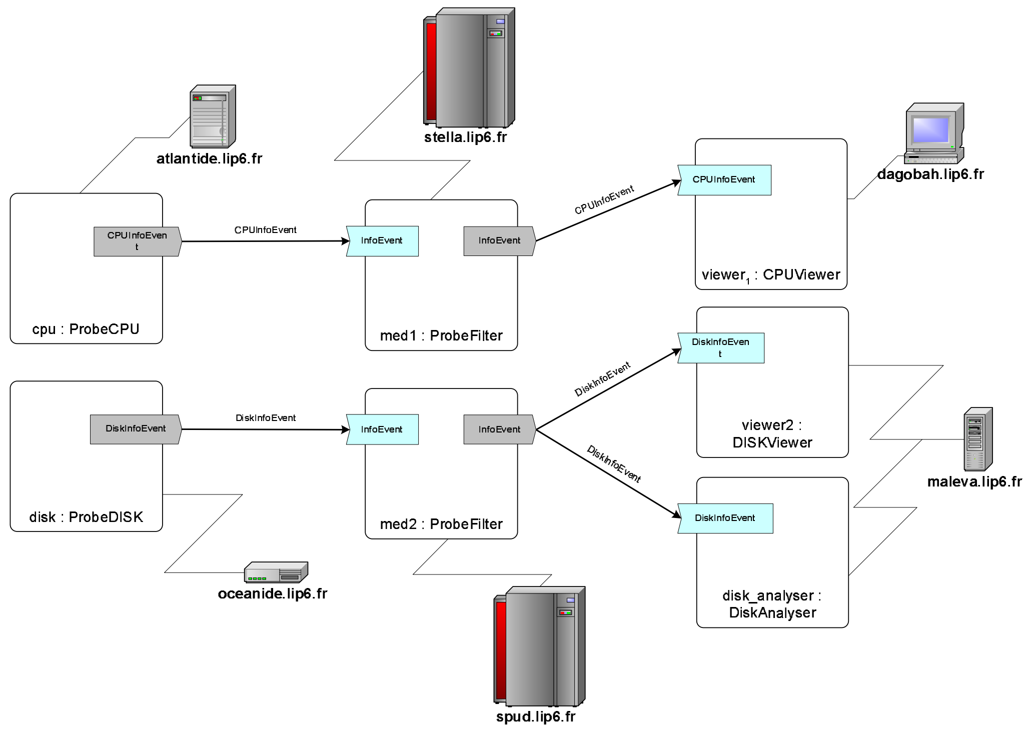


FIG. 7.13 – *Médiateurs concurrents : approche structurelle*

Une autre possibilité, particulièrement intéressante si des machines à processeurs multiples sont disponibles, concerne le remplacement du modèle par défaut par un comportement multi-session pour opérationnaliser le médiateur. Cette solution, complète car le médiateur est sans état explicite, est exprimée en figure 7.14 page ci-contre.

Une étape préliminaire consiste à décrire les conditions d'ouverture de nouvelles sessions. Pour le composant médiateur, nous allons utiliser uniquement des sessions génériques permettant d'associer une activité propre à

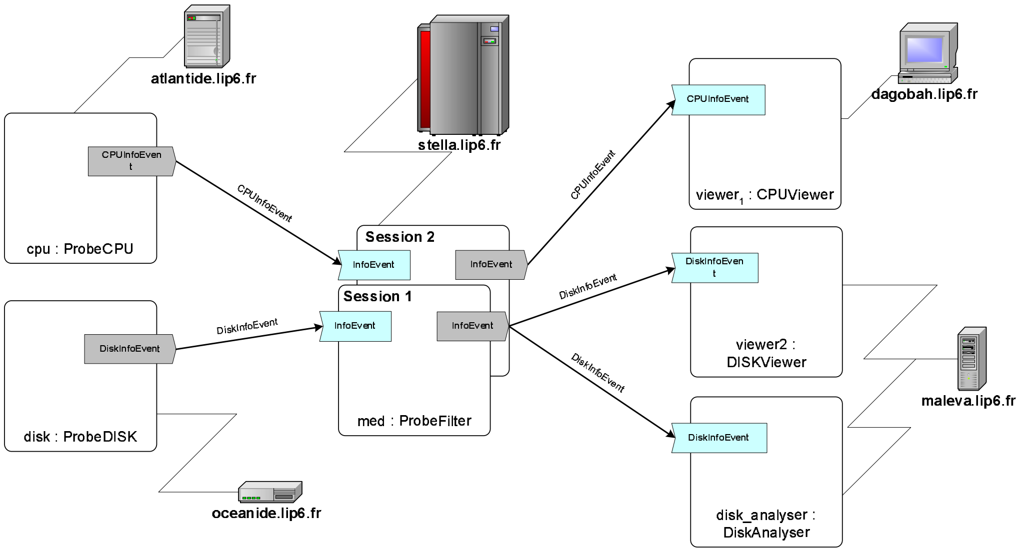


FIG. 7.14 – *Médiateurs concurrents : approche comportementale*

la chaîne de traitement d'événements d'un type donné (ou de sous-types). En anticipant un peu sur les classes d'implantation des modèles de comportement (décrites au chapitre 10), nous pouvons donner la forme générale des médiateurs concurrents :

```
component ConcurrentProbeFilter extends MSessionComponent {
  receive InfoEvent;
  send InfoEvent;

  // création implicite de session
  void connect_destination(String event_type, ...) {
    super.connect_destination(event_type, ...);
    open_session(event_type);
  }

  when (InfoEvent info) {
    send(info);
  }
}
```

La différence fondamentale avec les composants ProbeFilter concerne l'emploi du modèle de comportement MSessionBehavior, ainsi que l'enregistrement implicite des sessions. En effet, nous nous rappelons du caractère implicite des souscriptions; propriété que nous ne désirerons pas «perdre» par modification du modèle de comportement. La solution passe par la spécialisation des opérations internes de connexion qui correspondent aux schémas opérationnels de mise en place des canaux d'entrée/sortie.

Nous pouvons voir sur cet exemple qu'encore une fois, malgré la nature complexe du changement opérationnel proposé dans cette section, l'impact au niveau de la description fonctionnelle des composants est relativement modéré.

« Rien n'est permanent, sauf le changement.»
Héraclite d'Ephèse

« Il n'existe rien de constant, si ce n'est le changement.»
Bouddha

« La réalité est un yo-yo, le changement est la seule constante. »
Paul Auster

8 Protocoles et rôles

Les composants et événements, s'ils «règnent» hégémoniquement sur l'opérationnalisation des systèmes basés sur l'interstitiel Comet, ne se suffisent cependant pas à eux même pour décrire des applicatifs complets. La propriété d'indépendance structurelle impose aux composants - comme nous l'avons maintes fois répété - de ne pas se référencer explicitement. Dans une interprétation stricte de cette propriété, les composants ne peuvent directement intervenir sur la structure des applications. Pourtant, aucune application ne peut être construite dans notre approche sans modification structurelle : ajout de composant ou établissement de connexion. En effet, rappelons-nous qu'au temps initial, tout système se résume à l'ensemble vide de composants et de connexions.

Il nous faut donc introduire un concept dont le but premier concerne la modification - ou adaptation - dynamique du système en cours d'opérationnalisation; cette responsabilité échoit au concept de *protocole*.

Plus qu'uniquement conceptuels, les protocoles Comet correspondent à des *abstractions* de notre infrastructure linguistique héritée du langage Scope. Un protocole décrit une modification ou un ensemble de modifications possibles sur l'environnement et est interprété par la plateforme pour opérationnaliser ces modifications à partir de structures linguistiques formant les *commandes* de notre *langage d'interactions distribuées* [1], sorte de version distribuée du langage de composition Scope.

8.1 Adaptations structurelles

Dans une première approche, les protocoles Comet permettent d'apporter des modifications structurelles à un environnement préexistant. Nous parlerons aussi - à un niveau plus pragmatique - de *scripts* de modifications même si l'utilisation privilégiée du terme «protocole» se trouvera justifiée tout au long de notre discours (et notamment par l'introduction des rôles en section 8.2 page 148).

8.1.1 Déploiement

La forme la plus simple de modification ou d'adaptation dynamique de cet environnement concerne le *déploiement*, c'est-à-dire :

- l'ajout - ou déploiement proprement dit - de nouveaux composants,
- la mise en place de connexions entre ces composants,
- les activations ou initialisations éventuelles des composants déployés.

1. Traduction littérale de *Distributed Interaction Language* [Stu96] dont les concepts - extirpés du monde des acteurs - se retrouvent dans notre proposition.

Comme annoncé en préambule, les protocoles permettent de décrire des modifications - ici un déploiement - par l'emploi de commandes adaptées. Retrouvons l'exemple de protocole *publish/subscribe* pour la notification système décrit au chapitre 7 et retranscrit en figure 8.1.

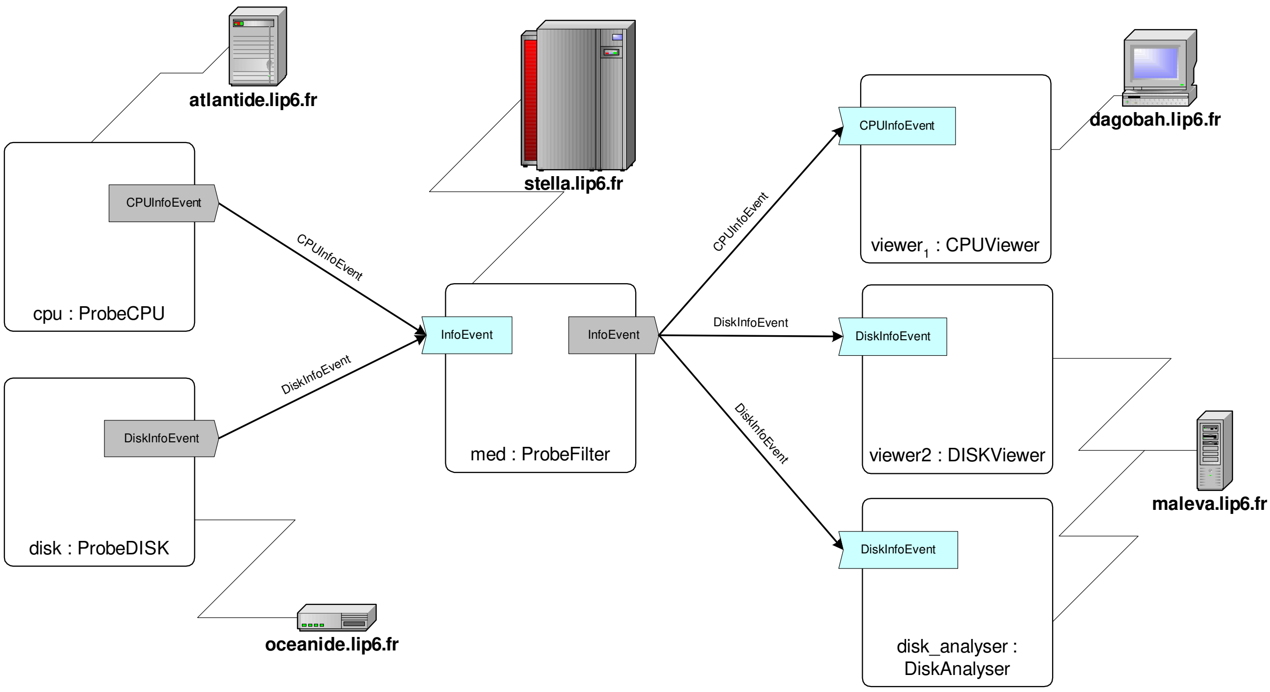


FIG. 8.1 – *Service de notifications systèmes avec localisations*

Si nous connaissons les principes formels sous-jacents du déploiement d'un tel système (principes définis au niveau du modèle structurel et du langage Scope associé), l'immersion en environnement distribué réclame quelques précisions d'ordre plus pragmatique. Sur la figure 8.1, nous pouvons voir que les différents composants du système de notification sont associés à des *localisations* variées. Ces localisations, ou *sites* dans notre terminologie, correspondent - comme décrit plus précisément au chapitre 10 - à des couples (*machine*,*port*) (ou juste une adresse de machine en utilisant le numéro de port par défaut). Ainsi, pour déployer des applications Comet, les commandes à utiliser (et résumées en table 8.1 page ci-contre) référencent explicitement des localisations.

Commandes de déploiement de composants
upload(String component, String address [, **int** port]) **Définition :** Déploiement du code de la définiton component sur le site address:port
String id = **instantiate**(String component, String address [, **int** port]) **Définition :** Création distante d'une instance de composant

Commandes de connexions
connect(String event_type, String source_id, String saddress [, **int** sport], String dest_id, String daddress [, **int** dport]) **Définition :** Commande de connexion avec localisation explicite des composants source (identifié par source_id) et destination (dest_id)

Commandes d'activation et d'initialisation
activate(ActivateEvent activation, String id, String address [, **int** port]) **Définition :** Requête d'activation distante d'un composant id sur le serveur address:port
initialize(InitializeEvent init, String id, String address [, **int** port]) **Définition :** Requête d'initialisation à distance d'un composant id

TAB. 8.1 – *Plateforme Comet : commandes de déploiement*

En anticipant certains aspects purement «implantatoires» (comme la gestion du code des composants), essayons de nous inspirer de ces commandes pour déployer notre exemple de notification système :

```
// Déploiement du code
upload("ProbeCPU","atlantide.lip6.fr");
upload("ProbeDISK","oceanide.lip6.fr");
upload("ProbeFilter","stella.lip6.fr");
upload("CPUViewer","dagobah.lip6.fr");
upload("DISKViewer","maleva.lip6.fr");
upload("DISKAnalyzer","maleva.lip6.fr");

// Déploiement des composants
String cpu_id = instantiate("ProbeCPU","atlantide.lip6.fr");
String disk_id = instantiate("ProbeDISK","oceanide.lip6.fr");
String filter_id = instantiate("ProbeFilter","stella.lip6.fr");
String cpuview_id = instantiate("CPUViewer","dagobah.lip6.fr");
String diskview_id = instantiate("DISKViewer","maleva.lip6.fr");
String diskalyzer_id = instantiate("DISKAnalyzer","maleva.lip6.fr");

// Etablissement des connexions
connect("CPUInfoEvent", cpu_id, "atlantide.lip6.fr", filter_id, "stella.lip6.fr");
connect("DISKInfoEvent", disk_id, "oceanide.lip6.fr", filter_id, "stella.lip6.fr");
connect("CPUInfoEvent", filter_id, "stella.lip6.fr", cpuview_id, "dagobah.lip6.fr");
connect("DISKInfoEvent", filter_id, "stella.lip6.fr", diskview_id, "maleva.lip6.fr");
connect("DISKInfoEvent", filter_id, "stella.lip6.fr", diskalyzer_id, "maleva.lip6.fr");

// Activations et intialisations
activate(new ActivateEvent(), cpu_id, "atlantide.lip6.fr");
activate(new ActivateEvent(), disk_id, "oceanide.lip6.fr");
```

Ce script décompose en quatre phases distinctes la problématique du déploiement effectif :

1. Le déploiement du code des composants, par l'utilisation de commandes upload,
2. la création des instances de composants avec instantiate qui retourne un identificateur de composant (sous forme de chaîne de caractères),
3. l'interconnexion des composants par connect, et

4. l'activation (activate) ou l'initialisation (initialize) de certains composants du système.

La distinction entre déploiement du code et des instances, s'il représente un détail d'un point de vue formel, revêtira une grande importance pour le modèle d'implantation (cf. chapitre 10).

8.1.2 Constructions linguistiques

Le script de déploiement décrit précédemment correspond au contenu d'un protocole. Il nous faut décrire un cadre pour ce genre de définition. Ce cadre - la définition de protocole elle-même - est structuré grammaticalement de la façon suivante :

<protocol>	→	`protocol` *<protocol-name> (*`extends` *<protocol-name>)?* `{` *<protocol-contents>* `}`
<protocol-contents>	→	*<protocol-state>* * *<protocol-function>* * *<protocol-main>?*
<protocol-state>	→	*<java-class-attribute>*
<protocol-function>	→	*<java-class-method>*
<protocol-main>	→	`main(`*(*`String args[]`*)?*`) {` *<protocol-script>* `}`

Cette grammaire simplifiée introduit le littéral protocol qui sert de cadre aux définitions de protocoles. Ces définitions peuvent contenir :

- Un *état*, sous la forme (pour notre implantation dans le langage Java) d'un ensemble d'attributs de classe,
- des *fonctionnalités* correspondant à des méthodes de classe Java, et
- un *script* global, sorte de point d'entrée du protocole.

L'interprétation d'une telle construction syntaxique correspond à la génération d'une classe *singleton* Java, éventuellement héritée d'une autre définition de protocole (entre autre de la classe *ancêtre* Protocol décrite au chapitre 10).

Pour notre exemple de notification système, la définition aura la structure suivante :

```
protocol ProbeSysDeployment1 {

  main() {
    // Contenu du script de déploiement (cf. ci-dessus)
  }

}
```

Ce protocole, nommé ProbeSysDeployment1, ne comporte qu'une partie script et est donc sans état ni fonctionnalité.

Nous pouvons remarquer, dans cette définition, qu'en plus des références aux composants mis en jeu, les nécessaires références aux localisations qui les gèrent. Ceci est très important car le *service de nommage* de Comet est un système de bas niveau assurant l'importante propriété d'unicité des composants mais ne s'occupant aucunement de leur localisation (nous décrirons un service de nommage - ou plutôt de localisation - de plus haut niveau au chapitre 9)[2].

8.1.3 Protocoles à état

Une première approche pour simplifier l'écriture de scripts de déploiement passe par la définition de protocoles avec état et fonctionnalités associées. La définition de protocole proposée ci-dessous permet de mettre en place un «service minimal» de localisation :

2. La migration de composant (encore expérimentale dans Comet) est un exemple «flagrant» où cette distinction entre nommage et localisation devient primordiale. La persistance est un autre service permettant la relocalisation possible d'un composant donné.

```
protocol LocalNameService {

  // état du protocol
  HashMap _comps;

  // fonctionnalités associées
  String fetchComponent(String id) {
    return (String) _comps.get(id);
  }

  void registerComponent(String id, String address) {
    _comps.put(id,address);
  }

  String upload_and_instantiate(String component, String address) {
    super.upload(component, address);
    String id = super.instantiate(component, address);
    registerComponent(id,address);
    return id;
  }

  void connect(String event_type, String source, String dest) {
    super.connect(event_type, source, fetchComponent(source), dest, fetchComponent(dest));
  }


// etc ...

}
```

Ce protocole gère un état et des fonctionnalités associées maintenant un tableau associatif de couples (*identificateur*,*localisation*). Il fournit également des variantes des commandes de déploiement[3] en rendant implicite la localisation des composants. Puisque aucune section main n'est fournie, ce protocole n'est pas un script et n'est donc pas utilisable directement. Il peut cependant être étendu par simple héritage de définition (ou de classe dans la terminologie de notre environnement orienté objet sous-jacent); ce qui nous permet de simplifier notre script de déploiement :

```
protocol ProbeSysDeployement2 extends LocalNameService {

  main() {
    // Déploiement des composants
    String cpu_id = upload_and_instantiate("ProbeCPU","atlantide.lip6.fr");
    String disk_id = upload_and_instantiate("ProbeDISK","oceanide.lip6.fr");
    String filter_id = upload_and_instantiate("ProbeFilter","stella.lip6.fr");
    String cpuview_id = upload_and_instantiate("CPUViewer","dagobah.lip6.fr");
    String diskview_id = upload_and_instantiate("DISKViewer","maleva.lip6.fr");
    String diskalyzer_id = upload_and_instantiate("DISKAnalyzer","maleva.lip6.fr");

    // Etablissement des connexions
    connect("CPUInfoEvent", cpu_id, filter_id);
    connect("DISKInfoEvent", disk_id, filter_id);
    connect("CPUInfoEvent", filter_id, cpuview_id);
    connect("DISKInfoEvent", filter_id, diskview_id);
    connect("DISKInfoEvent", filter_id, diskalyzer_id);

    // Activations et intialisations
    activate(new ActivateEvent(), cpu_id);
    activate(new ActivateEvent(), disk_id);
  }
}
```

3. La commande upload_and_instantiate, comme son nom l'indique, regroupe les phases de déploiement du code et d'instantiation de composants.

Cette variante est certes plus lisible que la première version du protocole mais il faut garder à l'esprit que le service de nom ainsi mis en œuvre n'offre qu'une vision locale des composants déployés. A la fin du déploiement, les références et localisations des composants concernés seront perdues [4].

Une autre possibilité consiste en l'instanciation, au sein même d'un composant, d'un protocole (et pas forcément un script). En effet, si les composants ne peuvent directement jouer le rôle de protocole, rien ne les empêche d'*utiliser* des protocoles.

Il est important de noter, en conclusion, que les protocoles «structurels» (nous découvrirons leurs pendants «comportementaux» dans la section suivante) ne concernent pas uniquement le déploiement; les opérations «inverses» sont bien sûr disponibles : retraits de composant ou de connexion (cf. table 8.2).

Retrait de composant
delete(String id, String address [, **int** port]) **Définition :** Retrait (et éventuelle déconnexion) du composant id sur le site address:port

Commandes de déconnexion
disconnect(String event_type, String source_id, String saddress [, **int** sport], String dest_id) **Définition :** Commande de déconnexion, seul le composant source doit être localisé.
disconnect_all(String source_id, String saddress [, **int** sport], String dest_id) **Définition :** Commande de déconnexion généralisée.
disconnect_component(String id, String address [, **int** port]) **Définition :** Commande de déconnexion d'un composant donné, identifié par id.

TAB. 8.2 – *Plateforme Comet : commandes de retrait et de déconnexion*

8.2 Adaptations de grain fin

Nous avons défini la notion de protocole comme le vecteur de modifications dynamiques de l'environnement exécutif. Pour le *grand dictionnaire terminologique* [Dic], la définition de ce terme est la suivante :

> *Ensemble des conventions nécessaires pour faire coopérer des entités généralement distantes, en particulier pour établir et entretenir des échanges d'informations entre ces entités.*

Cette définition cadre bien avec le principe de nos protocoles, mêmes lorsque leur portée est restreinte au simple déploiement d'application. Les protocoles Comet, si on les complète par les définitions des événements et composants associés, cadrent bien avec cette définition. Cependant, ce sont les composants et non les protocoles directement qui décrivent la partie opérationnelle des échanges applicatifs.

Il existe cependant une vision plus stricte des protocoles qui, pris indépendamment de toute référence externe, s'intègre totalement dans la définition ci-dessus. Ces protocoles, s'ils décrivent toujours des modifications apportées à l'environnement en cours d'exécution, donnent également toutes les règles de comportement que doivent suivre les composants participant pour «générer» les interactions associées.

Cette catégorie de protocole doit donc décrire ce qu'il faut changer dans un composant pour participer aux nouvelles interactions proposées. Ainsi, le niveau de granularité des modifications se raffine et nous sommes obligés - en quelque sorte - d'entrer dans les composants pour supporter ces nouveaux protocoles.

Les *rôles* Comet, associés aux protocoles de modification de grain fin, décrivent ce que les participants aux protocoles doivent fournir - tant en terme de comportement que de structure - pour supporter les interactions sous-jacentes du protocole concerné. Par exemple, on parlera d'un rôle de client et d'un rôle de serveur pour un protocole de type client/serveur, dénotant des interactions du même type (requêtes/réponses).

Mais il est important de conserver à l'esprit que l'intérêt de tels protocoles se restreint à des modifications de grain fin guidées selon quelques tendances fondamentales :

4. Nous verrons, cependant, que des requêtes permettent de «retrouver» des composants en cours d'exécution.

1. La description de ces modifications doit revêtir un intérêt certain pour une gamme d'application, et ce, indépendamment de ces mêmes applications. En d'autre terme, un protocole de cette catégorie n'a de l'intérêt que s'il correspond à un service susceptible d'être partagé par plusieurs applicatifs. C'est une différence fondamentale avec les protocoles de déploiement qui sont, eux, généralement très spécifiques. En guise d'exemple, les services décrits dans cette thèse ont, pour la plupart d'entre eux, une forte connotation *système* et donc à portée relativement générique.
2. Ces protocoles décrivent des modifications dynamiques, ils ne sont en aucun cas utilisés pour décrire des interactions *complètes* et *indépendantes* des composants qui devront les supporter.
3. Les modifications temporaires fournissent un autre cadre d'application très intéressant pour cette catégorie de protocoles.

Au niveau de l'opérationnalisation de tels protocoles, le principal «moteur» concerne l'introduction d'une nouvelle commande de modification permettant l'*assignation dynamique* d'un rôle à un composant donné. Cette commande complète les constructions linguistiques primitives de notre langage d'interactions distribuées.

8.2.1 Catégories de rôle

Selon le même fil conducteur qui sous-tend systématiquement notre thèse, il nous faut passer des concepts - ici, de rôle et de protocole associé - à leur concrétisation, toujours sous forme d'abstractions structurantes de notre langage d'interaction.

Nous connaissons déjà la structure des protocoles, qui est complétée par la structure des rôles, définie ainsi :

```
<role>              →  role <role-name> (extends <role-name>)? { <role-type>* <role-contents>* }
<role-type>         →  <component-type>
                        | <pre-hook-type>
                        | <post-hook-type>
                        | <in-filter-type>
                        | <out-filter-type>
<pre-hook-type>     →  prehook <event-type>;
<post-hook-type>    →  posthook <event-type>;
<in-filter-type>    →  infilter <event-type>;
<out-filter-type>   →  outfilter <event-type>;
<role-contents>     →  <component-part>
                        | <role-part>
<role-part>         →  <role-pre-hook>
                        | <role-post-hook>
                        | <role-in-filter>
                        | <role-out-filter>
<role-pre-hook>     →  before(<event-type> <event-name>) { <java-code> }
<role-post-hook>    →  after(<event-type> <event-name>) { <java-code> }
<role-in-filter>    →  boolean filter in(<event-type> <event-name>) { <java-code> }
<role-out-filter>   →  boolean filter out(<event-type> <event-name>) { <java-code> }
```

Nous allons illustrer cette définition assez mystérieuse qui ressemble fortement à la définition des composants proposées par le langage Scope (cf. chapitre 6).

De façon assez empirique, on peut décrire un rôle comme une forme de composant dont la principale différence structurelle réside principalement dans la présence du mot-clé role (en lieu et place de component) et en l'ajout de propriétés spécifiques que nous allons décrire dans le reste de cette section. Il est toutefois important de noter qu'un rôle ne dénote aucun comportement propre (et ne propose donc pas de niveau méta-comportement); ce dernier étant indissociable du composant auquel il sera assigné.

Les propriétés spécifiques des rôles permettent de décomposer le concept en trois catégories non-exclusives :

- les rôles *fonctionnels* qui ajoutent de nouvelles fonctionnalités aux composants auxquels ils sont assignés,
- les rôles *crochets* qui permettent d'encadrer l'exécution des fonctionnalités des mêmes composants, et
- les rôles *filtres* qui ajoutent la possibilité de filtrer les événements reçus et/ou émis par les composants modifiés.

Nous détaillons ci-dessous ces trois catégories sur deux exemples de services adaptatifs dans les domaines du *filtrage de contenu* et de la *résistance aux pannes*.

Rôles fonctionnels

Le propos d'un rôle fonctionnel est - comme son nom l'indique - d'ajouter une nouvelle fonctionnalité à un composant existant et, potentiellement, en cours de fonctionnement au moment de l'ajout. Nous avons vu qu'une fonctionnalité de composant correspond au traitement d'un type d'événement donné. Ainsi, un rôle fonctionnel décrit la réception d'un *nouveau* type d'événement, auparavant non supporté par le composant.

Considérons à nouveau notre exemple de protocole *publish/subscribe* et son application pour les notifications systèmes. Dans la configuration de type *émetteurs/médiateur/récepteurs*, nous avons vu que le rôle du médiateur était de servir d'aiguilleur ou de filtre entre les émetteurs et les récepteurs. Si la définition de ce médiateur était très simple, c'est que nous pouvions nous reposer presque entièrement sur le mode de communication spécifique de la plateforme Comet. Cependant, le filtrage exercé par le médiateur ne pouvait se faire - pour les mêmes raisons - qu'en fonction du type des événements reçus.

Une extension souhaitable de ce *filtrage en fonction du type* (*type-based filtering*) - et donc relativement «grossier» - consisterait à sélectionner les événements à communiquer selon leur *contenu* (*content-based filtering*). La figure 8.2 représente une configuration pour notre service de notification avec filtrage de contenu. Ainsi, par exemple, les événements dirigés vers l'analyseur de disque (composant de type DiskAnalyzer) sont contraints par la règle de filtrage :

(getImmediateValue() > 500) **and** (getAverageValue() <= 400)

Autrement dit, les événements de type DiskInfoEvent ne sont transmis à l'analyseur que si l'indice immédiat dépasse la valeur de 500 et l'indice en moyenne est inférieur à 400[5].

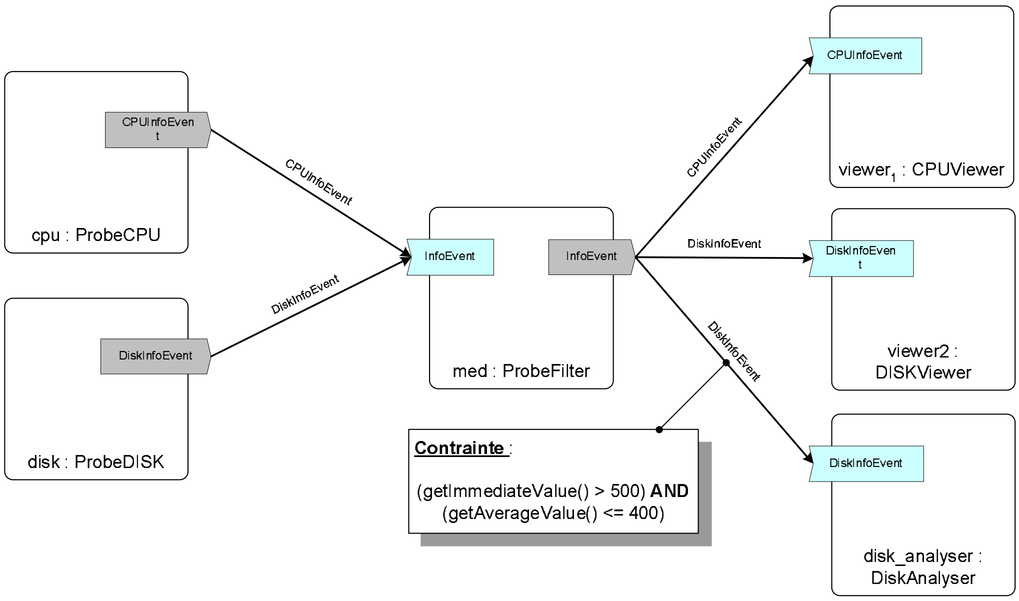


FIG. 8.2 – *Contrainte de filtrage sur le système de notification*

Pour réaliser ce filtrage en fonction du contenu, la première approche possible, celle que l'on trouve en général dans les *frameworks* pour le *publish/subscribe*, serait d'opérer de façon *ad-hoc*, en créant un service spécifique[6] rendant explicite les différentes phases du protocole :

- Enregistrement des émetteurs et récepteurs auprès du médiateur.
- Configuration du médiateur pour opérer du filtrage en fonction du type ou du contenu.

5. De façon intuitive, cette règle sélectionne des indices en crête pour une unité de disque modérément chargée en moyenne.
6. Pour l'instant, ce service n'existe pas en tant que tel puisqu'il est fourni *directement* par la plateforme.

Cette approche, quoique intéressante, nous obligerait cependant à définir des hiérarchies séparées de types d'événement pour les protocoles *publish/subscribe* (par exemple pour gérer l'enregistrement des émetteurs/récepteurs). Il faudrait également créer des définitions de composants plus spécifiques pour gérer ces événements. En d'autres termes, le protocole lui-même se verrait spécialisé, remettant en cause sa caractéristique de généricité.

Les protocoles et rôles Comet nous offrent une solution à la fois *plus générale* et *plus simple* pour supporter ce filtrage en fonction du contenu. L'idée est de sortir de la restriction du domaine de la notification asynchrone et de réfléchir en terme de *service de filtrage de contenu* indépendant et générique. Pour l'instant, nous allons simplement nous intéresser au problème de la configuration des filtres de contenu, nous expliquerons le procédé de filtrage proprement dit dans la section suivante.

Du point de vue de l'interstitiel Comet, un filtre de contenu est une entité dont l'objectif est de sélectionner des événements à partir de règles de filtrage opérant sur le contenu de ces événements.

La classe Java FilterRule et ses sous-classes, organisées en *motif composite* (cf. diagramme de classe UML de la figure 8.3) permettent de définir de tels filtres, dans un formalisme basé sur la logique propositionnelle[7].

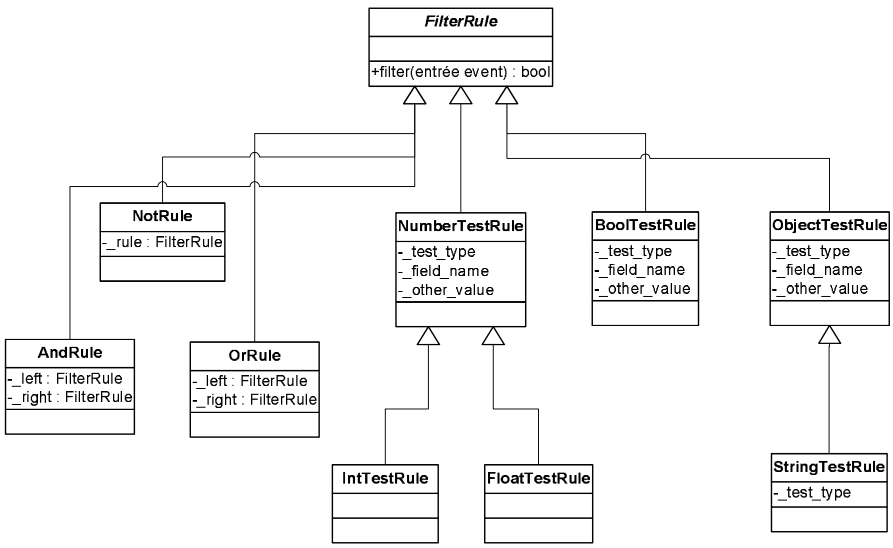


FIG. 8.3 – *Hiérarchie des classes de filtrage de contenu (extrait)*

Chaque filtre axiomatique propose une méthode filter() qui prend en paramètre un événement Comet et retourne un booléen indiquant si l'événement est filtré positivement ou négativement par la règle. Les filtres sont combinables grâce aux axiomes AndRule (et logique), OrRule (ou logique) et NotRule (non logique). Les classes de test permettent de filtrer des types de base de Java ou des objets, à partir d'accesseurs[8] des événements considérés. Le types de tests sont définis par des constantes symboliques (cf. table 8.3 page suivante).

Par exemple, pour créer la règle de filtrage associée au composant disk_analyzer de la figure 8.2 page précédente, on écrira :

```
disk_filter = new AndRule( new IntTest("getImmediateValue", GREATER, 500),
                           new IntTest("getAverageValue", LOWER_OR_EQUAL , 400));
```

Les événements génériques, de type FilterConfigureEvent, permettent d'envoyer une configuration de filtre à un composant distant. Les principaux accesseurs sont getFilterRule() et isPositiveFilter(). Le premier est utilisé pour récupérer la configuration de filtrage, sous la forme d'une règle FilterRule. Le deuxième indique si le filtre est *positif* ou *négatif*. Dans le premier cas, un événement testé positivement par la règle de filtre continuera «tranquillement» son cheminement au sein du composant. Tous les autres événements seront en revanche rejetés. Un filtre négatif effectue exactement le travail inverse.

Pour ce qui concerne la partie configuration de filtre de contenu, il est donc nécessaire de disposer d'un composant acceptant et sachant interpréter ces événements FilterConfigureEvent. Une première possibilité consisterait

7. Le filtre est lui-même quantifié universellement sur le type d'événement qu'il est censé filtrer.
8. Le statut d'accesseur est généralement attribué à une méthode sans argument et retournant la valeur d'un champ de l'événement.

Classe associée	Constante de test	Params	Règle de filtrage
NumberTest	TYPE	f	filtré si champ f est un nombre (du bon type)
	POSITIVE	f	filtré si champ f positif
	STRICT_POSITIVE	f	idem strictement positif
	LOWER	(f,v)	champ f strictement inférieur à valeur v
	LOWER_OR_EQUAL	(f,v)	idem supérieur ou égal
	GREATER	(f,v)	idem strictement supérieur
	GREATER_OR_EQUAL	(f,v)	idem supérieur ou égal
BoolTest	TYPE	f	filtré si champ f est booléen
	TRUE	f	filtré si champ f vrai
	FALSE	f	filtré si champ f faux
	COMPARE	(f,v)	comparaison avec la valeur booléenne v
ObjectTest	TYPE	f	filtré si champ f est un objet
	STRUCT_EQUAL	(f,o)	équivalence structurelle avec autre objet o
	REF_EQUAL	(f,o)	équivalence référentielle avec autre objet o
StringTest	TYPE	f	filtré si champ f est un chaîne de caractères
	EMPTY	f	filtré si champ f est la chaîne vide
	NON_EMPTY	f	test «inverse»
	EQUAL	(f,s)	égalité avec la chaîne s
	CONTAINS	(f,s)	filtré si f est dans s
	REGEXP	(f,r)	filtré si f est reconnu par l'expression rationnelle r

TAB. 8.3 – *Catégories de tests disponibles pour le filtrage de contenu (extrait)*

à spécialiser notre médiateur pour ajouter cette fonctionnalité de configuration de filtre. Encore une fois, nous pouvons éviter cette restriction (ici, aux médiateurs pour le service de notification) en décrivant un rôle fonctionnel prenant en charge ces configurations :

```
role GenericFilterRole {

  receive FilterConfigureEvent;

  when (FilterConfigureEvent config) {
    _filter_rule = config.getFilterRule();
    _positive_filter = config.isPositiveFilter();
  }

  protected FilterRule _filter_rule = null;
  protected boolean _positive_filter = false;
}
```

Ce rôle peut ensuite être *assigné dynamiquement* à un composant déjà déployé, en utilisant la commande assign (décrite en table 8.4).

Assignations de rôle
String **assign**(String id, String address [, **int** port], String protocol, String role) **Définition :** Assignation de role, pour le protocole protocol, au composant id
unassign(String rid, String cid, String address [, **int** port]) **Définition :** Désassignation du rôle identifié par rid du composant cid

TAB. 8.4 – *Plateforme Comet : commandes d'assignation dynamique de rôle*

Considérons par exemple l'assignation de ce rôle au médiateur de notification :

```
String role_id = assign("GenericFilterRole", mediateur_id, mediateur_address);
```

Dans cet extrait de protocole, les variables mediateur_id et mediateur_address représentent respectivement l'identificateur et la localisation du composant de médiation. La chaîne de caractère role_id permet d'identifier un rôle de façon unique pour un composant donné[9] et est principalement utilisée par la commande de retrait de rôle : unassign().

Un rôle ne pouvant être fourni en dehors d'une définition de protocole, nous devons proposer une «enveloppe» pour l'ajout dynamique de filtrage de contenu, par exemple :

```
protocol ContentBasedFiltering {

  String add_filter(String id, String address) {
    String role_id = assign("GenericFilterRole", id, address);
    return role_id;
  }
}
```

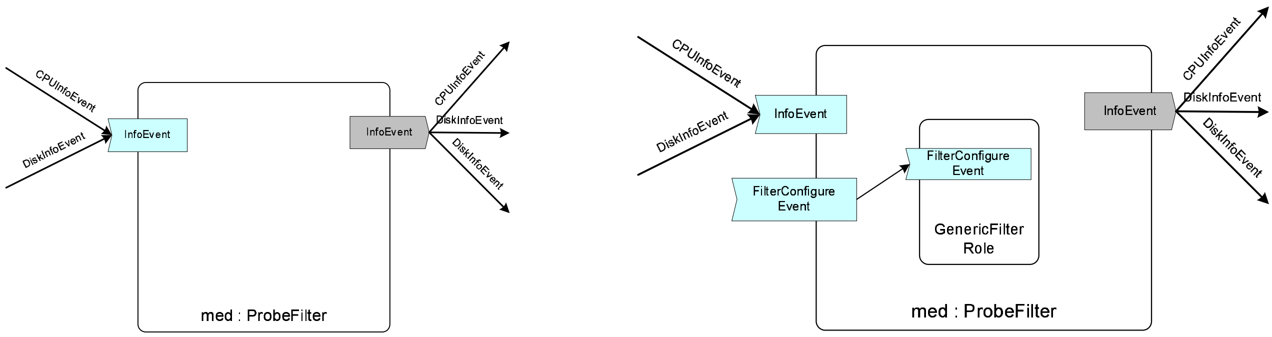
La fonctionnalité add_filter peut être utilisée par toute définition héritée de ContentBasedFiltering pour ajouter un rôle de filtrage de contenu à *n'importe quel* composant. Ce protocole décrit exactement ce qu'il est nécessaire de fournir pour supporter le filtrage : un composant donné sur lequel on a greffé dynamiquement le rôle nécessaire.

Revenons à notre exemple en considérant le médiateur de notification système avant l'assignation d'un rôle de filtrage de contenu. Ce composant se trouve dans la configuration présentée en figure 8.4(a).

Nous pouvons maintenant écrire un script permettant de réaliser l'assignation :

```
protocol CBFExample1 extends ContentBasedFiltering {
  main(String mediateur_id, String mediateur_address) {
    add_filter(mediateur_id, mediateur_address);
  }
}
```

Après invocation de la fonction main() du protocole, *sans pour autant interrompre le fonctionnement du médiateur*, ce même composant passe dans la configuration représentée en figure 8.4(b).



(a) Configuration normale (b) Après assignation du rôle de filtrage

FIG. 8.4 – *Adaptation du composant médiateur pour le filtrage de contenu*

Tout se passe comme si nous avions dynamiquement ajouté une fonctionnalité (et donc un type entrant) au composant, sans le perturber et en préservant son identité et son «propos».

Dorénavant, notre médiateur dispose d'une nouvelle fonctionnalité - pour l'instant partielle - de filtrage qui peut être configurée. Avant l'assignation du rôle GenericFilterRole, le composant ne «savait» pas comment configurer le filtre de contenu; ce qu'il «sait» faire désormais.

Il existe un large panel d'adaptations dynamiques pouvant être prises en charge par des rôles fonctionnels. Ils permettent par exemple la mise en place des systèmes de *plugins*, dans une approche plus structurée que la plupart des systèmes comparables[10]. De fait, il n'existe pratiquement aucune limite sur la portée de tels rôles,

9. Il n'y a pas d'unicité globale pour l'identification des rôles. Deux rôles assignés à deux composants différents peuvent donc partager un même identificateur.

10. Les rôles et les protocoles fournissent un cadre de plus haut-niveau que le respect d'une interface, par exemple de bibliothèque utilitaire pour le langage sous-jacent.

outre le fait qu'il faudra proposer des mécanismes opérationnels pour vérifier l'impact de leur assignation sur les composants du système. Nous reviendrons sur ces points précis par la suite, après avoir introduit les deux autres catégories principales de rôles.

Rôles filtres

Si l'assignation dynamique d'un rôle fonctionnel ouvre de grandes perspectives d'adaptation de grain fin, notre exemple n'est pour l'instant que peu explicite à ce sujet puisque la fonctionnalité attendue de filtrage de contenu n'est pas réalisable «facilement» par ce type de greffe.

Ce genre de fonctionnalité - au large potentiel de spécialisation - est pris en charge par une catégorie de rôle dédiée : les *rôles filtres*. Contrairement aux rôles fonctionnels qui - structurellement parlant - ne font qu'ajouter des fonctionnalités et des types entrant (et/ou sortant) aux composants, sans interférer - toujours structurellement parlant - avec ces derniers, les rôles filtres «perturbent» le fonctionnement normal (ou plutôt, avant assignation) des composants, selon deux procédés complémentaires :

- le filtrage des événements *reçus*, et
- le filtrage des événements *émis*.

Dans le premier cas, on parlera de *filtre entrant* (ou *in filter*), et dans le deuxième cas, de *filtre sortant* (ou *out filter*).

Dans les deux cas, un filtre représente une fonctionnalité de rôle, associée à un type d'événement (entrant ou sortant, suivant la catégorie de filtre) et retournant un booléen. Si le booléen retourné est true (ou vrai), l'événement est «extrait» des files d'attente (entrante ou sortante) du composant considéré. Le booléen false dénote une poursuite «normale» du traitement de l'événement filtré. Cette règle correspond à un *filtrage négatif* par défaut. En cas de *filtrage positif*, il faudra donc retourner le booléen opposé.

Pour le filtre entrant, l'événement est filtré *avant* exécution de la fonctionnalité associée à l'événement. Le filtre sortant réalise pour sa part son «travail» *avant* émission proprement dite de l'événement.

Pour réaliser un rôle générique de filtrage de contenu, nous pouvons choisir l'une des deux solutions : filtre entrant ou sortant. Nous devons pour cela «augmenter» le rôle GenericFilterRole, ce qui peut se faire par simple héritage.

Par exemple, pour le filtrage entrant, nous écrirons :

```
role GenericInFilterRole extends GenericFilterRole {
  infilter Event;

  boolean filter in (Event event) {
    if(_filter_rule==null)
      return false;
    else if(_positive_filter)
      return !_filter_rule.filter(event);
    else
      return _filter_rule.filter(event);
  }
}
```

En plus de la fonctionnalité de configuration de filtre, le bloc de filtrage filter in (correspondant à la déclaration de type infilter Event) intercepte tout événement de type Event ou d'un sous-type[11] puis vérifie si le contenu de cet événement est filtré par la règle _filter_rule associée au rôle (si elle existe).

L'assignation de ce nouveau rôle au médiateur correspond à la figure 8.5 page suivante.

Nous pouvons voir qu'en plus de l'ajout d'une fonctionnalité (configuration du filtre), le rôle intercepte les événements reçus par le composant. Ceci implique bien évidemment un test de cohérence structurelle puisque le composant doit recevoir des événement «au moins» compatibles avec le type filtré par le rôle. Dans notre exemple, cette condition est trivialement vérifiée puisque le composant reçoit des InfoEvent et que le rôle filtre les Event, plus génériques.

11. Nous sommes donc bien ici en présence d'un rôle de filtrage générique puisque tous les types d'événements sont sous-types de Event.

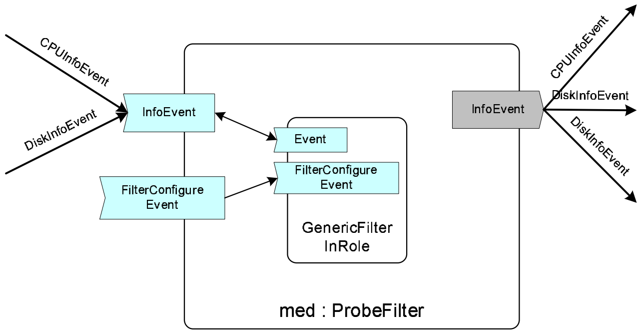


FIG. 8.5 – *Adaptation complète pour le filtrage de contenu en entrée*

Nous pouvons de même décrire un rôle générique de filtrage sortant, de la façon suivante :

```
role GenericOutFilterRole extends GenericFilterRole {
  outfilter Event;

  boolean filter out (Event event) {
    if(_filter_rule==null)
      return false;
    else if(_positive_filter)
      return !_filter_rule.filter(event);
    else
      return _filter_rule.filter(event);
  }
}
```

Cette fois-ci, l'assignation au médiateur de ce rôle correspondra à la figure 8.6. La cohérence structurelle de cette assignation porte désormais sur l'adéquation entre le type filtré par le rôle et le type sortant au niveau du composant.

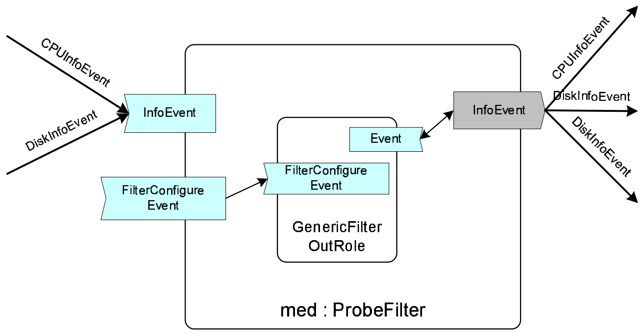


FIG. 8.6 – *Adaptation pour le filtrage de contenu en sortie*

Si le propos général des rôles filtres - comme leur dénomination l'indique - concerne le filtrage des événements, leur caractérisation spécifique permet de faire totalement abstraction de la méthode employée pour le filtrage. L'exemple choisi pour illustrer cette section, s'il repose sur un modèle de filtrage assez simple (règles d'ordre 0), est particulièrement intéressant par sa caractéristique de *généricité* : le filtrage de contenu est disponible *partout* et *à tout moment*.

Rôles crochets

La dernière catégorie importante de rôle est en fait une variante moins intrusive que les filtres : les rôles *crochets* (ou *hooks* en anglais). Leur propos rejoint un mécanisme fréquemment rencontré dans les frameworks réflexifs et permettant «d'encadrer» une fonctionnalité de composant.

Un crochet *avant* (ou *prehook*) sera ainsi déclenché immédiatement avant le traitement «normal» d'un événement reçu, puis ce sera le tour de l'éventuel crochet *après* (ou *posthook*). Ces crochets ne peuvent, contrairement aux filtres, interrompre le cheminement des événements mais ils peuvent en revanche effectuer des prétraitements et des posttraitements variés : modification de l'état du composant, émission d'événement, etc.

Nous allons illustrer ce type de rôle en mettant en place un protocole de *réplication*, outil très intéressant dans le cadre de la résistance aux pannes.

Pour ce type de service, on distingue généralement trois types d'entité :

1. les *clients* qui émettent des requêtes,
2. les *serveurs* qui traitent ces requêtes (et, potentiellement, retournent des réponses),
3. les *répliques* qui maintiennent la copie de propriétés liées aux serveurs.

En cas de panne d'un serveur, on utilise les informations disponibles au niveau de la (ou des) réplique(s) associée(s) pour «reconstruire» dynamiquement le service. Diverses catégories de réplication sont possibles, selon la nature des répliques. Aux deux extrémités de ce spectre, on trouve la réplication *active* et la réplication *passive*. Dans le premier cas, les répliques sont des clones des serveurs qui, à un moment donné, ont le même état que le serveur associé (mais à un moment antérieur, dit *sûr*). Ils traitent donc les mêmes requêtes que le serveur et possèdent les mêmes propriétés. Au contraire, la réplication *passive* met en œuvre des répliques qui ne font que stocker les requêtes des serveurs, sans les traiter.

En cas de panne, ces deux catégories vont bien évidemment réagir de façon distincte. Dans le cas *actif*, une réplique devra être élue comme nouveau serveur, ce qui n'est pas très coûteux. En revanche, un réplique passive, après avoir été choisie, servira à rejouer les requêtes perdues lors de la panne; procédé dit de *rollback* et clairement plus coûteux que le cas actif. De façon quelque peu empirique, on peut dire que l'approche passive favorise le fonctionnement normal alors que la réplication active fournit les meilleures réactions en cas de panne. Bien sûr, les solutions généralement proposées dans la littérature reposent sur des compromis, certaines plutôt actives et d'autres plutôt passives.

Pour revenir à notre propos, qui concerne l'illustration des rôles crochets, nous pouvons remarquer que pour toute forme de réplication, il existe un préliminaire incontournable : l'interception des requêtes reçues au niveau du serveur.

Supposons que le composant serveur propose l'interface suivante :

```
component Serveur {
  receive RequestEvent;
  send ReplyEvent;

  when (RequestEvent request) {
    // Traitement de la requête
    send( new ReplyEvent( ... ) );
  }
}
```

Il nous est possible de décrire un rôle générique (pour le type RequestEvent) offrant le service attendu de recopie transparente des réceptions de requêtes au niveau du serveur :

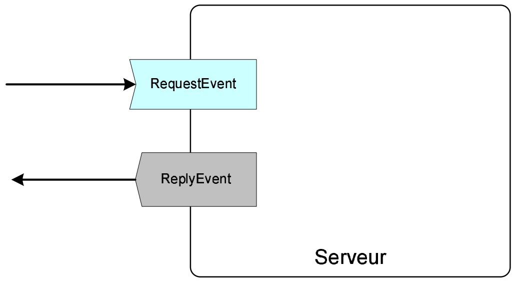
```
role ReplicationRole {
  prehook RequestEvent;
  send RequestEvent;

  pre hook (RequestEvent request) {
    send(request);
  }
}
```

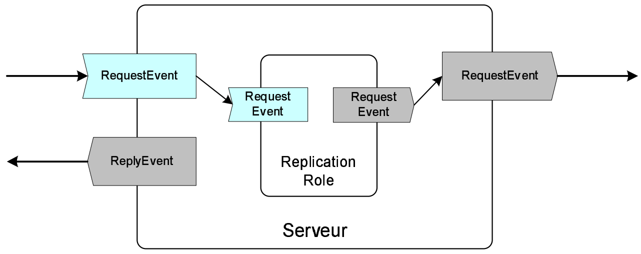
Ce rôle définit un crochet avant traitement des événements de type RequestEvent dont le contenu concerne leur émission pure et simple. Le traitement de l'événement ne se trouvera bien sûr pas perturbé par ce «détour» (modulo ralentissement possible du composant qui se voit assigner ce rôle).

Après assignation de ce rôle au composant serveur de la figure 8.7(a) (page 157), nous obtenons le composant de la figure 8.7(b), adapté dynamiquement selon deux aspects :

– aspect structurel : le serveur dispose d'un nouveau type sortant, ici RequestEvent, émis par le rôle.



(a) Serveur en configuration normale



(b) Serveur prêt pour la réplication

FIG. 8.7 – *Adaptation d'un serveur pour la réplication*

- aspect comportemental : les requêtes sont interceptées au niveau du crochet; cette interception ayant pour principale conséquence l'émission d'une copie de ces requêtes.

Ceci nous permet de construire un protocole de réplication (par exemple passive), dont l'interprétation correspondra à l'adaptation, bien sûr dynamique et sans modification des descriptions des composants clients et serveurs, représentée sur les figures successives 8.8 (sans réplication) et 8.9 (avec réplication). Nous pouvons constater encore une fois l'intérêt de la diffusion multi-point implicite en fonction du type, qui permet de faire varier dynamiquement et de façon transparente le nombre de répliques associées à un serveur donné.

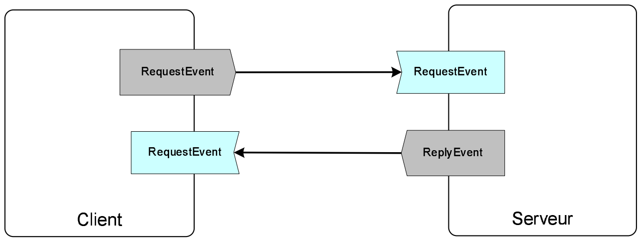


FIG. 8.8 – *Système client/serveur en configuration normale*

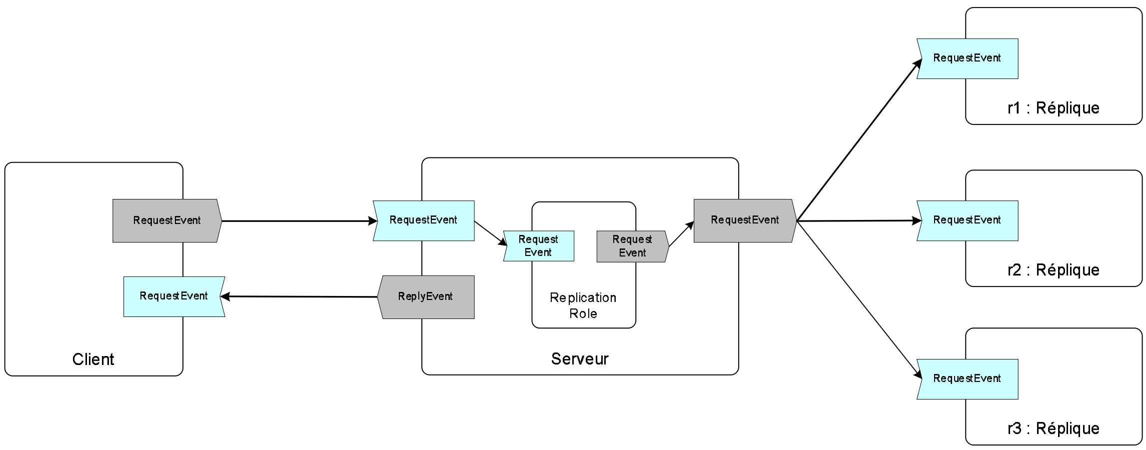


FIG. 8.9 – *Système client/serveur adapté dynamiquement pour la réplication*

Dans le cadre de l'utilisation du modèle de comportement par défaut (enrichi des assignations dynamiques de rôles), nous pouvons de plus assurer - grâce à l'acheminement fiable et ordonné du modèle de communication associé - que les requêtes seront reçues par les différentes répliques dans le même ordre que leur réception au niveau du serveur, pour une connexion client/serveur donnée [12]. En pratique, il faudra quelque peu «tempérer» cette propriété d'ordre puisqu'elle dépendra de l'infrastructure de communication de bas-niveau.

12. Le cas des clients multiples pourra conduire à des entrelacements non-déterministes de requêtes mais pour des clients *différents*.

Rôles hybrides

Il est évident que les concepts de rôles fonctionnels, de rôles filtres et de rôles crochets sont tout à fait compatibles et peuvent donc être composés. Nous illustrerons ce fait sur un exemple très intéressant de *trace conditionnelle généralisée*, qui montre de plus les possibilités nombreuses de réutilisation de rôles tels que ceux décrits précédemment.

Les mécanismes de trace en cours d'exécution sont souvent complexes et en général décrits au sein même du code fonctionnel des entités logicielles dont on désire connaître le fonctionnement effectif. En effet, il semble difficile de tracer des informations opérationnelles sans connaître les caractéristiques principales de ces informations. Cependant, certaines boîtes-à-outils réflexives permettent de séparer les descriptions de trace des fonctionnalités concernées mais la mécanique sous-jacente est généralement assez lourde et de bas niveau.

Les rôles Comet offrent des avantages indéniables concernant la mise en œuvre de ces mécanismes de trace sophistiqués pour ce qui concerne le cheminement des événements au sein des composants. Les crochets permettent par exemple de détecter simplement le passage d'événements au niveau de l'étage de traitement des composant. Les filtres sortant sont pour leur part très utiles dans la détection des événements émis par un composant donné. Puisque dans les deux cas, la mise en œuvre est quasiment identique, nous nous limiterons dans cet exposé aux traces «crochets».

Dans notre démarche de réutilisation, le filtrage de contenu peut être employé à nouveau pour contrôler de manière fine quels événements doivent être signalés ou non. Nous pouvons également tirer parti des fonctionnalités de rôles pour configurer la trace ou encore autoriser ces «traces» à être communiquées (voire diffusées) au monde extérieur.

Commençons par décrire les deux catégories d'événements mises en jeu dans le cadre de la trace : configuration et signalement.

```
// Configuration d'une trace
event TraceConfigureEvent is FilterConfigureEvent {
  slot _ctrl type enum[ADD,REMOVE];

  // Ajout ou retrait d'une règle de trace ?
  int getControl() { return _ctrl; }
}

// Evénement de trace
event TraceEvent {
  slot _traced type Event;
  slot _timestamp type int;

  // Construction d'une trace
  TraceEvent(Event traced, int ts) {
    _traced = traced;
    _timestamp = ts;
  }

  // Evénément tracé
  Event getTrace() { return _traced; }

  // Temps local à la source pour la trace
  int getTimestamp() { return _timestamp; }
}
```

Les événements de configuration propose une information de contrôle (ajout ou retrait d'une règle de trace) ainsi qu'un règle de filtrage de contenu indiquant quels événements doivent être signalés. Il est intéressant de constater que cette définition est pratiquement identique à celle utilisée pour les événements de filtrage générique. Les traces proprement dites sont des évenements englobant l'événement tracé ainsi qu'un estampillage permettant de conserver l'ordre de signalement des événements[13].

À partir de ces informations, nous pouvons maintenant définir notre rôle de traces conditionnelles pour l'étage d'exécution des composants Comet :

13. Quoique non montré ici, cet estampillage est utile pour optimiser l'émission des traces, en attendant un certain temps et/ou un certain nombre de traces avant de les signaler.

```
role GenericExecutionTracer {
  prehook Event;
  receive TraceConfigureEvent;
  send TraceEvent;

  // Configuration
  when(TraceConfigureEvent config) {
    if(config.getControl()==TraceConfigureEvent.ADD)
      _filter_rules.add(config.getFilterRule());
    else _filter_rules.remove(config.getFilterRule());
  }

  // Crochet de trace
  before (Event event) {
    iterator rules_iter = _filter_rules.iterator();
    while(rules_iter.hasNext())
      if(((FilterRule) rules_iter.next()).filter(event))
        send(new TraceEvent(event,_timestamp++);
  }

  // Règles de filtrage
  protected ArrayList _filter_rules = new ArrayList();

  // Estampillage local
  protected int _timestamp = 0;
}
```

Le crochet entrant décrit ci-dessus permet de d'effectuer la sélection entre les événements devant être tracés et ceux qui ne seront pas signalés avant traitement. Pour effectuer ce choix, le rôle de trace gère une liste de règles de filtrage qui peuvent être ajoutées ou retranchées du rôle grâce à la fonctionnalité de configuration.

Nous donnons ci-dessous un extrait de protocole pour exploiter ces possibilités adaptatives de trace :

```
protocol GenericTraceProtocol {

  // Assignation d'un rôle de trace à un composant donné
  String traceExecution(String id, String address) {
    String trace_id = assign("GenericExecutionTracer", id, address);
    return trace_id;
  }

  // ajout/retrait de règle de trace
  void addTrace(FilterRule fule, String id, String address) {
    send(new TraceConfigureEvent(ADD,rule), id, address);
  }
  void removeTrace(FilterRule fule, String id, String address) {
    send(new TraceConfigureEvent(REMOVE,rule), id, address);
  }
}
```

Prenons l'exemple d'un composant quelconque, dont l'identificateur est spam et qui tourne sur la machine egg. Commençons par ajouter (dans un script intéractif), un rôle de trace à ce composant :

```
traceExecution("spam","egg");
==> "role1"
```

Supposons que ce composant émettent des événements de différents types mais que nous soyons intéressés principalement par les événement de type MontyEvent. Dans ce cas, nous pourrons utiliser une règle de filtrage simple :

```
addTrace( new ObjectTestRule(TYPE, "MontyEvent"), "spam", "egg");
```

Désormais, avant chaque traitement d'un événement du type MontyEvent, un signal sera émis à tout composant connecté au composant pour le type TraceEvent. Le dévermineur intégré à l'environnement de contrôle Halley (cf. chapitre 10) est un bon exemple d'un tel composant.

Notre exemple très énigmatique montre que ce protocole de trace est très générique et peut être employé pour observer l'exécution de *n'importe quel composant Comet*. Si par exemple nous voulons savoir quand le médiateur de notre service de sondes vu au chapitre précédent diffuse des événements dont l'indice immédiat est supérieur à 500 et dont l'indice moyen ne dépasse pas 400, on pourra ainsi écrire :

```
traceExecution("med","stella.lip6.fr");
==> "role1"
trace_rule = new AndRule( new IntTest("getImmediateValue", GREATER, 500),
                          new IntTest("getAverageValue", LOWER_OR_EQUAL, 400));
addTrace(trace_rule, "med", "stella.lip6.fr");
```

Par la suite, lorsque ces traces particulières ne nous intéresseront plus, nous pourrons simplement invoquer :

```
removeTrace(trace_rule, "med", "stella.lip6.fr");
```

Et si au final, l'on désire rendre au composant sa «tranquilité», il est très simple de le soustraire à ce protocole de trace :

```
unassign("role1", "med", "stella.lip6.fr");
```

Ces exemples interactifs montrent, assez efficacement selon nous, les caractéristiques de généricité et de transparence du protocole de trace mis en place. Nous n'avons de plus fait qu'en survoler les principes de base puisqu'il est possible de raffiner les rôles et protocoles de trace pour les adapter à de multiples besoins. Ces raffinements seront (à terme) disponibles dans l'environnement de contrôle des systèmes Comet.

Pour terminer, il est important de signaler que nous trouverons une autre (ré)utilisation assez avancées des rôles et des concepts associés au chapitre 9, lorsque nous mettrons en place des protocoles de synchronisation de flux.

8.2.2 Impact structurel

Notre description des rôles, au travers d'exemples applicatifs *opérationnels*, n'en ressemble pas moins à une sorte de «pétition de principe». En effet, rien ne nous dit pour l'heure si les constructions linguistiques et mécanismes sous-jacents sont réalistes et s'ils peuvent être mis en place *formellement* dans notre modèle. Bien sûr, leur «découverte» correspond à une sorte de parcours inverse qui a permis d'identifier dans le modèle les «lieux» possibles de spécialisation. L'ajout du support pour les rôles correspond donc elle aussi à une spécialisation du modèle de comportement canonique (décrit au chapitre 7).

Schéma d'état

Tout d'abord, un rôle est défini formellement par le schéma suivant :

Role

$id : IDENTIFIER$

$in\_type : \mathbb{P}\ TYPE$
$out\_type : \mathbb{P}\ TYPE$

$in\_filters : \mathbb{P}\ FilterFunctionality$
$out\_filters : \mathbb{P}\ FilterFunctionality$

$pre\_hooks : \mathbb{P}\ Functionality$
$post\_hooks : \mathbb{P}\ Functionality$

$functs : \mathbb{P}\ Functionality$

$\forall t : in\_type \bullet \exists f : functs \bullet t \preceq fs.type$

Les variables *id*, *in_type* et *out_type* décrivent l'identificateur et le type du rôle, avec une définition identique à celle des composants standard (cf schéma *DefaultComponent* page 129). Les filtres sont déclarés par *in_filters* (filtres entrant) et *out_filters* (filtres sortant).

Le type *FilterFunctionality* décrit une action de filtre qui correspond au schéma *Functionality* (décrit page 120) mais en ajoutant une action de filtrage :

$$
\begin{array}{|l}
\textit{FilterFunctionality} \\
\hline
type : TYPE \\
activation : \mathbb{P}\,\mathrm{seq}\,FilterAction \\
\hline
\forall\, actions : activation \bullet \mathrm{last}\ actions \in (FilterPositiveAction \cup FilterNegativeAction) \\
\hline
\end{array}
$$

avec :

$$FilterAction \mathrel{\widehat{=}} SendAction \vee SendExplicitAction \vee FilterPositiveAction \vee FilterNegativeAction$$

En pratique, il faudra toujours s'assurer de terminer une activation (ou séquence d'actions) par *FilterPositiveAction* (poursuite normale du traitement) ou *FilterNegativeAction* (rejet de l'événement).

A partir de cette définition, il nous faut proposer un moyen simple de résumer l'activation d'un filtre à une valeur booléenne : *true* si l'événement est filtré positivement et est donc rejeté; et *false* si l'événement est accepté. Ces valeurs sont introduites par une définition de syntaxe :

$$bool ::= false \mid true$$

Pour savoir si un filtrage est positif ou non, nous utiliserons la fonction suivante :

$$
\begin{array}{|l}
isFiltered : (\mathrm{seq}\,FilterAction, Event) \rightarrow\!\!\!\!\!+ bool \\
\hline
\forall (f,e) : \mathrm{dom}\,isFiltered \bullet f \neq \langle\rangle \wedge e \neq \emptyset \\
isFiltered(f,e) = true \Leftrightarrow \mathrm{last}\ f = FilterPositiveAction \\
isFiltered(f,e) = false \Leftrightarrow \mathrm{last}\ f = FilterNegativeAction
\end{array}
$$

A partir d'un couple (activation de filtre,événement reçu) *non vide*, la fonction *isFiltered* retourne *true* si la séquence d'actions correspondant à l'activation se termine par un filtrage positif ou *false* s'il s'agit d'un filtrage négatif.

Les crochets *pre_hooks* (avant) ou *post_hooks* (après), pour leur part, sont des fonctionnalités «comme les autres», elle même dénotées par *functs*.

L'unique condition d'invariant du schéma *Role* porte sur la vérification du type de ce dernier. Ainsi, il faudra s'assurer qu'il existe bien une fonctionnalité associée à chaque type entrant du composant.

Nous pouvons voir sur ce schéma d'état que les rôles sont bien, formellement parlant, des sortes de «composants dégénérés», avec une partie structurelle plus riche et une partie comportementale pratiquement vide (pas de comportement associé mais uniquement des fonctionnalités).

Modèle de composant avec support pour les rôles

Les rôles sont manipulés par les composants, tant au niveau structurel que comportemental. Décrivons pour l'instant l'impact structurel sur le schéma *DefaultComponent* qui nous servira de base à la spécialisation proposée[14] :

14. Contrairement aux autres spécialisations proposés dans cette thèse, le modèle avec support des rôles - en raison de l'importance du concept - ne renomme pas les schémas raffinés. Il s'agit donc d'une extension du modèle canonique, plus qu'une simple spécialisation.

$$
\begin{array}{|l}
\hline
\textit{DefaultComponent} \\
\textit{AbstractComponent} \\
\textit{DefaultBehavior} \\
\\
\textit{role} : \mathbb{P}\,\textit{Role} \\
\\
\textit{basic\_in\_type} : \mathbb{P}\,\textit{TYPE} \\
\textit{basic\_out\_type} : \mathbb{P}\,\textit{TYPE} \\
\hline
\forall\,\textit{inct} : \{c : \textit{in\_channels} \bullet c.\textit{type}\} \bullet (\exists\,t : \textit{in\_type} \bullet t \succeq \textit{inct}) \\
\forall\,\textit{outct} : \{c : \textit{out\_channels} \bullet c.\textit{type}\} \bullet (\exists\,t : \textit{out\_type} \bullet t \succeq \textit{outct}) \\
\\
\forall\,t : \textit{in\_type} \bullet \exists\,\textit{fs} : \textit{functs} \bullet t \preceq \textit{fs}.\textit{type} \\
\\
\forall\,r : \textit{role} \bullet \\
\qquad \forall\,r_2 : \textit{role} \bullet r.\textit{id} = r_2.\textit{id} \Rightarrow r = r_2 \land \\
\qquad r.\textit{in\_type} \subseteq \textit{in\_type} \land \\
\qquad r.\textit{out\_type} \subseteq \textit{out\_type} \\
\hline
\end{array}
$$

Les rôles sont ajoutés au composant dans l'ensemble *role* dont chaque élément est bien sûr du type *Role*. Cette variable est associée aux propriétés décrites dans la dernière condition de l'invariant, qui vérifie pour chaque rôle :

- L'unicité de l'identification des rôles, au niveau du composant local (et non du système complet).
- L'inclusion de l'ensemble des types entrant du rôle dans l'ensemble des types entrant du composant.
- L'inclusion, similaire, du type sortant du rôle dans le type sortant du composant.

Bien sûr, cette modification structurelle a des portées comportementales que nous détaillerons par la suite. Les variables *basic_in_type* et *basic_out_type* permettent de conserver «en mémoire» le type de base du composant, indépendamment des rôles qui lui sont assignés. Il ne faut pas oublier, lors de l'initialisation, de construire ces deux ensembles.

Opération d'assignation

L'unique opération associée aux rôles, leur assignation, ne concerne que la spécialisation du modèle de composant. En effet, il ne s'agit pas ici de modifier le rôle mais plutôt le composant auquel il est assigné.

Cette opération fondamentale qui donne la sémantique de la primitive assign est définie de la façon suivante :

$$
\begin{array}{|l}
\hline
\textit{AssignRole} \\
\Delta\textit{DefaultComponent} \\
\\
\textit{role}? : \textit{Role} \\
\hline
\textit{role}?.\textit{id} \notin \{r : \textit{role} \bullet r.\textit{id}\} \\
\\
\forall\,\textit{if} : \textit{role}?.\textit{in\_filter} \bullet \exists\,t : \textit{in\_type} \bullet t \sim \textit{if}.\textit{type} \\
\forall\,\textit{of} : \textit{role}?.\textit{out\_filter} \bullet \exists\,t : \textit{out\_type} \bullet t \sim \textit{of}.\textit{type} \\
\\
\forall\,\textit{ph} : \textit{role}?.\textit{pre\_hooks} \bullet \exists\,t : \textit{in\_type} \bullet t \sim \textit{ph}.\textit{type} \\
\forall\,\textit{ph} : \textit{role}?.\textit{post\_hoooks} \bullet \exists\,t : \textit{in\_type} \bullet t \sim \textit{ph}.\textit{type} \\
\\
\textit{roles}' = \textit{roles} \cup \{\textit{role}?\} \\
\\
\textit{in\_type}' = \textit{in\_type} \cup \textit{role}?.\textit{in\_type} \\
\textit{out\_type}' = \textit{out\_type} \cup \textit{role}?.\textit{out\_type} \\
\hline
\end{array}
$$

L'opération *AssignRole* ne prend qu'un unique paramètre : *role*? et modifie en conséquence le composant qui se voit assigner ce rôle. La première précondition vérifie l'unicité, en terme d'identification, du rôle assigné.

Bien sûr, une telle assignation ne peut se faire de façon totalement arbitraire. Au niveau purement structurel, notre système de type et la relation de sous-typage associée vont nous permettre d'effectuer d'importantes vérifications préliminaires. Ces vérifications sont établies par des conditions d'invariant et portent sur :

- La vérification des propriétés de filtrage du rôle : chaque filtre entrant doit être d'un type compatible avec le type entrant du composant avant ajout (variable *in_type*). La même relation doit être vérifiée entre les filtres sortant du rôle et le type sortant du composant.
- La vérification des propriétés des crochets : ces derniers doivent encadrer une fonctionnalité existante du composant. Ils doivent donc être définis pour un type relié à au moins un type entrant du composant.

En d'autres termes, il ne sera pas possible d'assigner un rôle si les filtres et crochets qu'il définit ne sont pas compatibles, au sens du système de type de notre langage, avec le composant cible de l'assignation.

Si le rôle est structurellement acceptable, alors il est ajouté à l'ensemble des rôles gérés par le composant (variable $roles'$). De plus, les types entrant et sortant du composant après ajout (in_type' et out_type') sont modifiés.

Ce dernier «effet de bord» a un impact au niveau du système puisqu'il sera désormais possible d'établir des connexions de et vers le composant modifié pour interagir avec le rôle assigné, en particulier si ce dernier est un rôle fonctionnel. Réciproquement, le retrait d'un rôle devra s'effectuer uniquement si aucune connexion n'est établie spécifiquement pour le rôle précité. En dehors de ces considérations, le retrait de rôle n'est pas d'une grande complexité :

$$
\begin{array}{l}
\textit{UnAssignRole} \\
\hline
\Delta \textit{DefaultComponent} \\
\textit{id}? : \textit{IDENTIFIER} \\
\hline
\exists\, r : \textit{roles} \bullet r.\textit{id} = \textit{id}? \\
\textbf{let } \textit{role}? == \mu\, r : \textit{roles} \bullet r.\textit{id} = \textit{id}? \bullet \\
\quad (\forall\, \textit{inct} : \{c : \textit{in\_channels} \bullet c.\textit{type}\} \bullet (\forall\, \textit{it} : \textit{inct} \bullet \\
\quad\quad (\exists\, f : \textit{functs} \bullet \textit{it} \preceq f.\textit{type}) \vee \\
\quad\quad (\exists\, r : \textit{roles} \setminus \{\textit{role}?\} \bullet \textit{it} \preceq r.\textit{in\_type}))) \wedge \\
\quad (\forall\, \textit{outct} : \{c : \textit{out\_channels} \bullet c.\textit{type}\} \bullet (\forall\, \textit{ot} : \textit{outct} \bullet \\
\quad\quad \exists\, r : \textit{roles} \setminus \{\textit{role}?\} \bullet \textit{ot} \preceq r.\textit{out\_type})) \wedge \\
\quad \textit{roles}' = \textit{roles} \setminus \{\textit{role}?\} \wedge \\
\quad \textit{in\_type}' = \textit{basic\_in\_type} \cup \bigcup\{r : \textit{roles} \bullet r.\textit{in\_type}\} \wedge \\
\quad \textit{out\_type}' = \textit{basic\_out\_type} \cup \bigcup\{r : \textit{roles} \bullet r.\textit{out\_type}\} \\
\hline
\end{array}
$$

Le paramètre de «désassignation» est l'identificateur du rôle concerné qui doit bien entendu être présent dans le composant (première précondition). La variable locale $role?$ identifie ce rôle dans tout le reste du schéma. Tout d'abord, il est important de vérifier que les connexions effectives associées au composant (et rendues tangibles localement grâce aux canaux de communication qui les supportent) peuvent toutes être «traitées» par les fonctionnalités du composant ou les rôles restant après assignation. En d'autres termes, il faut s'assurer que le rôle que l'on se propose de retirer n'est pas «indispensable» au fonctionnement du composant.

Une fois cette vérification fondamentale effectuée, le rôle peut alors être retiré du composant (modification de la variable $roles'$). Pour mettre à jour le type du composant, la méthode consistant à retirer les types du rôle enlevé de *in_type* et *out_type* n'est pas adéquate puisque d'autres rôles (ou même le composant «originel») peuvent avoir besoin de ces types (cf. section 8.2.4 page 170). Une solution «propre» passe par la restauration du type primitif du composant et l'ajout par union généralisée (dénotée par le symbole $\bigcup$) de tous les types introduits par les différents rôles assignés au composant, à l'exception bien sûr du rôle retiré.

8.2.3 Impact comportemental

Les schémas de la section précédente nous permettent de comprendre ce que représente l'assignation dynamique d'un rôle à un composant donné en terme de perturbation de structure. Il nous reste à éclaircir les questions concernant les interactions incontournables avec le niveau méta-comportement.

Une des questions les plus importantes qu'il faut se poser à ce niveau est la suivante :

Que représente le «moment» d'activation d'un rôle ?

En d'autres termes, il est important de définir proprement ce qui fait la différence entre l'état du composant *avant* et *après* assignation du rôle.

Encore une fois, le raffinement des méta-objets comportementaux et des schémas associés sera notre principal outil pour construire les réponses formelles à nos questionnements concernant l'impact comportemental des rôles. Rappelons que les opérations comportementales, décrites dans leur version sans support pour les rôles au chapitre 7, portent sur le schéma d'état *DefaultComponent* (défini page 161) et en particulier la partie comportementale définie par le schéma *DefaultBehavior* (cf. page 116).

Il est intéressant de noter que l'impact comportemental des rôles se restreint à trois fonctionnalités comportementales :

- L'opération *DefaultMetaInFetch*, modifiée pour prendre en compte les rôles filtres entrant.
- L'opération *DefaultMetaOutFetch* pour les rôles filtres sortant.
- L'opération *DefaultMetaExec*, encadrée par les crochets avant et après et augmentée par les fonctionnalités de rôles.

Rôles filtres

Le principe d'un rôle filtre (entrant ou sortant) est, comme nous l'avons vu sur l'exemple de filtrage de contenu, d'intercepter les événements reçus ou à émettre avant leur traitement ou leur envoi pour éventuellement les filtrer. Il ne faut pas intervenir sur les phases de réception ou d'émission effectives qui doivent rester les plus efficaces possibles et ainsi préserver au maximum le caractère asynchrone du comportement. Pour les filtres entrant, l'idée est donc d'interposer entre la file d'attente en entrée (*DefaultMetaInQueue*) et la phase d'extraction (*DefaultMetaInFetch*) le filtre concerné. Ce principe est décrit pour le cas entrant en figure 8.10.

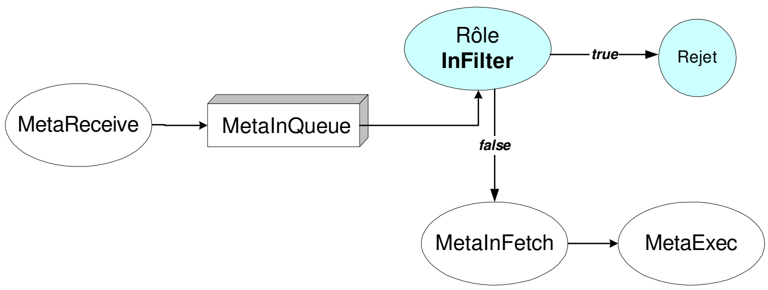


FIG. 8.10 – *Impact comportemental des rôles filtres entrant*

Au niveau du filtrage en sortie (avant émission), l'opération «pivot» est *DefaultMetaOutFetch*, comme le montre la figure 8.11.

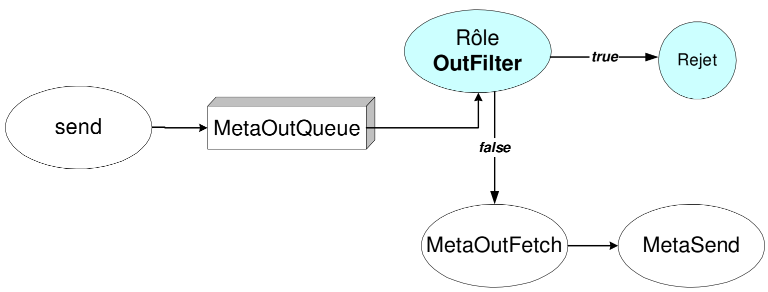


FIG. 8.11 – *Impact comportemental des rôles filtres sortant*

Bien sûr, il ne s'agit pas dans ces deux cas d'ajouter de nouvelles fonctionnalités comportementales à part entière, mais plutôt d'activer les filtres depuis des opérations existantes au niveau méta-comportement : *DefaultMetaInFetch* et *DefaultMetaOutFetch*.

Une première partie commune de ces deux fonctionnalités adaptées aux filtres de rôle prend en compte le filtrage négatif :

FetchFilterNegative

$queue? : \mathrm{seq}\, Event$

$activations? : \mathbb{P}\, \mathrm{seq}\, FilterActions$

$true \notin \{a : activations? \bullet isFiltered(a, \mathrm{last}\ queue?)\}$

Ce schéma vérifie qu'aucun filtrage positif ne résulte du «passage» du dernier événement de la file d'attente considérée dans les séquences d'actions associées aux activations passées en paramètres. En d'autres termes, ce schéma n'est valide que si le filtrage «totale» de l'événement considéré est négatif. Il est important de remarquer que cette opération n'est valide que si un événement est bien disponible dans la file d'attente (car *isFiltered* n'est pas valide avec un de ses arguments vide).

Le schéma complémentaire est le suivant :

FetchFilterPositive

$queue? : \mathrm{seq}\, Event$

$activations? : \mathbb{P}\, \mathrm{seq}\, FilterActions$

$true \in \{a : activations? \bullet isFiltered(a, \mathrm{last}\ queue?)\}$

Nous pouvons légitimement nous demander pourquoi dans ces schémas un ensemble d'activations est passé en paramètre alors qu'un filtre ne peut activer qu'une séquence d'action à un moment donné. En fait, il faut ici anticiper sur la section 8.2.4 page 170 en remarquant que plus d'un rôle filtre pourra être activé pour un événement reçu. En revanche, pour chaque filtre activé, il est impératif d'effectuer un choix *unique* concernant l'activation spécifique choisie[15].

Puisque nous n'avons pas accès, à ce niveau de raffinement, au contenu sémantique du filtre, le seul choix que nous pouvons faire est non-déterministe. Nous pouvons en conséquence réutiliser la fonction *oneElement* définie au chapitre précédent (page 119) pour construire le schéma de sélection des activations de filtre :

FetchFilterSelectActivations

$filters? : \mathbb{P}\, FilterFunctionality$

$event? : Event$

$activations! : \mathbb{P}\, \mathrm{seq}\, FilterActions$

$activations! = \{f : filters? \mid event?.type \preceq f.type \bullet oneElement(f.activation)\}$

Le propos de ce schéma est de sélectionner (dans la variable de sortie *activations*) exactement une activation par filtre passé en paramètre dans la variable *filters*?, à partir du moment ou le filtre est valide pour l'élément considéré.

La sélection des filtres va également dépendre de la catégorie de ces derniers : entrant ou sortant. Considérons le premier cas :

FetchFilterSelectInFilters

$DefaultComponent$

$filters! : \mathbb{P}\, FilterFunctionality$

$queue! : \mathrm{seq}\, Event$

$queue! = in\_queue$

$filters! = \{r : roles \bullet r.in\_filters\}$

N'oublions pas que le schéma *DefaultComponent* adapté introduit la variable *roles* dénotant l'ensemble des rôles assigné au protocole considéré. La variable *filters*! contient en sortie de cette opération partielle l'ensemble de tous les filtres entrant définis par l'ensemble des rôles assignés. De la même manière, la variable *queue*! contient la file d'attente adéquate, entrante dans le cas présent.

15. La présence de plus d'une activation possible par filtre résulte de l'emploi d'alternatives dans la description de ces activations.

Le cas des filtres sortant est quasiment similaire :

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{FetchFilterSelectOutFilters}} \\
\hline
\textit{DefaultComponent} \\
\textit{filters}! : \mathbb{P}\, \textit{FilterFunctionality} \\
\textit{queue}! : \operatorname{seq} \textit{Event} \\
\hline
\textit{queue}! = \textit{out_queue} \\
\textit{filters}! = \{ r : \textit{roles} \bullet r.\textit{out_filters} \} \\
\hline
\end{array}$$

Après la sélection des filtres et activations (schémas *FetchFilterSelect*) et la prise en compte du filtrage (schémas *FetchFilterPositive* ou *FetchFilterNegative*), nous pouvons nous intéresser aux effets de ce filtrage sur la fonctionnalité d'extraction. Pour les filtres entrant, ces effets (contrairement à l'ancienne version de *DefaultMetaInFetch*, décrite page 121) se subdivisent en deux sous-cas :

- Cas du filtrage positif où il ne reste plus rien à faire (l'événement est rejeté).
- Cas du filtrage négatif qui correspond à l'ancienne version de *DefaultMetaInFetch*.

Pour ne pas interférer avec la définition de *DefaultMetaExec* (cf. page 123), nous allons «simuler» le premier cas par la génération d'un événement vide :

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{FetchFilterReject}} \\
\hline
\Xi \textit{DefaultBehavior} \\
\textit{event}! : \textit{Event} \\
\hline
\textit{event}! = \emptyset \\
\hline
\end{array}$$

En revanche, pour le filtrage négatif, ce schéma devient :

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{FetchFilterAccept}} \\
\hline
\Delta \textit{DefaultBehavior} \\
\textit{event}! : \textit{Event} \\
\hline
\# \textit{in_queue} > 0 \\
\textit{event}! = \operatorname{last} \textit{in_queue} \\
\textit{in_queue}' = \operatorname{front} \textit{in_queue} \\
\hline
\end{array}$$

Il s'agit d'une copie conforme de l'ancienne version de *DefaultMetaInFetch* qui consistait à prendre le dernier événement disponible dans la file d'attente en entrée.

La nouvelle version de *DefaultMetaInFetch* correspond finalement à l'assemblage des schémas précédents, de la façon suivante :

$$\begin{aligned}
&\textit{DefaultMetaInFetch} \mathrel{\widehat=} \textit{FetchFilterSelectInFilters} \gg \textit{FetchFilterSelectActivations} \gg \\
&\quad (\textit{FetchFilterPositive} \land \textit{FetchFilterReject}) \lor \\
&\quad (\textit{FetchFilterNegative} \land \textit{FetchFilterAccept})
\end{aligned}$$

La séquence d'opérations décrite ci-dessus résume notre algorithme d'extraction des événements potentiellement filtrés en entrée; algorithme décomposé en 3 phases :

1. Sélection pour chaque rôle assigné des filtres entrant et de la file d'attente associée (opération *FetchFilterSelectInFilters*).
2. Restriction des filtres au type de l'événement à extraire et sélection d'une activation pour chacun des filtres retenus (opération *FetchFilterSelect*).
3. Filtrage positif (*FetchFilterPositive*) ou négatif (*FetchFilterNegative*) et effet conséquent : rejet (*FetchFilterReject*) ou acceptation (*FetchFilterAccept*).

Nous pouvons en déduire simplement la modification de la primitive d'extraction pour les filtres sortant :

$$
\begin{array}{l}
\mathit{DefaultMetaOutFetch} \mathrel{\widehat=} \mathit{FetchFilterSelectOutFilters} \gg \mathit{FetchFilterSelectActivations} \gg \\
\quad (\mathit{FetchFilterPositive} \wedge \mathit{FetchFilterReject}) \vee \\
\quad (\mathit{FetchFilterNegatif} \wedge \mathit{FetchFilterAccept})
\end{array}
$$

La modularité de notre modèle formel nous permet de ne modifier qu'un seul schéma entre les cas entrant et sortant. De même, pour ce qui concerne les filtres, cette modification s'intègre parfaitement dans le modèle de comportement par défaut sans gestion des filtres.

Rôles fonctionnels et crochets

En ce qui concerne les autres types de rôle - filtres fonctionnels, crochets entrant ou sortant - les modifications à apporter se concentrent au niveau de l'encadrement du méta-objet *DefaultMetaExec*. Pour un événement généré (et éventuellement filtré négativement) par *DefaultMetaInFetch*, tous les crochets avant compatibles doivent être exécutés avant les fonctionnalités de rôle ou du composant. Finalement, tous les crochets sortant sont à exécuter en dernier. Ces indirections sont présentées en figure 8.12.

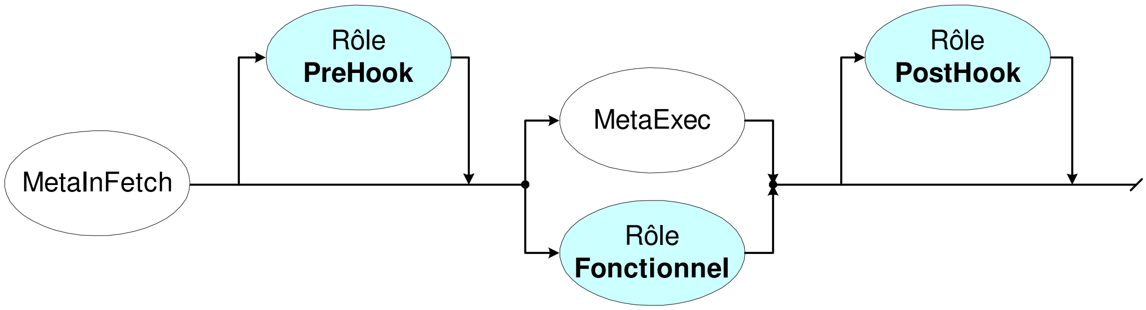


FIG. 8.12 – *Impact comportemental des rôles filtres fonctionnels crochets*

Pour obtenir ce comportement spécialisé, nous pouvons recourir à la modification simple du schéma *SelectActions* (cf. page 122), sans toucher à celle de *DefaultMetaExec* (cf. page 123).

L'ancienne version de ce schéma peut être conservée pour extraire (sous forme d'une variable *fun_actions*) l'ensemble des actions à exécuter au niveau du composant :

SelectComponentActions

$$
\begin{array}{l}
\mathit{DefaultBehavior} \\
\mathit{event?} : \mathit{Event} \\
\mathit{fun\_actions!} : \mathrm{seq}\, \mathit{Action} \\
\hline
\mathbf{let}\ \mathit{fun\_types} == \{\mathit{fs} : \mathit{functs} \bullet \mathit{fs}.\mathit{type}\} \bullet \\
\quad \mathbf{let}\ \mathit{best\_type} == \mu\, t : \mathit{fun\_types} \bullet t = \mathit{tmin}(\{\mathit{ft} : \mathit{fun\_types} \mid \mathit{ft} \succeq \mathit{event?}.\mathit{type}\}) \bullet \\
\quad\quad \mathbf{let}\ \mathit{all\_actions} == (\mu\, \mathit{fs} : \mathit{functs} \bullet \mathit{fs}.\mathit{type} = \mathit{best\_type}).\mathit{activation} \bullet \\
\quad\quad\quad \mathit{fun\_actions!} \in \mathit{all\_actions}
\end{array}
$$

Pour les fonctionnalités de rôle et autres crochets, les définitions sont plus complexes du fait de la multiplicité potentielle des activations associées. Tout d'abord, il est important d'imposer un ordre sur les rôles puisqu'au final, nous aurons besoin de construire une séquence d'activations.

Pour commencer, nous sélectionnons l'intégralité des rôles assignés :

SelectRoles

$$
\begin{array}{l}
\mathit{DefaultComponent} \\
\mathit{roles!} : \mathrm{seq}\, \mathit{Role} \\
\hline
\forall\, r : \mathit{roles} \bullet r \in \mathit{roles!}
\end{array}
$$

Cette définition n'est pas déterministe puisque son unique condition, si elle impose un ordre sur les rôles (puisque la variable de sortie *roles*! est une séquence), ne donne pas pour autant les propriétés de cet ordre. En effet, plusieurs choix «sensés» sont possibles à cet endroit; nous reportons donc cette discussion à la section suivante.

La deuxième étape consiste en la sélection des crochets *avant* qui doivent être exécutés en tout premier lieu :

$$
\begin{array}{|l}
\hline
\textit{SelectionPreHookActions} \\
\textit{DefaultComponent} \\
\textit{event}? : \textit{Event} \\
\textit{roles}? : \operatorname{seq} \textit{Role} \\
\textit{pre\_hook\_actions}! : \operatorname{seq} \textit{Action} \\
\hline
\textbf{let } \textit{valid\_roles} == \textit{roles}? \upharpoonright \{ r : \operatorname{ran} \textit{roles}? \mid \exists\, \textit{ph} : r.\textit{pre\_hooks} \bullet \textit{event}?.\textit{type} \preceq \textit{ph}.\textit{type} \} \bullet \\
\quad \textbf{let } \textit{all\_activations} = \{ n \mapsto r : \textit{valid\_roles} \bullet n \mapsto \textit{oneElement}(r.\textit{activation}) \} \bullet \\
\qquad \textit{pre\_hook\_actions}! = {}^\frown/\ \textit{all\_activations} \\
\hline
\end{array}
$$

La variable *valid_roles* est définie comme une sous-séquence de *roles*? obtenue par *extraction*. L'expression $S_2 = S_1 \upharpoonright E$ définit la séquence S_2 comme une sous-séquence de S_1 dans laquelle ne sont «retenus» que des éléments appartenant à l'ensemble E (qui doit être un sous-ensemble de $\operatorname{ran} S_1$)[16]. Ici, ne sont retenus que les roles (appartenant au co-domaine de *roles*? possèdant un crochet entrant dont le type est compatible (un super-type, donc) avec le type de l'événement à traiter.

Ensuite, une séquence d'activation est composée de sorte que pour chaque élément de *valid_roles* (une relation entre un indice n et un rôle r) y corresponde un choix non-déterministe d'activation (relation entre le même indice n et l'activation $\textit{oneElement}(r.\textit{activation})$). Le résultat, nommé *all_activations* est donc une séquence de séquences d'actions que l'on distribue (avec ${}^\frown/$) dans la séquence *pre_hook_actions*!. Au final, nous obtenons l'activation en séquence des rôles proposant un crochet entrant associé au type de l'événement à traiter. Nous pouvons remarquer que si l'ordonnancement de ces activations n'est pas défini (choix non-déterministe), la séquence obtenue suit le même ordre que la séquence des rôles[17]

Le schéma de sélection des crochets sortant est quasiment identique :

$$
\begin{array}{|l}
\hline
\textit{SelectionPostHookActions} \\
\textit{DefaultComponent} \\
\textit{event}? : \textit{Event} \\
\textit{roles}? : \operatorname{seq} \textit{Role} \\
\textit{post\_hook\_actions}! : \operatorname{seq} \textit{Action} \\
\hline
\textbf{let } \textit{valid\_roles} == \textit{roles}? \upharpoonright \{ r : \operatorname{ran} \textit{roles}? \mid \exists\, \textit{ph} : r.\textit{post\_hooks} \bullet \textit{event}?.\textit{type} \preceq \textit{ph}.\textit{type} \} \bullet \\
\quad \textbf{let } \textit{all\_activations} = \{ n \mapsto r : \textit{valid\_roles} \bullet n \mapsto \textit{oneElement}(r.\textit{activation}) \} \bullet \\
\qquad \textit{post\_hook\_actions}! = {}^\frown/\ \textit{all\_activations} \\
\hline
\end{array}
$$

En ce qui concerne la sélection des fonctionnalités, nous nous retrouvons dans les mêmes conditions que pour les fonctionnalités de composant, c'est-à-dire qu'il ne peut exister qu'une unique fonctionnalité activable pour un événement donné. Le schéma de sélection résultant est donc une sorte de mélange entre les trois schémas précédents :

16. Le type de l'opérateur $\upharpoonright$ est donc : $\operatorname{seq} X \times \mathbb{P}\, X \to \operatorname{seq} X$.
17. Nous ne rajoutons donc pas un problème d'ordre ici; nous ne faisons que propager notre problème général de l'ordre relatif des rôles potentiellement conflictuels.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{SelectionRolesActions}\ }}\\
\mathit{DefaultComponent}\\[4pt]
\mathit{event?} : \mathit{Event}\\
\mathit{roles?} : \mathrm{seq}\,\mathit{Role}\\[4pt]
\mathit{role\_actions} : \mathrm{seq}\,\mathit{Action}\\
\hline
\textbf{let } \mathit{valid\_roles} == \mathit{roles?} \upharpoonright \{r : \mathrm{ran}\,\mathit{roles?} \mid \exists \mathit{fun} : r.\mathit{functs} \bullet \mathit{event?}.\mathit{type} \preceq \mathit{fun}.\mathit{type}\} \bullet\\
\quad \textbf{let } \mathit{valid\_functs} == (\{n \mapsto r : \mathit{valid\_roles} \bullet\\
\quad\quad \textbf{let } \mathit{fun\_types} == \{f : r.\mathit{functs} \bullet f.\mathit{type}\} \bullet\\
\quad\quad\quad \textbf{let } \mathit{best\_type} == (\mu\, t : \mathit{fun\_types} \bullet t = \mathit{tmin}(\{\mathit{ft} : \mathit{fun\_types} \mid \mathit{ft} \succeq \mathit{event?}.\mathit{type}\})) \bullet\\
\quad\quad\quad\quad \textbf{let } \mathit{best\_fun} == (\mu\, f : r.\mathit{functs} \bullet f.\mathit{type} = \mathit{best\_type}) \bullet\\
\quad\quad\quad\quad\quad n \mapsto \mathit{best\_fun}\})\\
\quad\quad \textbf{let } \mathit{all\_activations} = \{n \mapsto f : \mathit{valid\_functs} \bullet n \mapsto \mathit{oneElement}(f.\mathit{activation})\} \bullet\\
\quad\quad\quad \mathit{role\_actions!} = {}^\frown/\ \mathit{all\_activations}\\
\hline
\end{array}
$$

Tout d'abord, la phase d'extraction des rôles valides est réalisée puis, selon les mêmes principes, nous construisons une séquence associant pour chaque indice de rôle, la «meilleure» fonctionnalité pour l'événement considéré. Ceci passe par la sélection du meilleur type possible et nommé *best_type* (cf. schéma *SelectComponentActions* ci-dessus). Finalement, les activations sont prises dans cette séquence de fonctionnalités, toujours de façon non-déterministe.

Nous pouvons, dans une dernière étape de sélection, réunir les précédents schémas pour former une unique séquence d'activation :

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{SelectAllActions}\ }}\\
\mathit{pre\_hook\_actions?} : \mathrm{seq}\,\mathit{Action}\\
\mathit{role\_actions?} : \mathrm{seq}\,\mathit{Action}\\
\mathit{fun\_actions?} : \mathrm{seq}\,\mathit{Action}\\
\mathit{post\_hook\_actions?} : \mathrm{seq}\,\mathit{Action}\\[4pt]
\mathit{actions!} : \mathrm{seq}\,\mathit{Action}\\
\hline
\mathit{actions!} = \mathit{pre\_hook\_actions?} \frown \mathit{role\_actions?} \frown \mathit{fun\_actions?} \frown \mathit{post\_hook\_actions?}\\
\hline
\end{array}
$$

Il est important de noter que ce schéma effectue un choix très important : privilégier les fonctionnalités des rôles sur celles des composants. En effet, si une activation est disponible pour les rôles, alors les actions correspondantes seront exécutées avant les actions potentielles au niveau du composant.

Une contrainte plus forte peut être posée : interdire l'exécution mixte de fonctionnalité de rôle et de protocole. Pour se faire, nous pouvons remplacer la condition d'invariant de *SelectAllActions* par :

$$
\begin{array}{l}
\textbf{if } \mathit{role\_actions} \neq \langle\rangle \textbf{ then}\\
\quad \mathit{actions!} = \mathit{pre\_hook\_actions?} \frown \mathit{role\_actions?} \frown \mathit{post\_hook\_actions?} \textbf{ else}\\
\quad \mathit{actions!} = \mathit{pre\_hook\_actions?} \frown \mathit{fun\_actions?} \frown \mathit{post\_hook\_actions?}
\end{array}
$$

En maintenant notre priorité pour les rôles, cette modification a pour principale conséquence qu'une assignation de rôle peut désormais inhiber une fonctionnalité de composants et ainsi simuler un remplacement de fonctionnalité. Si la priorité était donnée au composant, ceci ne serait plus possible.

Nous pouvons également voir que nos schémas autorisent l'exécution en séquence de plusieurs fonctionnalités pour le même événement. Toutes les autres variantes, qui ne demandent que de «légères» modification dans ces schémas, sont également à considérer :

– Autoriser les activations multiples de fonctionnalités, mais en donnant la priorité aux composant.
– Pour un événement donné, ne permettre qu'une seule activation, en privilégiant les fonctionnalités de rôles (il faudra donc choisir un unique rôle).
– Activation unique mais en privilégiant le composant.

Nous pouvons en conclusion construire précisément l'ensemble des actions à éxécuter en réaction de la réception d'un événement. Cette construction consiste en la réécriture du schéma *SelectActions*, désormais adapté (et donc largement complexifié pour le support des rôles) :

$$\begin{aligned}&SelectActions \mathrel{\widehat{=}} (SelectComponentActions \land \\ &\qquad SelectPreHookActions \land SelectPostHookActions \land \\ &\qquad SelectRoleActions) \gg SelectAllActions\end{aligned}$$

Ce schéma composé doit être comparé à l'ancienne version de *SelectActions*, définie page 122.

8.2.4 Conflits et résolution

Le modèle formel décrit ci-dessus expose, comme nous l'avons signalé à plusieurs reprises, un certain nombre de points de choix. Ces choix dont la plupart n'ont pas été faits (non-déterminisme) sont autant de sources de réflexion concernant l'opérationnalisation du modèle[18].

Cependant, si ces choix doivent d'une façon ou d'une autre être effectués, il n'existe pas à notre connaissance de solution générale et parfaite; de sorte que notre approche consiste à se reposer sur notre modèle très ouvert en n'imposant aucun choix «transcendant» mais plutôt d'autoriser le raffinement, au niveau de granularité du composant, de ces critères de choix. Bien sûr, dans les modèles de comportement par défaut, ces choix sont faits mais il est possible, parfois souhaitable et relativement aisé, de les contredire.

Conflits structurels

Les ambiguïtés afférentes à nos critères de choix sont pour la plupart en corrélation avec un phénomène de *conflit* se produisant lorsque plus d'un rôle peut être activé pour une réception d'événement donnée.

Considérons en guise d'exemple la figure 8.13 dans laquelle deux rôles de filtrage de contenu ont été assignés pour les types CPUInfoEvent et DiskInfoEvent.

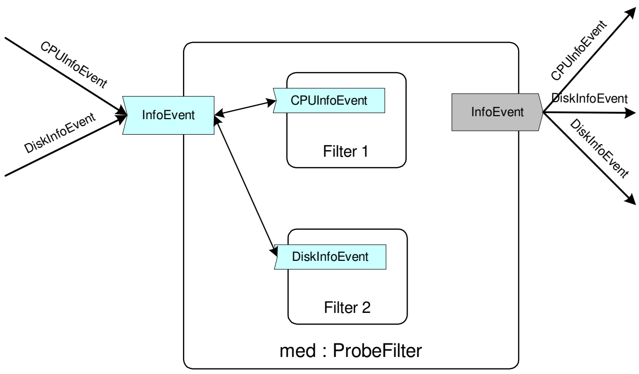


FIG. 8.13 – *Rôles conflictuels pour le type InfoEvent*

Dans cet exemple, si le composant med reçoit des événements de types CPUInfoEvent ou DiskInfoEvent, alors aucun problème ne se pose puisqu'un seul des deux filtres sera activable dans chaque cas. Si en revanche, un événement d'un sous-type commun de ces deux types est reçu (en supposant qu'un tel type existe), alors les rôles filtres deviennent activables puisqu'ils peuvent tous deux interpréter cet événement.

Dans le modèle formel établi dans la section précédente, ceci résulte en la sélection de plus d'un rôle dans la séquence des rôles activables. Pour l'instant, nous avons contourné le problème en ordonnant les rôles ainsi sélectionnés de façon arbitraire.

Une première heuristique de choix possible concerne l'ordonnancement des rôles selon leur spécificité. On privilégie dans ce cas les rôles activés pour les types minimaux au sens du sous-typage. Les rôles activés pour le même type étant ensuite placé de façon arbitraire.

Une autre heuristique intéressante (celle utilisée en pratique dans notre implantation par défaut) concerne l'ordonnancement des activations de rôle selon les moments d'assignation des rôles. L'assignation d'un rôle est

18. L'avantage de disposer d'un modèle formel est évident ici : nous connaissons nos points de choix.

en effet une opération localement atomique, c'est-à-dire qu'il est possible de discrétiser les moments de ces assignations. Le rôle le plus ancien pourra ainsi être considéré comme ayant une priorité d'activation maximale (notre choix) ou au contraire minimale.

L'idée de favoriser les plus anciennes assignations est de partir du principe que les nouvelles adaptations doivent se placer «au dessus» des anciennes et non l'inverse. Mais ce choix est parfois contestable, par exemple si l'on considère un filtrage positif (événement rejeté) et que l'on voudrait rendre potentiellement négatif (événement accepté). Pour ce faire, une solution consisterait à placer un nouveau rôle filtre avant l'ancien, ce qui n'est pas possible avec notre heuristique par défaut [19]. Il est donc important de pouvoir paramétrer finement ces critères de choix, et ceci par instance de composant déployé.

Conflits comportementaux

Les conflits les plus difficiles à résoudre ne sont de fait pas capturés par notre modèle opérationnel. En pratique, les rôles peuvent être amenés à interférer à un niveau de granularité plus fin que le «détournement» ou le «crochetage» d'événements normalement traités par les fonctionnalités de base des composants.

Dans notre modèle opérationnel (non formel), nous autorisons ainsi les rôles assignés à interagir finement avec les propriétés de base du composant, à savoir son *état* et ses *fonctionnalités*. Ces possibilités d'interactions - toujours contrôlée au niveau méta-comportement implanté - sont de deux ordres :

- *Introspection* : les actions des rôles peuvent, si le niveau méta-comportement en fournit le moyen, récupérer certaines informations opérationnelles sur le composant auquel ils sont assignés. Ces informations peuvent concerner l'état interne du composant (nom, type et valeur d'un attribut), l'isolation du rôle en termes d'effet de bord (*sandboxing*) ou encore ses fonctionnalités (blocs d'activation `when` ou méthodes internes).
- *Intercession* : certains modèles de comportement opérationnels permettent aux rôles - toujours de façon strictement contrôlée - de lire ou d'écrire des attributs de l'état interne ou encore de composer des invocations dynamiques de fonctionnalités proposées au niveau de base du composant.

D'un point de vue formel, nous ne possédons pas les «armes» (diplomatiques bien sûr) nécessaires pour résoudre ces conflits de grain très fin. Une première approche drastique consisterait à interdire ces interactions composant/rôle. S'il s'agit d'une solution parfois intéressante car sûre, il existe de nombreux cas où l'on se coupe alors d'un vaste domaine d'adaptations utiles; de sorte que cette restriction n'est pas imposée dans l'absolu.

Dans le prototype opérationnel que nous proposons et décrivons au chapitre 10, deux mécanismes sont mis en œuvre pour servir de fondations aux vérifications de granularité fine :

- Un mécanisme de vérification du rôle par le composant, ce qui permet à ce dernier de refuser - pour des critères indépendant des problèmes structurels - de refuser une assignation.
- Un mécanisme complémentaire (mais moins prioritaire) de vérification du composant par le rôle.

A partir de ces deux mécanismes, de nombreuses pistes peuvent être explorées comme la mise en place de protocoles d'authentification (pour s'assurer qu'un rôle provient d'une source «fiable» par exemple) ou encore le support de preuves embarquées dans les deux parties concernées par l'assignation.

19. Dans ce cas précis, une solution consisterait à retirer le rôle filtre problématique et en assigner un autre - plus adéquat - à la place.

« N'ayez pas de bonnes idées si vous ne voulez pas en être responsable. »

Alan Perlis

9
Modèles client/serveur

De notre discussion précédente, nous pourrions être tenté de penser que l'obtention d'un modèle *naturellement* dynamique et adaptable n'était possible que via d'importantes concessions en comparaison de modèles plus courants (et donc essentiellement à objets). En d'autres termes, la plateforme Comet et son infrastructure linguistique apparaît quelque peu «exotique»; ce qui représente la critique la plus fréquente à l'encontre des approches asynchrones.

Pourtant, des travaux fort influents ont montré qu'il était tout à fait sensé - et même profitable - de considérer les interactions synchrones comme des compositions d'interactions plus primitives et de nature asynchrone. Les interstitiels courants (RMI/CORBA) prennent plutôt le point de vue inverse : une invocation asynchrone de méthode y est considérée comme un cas particulier d'invocation synchrone. L'idée est que les interactions synchrones supportent *naturellement* le modèle client/serveur qui domine le monde des applications distribuées. Cette dichotomie client/serveur - s'il faut parfois s'en abstraire [1] - se trouve pleinement justifiée dans de nombreuses situations.

Ce chapitre est donc très important en ceci qu'il réconcilie nos prises de position divergentes avec les modes d'interactions standard dans l'informatique distribuée d'aujourd'hui. L'objectif ici n'est cependant pas de «retrouver» une copie exacte de l'application des mécanismes objet au monde du client/serveur mais plutôt de construire des mécanismes proches quoique disposant de propriétés qui leur sont propres.

9.1 Approche objet

Pour aborder notre problématique dans un esprit le plus ouvert possible, nous allons montrer dans cette section qu'il n'existe pas un unique modèle pour les interactions de type client/serveur mais bien un spectre assez large que l'on peut faire varier selon divers critères.

Les communications client/serveur ont de nombreuses propriétés communes avec le mécanisme d'invocation de méthodes sous-jacent des modèles objets. Tout d'abord, observons ces rapports d'un point de vue structurel :

- Distinction entre une *entité invoquante*, le *client*, où est posé l'ordre d'invocation et une *entité invoquée*, le *serveur*, qui supporte le traitement associé,
- envoi d'informations du client vers le serveur permettant de caractériser une requête, et
- renvoi d'une réponse par le chemin inverse.

Si l'on se place dans le cadre d'une invocation de méthode, nous pouvons identifier :

- l'objet source de l'invocation est l'*objet client*,

1. Les applications de type *peer to peer* ou *publish/subscribe* sortent par exemple du cadre client/serveur strict.

- l'objet siège de cette même invocation est l'*objet serveur*,
- les informations de requêtes correspondent aux arguments de l'invocation, et
- la valeur de retour dénote la réponse du serveur.

D'un point de vue opérationnel, les relations sont également étroites :

- Blocage implicite du client en attente de la réponse depuis le serveur.
- Prise de contrôle exclusive du serveur pour réaliser le traitement.

Pour illustrer ces divers critères, prenons l'exemple d'une application client/serveur primordiale : un service de nommage[2].

Pour ne pas se perdre dans les détails d'implantation, nous allons supposer que notre serveur de nom fera le lien entre des références d'objets serveurs sous forme de chaînes de caractères et des identifications associées de localisation, également sous forme de chaînes de caractère de format : "machine:port".

Donnons une implantation possible de cet objet distribué serveur de nom dans le langage Java :

```
class NameServer {

  // Conteneur des associations nom=>localisation
  protected HashMap _names = new HashMap();

  // Enregistrement d'un nom
  void bind(String nom, String machine, int port) {
    _names.put(nom, machine+":"+Integer.toString(port));
  }

  // Requete de nommage
  String lookup(String nom) {
    return _names.get(nom);
  }
}
```

Bien que simplifiée, ce serveur de nom est proche de ce qu'il faudrait écrire en pratique[3]. Du côté du client, une requête de nommage pourra être effectuée de la façon suivante :

```
class Client {

  // Une méthode quelconque
  void exemple(NameServer myNameServer, String which) {
    String localisation = myNameServer.lookup(which);
    // instructions suivantes
  }
}
```

La figure 9.1 page suivante donne une interprétation graphique d'une telle invocation.

Pour catégoriser ce modèle d'interaction, nous dirons qu'il est :

- *bidirectionnel* : en terme structurel, il existe un lien allant du client vers le serveur (passage des paramètres, ici nom) et un autre lien allant du serveur vers le client (valeur de retour localisation).
- *synchrone côté client* : le client voit son exécution bloquée en attente de la réponse du serveur.
- *synchrone côté serveur* : le serveur «capture» le fil d'exécution associé au serveur et ne le rend que lorsqu'il est prêt à retourner la réponse.

2. De notre point de vue, il s'agit du programme «bonjour, monde» le plus intéressant dans le cadre des systèmes client/serveur. En effet, le service de nommage a pour composant fondamental une entité qui est elle-même un objet serveur. Un problème non-trivial réside d'ailleurs dans le nommage de cet objet.

3. Pour obtenir un serveur complet, il faudrait ajouter les constructions spécifiques à Java/RMI, traiter le cas de renommage, etc.

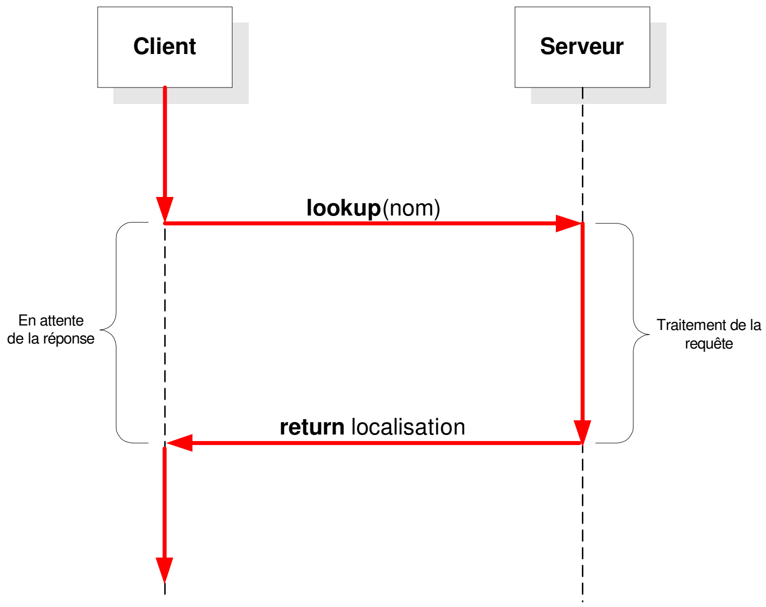


Fig. 9.1 – *Client/Serveur par invocation de méthode*

Ces critères reposent en général sur des mécanismes implantés dans les couches les plus basses des interstitiels objet. De sorte qu'il est en général très difficile de pouvoir «jouer» avec ces critères et qu'il s'agit quasi-systématiquement d'interventions de bas niveau.

Considérons par exemple le critère de synchronisation côté serveur, qui représente un élément important du mode d'interaction décrit par la figure 9.1. Dans l'hypothèse d'un support du mode client/serveur via le mécanisme d'invocation de méthode dans une version stricte, alors il n'est pas possible, pour plusieurs clients, d'interagir *en même temps* avec le serveur. Bien sûr, ce mode d'interaction n'est pas viable en pratique. Ainsi, les interstitiels objets du marché (qu'ils soient basés sur CORBA, Java/RMI ou COM+) autorisent les clients multiples selon deux heuristiques principales :

- une seule session serveur active, les autres invocations sont mises en file d'attente, ou
- plusieurs sessions activables en parallèle.

Si le premier mode d'activation est le plus simple à mettre en œuvre, l'activation multiple offre potentiellement de meilleures performances. Mais ce gain se fait au prix de l'intrusion au niveau du code du serveur de contraintes liées au contrôle. Il faudra synchroniser finement les accès potentiellement concurrents aux données gérées par l'objet serveur. Retenons que dans les deux cas, nous sortons quoiqu'il arrive du cadre strict des interactions du paradigme objet; de nouveaux mécanismes de support sont nécessaires : adaptateurs d'objets, primitives de synchronisation, etc.

Il n'est donc pas «évident» de manipuler ce genre de propriété pourtant fondamentale des modes d'interactions client/serveur. Notre parti-pris, dans la plateforme Comet, est de rendre tangible ces propriétés au niveau du modèle formel qui la sous-tend. Nous proposons donc dans ce qui suit les interventions - de nature formelle - qui sont nécessaires pour couvrir la quasi-totalité du spectre des modes d'interaction client/serveur.

9.2 Interactions bidirectionnelles

Si les invocations de méthode représentent un modèle plutôt adéquat (quoique non suffisant) pour le support d'interactions client/serveur, le résultat est généralement peu flexible. Ainsi, effectuer des échanges asynchrones est par exemple loin d'être trivial et demande des modifications au niveau des mécanismes de bas niveau ou même, dans certains cas, une refonte du système (exemple : CORBA AMI). C'est d'autant plus étonnant que les mécanismes de plus bas niveau sont, eux, asynchrones (une histoire de «chaussettes» BSD ...) !

Notre argument, rappelons-le, est que l'approche proposée dans le cadre de Comet dispose d'une plus grande flexibilité et est surtout plus «propre» car formalisée. Illustrons ce propos sur une version adaptée du service de nommage. Dans un premier temps, il nous faut introduire des événements de requête et de réponse :

```
// Evénement de requête de nommage
event NameRequest {

  slot _name type String;

  NameRequest(String name) { _name = name; }

  String getName() { return _name; }
}

// Evénement de réponse de nommage
event NameReply {

  slot _location type String;

  NameReply(String location) { _location = location; }

  String getLocation() { return _location; }
}
```

Si les types d'événements définis ci-dessus gèrent tous deux un contenu sous forme de chaîne de caractère, nous préférons utiliser deux types différents pour bien marquer leur différence fondamentale :

- NameRequest décrit les informations circulant du client vers le serveur.
- NameReply concerne la seule réponse du serveur.

Nous pouvons de même introduire un type d'événement pour l'enregistrement de nom :

```
event NameRegister {
  slot _name type String;
  slot _location type String;

  NameRegister(String name, String location) { _name = name; _location = location; }

  String getName() { return _name; }
  String getLocation() { return _location; }

}
```

Il nous faut ensuite transformer nos objets client/serveur en composants Comet. Commençons par le serveur :

```
component NameServer {

  // Conteneur des associations nom=>localisation
  protected HashMap _names = new HashMap();

  // Enregistrement d'un nom
  when (NameRegister register) {
    _names.put(register.getName(), register.getLocation());
  }

  // Récupération d'un nom
  when (NameRequest request) {
    String location = _names.get(request.getName());
    NameReply reply = new NameReply(location);
    send(reply);
  }
```

Ce qui était auparavant géré par des méthodes est ici pris en charge par des blocs when pour les différents types de requêtes : enregistrement ou récupération d'une localisation.

Pour le client, nous pouvons écrire :

```
component Client {

  // une méthode quelconque
  void exemple(String which) {
    send( new NameRequest(which))
    // instructions suivantes
  }
  // Réception de la requête
  when (NameReply reply) {
    String location = reply.getLocation();
    // etc...
  }
}
```

L'application obtenue est, nous en convenons, légèrement plus complexe que la version objet. Par exemple, la distinction entre réception et émission des événements oblige à s'orienter vers une architecture réactive. Mais contrairement à la version objet, la plateforme Comet peut directement interpréter les constructions ci-dessus; il ne s'agit pas, comme précédemment, d'une version simplifiée.

Pour comparer les deux versions de façon plus approfondie, considérons la figure 9.2.

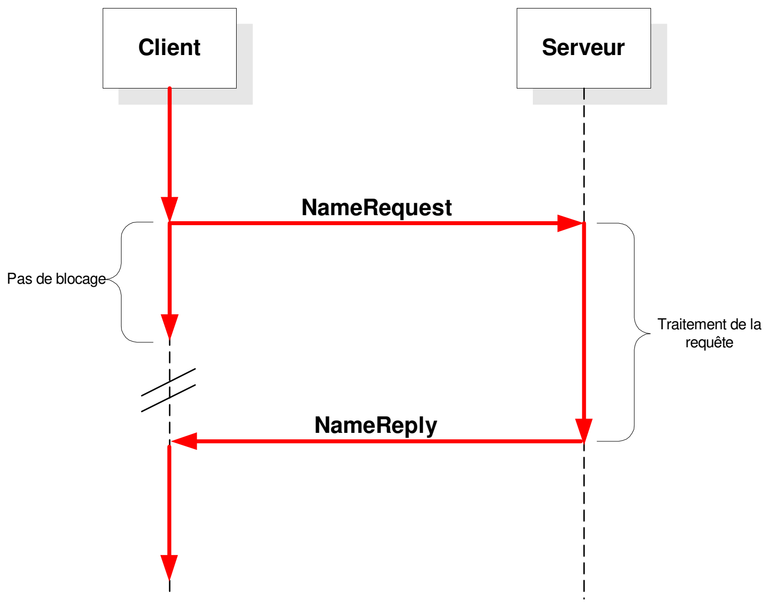


FIG. 9.2 – *Client/Serveur en Comet : modèle de base*

La différence la plus « visible » ici résulte directement des propriétés particulières de notre modèle : le client n'est pas bloqué par l'émission de la requête. Une autre différence importante est que sans ajout de mécanisme supplémentaire, le serveur peut recevoir de nouvelles requêtes d'autres clients qui seront mises en file d'attente avant traitement. Selon nos critères, nous obtenons donc un modèle :

- *bidirectionnel* : nous sommes toujours en présence d'un aller/retour d'information,
- *asynchrone côté client* : l'émission de la requête est non-bloquante,
- *asynchrone côté serveur* : d'autres requêtes peuvent être reçues,
- *mono-session* : une seule de ces requêtes est traitée à un moment donné.

La limitation à une seule session peut-être levée par l'emploi d'un modèle de comportement adapté (comme celui décrit au chapitre 7). Mais contrairement au cas par défaut et de façon similaire aux approches objet, il faudra ajouter des contraintes de synchronisation pour assurer la cohérence du serveur en cas d'activations concurrentes.

Cependant, un autre problème plus subtil est posé par l'approche client/serveur proposée ci-dessus. Avant de pouvoir communiquer, le client et le serveur concernés doivent être connectés (par exemple pour les types

NameRequest à l'aller et NameReply au retour) avant de pouvoir communiquer. Que se passe-t-il alors si plusieurs clients se connectent à plusieurs serveurs ? Le cas 1 client N serveurs ne soulève aucune question et sera même à la base d'interactions très intéressantes (cf. section 9.3 page 181. En revanche le cas N clients 1 serveur (dont une instance est donnée en figure 9.3) est plus complexe, tout comme dans le monde objet.

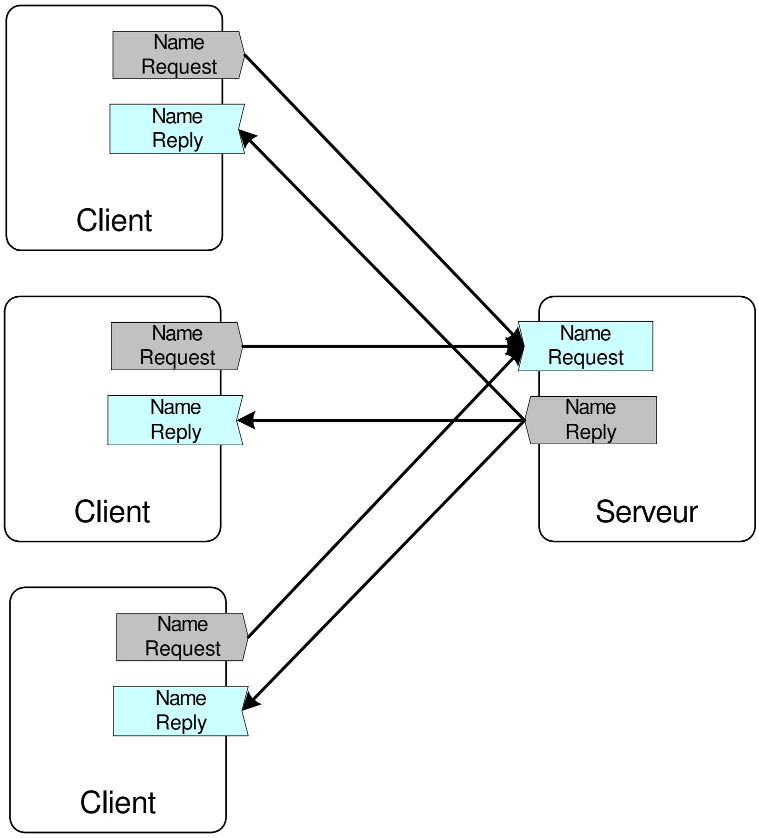


FIG. 9.3 – *Clients multiples*

Nous avons vu que même dans le cadre d'un serveur à session unique, les requêtes potentiellement concurrentes étaient automatiquement sérialisées (au sens transactionnel du terme) grâce au système de file d'attente en entrée. Le problème ne se situe donc pas à la réception des requêtes mais plutôt au niveau de l'émission des réponses. Dans notre exemple, le serveur est connecté à trois clients pour le type NameReply. L'émission d'un événement de ce type sera - d'après notre modèle de communication - multi-diffusé sur toutes les connexions compatibles avec son type. Ici, les trois clients recevront donc implicitement toutes les réponses émises par chacun d'eux; ce qui ne représente bien évidemment pas un fonctionnement correct dans le cadre du client/serveur.

Le problème est que la contrainte de *bidirectionnalité* n'est en fait pas respectée dans le mode de communication par défaut. Pour obtenir une véritable communication restreinte à deux entités (le client concerné et «son» serveur), il faut conserver quelque part l'identité du client ayant émis la requête pour que le serveur puisse lui répondre *explicitement*.

Le champ *sender_id* des événements (schéma *Event* décrit page 116) pourrait jouer ce rôle mais l'utilisation d'un champ atomique empêcherait la délégation des requêtes. Pour prendre en compte la possibilité pour un serveur de réémettre une requête (et donc de devenir client pour un autre serveur), l'idée est de mettre en œuvre une pile de clients.

La structure d'événement associée est donnée ci-dessous :

ClientServerEvent
Event

call_chain : seq *IDENTIFIER*

La séquence *call_chain* contient la pile des émissions de requêtes, représentée par la séquence des émetteurs successifs.

Les opérations d'émission de requête et de réponse sont représentées formellement par des variantes de *SendAction* et *SendExplicitAction* définies par le modèle de comportement par défaut (cf. chapitre 7, page 122). Pour l'émission d'une requête, le schéma *SendAction* sera complété de la façon suivante :

SendRequestAction

$$
\begin{array}{l}
\mathit{DefaultBehavior} \\[4pt]
\mathit{event?} : \mathit{ClientServerEvent} \\
\mathit{event!} : \mathit{ClientServerEvent} \\
\hline
\mathit{event!.type} = \mathit{event?.type} \\
\mathit{event!.sender\_id} = \mathit{id} \\
\mathit{event!.receiver\_id} = \emptyset \\[4pt]
\mathit{event!.call\_chain} = \langle \mathit{id} \rangle \frown \mathit{event?.call\_chain}
\end{array}
$$

L'identificateur du composant (variable *id*) est placé en sommet de pile (en fait en début de séquence) avant envoi de l'événement *event*!.

Il est bien sûr possible d'émettre une requête en référençant explicitement le(s) destinataire(s) :

SendRequestExplicitAction

$$
\begin{array}{l}
\mathit{DefaultBehavior} \\[4pt]
\mathit{event?} : \mathit{ClientServerEvent} \\
\mathit{rids?} : \mathbb{P}\,\mathit{IDENTIFIER} \\
\mathit{event!} : \mathit{ClientServerEvent} \\
\hline
\mathit{event!.type} = \mathit{event?.type} \\
\mathit{event!.sender\_id} = \mathit{id} \\
\mathit{event!.receiver\_id} = \mathit{rids?} \\[4pt]
\mathit{event!.call\_chain} = \langle \mathit{id} \rangle \frown \mathit{event?.call\_chain}
\end{array}
$$

Du côté du serveur, l'opération d'émission de réponse doit être modifiée en sélectionnant explicitement le receveur comme premier élément de la pile :

SendReplyAction

$$
\begin{array}{l}
\mathit{DefaultBehavior} \\[4pt]
\mathit{event?} : \mathit{ClientServerEvent} \\
\mathit{event!} : \mathit{ClientServerEvent} \\
\hline
\mathit{event!.type} = \mathit{event?.type} \\
\mathit{event!.sender\_id} = \mathit{id} \\[4pt]
\mathit{event!.receiver\_id} = \mathrm{head}\ \mathit{event?.call\_chain} \\[4pt]
\mathit{event!.call\_chain} = \mathrm{tail}\ \mathit{event?.call\_chain}
\end{array}
$$

Le récepteur explicite choisi est donc ici le dernier émetteur de requête placé en sommet de pile. Cet élément est finalement retiré de la pile dans l'événement de réponse *event*!.

A un niveau plus «terre à terre», ces opérations correspondent à de nouvelles primitives linguistiques pour supporter ces échanges client/serveur. Ces primitives sont listées en table 9.1 page suivante.

Pour illustrer ces primitives, prenons l'exemple d'une chaîne de deux échanges bidirectionnels comme indiqués sur la figure 9.4 page suivante.

En premier lieu, le client émet une requête au serveur et place en conséquence son identificateur sur la pile d'appel. Au moment de la réception, le serveur délègue le traitement de cette requête au composant 3tier; il s'identifie donc également dans la chaîne d'appel. Ce composant «troisième tiers» renvoie alors une réponse au composant situé en sommet de la pile, ici le composant Server. La réponse sera finalement reçue par le client puisque ce composant se retrouve naturellement en sommet de pile.

Emission de requête
sendRequest(ClientServerEvent request) **Définition :** Emission d'une requête pour pour serveur(s) en mode multi-point implicite
sendRequest(ClientServerEvent request, String id) **Définition :** Emission de requête en point-à-point
sendRequest(ClientServerEvent request, String[] id) **Définition :** Emission de requête en mode multi-point explicite

Emission de réponse
sendReply(ClientServerEvent reply) **Définition :** Emission de réponse par un composant serveur en mode point-à-point implicite.

TAB. 9.1 – *Plateforme Comet : primitives de communications client/serveur*

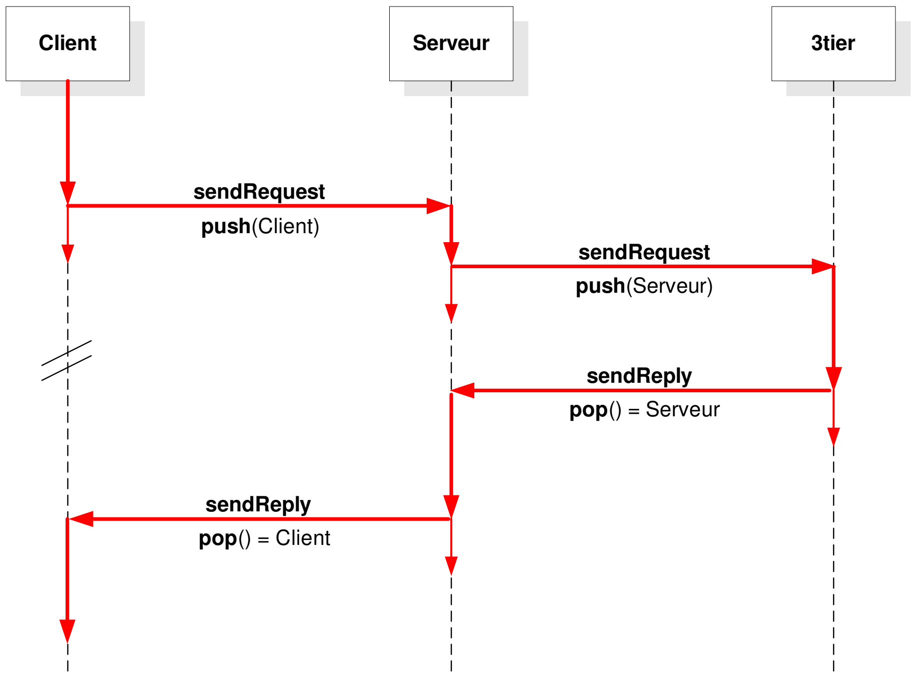


FIG. 9.4 – *Client/Serveur asynchrone : gestion de la pile d'appel*

Il est intéressant de noter qu'à tout moment, un composant peut analyser la pile d'appel et décider d'y effectuer des «sauts». Ceci est un moyen très peu couteux de mettre un œuvre un *mécanisme de rappel* (cf. [Van01] pour l'équivalent en Java/RMI). Il s'agit également d'un outil très intéressant pour mettre en œuvre des *continuations orientées jeton* [VA01] (où les jetons seront représentés bien évidemment par des événements). Ces critères, en plus du caractère très peu intrusif des modifications nécessaires au niveau du modèle formel, ont conduit au choix de placer la chaîne d'appel dans l'événement et non sur les différents composants «traversés» par les requêtes/réponses.

9.3 Synchronisation : point de vue client

Dans la section précédente, nous avons montré le modèle minimal pour supporter des interactions client/serveur en environnement Comet. Ce modèle offre un certain nombre d'avantages si nous le comparons aux approches «traditionnelles» :

- Client asynchrone : les émissions de requêtes sont non-bloquantes.
- Diffusion implicite des requêtes en fonction de leur type.
- Serveur asynchrone : le modèle de comportement par défaut gère les requêtes concurrentes (en les sérialisant).
- Peu intrusif au niveau du modèle formel : la seule modification nécessaire concerne la gestion de la pile d'appel.
- Approche flexible : support aisé de mécanismes de rappel complexes ou de continuation à jeton, etc.

La limite la plus tangible de l'approche concerne le goulet d'étranglement représenté par la sérialisation des événements[4] au niveau du serveur. Les solutions passent invariablement par l'emploi d'un modèle de comportement multi-session, comme celui présenté au chapitre 7. Il faudra lui adjoindre des propriétés de synchronisation interne comme ceux proposés par Jeff Mc Affer dans sa thèse sur CodA [McA95b]. Ces modèles de synchronisation que nous avons pratiquement tous portés dans l'environnement Comet y sont décrits avec moult détails; nous référons donc le lecteur vers ce document de grand intérêt.

L'autre inconvénient majeur, outre cet aspect performance, est plutôt d'ordre qualitatif et concerne le modèle de développement - côté client - sous-jacent de notre approche. L'approche asynchrone entraîne en effet la séparation entre les phases d'émission de requête et de réception des réponses correspondantes. Supposons que dans une des fonctionnalités du client, la séquence de pseudo-code suivant soit présente :

```
// Dans une fonctionnalité du client
RequestType request = new RequestType(Args...);
// Emission de requête
sendRequest(request);
// Instruction suivante
next();
// etc ...
```

Lors de l'interprétation de cette séquence, l'instruction qui suit l'émission de requête (ici représentée par un appel à un fonction next()) est exécutée «immédiatement». De plus, il faudra indiquer à un autre endroit du client ce qu'il faut faire au moment du retour de la réponse :

```
// Dans le code du client
when(ReplyType reply) {
  // Traiter la réponse
}
```

Pour les développeurs habitués aux formalismes réactifs, ce découplage émission/réception est somme toute plutôt naturel (et très intéressant pour optimiser le degré potentiel de parallélisme dans les échanges). En revanche, du côté des usagers - plus nombreux - de la programmation orientée-objet, on s'attendrait plutôt à une fusion des deux phases :

4. Nous parlons de sérialisation temporelle, pas de sérialisation au sens de Java dont les concepteurs ont selon nous opérés quelques choix catastrophiques quant à la terminologie employée !

```
// Dans une fonctionnalité du client
RequestType request = new RequestType(Args...);
// Communication avec le serveur
ReplyType reply[] = sendRequest+receiveReply(request, ReplyType.class);
// Instruction suivante
next();
// etc ...
```

Ce pseudo-code conserve les propriétés d'indépendance structurelle (aucune référence extérieure explicite) mais ressemble plus fortement que la version «réactive» à une invocation de méthode standard. Cependant, il faut toujours se rappeler qu'il s'agit d'une interaction Comet et donc basée uniquement sur des émissions/réceptions d'événements d'où l'utilisation du nom sendRequest+receiveReply pour la caractériser. En pratique, le nom exact de la primitive est sendreceive (cf. table 9.2).

Requêtes synchrones
ReplyType[] = **sendreceive**(ClientServerEvent request, Class ReplyType) [, **long** timeout])   **Définition :** Emission d'une requête synchrone diffusée implicitement. ReplyType donne le type des réponses.
ReplyType = **sendreceive**(ClientServerEvent request, Class ReplyType, String id [, **long** timeout])   **Définition :** Emission synchrone en mode point-à-point.
ReplyType[] = **sendreceive**(ClientServerEvent request, Class ReplyType, String[] id [, **long** timeout])   **Définition :** Emission synchrone en mode multi-point explicite.

Réception bloquante
ReceiveType = **receive**(Class ReceiveType [, **long** timeout])   **Définition :** Mise en attente du composant jusqu'à réception d'un événement de type adéquat.

Tab. 9.2 – *Plateforme Comet : primitives client/serveur synchrones*

Dans ce contexte, il est évident que l'on s'attend à ce que l'instruction next() ne soit exécutée qu'à la fin de l'échange client/serveur; c'est-à-dire après l'émission de la requête *puis* la réception des réponses. Nous employons le pluriel car la diffusion implicite en fonction du type fait que le nombre de réponses possibles dépend du nombre de connexions compatibles en sortie, d'où l'utilisation d'un tableau de réponses.

Notez qu'il est possible de placer un composant synchrone en mode d'attente de la réception d'un événement de type particulier. La primitive receive correspondant à cet sémantique est une variante intéressante et encore une fois uniquement réalisable par décomposition des phases d'émission de requête et de réception de réponse.

9.3.1 Modèle comportemental

Pour obtenir un mode de communication *synchrone côté client*, les approches «classiques» (type Corba) interviennent au niveau de l'implantation en construisant des abstractions synchrones au-dessus des mécanismes de plus bas niveau et eux asynchrones. Cette démarche d'abstraction système est loin d'être simple, contrairement à ce qu'en pensent nombre de développeurs d'applications distribuées. Dans le cadre de Comet, les interventions nécessaires sont également assez intrusives mais elles se font systématiquement d'un point de vue formel, par spécialisation (i.e. raffinement formel) du modèle comportemental canonique. Les solutions que nous proposons sont donc de complexité comparable aux problématiques de bus d'objets répartis mais le discours se déplace sensiblement vers la manipulation de modèles algébriques, selon nous beaucoup plus flexibles et «contrôlables» que les opérationnalisations de bas niveau.

Contrairement aux composants Comet canoniques, un client disposant de propriétés synchrones peut se retrouver dans trois états différents :

- comportement «normal» (mode *normal*),
- en attente de réponse de la part d'un serveur (mode *wait*),
- une réponse a été reçue et doit être traitée (mode *signal*).

Ces différents modes de comportements sont représentés par le type énuméré *SYNCMODE* :

$$SYNCMODE ::= normal \mid wait \mid signal$$

Le schema de comportement par défaut (cf. schéma *DefaultBehavior* page 116) doit être modifié pour prendre en compte ce mode de fonctionnement :

SyncBehavior

$DefaultBehavior$

$reply\_queue : \mathrm{seq}\ Event$

$syncmode : SYNCMODE$

$local\_ts : \mathbb{N}$

$nb\_replies : \mathbb{N}$

$\#reply\_queue \leq nb\_replies$

La variable *syncmode* est donc ajoutée au schéma, de même que trois variables importantes : *reply_queue* conservant les réponse en cours de réception, *local_ts* qui permet l'identification des événements et *nb_replies* qui conserve le nombre de réponse attendues. La condition d'invariant vérifie que le nombre de requêtes déjà reçues (dans *reply_queue*) est bien inférieur au nombre de requêtes que l'on s'attend à recevoir.

L'initialisation de ce nouveau schéma de comportement se fait de la façon suivante :

InitSyncBehavior

$SyncBehavior'$

$InitDefaultBehavior$

$syncmode = normal$
$reply\_queue = \langle\rangle$
$local\_ts = 0$
$nb\_replies = 0$

Dans l'état initial, le composant est donc en mode de fonctionnement *normal*. Les transitions possibles ensuite sont présentées en figure 9.5.

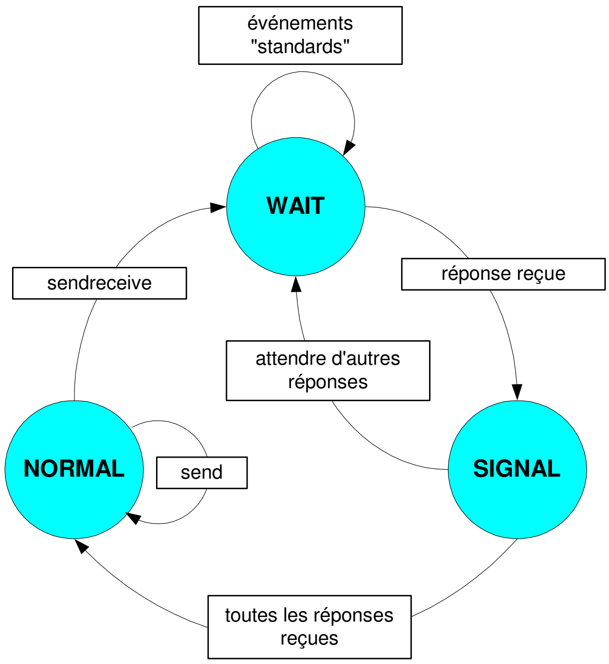


FIG. 9.5 – *Client/Serveur synchrone : états comportementaux*

Dans l'état *normal*, seule une émission synchrone - par l'intermédiaire de la primitive sendreceive - peut faire basculer le composant en mode *wait*. Par exemple, une émission aysnchrone avec send ne change pas le mode de fonctionnement.

Pour initier une interaction client/serveur en mode synchrone client, nous devons faire appel à un nouveau schéma d'émission explicite. Ce schéma devra gérer un nouveau type d'événement client/serveur ajoutant une information permettant d'identifier l'événement de manière unique :

$$
\begin{array}{|l}
\hline
\textit{SyncEvent} \\
\textit{ClientServerEvent} \\
\\
\textit{global\_id} : \textit{IDENTIFIER} \\
\textit{local\_id} : \mathbb{Z} \\
\hline
\end{array}
$$

L'identification de l'événement est divisée en une partie globale (un identificateur de composant) et une partie locale (un simple entier).

Concernant le client, l'émission d'un événement synchrone correspond à un raffinement de *SendRequestAction* (cf. page 179) :

$$
\begin{array}{|l}
\hline
\textit{SendReceiveAction} \\
\Delta \textit{SyncBehavior} \\
\\
\textit{SendRequestAction} \\
\\
\textit{event}! : \textit{SyncEvent} \\
\hline
\textit{syncmode} = \textit{normal} \\
\textit{reply\_queue} = \langle\rangle \\
\\
\textit{event}!.\textit{local\_id} = \textit{local\_ts} \\
\\
\textit{nb\_replies}' = \#\{c : \textit{out\_channels} \mid (\exists\, t : c.\textit{type} \bullet t \preceq \textit{event}?.\textit{type})\} \\
\\
\textit{syncmode}' = \textit{wait} \\
\hline
\end{array}
$$

Il est intéressant de noter que ce schéma modifie le comportement du composant. Nous vérifions tout d'abord que le composant est en mode *normal* et qu'aucune réponse n'est présente dans la file spécialisée. La valeur de la variable $\textit{nb\_replies}'$ correspond au nombre de canaux sortant qui seront sélectionnés pour l'émission implicite de l'événement. L'état du comportement passe alors en mode *wait*; ce qui doit correspondre, en terme de contrôle, à une suspension de l'activité principale du composant, à savoir l'opération *DefaultMetaExec* et consorts. Nous reviendrons sur ce point par la suite. La variante d'émission/réception explicite est donnée ci-dessous :

$$
\begin{array}{|l}
\hline
\textit{SendReceiveExplicitAction} \\
\Delta \textit{SyncBehavior} \\
\\
\textit{SendRequestExplicitAction} \\
\hline
\textit{syncmode} = \textit{normal} \\
\textit{reply\_queue} = \langle\rangle \\
\\
\textit{event}!.\textit{local\_id} = \textit{local\_ts} \\
\\
\textit{nb\_replies}' = \#\textit{rids}? \\
\\
\textit{syncmode} = \textit{wait} \\
\hline
\end{array}
$$

Dans ce cas, le nombre de réponses attendues est tout simplement égal au cardinal de l'ensemble des receveurs désignés par *rids*?.

Du côté du serveur, la seule modification par rapport à *SendReplyAction* (cf. page 179) concerne la restauration de l'identification de l'événement :

SendSyncReplyAction

$DefaultBehavior$

$SendReplyAction$

$event? : SyncEvent$
$event! : SyncEvent$

$event!.local\_id = event?.local\_id$

$event!.global\_id = \text{head } event?.call\_chain$

Il est très important de remarquer sur ce schéma que le serveur n'a pas besoin d'un comportement de type synchrone : cette opération s'effectue sur l'état *DefaultBehavior*. En revanche, les événements passés doivent être de type *SyncEvent*. L'identificateur local doit être restauré, de même que l'identificateur global, correspondant au destinataire de l'envoi.

D'après la figure 9.5, au moment de la réception d'un événement par le client - alors que ce dernier est bloqué (en mode *wait*) - plusieurs possibilités sont à considérer :

1. L'événement reçu n'est pas une réponse, ce qui ne doit pas empêcher le client de le recevoir. Bien sûr, il ne sera capable de traiter cette événement que lorsque toutes les réponses seront reçues.
2. Il s'agit d'une réponse valide mais il reste d'autres réponses en attente. L'événement est mis dans une file d'attente provisoire des réponses reçues.
3. La réponse reçue est la dernière attendue. Dans ce cas, le composant doit revenir en mode de fonctionnement *normal*.

Pour effectuer le *dispatch* entre ces différents cas, nous allons introduire une variation du couple *DefaultMetaReceive*/*DefaultMetaInQueue* (cf. page 121). La figure 9.6 résume les modifications structurelles nécessaires (la prochaine sous-section aborde la problématique du contrôle).

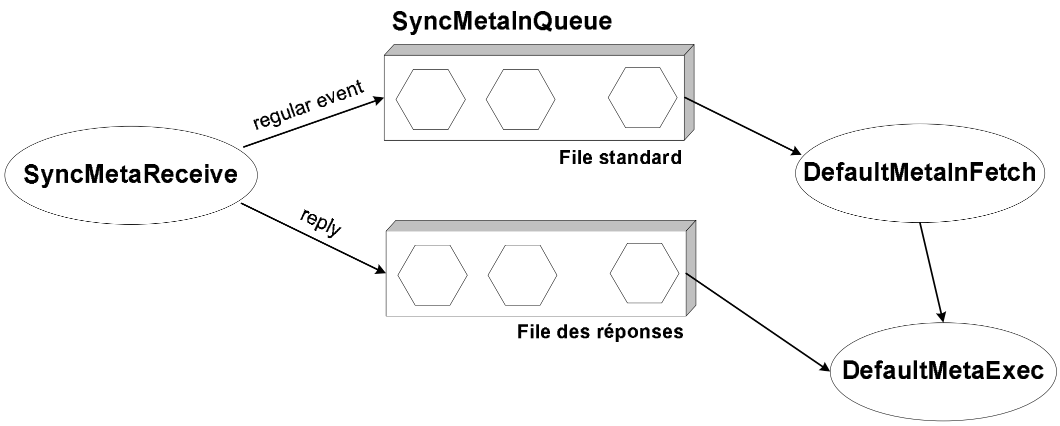


FIG. 9.6 – *Client/Serveur synchrone : modifications comportementales*

La partie réception est agrémentée d'un schéma de sélection de la file qui peut prendre les valeurs *standard* ou *reply* :

$QUEUE\_TYPE ::= standard \mid reply$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SyncMetaReceive}} \\
\hline
\textit{SyncBehavior} \\
\\
event? : Event \\
event! : Event \\
\\
queue\_type! : QUEUE\_TYPE \\
\hline
event! = event? \\
\\
\textbf{if } syncmode = normal \textbf{ then} \\
\quad queue\_type! = standard \\
\textbf{else if } (event?.type \preceq SyncEvent) \wedge (id = event?.global\_id) \wedge (local\_ts = event?.local\_id) \textbf{ then} \\
\quad queue\_type! = reply \\
\textbf{else} \\
\quad queue\_type! = standard \\
\hline
\end{array}
$$

Dans le mode de fonctionnement *normal*, la file sélectionnée (dans *queue*!) est bien sûr la file normale. Si en revanche le composant est en mode *wait* alors plusieurs vérifications sont nécessaires pour savoir s'il s'agit réellement d'une réponse attendue. Tout d'abord, le type de l'événement doit être un sous-type de *SyncEvent* et son identificateur *global_id* doit correspondre à l'identificateur *id* du composant. L'estampillage *local_ts* doit de plus être égal à l'identificateur numérique local de l'événement reçu. Il est facile de montrer que ce couple permet de caractériser un événement particulier au niveau du système global. En effet, la partie *global_id* identifie comme nous l'avons vu un composant de manière unique. Et pour ce composant, le champ *local_id* est également unique pour chaque invocation synchrone (cf. ci-dessous). Finalement, si l'un de ces trois tests échoue, alors la file normale est utilisée[5].

Il est désormais possible de raffiner la mise en file d'attente en prenant en paramètre la file choisie :

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SyncMetaInQueue}} \\
\hline
\Delta \textit{SyncBehavior} \\
\\
event? : Event \\
queue\_type? : queue\_type \\
\hline
\textbf{if } queue\_type = reply \textbf{ then} \\
\quad (reply\_queue' = \langle event? \rangle \frown reply\_queue) \wedge (syncmode' = signal) \\
\textbf{else} \\
\quad in\_queue' = \langle event? \rangle \frown in\_queue \\
\hline
\end{array}
$$

Remarquons que dans le cadre d'une modification de la file des réponses, le mode de fonctionnement du composant passe dans l'état *signal*. Ceci revêt principalement une importance en terme de contrôle. Il ne nous reste plus qu'à vérifier si, dans le cadre d'une réception de réponse, ce mode doit être remis à l'état *normal* (toutes les réponses reçues) ou *wait* (il reste des réponses à recevoir) :

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SyncResetMode}} \\
\hline
\Delta \textit{SyncBehavior} \\
\hline
\textbf{if } syncmode = signal \textbf{ then} \\
\quad \textbf{if } \#reply\_queue = nb\_replies \textbf{ then} \\
\quad\quad syncmode' = normal \wedge local\_ts' = local\_ts + 1 \\
\quad \textbf{else} \\
\quad\quad syncmode' = wait \\
\hline
\end{array}
$$

La restauration du mode n'est activée que si sa valeur est *signal*. Dans le cas ou toutes les réponses ont été reçues (comparaison du cardinal de la file des réponse avec la variable *nb_replies*), alors le mode *normal* est restauré. De plus, l'estampillage local est avancé; ce qui permet de s'assurer que la prochaine invocation synchrone sera identifié de manière unique au niveau local. Si d'autres réponses doivent être reçues, alors le mode de restauration est bien sûr *wait* (sans modification de l'estampillage).

5. Il est donc possible de faire transiter des événements synchrones par des composants qui les traiteront normalement. Ceci ouvre d'intéressantes possibilités en matière de mécanisme de rappel par exemple.

La phase complète de réception devient ainsi :

$$SyncReceive = (DefaultMetaReceive \gg SyncMetaReceive \gg SyncMetaInQueue) \fatsemi SyncResetMode$$

Ainsi, après réception effective (l'ancien *DefaultMetaReceive* est toujours valide à ce niveau), l'événement reçu est routé par *SyncMetaReceive* puis mis dans une des deux files d'attente disponibles. Finalement le mode de fonctionnement est mis à jour.

La seule opération qu'il nous faudrait écrire pour finaliser notre modèle de comportement concerne la récupération des requêtes reçues pour les retourner dans le cadre de l'opération *SendReceiveAction*. Cette simple recopie n'est cependant vraiment utile qu'au niveau de l'implémentation des primitives linguistiques associées aux interactions synchrones.

9.3.2 Réseaux d'activations concurrentes

Le terme *synchronisation* dénote une propriété que l'on associe généralement à un problème de *contrôle*. Le modèle de comportement *SyncBehavior* est donc assez intrusif sur le contrôle des composants qui l'emploient. Cet impact dépend fortement du mode de comportement dans lequel se trouve le composant : *normal*, *wait* ou *signal*.

Mode normal

En mode *normal*, le composant se comporte pratiquement comme s'il employait le modèle par défaut *DefaultBehavior*. Cependant, une nouvelle action (*SendReceiveAction*) peut faire basculer ce mode de fonctionnement en mode de synchronisation *wait*.

A un niveau plus formel, il nous faut établir les réseaux d'activation pour les méta-objets mis en œuvre dans le modèle *SyncBehavior*. Au niveau de la phase d'émission des événements, nous avons vu que les méta-objets concernés restaient en fait inchangés par rapport au mode de comportement par défaut. Nous restreindrons donc notre étude aux méta-objets concernés par les phases de réception et de traitement des événements. Considérons tout d'abord le réseau partiel représenté sur la figure 9.7 page suivante.

Ce réseau montre tout d'abord les propriétés d'activation concurrente des méta-objets *SyncMetaReceive* et *SyncMetaInQueue* (phase de réception) en mode *normal*. Dans ce mode de fonctionnement, les deux méta-objets ne peuvent être activés qu'en séquence. Pour marquer la place *InQueueNormal* du réseau, il faut d'une part disposer d'un événement à mettre en file d'attente (*SyncMetaInQueue* marqué), mais également prendre possession des jetons se trouvant respectivement dans la place *NormalMode* (drapeau de mode) et *InQueueLock* (verrou). Juste avant de marquer *InQueueNormal*, un jeton est «rendu» à la place *EventInChannel* indiquant que l'on peut recommencer à recevoir des événements. N'oublions pas que cette ressource est critique et doit donc être restaurée le plus rapidement possible.

Après passage dans *InQueueNormal*, la transition suivante relâche le verrou *InQueueLock* et replace également le jeton pris dans *NormalMode*. Les événements ainsi reçus sont stockés dans la place *InQueueEvents* qui est une place non-bornée (d'ou la présence du signe *infini* à côté de cette place).

Le méta-objet d'extraction des événements entrant *SyncMetaInFetch* est également synchronisé sur le verrou entrant mais peut - comme dans le modèle par défaut - s'exécuter en parallèle de *SyncMetaReceive*. Suit l'étage d'exécution *SyncMetaExec* qui pour l'instant ne considère que l'action *SendAction* donnée à titre d'exemple puisque cette phase d'exécution à déjà été introduite au chapitre 7.

Mode wait

Les propriétés d'activation les plus intéressantes concernent le passage en mode *wait* et la gestion de ce mode de fonctionnement au fur et à mesure de la réception des requêtes. Le réseau d'activation associé est présenté en figure 9.8 page suivante.

Pour simplifier l'exposé de ce réseau nettement plus complexe que ce que nous avons étudié jusqu'à présent, les places, transitions et arcs spécifiques au mode de fonctionnement *normal* n'y sont pas figurés.

Le composant a l'origine se trouve en mode *normal*. Pour passer en mode *wait*, il faut initier une requête synchrone correspondant au marquage de la place *SendReceiveAction*. Le réseau de Petri montre qu'il faut acquérir deux ressources très importantes pour atteindre cet objectif :

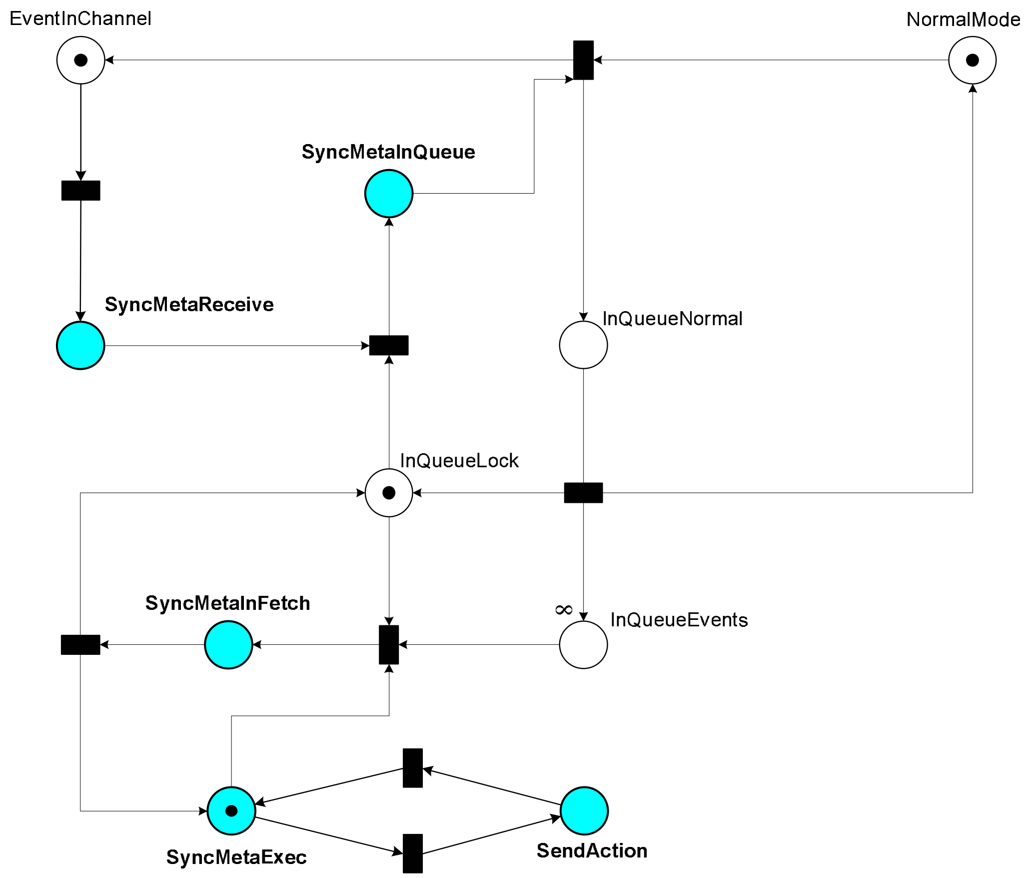


FIG. 9.7 – *Modèle de comportement synchrone en mode normal*

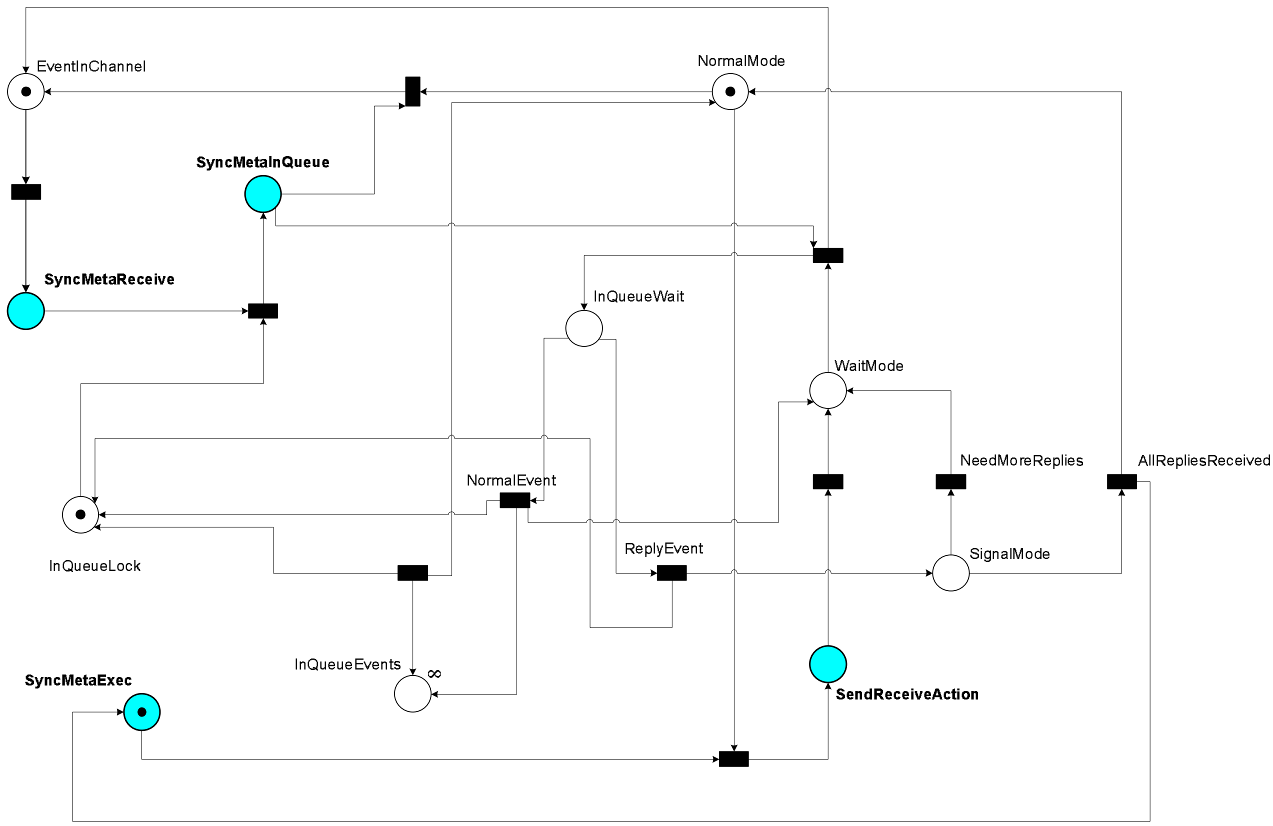


FIG. 9.8 – *Modèle de comportement synchrone en mode wait*

1. le jeton qui marque le fonctionnement en mode normal (place *NormalMode*), et
2. le jeton d'activation de l'étage de traitement se trouvant dans *SyncMetaExec* à l'état initial.

Une fois ces deux conditions réunies, la place *SendReceiveAction* peut être marquée. Il est très important de constater que l'étage de traitement des événements se trouve ainsi désactivé au moment de l'activation d'une requête synchrone[6]. La deuxième conséquence importante de ce marquage concerne, après la transition suivante, le passage effectif en mode de fonctionnement *wait*. Ce fait correspond au marquage de la place *WaitMode*.

A partir de ce moment, le traitement des événements est donc inhibé et la seule activité possible concerne la réception de nouveaux événements. Dans ce cas, l'étage de mise en file d'attente *SyncMetaInQueue* ne peut prendre possession d'un jeton dans *NormalMode*. Ce dernier a effectivement été pris par l'événement ayant déclenché la requête synchrone et se trouve maintenant - en quelque sorte - dans *WaitMode*. La seule transition franchissable est donc celle qui entraîne le marquage de *InQueueWait* (le pendant de *InQueueNormal* du réseau précédent). Nous pouvons remarquer que cette transition ne peut être franchie que si le composant se trouve en mode *wait*; c'est-à-dire qu'il y a bien un jeton dans *WaitMode*. Tout comme dans le cas *normal*, la réception des événements est réactivée (marquage de *EventInChannel*).

Dès lors, l'événement reçu dans *InQueueWait*[7] peut circuler selon deux cheminements différents. Dans le modèle algébrique (schémas Z), le choix de la route était déterminé par la nature de l'événement. Au niveau de l'activation, il correspond plutôt à un franchissement non-déterministe d'un transition parmi deux possibles : *NormalEvent* ou *ReplyEvent*.

Le franchissement de la transition *NormalEvent* correspond à la réception d'un événement normal (et pas une réponse de requête synchrone). Puisque le composant fonctionne en mode *wait*, l'événement ne peut être traité et doit être mis en file d'attente pour traitement ultérieure. Il suffit pour cela de place l'événement (i.e. le jeton qui le caractérise) dans la file (i.e. la place) *InQueueEvents*. Ces événements ne sont pas traitables puisque l'opération *SyncMetaInFetch* ne peut acquérir le jeton d'activation de l'étage de traitement. La place *WaitMode* doit bien sûr être de nouveau marquée.

L'autre route possible correspond au choix (toujours non-déterministe ici) du franchissement de la transition *ReplyEvent* qui dénote la réception d'une des réponses attendues. Dans ce cas, le composant passe en mode *signal* (marquage de *SignalMode*), suivi de deux transitions possibles :

1. La réponse reçue était la dernière attendue, ou
2. il reste des réponses à recevoir.

Le deuxième choix (encore une fois non-déterministe pour le réseau de Petri) concerne le franchissement de *NeedMoreReplies* dont la conséquence est simplement le passage une nouvelle fois en mode *wait*.

En revanche, si la transition *AllRepliesReceived* est franchie, alors il ne reste plus de réponse à recevoir; ce qui a pour conséquences deux faits majeurs :

– Le composant retrouve le mode de fonctionnement *normal* (marquage de *NormalMode*), et
– l'étage d'exécution *SyncMetaExec* est de nouveau activé, ce qui permet éventuellement de traiter les événements «normaux» ayant été placés en attente pour autoriser la gestion prioritaire des réponses synchrones.

Nous comprenons désormais le choix du nom *signal* pour le troisième mode de fonctionnement mis en œuvre. En effet, il s'agit ici - potentiellement - de signaler à l'étage d'exécution qu'il peut reprendre son fonctionnement.

Réseau complet

Le réseau d'activation du mode de comportement *SyncBehavior* correpond à la fusion entre les deux réseaux présentés précédemment. Cette fusion est représentée en figure 9.9 page suivante.

6. Nous donnerons un sens plus formel à cette remarque lors de l'étude du réseau complet.
7. Il s'agit bien sûr d'une expression métaphorique car la place du réseau ne reçoit qu'un jeton; l'événement lui-même est géré par les opérations Z et non le réseau d'activation. Mais il est intéressant de constater que le caractère très intuitif de notre métaphore confirme l'intéressante complémentarité entre les deux points de vue.

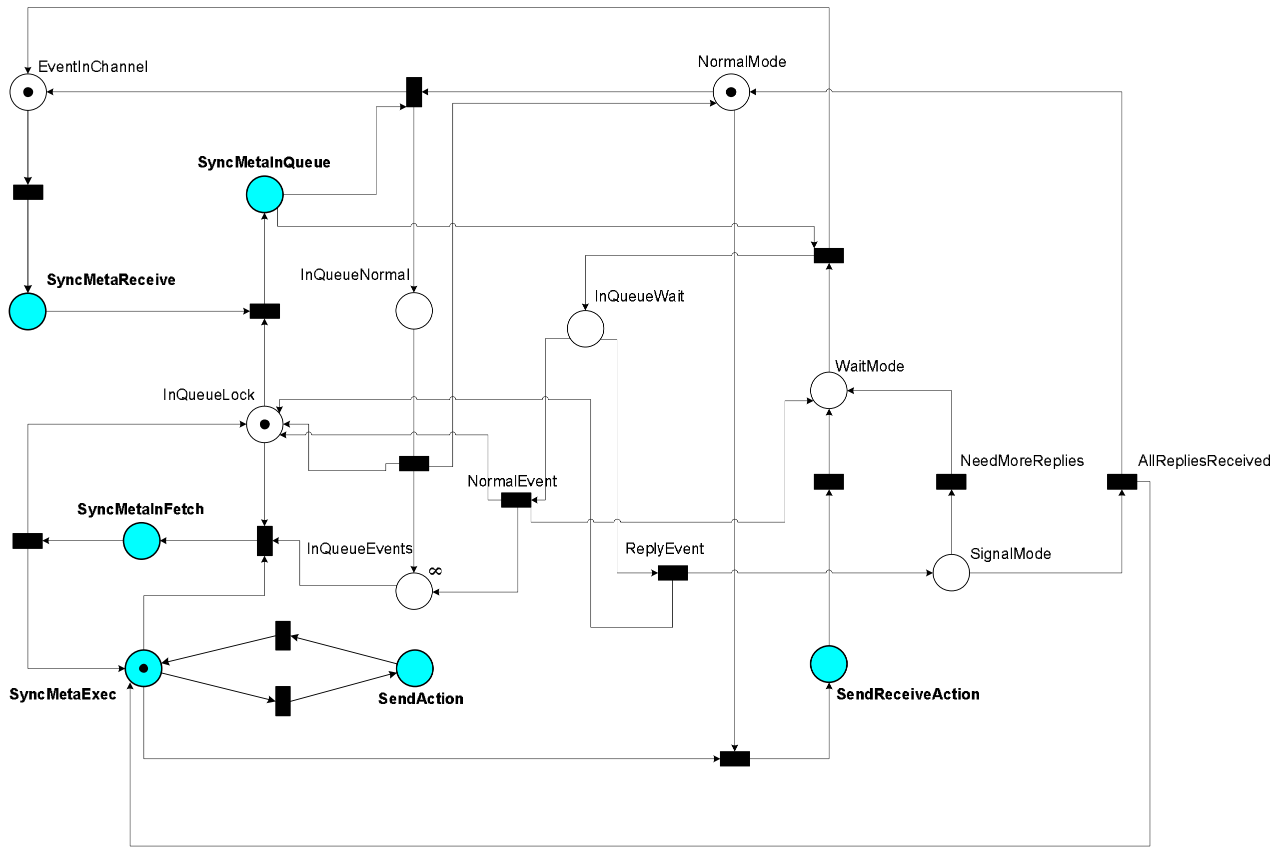


FIG. 9.9 – *Modèle de comportement synchrone : réseau d'actication complet*

Ce réseau complet se caractérise par sa relative complexité [8], du moins si on le compare aux réseaux d'activation que nous avons élaborés au préalable. Cependant, il n'est pas possible d'analyser formellement nos propriétés d'activation sans construire le réseau complet et le «passer à la moulinette» des outils disponibles. Parmi ces propriétés, nous retrouvons nos contraintes d'activations parallèles qui dénotent les obligations de preuves «étendues» au comportement concurrent des composants. En mode de fonctionnement normal, les contraintes d'activation du modèle *SyncBehavior* sont les mêmes que dans le cadre du comportement par défaut (cf. table 7.2 page 129). En revanche, en modes de fonctionnement *wait* ou *signal*, l'étage d'exécution n'est plus activable. Nous pouvons décrire cette propriété de nombreuses façons.

Donnons par exemple l'invariant suivant :

$$m(WaitMode) = 1 \vee m(SignalMode) = 1 \Rightarrow m(DefaultMetaInFetch) + m(DefaultMetaExec) = 0$$

De même, nous pouvons calculer une propriété très importante concernant la cohérence de notre modèle de comportement : l'exclusion mutuelle entre les différents mode de fonctionnement.

Du point de vue du réseau de Petri, nous pouvons calculer :

$$m(NormalMode) + m(WaitMode) + m(SignalMode) \leq 1$$

Les places associées aux différents modes de fonctionnement du composant sont donc bien mutuellement exclusives. Puisque ces places «décident» des différentes activations possibles pour les schémas d'opération, on peut en déduire l'exclusion «algébrique» entre les modes de fonctionnement.

Finalement, nous pouvons (ou plutôt les outils que nous utilisons peuvent) prouver - en posant initialement la place *InQueueEvents* comme non-bornée - que le réseau obtenu est à la fois *sûr* et *vivant*. Ceci indique que quoiqu'il arrive, le composant ne se retrouvera jamais dans un état «définitif» qui pourrait l'empêcher par

8. Cette complexité est toute relative puisque malgré ses 14 places, 15 transitions et 54 arcs, ce réseau de Petri est tout à fait modeste pour un ordinateur lui aussi bien modeste. En revanche, l'apréhension de ce réseau par un humain est probablement plus «difficile»...

exemple de traiter les événements reçus, de ne pas autoriser de nouvelle requête synchrone, ou encore de ne pas traiter correctement une réponse attendue.

En revanche, sous sa forme actuelle, le réseau n'est pas *équitable* en ce qui concerne la mise en file d'attente des événements (cette dernière peut grandir indéfiniment). Il faut cependant se rappeler que le réseau d'activation ne donne *a priori* que des propriétés d'activation. Or, il est évident que la gestion des systèmes de file d'attente dépasse le cadre de l'activation pure des schémas Z.

9.3.3 Exemple : vote transactionnel distribué

Nous avons vu que l'emploi de la primitive sendreceive (cf. table 9.1 page 180) permettait d'exprimer dans l'univers Comet des modes d'interactions client/serveur nous rapprochant des environnements objets, tout en conservant une identité propre. Bien sûr, ce rapprochement se fait sans mettre en doute les propriétés très intéressantes de la plateforme en mode asynchrone.

Mais, comme nous l'avons déjà signalé précédemment, le mode de diffusion multi-point implicite en fonction du type permet beaucoup plus. Il couvre en effet un territoire très vaste de possibilités, que nous pensons réellement innovantes, dans le cadre des échanges distribués, et notamment le client/serveur synchrone. En particulier, cela permet de dénoter, en une unique invocation de la primitive sendreceive, une gamme d'interactions de type *rendez-vous* qu'il faudrait en temps normal construire de toutes pièces.

Pour illustrer cette différence réellement décisive selon nous avec les approches «point-à-point» (interstitiels classiques, langages d'acteurs, etc.), nous allons nous intéresser dans ce qui suit au domaine des *systèmes transactionnels* et notamment la phase de validation de transaction (*Commit*).

Ces systèmes transactionnels mettent (au moins) en œuvre les entités suivantes :

- les clients de transaction qui effectuent des requêtes, généralement restreintes à des lectures/écritures, dans le cadre de transactions spécifiques, sur des données partagées par les différentes transactions.
- les *coordinateurs* (ou *moniteurs*) de transactions qui gèrent ces dernières en terme de contrôle;
- les serveurs, appelés *workers*, réalisant les traitements requis, toujours dans le cadre de transactions données. Chaque *worker* est généralement associé à une partie de l'état modifié par les transactions et gère les accès concurrents correspondant à des transactions menées en parallèle.

Les propriétés associées aux interactions transactionnelles sont résumées par les célèbres initiales *ACID* :

- *Atomicité* : l'ensemble des ordres d'écriture/lecture associés à une transaction est une séquence atomique ; tous les ordres sont exécutés ou bien aucun ne l'est, aucun état intermédiaire ne peut être atteint.
- *Cohérence* : l'état du système transactionnel doit rester cohérent du point de vue global durant tout le déroulement de la transaction.
- *Isolation* : chaque transaction est totalement hermétique au monde extérieur, en particulier aux autres transactions potentiellement conflictuelles, tant qu'elle n'a pas été validée.
- *Durable* : après validation, les effets d'une transaction ne peuvent être remis en doute.

Pour satisfaire ces propriétés, l'un des aspects critiques d'un système transactionnel concerne la validation des transactions. La majorité des moniteurs transactionnels mettent en œuvre, principalement parce qu'ils offrent d'importantes propriétés relatives à la résistance aux pannes, le célèbre protocole de validation à deux phases (i.e. *two-phases commit protocol*).

Après communication depuis le client au coordinateur d'une requête de validation de transaction, ce même coordinateur doit demander à tous les *workers* mis en jeu que la transaction qu'il gère soit validée (*Commit*) ou annulée (*Abort*). Pour ce faire, le coordinateur demande à chaque *worker* d'émettre un choix *local* au sujet de la validation. On parle alors de protocole de *vote réparti* ou chaque *worker* doit répondre par *oui* ou *non* à la question : *la transaction peut-elle être validée ?*.

Donnons ci-dessous une définition possible pour les composants *workers* dans le cadre de cette phase de vote réparti de validation :

```
component Worker extends DefaultBehavior {
  receive CoordEvent;
  send VoteEvent;

  when(CoordEvent coord) {
    if(coord.getControl()==ASK_VOTE)
      if(commitTransaction(coord.getTransactionID())) {
        sendReply(new VoteEvent(YES));
      else
        sendReply(new VoteEvent(NO));
      }
  }
}
```

Lorsqu'un composant Worker reçoit un événement de type CoordEvent lui demandant (par l'accesseur getControl()) de voter pour la validation d'une transaction, cette dernière est d'abord vérifiée localement (méthode commitTransaction() non spécifiée ici), c'est-à-dire en fonction de l'état local du Worker considéré. Suivant le résultat de cette vérification, un vote est retourné, sa valeur de type VoteEvent pouvant être soit oui (YES) soit non (NO).

Il est intéressant de noter sur cet exemple que les parties serveurs de l'interaction considérée[9] ne possèdent (et n'ont besoin d') aucune information sur la nature synchrone de la requête. Le comportement des *workers* conserve donc son caractère asynchrone (utilisant le modèle de comportement par défaut); notre discussion peut donc - en toute indépendance - entièrement se tourner vers la problématique du contrôle côté client.

Chaque transaction est supportée par un coordinateur dédié qui doit être connecté - au sens propre du terme dans le cadre de Comet - aux différents *workers*. Voici la description de ce composant coordinateur pour tout ce qui concerne la phase de vote :

```
component Coordinator extends SyncBehavior {
  receive TransactionEvent, VoteEvent;
  send CoordEvent, TransactionEvent;

  when(TransactionEvent trans) {
    if(trans.getControl()==ASK_COMMIT) {
      VoteEvent votes[] = sendreceive(new CoordEvent(ASK_VOTE), VoteEvent.class);
      if(all_votes_ok(votes))
        sendReply(new TransactionEvent(COMMIT));
      else
        sendReply(new TransactionEvent(ABORT));
    }
  }
}
```

Au moment de la réception d'une demande de validation (requête de type CoordEvent et de valeur ASK_COMMIT), le coordinateur émet une requête de vote, c'est-à-dire un événement de type CoordEvent et de valeur ASK_VOTE. En raison du mode de diffusion multi-point implicite en fonction du type, cette requête de vote est émise à tout *worker* connecté en sortie du coordinateur pour le type CoordEvent (ou un super-type).

La figure 9.10 page suivante décrit une instance de notre protocole de validation à deux phases mettant en œuvre un client, un coordinateur et trois *workers*.

Nous pouvons voir que le coordinateur est bel et bien connecté (par des liens doubles) avec l'ensemble des *workers* mis en jeu. Lors de l'émission, par sendreceive, de la requête de vote, cette dernière est donc acheminée vers les trois composants connectés. Le coordinateur se met alors en attente de trois réponses qui serviront à «remplir» le tableau votes[]. Dans cet exemple, nous réalisons, *en une seule instruction atomique*, une *jonction implicite* des décisions prises par l'ensemble des *workers* mis en jeu et dont le nombre de *workers* correspond à une variable pour notre système. Dans une approche point-à-point traditionnelle, cette interaction serait bien plus complexe à mettre en œuvre, notamment si l'on veut que les *workers* puissent voter de façon concurrente.

9. Dans le cadre du vote réparti de validation de transaction, le coordinateur est considéré comme client des différents serveurs mis en jeu (i.e. les *workers*).

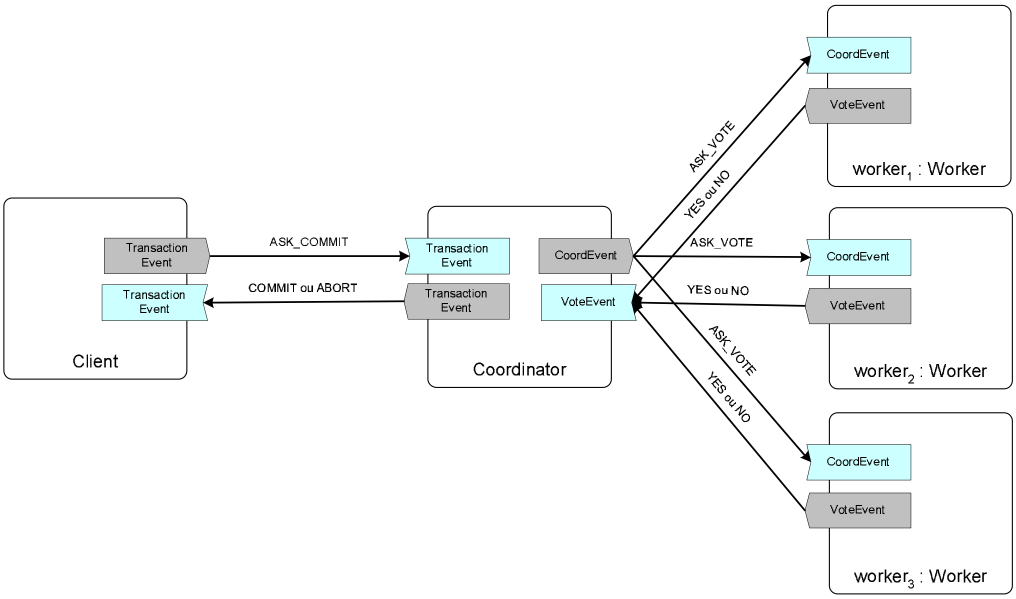


FIG. 9.10 – *Validation transactionnelle en Comet : phase de vote*

Le mode de diffusion multi-point implicite en fonction du type mis en œuvre dans la plateforme Comet représente donc, selon nous, ce qui marque notre différence avec (à notre connaissance) l'ensemble des approches interstitielles proposées dans l'industrie ou le monde de la recherche. Il s'agit bien de notre *principe premier*.

La phase suivante concerne l'analyse de ce tableau pour en déduire un choix de validation ou d'annulation de la transaction. Ce choix est assez drastique puisque si un seul des *workers* répond NO pour la phase de vote, alors la transaction est annulée et l'événement ABORT est retourné au client. En revanche, si tous les *workers* sont d'accords (tous les votes à YES), la transaction est validée (envoi de COMMIT au client).

Ce point de focalisation sur un élément important du moniteur transactionnel termine notre analyse fine du critère de synchronisation du point de vue du client.

9.4 Synchronisation : point de vue serveur

La synchronisation des systèmes concurents et/ou distribués est un vaste domaine d'une grande complexité que les approches interstitielles ont tendance à enfouir de manière assez profonde dans le bus d'objets. Très souvent, cet aspect pourtant central de l'informatique moderne se retrouve parfois relégué à une sorte de détail technique.

Du côté des objets serveurs, par exemple, la synchronisation liée au flux des requêtes émanant des clients et de ceux résultant de leurs traitement au sein des serveurs est un aspect critique mais généralement non manipulable simplement. Nous nous intéressons donc ici à une forme de manipulation assez directe de ces aspects de synchronisation considérés du point de vue des serveurs.

Notre but ici n'est pas tant de discuter sur ces aspects de synchronisation de flux que de montrer les ouvertures que propose notre modèle pour les aborder dans des conditions qui nous semblent plus satisfaisantes que, disons, en moyenne.

Bien sûr, nous nous limiterons ici à des expérimentations de nature purement pragmatique qui ont le principal intérêt de se situer au niveau de l'utilisation de notre interstitiel et non au niveau de son extension ou encore de son implantation.

Deux possibilités s'offrent principalement à nous concernant cette synchronisation de flux. Nous pouvons fournir une gamme de modèles comportementaux les prenant en charge ou bien les envisager sous un angle plus «adaptatif» en considérant cette synchronisation comme des protocoles, au sens Comet du terme. Nous nous focaliserons dans ce qui suit sur cette deuxième approche qui, si elle est réalisable, semble exploiter au mieux notre «philosophie» de conception de services adaptatifs.

Il existe pléthore de méthodes pour synchroniser des processus distribués sur une base de gestion de flux d'information. Il suffit de lire [KBH+01], par exemple, pour s'en convaincre. Dans nos exemples expérimentaux, nous allons nous restreindre à deux approches simples :

- la détection et le signalement de «sauts» dans le flux contrôlé, et
- la régulation dynamique des flux.

Ces deux approches complémentaires ont pour principal intérêt de représenter deux bornes bien définies du spectre méthodologique (et métaphoriquement policier) allant des approches répressives (exceptions) aux approches préventives (régulation).

9.4.1 Approches exceptionnelles

Commençons notre étude par une approche de type exceptionnelle, avant tout parce qu'il s'agit de la technique la plus simple. L'idée est de mettre en place, dans un premier temps, un mécanisme de détection d'anomalies dans le flux de contrôle au niveau d'un serveur.

Dans le cadre de la plateforme Comet, le serveur sera représenté par un composant et nous nous proposons de mettre en œuvre une version adaptative de décteur sous la forme d'un rôle que l'on peut «enficher» dans un serveur en cours d'exécution, selon le procédé bien connu désormais de l'assignation dynamique d'un rôle (cf. figure 9.11).

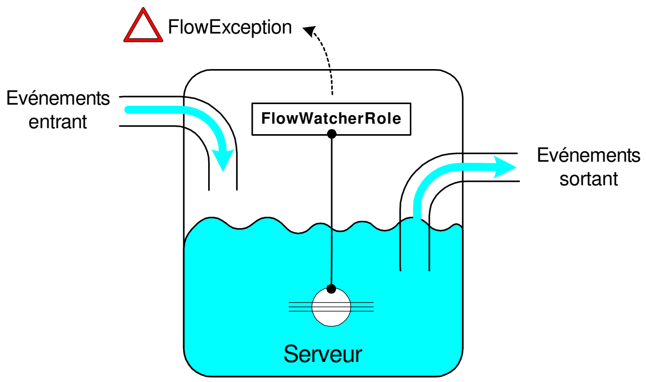


FIG. 9.11 – *Synchronisation de flux côté serveur : approche exceptionnelle*

Pour simplifier notre discours, nous allons baser la détection sur une donnée simple : l'intervalle de temps séparant le traitement de deux événements distincts au niveau du serveur. Bien sûr, de nombreux autres indices pourraient servir à la détection d'anomalies de flux mais peu satisfont les deux règles nécessaires à une détection efficace : la simplicité et le faible coût.

Il nous faut également donner un sens à l'expression «anomalies de flux» en relation avec cet indice d'intervalle temporelle de traitement. Nous établirons qu'un flux devient problématique si l'intervalle de traitement *moyen* dépasse une certaine limite fixée à l'avance (mais bien sûr adaptable à l'exécution). Nous donnons ci-dessous l'expression simplifiée d'un tel rôle de détecteur dans notre langage de description :

```
role GenericFlowWatcher {
  prehook Event;
  send FlowException;

  before(Event event) {
    _event_count++;
    long current = System.currentTimeMillis();
    _total_time = _total_time + current;
    _avg_time = _total_time / _avg_time;
    if(_avg_time > _avg_limit)
      send(new FlowException(_avg_time));
  }
}
```

Encore une fois, aux détails prêts de la configuration du détecteur et de la déclaration des attributs du rôle, la description ci-dessus - malgré sa concision - est quasiment complète. Le rôle GenericFLowWatcher met en œuvre un simple crochet avant traitement de tout événement au niveau du composant auquel il sera assigné. Lors du crochetage d'un tel événement, le nombre d'événement traités _event_count est tout d'abord augmenté. Nous utilisons ensuite une très utile méthode standard de Java pour récupérer le temps local immédiat en millisecondes. Ce temps est ajouté au temps total conservé dans _total_time puis une moyenne est calculée. Si cette moyenne _avg_time dépasse un certain seuil _avg_limit, alors une exception est simplement levée.

De l'autre côté de ce rôle doit donc se trouver un rôle ou composant complémentaire permettant d'interpréter cette exception en terme de flot de contrôle. De nombreuses approches sont possibles : régulation (cf. ci-dessous), interdire la connexion de nouveaux clients, etc.

9.4.2 Approches à régulation de flux

De façon à la fois complémentaire et alternative à l'exemple décrit précédemment, nous allons construire pour clore notre débat sur les modèles client/serveur un rôle, encore une fois très concis de régulation de flux.

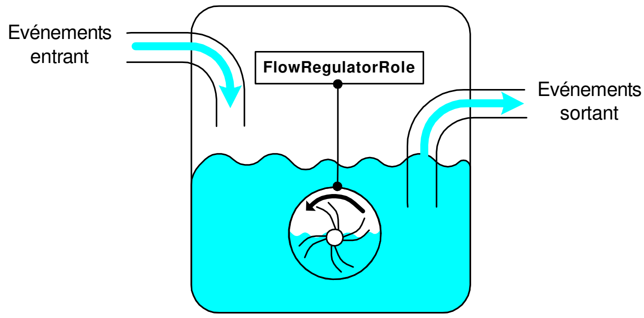


FIG. 9.12 – *Synchronisation de flux côté serveur : régulation*

Lorsque ce rôle - nommé GenericFlowRegulator - est assigné à un serveur, alors il impose une cadence minimale précise au traitement de chaque événement reçu (cf. 9.12). Encore une fois, il s'agit d'une des nombreuses variantes imaginables pour réguler des communications distribuées mais cet exemple sait rester encore une fois simple et relativement efficace. La définition proposée est la suivante :

```
role GenericFlowRegulator {
  prehook Event;
  posthook Event;

  before(Event event) {
    _start_time = System.currentTimeMillis();
  }
  after(Event event) {
    _end_time = System.currentTimeMillis();
    if(_end_time - _start_time < _min_rate)
      Thread.sleep(min_rate - (_end_time - _start_time));
  }
}
```

Ce rôle, une fois assigné, intercepte tous les événements reçus par un serveur. Avant le traitement de chaque événement (crochet avant), le temps local précis est récupéré dans la variable _start_time. Après traitement (crochet après), le temps local est une nouvelle fois demandé (variable _end_time). Nous savons donc, à ce moment, précisément (à la milliseconde prêt), combien de temps il a fallu pour traiter l'événement. Si ce temps est inférieur à la cadence minimale désirée (variable min_rate), alors on temporise pour que le prochain événement ne soit pas traité "trop tôt" relativement à la cadence que l'on s'impose.

Cette démarche préventive, quoique simple, permet de se sortir de nombreux pièges relatifs au mode de communication asynchrone privilégié dans Comet. Bien sûr, l'indice immédiat utilisé pourrait être associé à des indices en moyenne plus révélateurs du comportement du composant régulé. De la même manière, il faudrait pouvoir adapter la cadence minimale, par exemple en mettant des détecteurs côté client (pour rejoindre l'approche précédente). Mais encore une fois, les rôles Comet nous semblent particulièrement adaptés pour mettre en œuvre ces protocoles étendus de synchronisation de serveur de façon très simple.

On peut même étendre cette régulation non destructrice (aucun événement perdu) en imposant une cadence maximale qui, pour être satisfaite, nécessite la possibilité de filtrer des événements entrant. On pense tout naturellement aux rôles filtres entrant pour prendre en charge cette variante.

Nous pouvons donc clore notre débat sur l'adéquation de notre modèles aux exigences du monde client/serveur en remarquant que les mécanismes proposés ne sont pas plus complexes[10], que ce qu'il est nécessaire de mettre en œuvre dans les approches interstitielles plus classiques, essentiellement fondées sur le concept d'objet distribué. L'autre avantage est que dans tous les cas, nous sommes restés au niveau du modèle et ne sommes jamais «descendus» au niveau de l'opérationnalisation de grain fin de ce même modèle (i.e. son implantation).

Nous avons même montré dans notre dernière section sur la synchronisation côté serveur que certaines possibilités généralement absentes des approches de type interstitiel étaient disponibles, toujours au niveau du modèle, dans le cadre de Comet. Ces possibilités supplémentaires exploitent l'ensemble du modèle, ce qui nous permet désormais de «refermer» ce dernier ...

10. Je serais même tenté de penser qu'ils sont plus simples, pour avoir un peu «farfouillé» dans l'implantation du bus RMI. Mais il faudrait pour s'en convaincre une «force» de développement plus importante...

«L'optimisation prématurée est la racine du mal»
Donald E. Knuth

10
Noyau du système

Ce chapitre permet, après maints développements formels, d'entrer de plein pied dans la validation expérimentale, à la fois de notre modèle complet et des intuitions qui ont conduit à son élaboration.

10.1 Processus de développement

Comme nous l'avons introduit en préambule de notre thèse, nous nous plaçons dans la démarche de conception d'un interstitiel, non au sens «classique» - c'est-à-dire objet - du terme, mais plutôt dans l'idée de faire le lien entre un modèle de développement - quel qu'il soit - et une opérationnalisation dans un univers distribué.

Nous présentons en figure 10.1 le processus qui permet de passer des descriptions applicatives Comet à leur opérationnalisation.

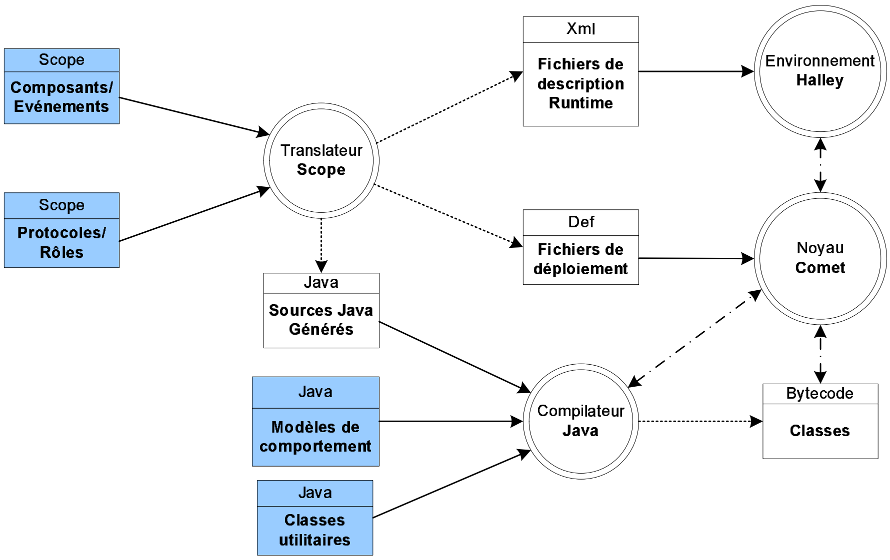


FIG. 10.1 – *Plateforme Comet : Processus de développement*

Il est important de garder à l'esprit que cette description correspond à une itération simple du processus plus général qui - lui - est totalement incrémentiel, puisque de nouveaux composants ou protocoles peuvent être conçus, implémentés puis ajoutés dynamiquement à un système en cours d'opérationnalisation. Sur le schéma, les boîtes grisées correspondent aux informations qu'il faut fournir au système alors que les boîtes blanches sont générées automatiquement. Les quatre «moteurs» opérationnels (encerclés) fournissent chacun un support spécifique pour une étape importante de l'itération.

10.1.1 Traduction

Comme nous l'avons largement décrit précédemment, le langage Scope (étendu pour le support des rôles et protocoles) fournit l'ensemble des constructions linguistiques permettant de décrire les différents éléments constitutifs des applications Comet.

Ces sources en langage Scope sont alors traduits dans le langage Java car nous avons bien vu dans les chapitres précédents que tout ce qui n'était pas du ressort direct de notre plateforme était délégué à un environnement sous-jacent. Notre choix s'est porté pour diverses raisons sur l'environnement Java de Sun Microsystems. Les principales raisons de ce choix sont autant pragmatiques (bibliothèques vastes, ouvertures sur le monde de la distribution, environnement gratuit) qu'intéressées (Java est populaire)[1]. Cependant, il était important selon nous de proposer au moins une variante de notre approche dans un langage objet, pour marquer le fait que nous ne remettons aucunement en cause l'intérêt global de l'approche objet. Il nous semble important de pouvoir implanter simplement et relativement «naturellement» les concepts que nous proposons dans des environnements traditionnels; nous n'avons pas vocation à l'exotisme.

La phase de traduction est également source d'informations relatives à la distribution et aux possibilités de compositions et d'adaptations dynamique, sous la forme de fichiers de déploiement ou de sources d'informations dynamiques. Nous reviendrons sur ces constituants du moteur par la suite.

10.1.2 Compilation

Les définitions Scope, une fois traduites en Java, ne forment pas l'intégralité de ce qu'il est possible de fournir à la plateforme Comet pour déployer des «morceaux» d'applications. Le langage Java est également utilisé pour décrire certains aspects de plus bas niveau. En particulier, c'est en Java que sont implantés les différents modèles comportementaux régissant le fonctionnement canonique des composants. Mais il peut également s'agir de définitions de classes utilitaires divers : structures de données, algorithmes, etc.

Arrive ensuite la phase de compilation qui se déroule ici entièrement dans le cadre du langage Java (à partir de la version 1.2). Nous obtenons alors un certain nombre de fichiers binaires, en bytecode Java.

10.1.3 Noyau Comet

La compilation des définitions spécifiques à la plateforme Comet se réfère à une relativement vaste[2] bibliothèque qui sert également d'environnement exécutif.

Le bytecode généré doit ensuite être interprété par l'association de la machine virtuelle Java (JVM) et de l'environnement de support de l'interstitiel Comet que l'on doit fournir sur chaque machine du système distribué physique. C'est plus précisément ce *runtime* que nous nous proposons de décrire dans les sections suivantes.

10.1.4 Environnement de contrôle

La plateforme Comet comprend également, en complément du noyau «bas-niveau», une application graphique permettant le contrôle «système» de l'infrastructure opérationnelle du système.

Cette application de contrôle, de nom de code Halley (sic), sert également d'outil de déploiement graphique, comme le montre la figure 10.2 page suivante.

Parmi les nombreuses facilités offertes par cet outil, citons entre autre :

- Gestion d'une base distribuée de définitions de composants et de protocoles.
- Gestion de sites physiques utilisables pour le déploiement.
- Déploiement graphique et dynamique de composants.

1. En toute franchise, notre préférence va aux langages plus fonctionnels et plus ouverts, qui sont hélas également moins populaires. Il existe ainsi une version - d'ailleurs plus «propre» - du langage Scope pour divers environnements Scheme/Common Lisp.
2. Environ une centaine de classes pour le noyau «pur», sans les extensions synchrones par exemple.

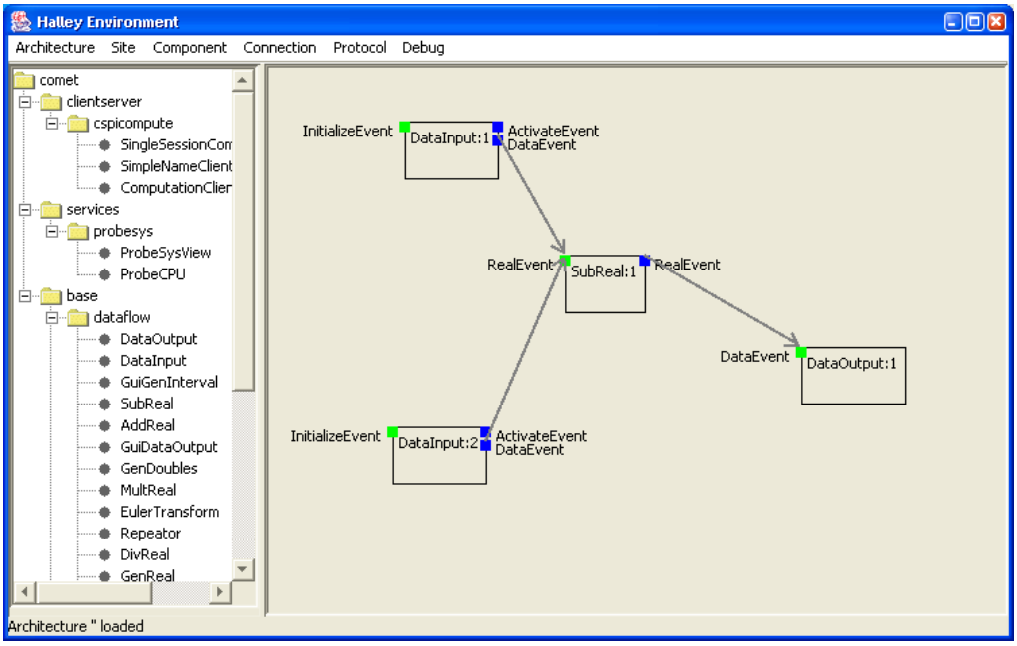


FIG. 10.2 – *Environnement de contrôle Halley*

- Manipulation graphique des systèmes (ajout/retrait de connexion).
- Application dynamique de protocoles.
- Récupération d'informations sur des composants déjà déployés.
- Sauvegarde d'un ensemble de modifications pouvant être rejouées ultérieurement (modèles de déploiement).
- Synchronisations contrôle/démons pour visualiser les modifications structurelles.
- Dévermineur distribué graphique.

Le dernier point, même s'il reste pour l'instant plutôt expérimental, est un aspect particulièrement intéressant de cet environnement. Il met notamment en œuvre une version étendue du protocole adaptatif de trace décrit au chapitre 8.

Il est important de signaler, enfin, que cet environnement Halley correspond à un protocole Comet «traditionnel» mais doté d'intéressantes capacitées d'interaction graphiques.

10.2 Infrastructure opérationnelle

La plateforme Comet a été conçue dès le départ comme un environnement hautement distribué, *sans aucun aspect centralisé*. L'infrastructure opérationnelle mise en œuvre est donc fortement liée à l'architecture physique du système.

10.2.1 Sites

Ceci nous conduit tout d'abord à définir la notion de *site*, constituant de base de l'environnement physique sur lequel repose le système Comet. Un site correspond, à ce niveau, à une machine «physique» au sens du protocole TCP/IP ; c'est-à-dire la réunion d'une adresse *Internet Protocol* et d'un démon *Internet Daemon* associé. Bien que simplificateur, nous considérerons dans nos schémas d'infrastructure opérationnelle qu'une machine physique au sens d'Internet correspond bien à une machine physique dans la «réalité».

10.2.2 Démons Comet

D'un point de vue global, un système Comet correspond à l'inter-relation de serveurs logiciels fonctionnant sur des sites généralement distincts mais éventuellement identiques. Un *démon Comet* identifie le logiciel de support local (pour un site donné) de l'environnement. Du point de vue système, un tel démon correspond à un *serveur TCP*, c'est-à-dire un couple (site, numéro de port). Chaque démon du système gère un ensemble de composants et de connexions locaux, ainsi que des «ponts» vers d'autres démons. Les architectures d'applications Comet étant transparentes au déploiement, une séparation «infranchissable» est posée entre leur manipulation logique (opérations Comet) et leur correspondance opérationnelle (échanges inter-démons de bas niveau). Par exemple, une même architecture logique peut être déployée physiquement selon de multiples configurations [3].

10.2.3 Identification

L'identification des composants est un problème relativement complexe dans l'environnement hautement distribué mis en œuvre par la plateforme Comet. Ainsi, plusieurs types d'identification sont possibles, à la fois pour les instances et les définitions de composants.

Comme nous l'avons vu dans le modèle structurel (cf. première partie), la séparation architecture logique/déploiement physique repose entre autre sur la mise en œuvre d'un système d'identification transparent à la localisation. Il s'agit de générer *localement* (i.e. sur le démon local), un identificateur *globalement* unique au niveau du système complet (tous les démons tournant ou non, dans une échelle de temps la plus large possible).

Nous proposons la forme suivante :

<classe>:*<hôte>*:*<port>*:*<ticks>*

où

- *<classe>* représente le nom de classe (au sens de Scope) du composant considéré,
- *<hôte>* correspond à l'adresse DNS (ou, à défaut, IP) qui a «créé» le composant,
- *<port>* correspond au numéro de port du même démon,
- *<ticks>* est un compteur 128 bits ne pouvant se reproduire (à l'échelle du temps «humain») deux fois sur le même site et donc unique pour tous les démons fonctionnant sur ce même site.

Nous pouvons montrer que ce choix satisfait «pratiquement» [4] les conditions formelles proposées pour l'identification des composants.

Dans le cadre de la *migration* (pour l'instant au stade expérimental), il est très important de signaler que l'identificateur unique de composant ne peut servir à localiser précisément un composant. En effet, les adresses et numéros de port référencés concernent le «lieu» de création du composant. Après migration, il est possible que ce composant soit localisé sur un démon différent.

Cependant, un protocole standard de nommage, fondé sur les principes exposés au chapitre 9 est proposé par l'environnement Comet. Ce protocole permet de donner un nom «logique» à une instance de composant. Cet identificateur (différent de l'identificateur physique) est unique pour un service de nom donné, ce qui ne signifie pas son unicité globale.

Le dernier type d'identification proposé par défaut concerne la «localisation» logique d'une définition de composant. Le nom de classe d'un composant est également unique au sens du système mais dénote un ensemble d'instances associées. Ce nom de classe est très important pour tout ce qui concerne la manipulation du code associé au composant (classes de composant, classes d'événement, classes et interfaces utilitaires, fichiers de déploiement, fichiers de ressources).

Encore une fois, cette identification logique est complétée par une localisation physique, indiquant où trouver le code associé à une définition de composant. Cette identification ne possède aucune caractéristique d'unicité. Le mécanisme de bas niveau proposé repose sur le mécanisme de gestion des paquetages du langage Java.

10.3 Architecture interne

Nous donnons ici quelques informations importantes concernant la mise en œuvre des concepts fondamentaux du langage Scope dans l'infrastructure Comet.

3. Exactement N^M configurations possibles pour un système à N démons et M composants.

4. Dans quelques milliards d'année, un identificateur redondant pourra être «découvert», mais même en cas d'extrême stabilité du système, le composant déjà identifié de la même façon aura probablement cessé d'exister depuis quelques autres milliards d'années.

10.3.1 Classes fondamentales

La stratégie d'implantation des concepts structurels de la plateforme Comet est fort simple : pour chacun de ces concepts, une classe Java est créée pour donner l'ensemble minimal de propriétés relatives à ce concept. Ainsi, nous trouvons seulement quatre classes que nous dirons fondamentales :

- La classe Component pour les composants. Cette dernière prend en charge tous les aspects relatifs à la structure des composants, comme la gestion du type ou encore l'identification.
- La classe Event pour les événements dont toutes les définitions événementielles héritent. Il s'agit entre autre de mettre en œuvre le protocole de sérialisation automatique associé aux événements Comet.
- La classe Protocol pour les protocoles. Cette dernière fournit l'interface objet de notre langage d'interactions distribuées; c'est-à-dire la syntaxe des commandes telles que upload, instantiate, connect, etc.
- La classe Role qui prend en charge les définitions de rôle, en fournissant notamment le protocole de vérification dynamique pour le contrôle fin des assignations.

Ces classes sont à la fois parties et fondements de l'implémentation canonique du système Comet et dénommé *paquetage Core*. Cette articulation est mise en relief sur la figure 10.3.

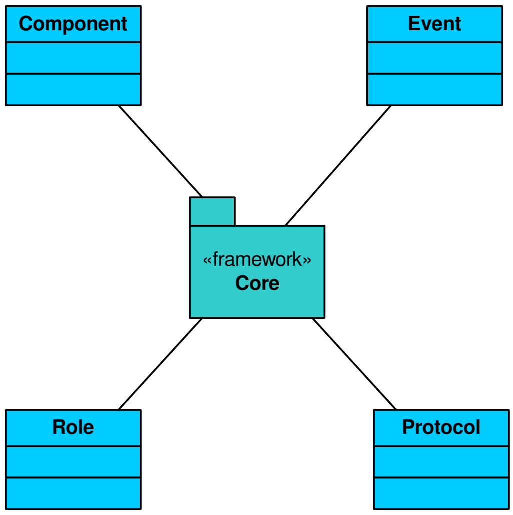


FIG. 10.3 – *Classes fondamentales du noyau Comet*

10.3.2 Niveau méta-comportement et spécialisations

Si au niveau formel, l'adjonction d'un comportement à une structure componentielle s'effectuait par raffinement, la variante implantée de ce modèle repose également sur des principes de génie logiciel traditionnels dans le monde objet : agrégation, délégation et héritage.

Nous pouvons voir sur la figure 10.4 page suivante les principales classes mises en jeu par l'implantation d'un modèle de comportement associé à la classe mère Component qui représente la structure immuable des composants. La classe DefaultComponent hérite de cette classe mère et fournit le comportement par défaut des composants sous forme de deux chaînes de délégations composées des :

- classes définies pour l'implantation des méta-objets nécessaires à la réception et au traitement des événements, et des
- classes des méta-objets mettant en œuvre les émissions événementielles.

Il est intéressant de remarquer que nous ne cassons pas ici notre règle de conception :

Un concept Comet = une classe Java

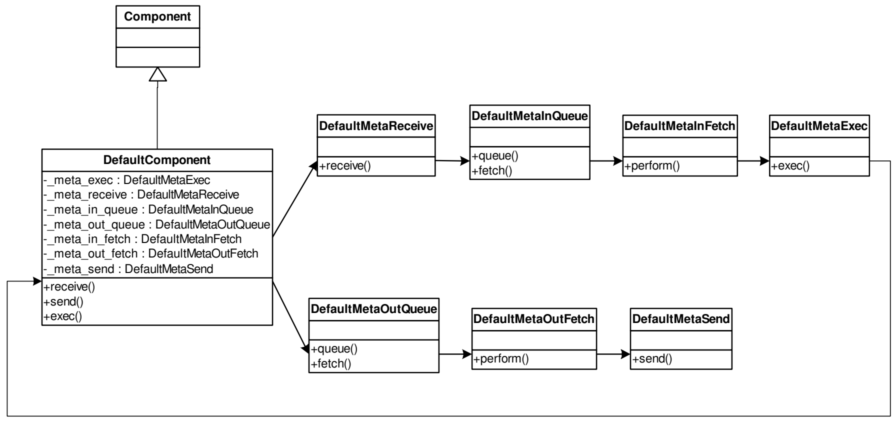


FIG. 10.4 – *Noyau Comet : raffinement comportemental*

Cette règle permet d'associer à chaque schéma du modèle formel une classe (ou plutôt une hiérarchie de classes) qui en implémente les propriétés. On pourrait ainsi envisager - à plus long terme - de construire les classes de base de l'implémentation comme raffinements des schémas formels.

Nous avons vu que l'intérêt principal de la décomposition du comportement des composants en méta-objets spécifiques concernait les nombreuses possibilités de spécialisations de ce même comportement. Nous avons également décrit certaines de ces spécialisations à un niveau formel; et notre implantation fournit bien sûr une version opérationnelle de ces modèles.

Mais puisqu'il existe une sorte d'isomorphisme entre le modèle formel et son implantation (pourtant non directement dérivée), il est possible d'approcher la problématique de spécialisation d'un point de vue plus pragmatique. Pour illustrer la relative facilité de spécialisation de ces modèles comportementaux implémentés, nous allons construire une extension simple mais utile du modèle canonique.

La philosophie sous-jacente de notre noyau technologique met en avant la séparation des préoccupations. Ainsi, au niveau fonctionnel, c'est-à-dire dans notre cas au niveau des fonctionnalités de réaction aux événements reçus, nous travaillons à un niveau local se limitant aux frontières du composant concerné. En particulier, répartition et concurrence ne sont pas (ou plutôt, ne devraient pas) être directement gérées à ce niveau mais déléguée à une interaction avec le niveau méta-comportement.

Le modèle que nous nous proposons de construire permet de faire tourner des tâches de fond nommées *co-routines* pour des raisons historiques mais qui, dans le cadre de Java, correspondent plutôt à des processus légers. Chacune de ces co-routines est exécutée concurremment à l'activité «normale» du composant, elle-même gérée par le méta-objet DefaultMetaExec. Une solution simple à ce problème de co-routinage consiste à raffiner le moteur d'exécution par simple héritage.

La définition (abrégée) du méta-objet CRMetaExec est la suivante :

```
class CRMetaExec extends DefaultMetaExec {
  // Construction
  CRMetaExec() {
    super();
    _coroutines = new ArrayList();
  }
  // Creation de co-routines cycliques
  CoRoutine newCyclicCoRoutine(String method_name, long interval) {
    CoRoutine cr = new CoRoutine(method_name, interval);
    _coroutines.add(cr);
  }
  // Autres types de coroutines
  ...

  ArrayList _coroutines;
}
```

La seule nouveauté introduite par la définition précédente, qui hérite des propriétés de DefaultMetaExec, réside dans les méthodes de créations comme newCyclicCoRoutine. Cette dernière permet de définir une co-routine exécutée à intervalle régulier; il existe bien sûr d'autres types de co-routines (exécution unique, exécution bornée, etc.). L'ensemble de ces méthodes fournit le langage permettant au niveau base d'interagir avec le niveau méta-comportement.

La classe CoRoutine implémente chaque coroutine qui correspond alors à une activité exécutant la fonctionnalité passée en paramètre (method_name). La définition de cette classe est triviale puisqu'il s'agit d'un raffinement très simple de la classe standard Thread de Java.

Finalement, pour mettre en œuvre ce nouveau méta-objet dans le cadre d'une définition de composant, nous pouvons simplement raffiner la classe DefaultComponent de la manière suivante :

```
abstract class CRComponent extends DefaultComponent {
  // Spécialisation du meta-exec
  void createMetaExec() {
    _meta_exec = new CRMetaExec();
  }
}
```

Nous découvrons ici le caractère assez flexible de notre boîte à outils de spécialisation du niveau méta-comportement. Dans la phase de construction de l'objet représentant le composant dans l'environnement d'exécution (instance de la classe définissant le composant, ici une sous-classe de CRComponent), le crochet standard createMetaExec est redéfini pour créer une instance spécialisée (CRMetaExec en lieu et place de DefaultMetaExec) de l'étage d'exécution.

L'environnement Comet utilise ce processus de spécialisation pour proposer un certain nombre de modèles comportementaux raffinés et notamment le support des sessions multiples définies au chapitre 7.

10.4 Protocoles inter-démons

Nous détaillons dans cette section l'opérationnalisation des différentes primitives distribuées supportées par la plateforme Comet. Le modèle proposé étant transparent à la localisation, tout requête peut être émise d'un site vers tout autre site enregistré du système.

D'un point de vue système, ces primitives correspondent donc à des communications point-à-point entre démons Comet dont la localisation physique est variable. Pour l'instant, l'amorçage complet du système n'a pas été réalisé, de sorte que le protocole de bas niveau pour la communication entre deux démons n'est pas un protocole au sens Comet du terme. Cependant, cet amorçage est en cours de réalisation; de sorte que nous ne présentons dans ce qui suit des échanges qui pourront, à l'avenir, être le fait de composants Comet «standard».

Concernant la communication entre démons, deux cas de figure peuvent se présenter :

1. présence de deux démons distincts : la requête est distribuée, c'est le cas le plus fréquent.
2. cas, plus simple, de deux démons identiques : la requête est locale.

Plutôt que de reposer sur une catégorie générique de protocoles client/serveur, nous introduisons par souci d'optimisation un «pôle» intermédiaire permettant de prendre en charge, le cas échéant, les requêtes locales et les sections communes des requêtes distribuées. Le fait d'isoler ces requêtes locales permet d'envisager un amorçage du noyau dans un avenir assez proche puisqu'il s'agit des seules interactions non-distribuées pour l'ensemble de la plateforme Comet.

De façon générale, le fil conducteur de cette section repose sur les nombreuses optimisations - génériques et spécifiques - mises en œuvre pour augmenter les performances sur tous les aspects système liés à notre implantation du modèle Comet.

10.4.1 Protocole générique

Le protocole générique proposé est mis en œuvre à travers deux classes système et trois classes spécifiques au protocole. Ce protocole est proposé, sous forme de diagramme de séquence UML, en figure 10.5 page suivante.

La requête, du côté du démon client, est déclenchée par un appel à une méthode de la classe Protocol qui fournit le «langage» de base des protocoles noyau. Cette invocation conduit à l'exécution d'une méthode de

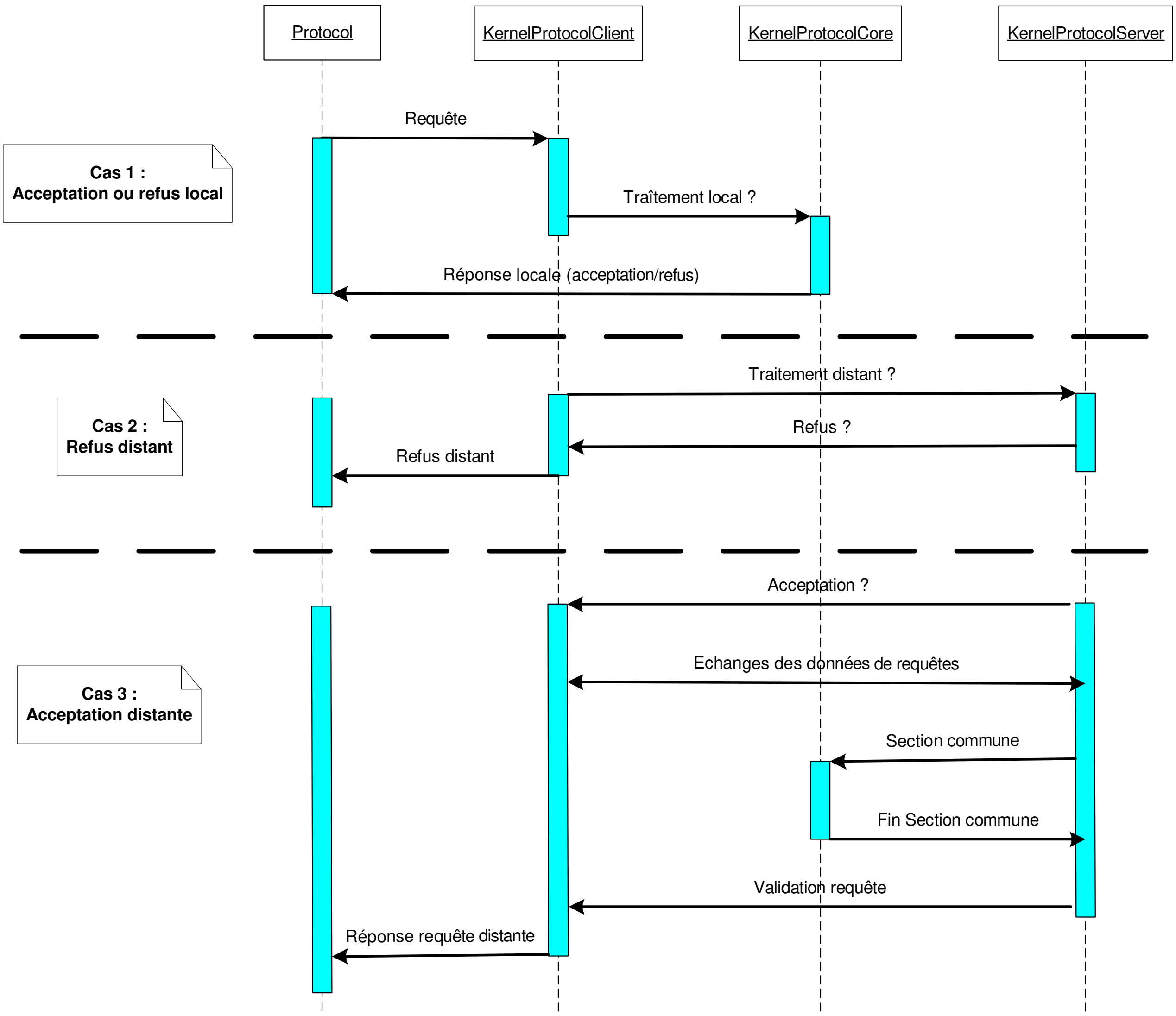


FIG. 10.5 – *Protocole générique de communication inter-démon*

bas-niveau, membre d'une sous-classe de KernelProtocolClient qui regroupe les parties clientes des protocoles d'interactions.

Cette méthode effectue tout d'abord un test de localité et délègue la continuation de la requête à une sous-classe de KernelProtocolCore si le test de localité est validé (cas 1).

En cas de requête distribuée (cas 2 et 3), une connexion est établie avec le démon serveur [5]. Chaque requête distante est identifiée de manière unique à ce niveau, ce qui conduit à une première négociation [6], notamment la classe spécifique de traitement côté serveur, sous-classe de KernelProtocolServer.

En cas de refus pour la requête (fin du cas 2), la connexion est rompue et le «client» est averti du problème (défaut de communication, requête impossible, etc.).

Si la requête est acceptée (début du cas 2 et cas 3) par le démon serveur, la sous-classe de KernelProtocolClient effectue les traitements spécifiques de la partie client du protocole en interaction étroite avec la «face» serveur.

Toute section commune au traitement local et distant est prise en charge par le troisième intervenant, une sous-classe de KernelProtocolCore spécifique au protocole considéré.

Finalement, après réponse et validation de la requête, la connexion est rompue [7] après satisfaction de la requête.

Le diagramme de classes proposé en figure 10.6 montre les relations entre les classes «mères» des protocoles inter-démons et leur spécialisations (ici, pour les requêtes de téléchargement et d'émissions de classes).

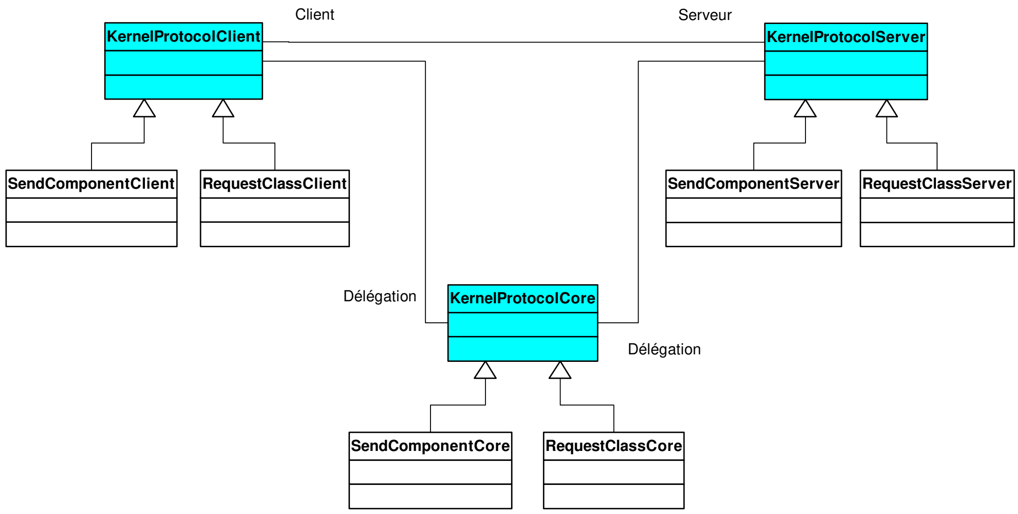


FIG. 10.6 – *Classes de protocole (téléchargement de composant, requêtes de classes*

Nous voyons sur ce diagramme que le troisième agent KernelProtocolCore agit en tant que délégué pour les parties client et serveur (les classes grisées sont les classes mères de l'architecture).

10.4.2 Téléchargement de composant

Lors du lancement d'un nouveau démon sur un site donné, le seul code disponible concerne le noyau du système et, optionnellement, les classes des composants standard (pour le client/serveur, la migration, etc.). Ainsi, lors du déploiement d'une application utilisateur, les codes des classes des événements et composants associés doivent être répartis sur les différents démons concernés par le déploiement.

La requête de téléchargement est déclenchée par un appel à la méthode upload() de la classe Protocol. Les classes systèmes mises en jeu par le protocole sont SendComponentClient (partie client), SendComponentServer (partie serveur) et SendComponentCore (partie partagée).

Ce protocole, instance du protocole d'interaction générique défini précédemment, est décrit - pour sa partie spécifique uniquement - en figure 10.7 page suivante.

5. Cette connexion est établie au niveau de la classe DaemonRequest qui n'est pas figurée sur le schéma de protocole.

6. Pour l'instant, les tests effectués concernent principalement les aspects réseaux - un échange est tenté - mais il s'agit avant tout d'un point d'entrée pour d'éventuelles négociations liées à la sécurité : identification du démon source, authentification, échanges de clés publiques, etc.

7. En fait, les connexions sont placées dans un cache et ne sont rompues effectivement qu'après dépassement d'un délai fixé.

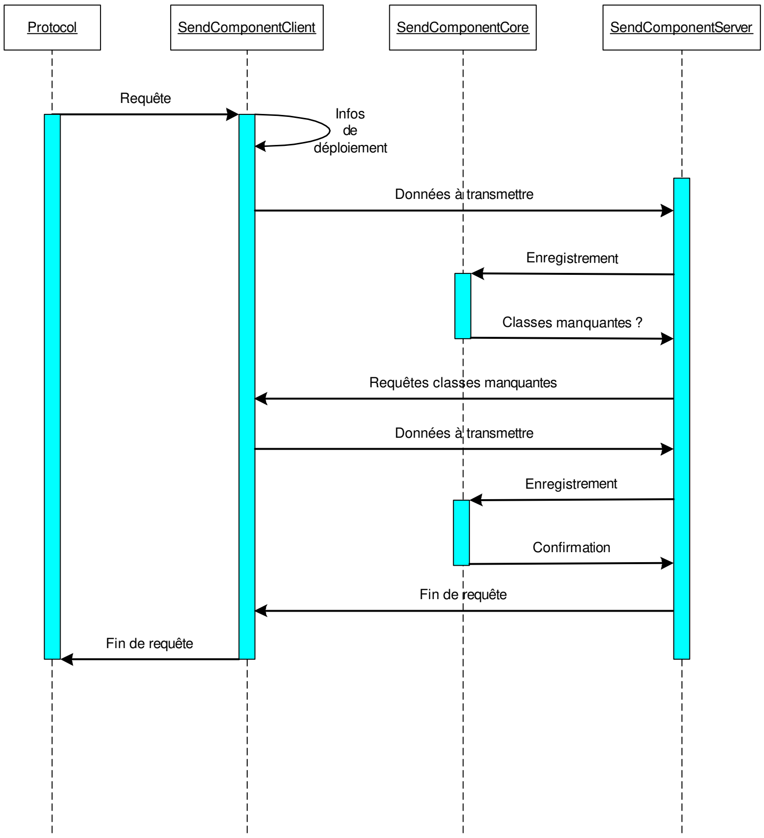


FIG. 10.7 – *Protocole spécifique de téléchargement de composant*

Si la requête est locale, il ne reste plus rien à faire bien sûr puisque le composant à émettre est déjà disponible localement. Dans le cadre d'une requête distribuée [8], en revanche, plusieurs échanges sont nécessaires.

Tout d'abord, le *fichier de déploiement* associé au composant est analysé, ce qui permet de dresser la liste des classes à transmettre. Rappelons que pour un composant donné, de nombreuses classes peuvent être mises en jeu : classes d'événements associés, classes utilitaires, interfaces *receivers*, *fetchers* et *senders*, etc. Il est également possible d'associer des ressources (comme des fichiers, des images, etc.) au déploiement d'un composant donné. Toutes ces informations sont contenues dans le fichier de déploiement (suffixe .def) du composant.

Considérons à nouveau l'exemple du composant Coordinator décrit au chapitre 9, page 192. Ce composant est associé au fichier de déploiement suivant [9] :

8. Un démon désire envoyer le code d'un composant à un autre démon.
9. Rappelons que ce fichier de déploiement est généré automatiquement par le traducteur Scope pour la plateforme Comet.

```
COMPONENT Coordinator {
  RECEIVE {
    CoordEvent
    TransactionEvent
  }

  SEND {
    CoordEvent
    TransactionEvent
  }

PRE_CLASS {
    SyncBehavior
    SyncMetaInFetch
    SyncMetaInQueue
    SyncMetaReceive
  }
}
```

Tout d'abord, les classes et interfaces associées aux événements reçus (section RECEIVE) et émis (section SEND) sont sélectionnées pour l'envoi. Les classes indiquées dans la section PRE_CLASS sont également sélectionnées pour un envoi préliminaire. Ici, nous envoyons les classes non-standard liées à la spécialisation du méta-comportement pour les interactions synchrones. Si nécessaire, les classes indiquées dans la section POST_CLASS sont également envoyées en fin de déploiement et les éventuels fichiers (section RESOURCE_FILES[10].) et autres ressources (section RESOURCES) associés sont également «empaquetés» pour émission.

Dans notre exemple, la liste des classes et ressources émises est la suivante :

- Classes d'événements : CoordEvent.class, TransactionEvent.class
- Interfaces associées : CoordEventReceiver.class, CoordEventSender.class, CoordEventFetcher.class, TransactionEventReceiver, TransactionEventSender.class, TransactionEventFetcher.class
  Classe(s) de définition du composant : Coordinator.class
- Classes utilitaires pré-chargées : SyncBehavior.class, SyncMetaInFetch.class, SyncMetaInQueue.class, SyncMetaReceive.class
- Fichiers et autres ressources : fichier de déploiement Coordinator.def

Nous pouvons voir sur cet exemple que la recherche, le chargement et l'émission de toutes ces informations peut se révéler assez couteux. Diverses optimisations liées à l'emploi de *caches* sont proposées pour remédier à ce «problème». Puisque seule la partie client du protocole «connaît» le composant à émettre, c'est elle qui prend en charge ces optimisations. Du côté serveur, des caches persistants sont utilisés pour conserver les définitions, au moins jusqu'aux premières instantiations[11]. L'encart page 208 décrit les deux niveaux de caches mis en œuvre dans le noyau.

Si une classe est signalée comme manquante, un système de récupération (cf. requête de bas-niveau RequestClass ci-dessous) est mis en œuvre. Celui-ci demande la classe manquante, qui peut provenir d'un fichier de déploiement incomplet[12]. L'algorithme côté client du protocole de téléchargement est résumé en figure 10.8 page suivante.

Du côté du serveur, les informations reçues sur le réseau concernent le minimum d'informations nécessaires pour pouvoir «honorer» les futures requêtes d'instanciation du composant concerné (cf. section suivante). Un système de chargement dynamique de classes (DaemonClassLoader) gère les réceptions de classes et interfaces (cf. encart page 209).

Finalement, si la requête se produit sans encombre[13], le site «récepteur» pourra instancier le composant ou réémettre son code vers d'autres démons.

10. Le fichier de définition est automatiquement ajouté aux fichiers de ressources et donc expédié avec le code du composant

11. Le ramasseur de miettes de Java (JVM de Sun) a en effet la fâcheuse tendance à collecter les classes non instanciées, «fonctionnalité» que nous avons mis un certain temps à comprendre et qui causait de nombreux «bogues» dans l'implantation de notre système.

12. Ainsi, il n'est pas erronné de fournir un fichier de déploiement vide au système, mais les échanges entre les démons émetteur et récepteur sont alors très peu performants puisque chaque classe émise correspondra à une négociation entre démons.

13. Dans le cas contraire, toutes les opérations sont annulées et tout se passe comme si rien n'avait été émis. Une exception est de plus déclenchée côté client.

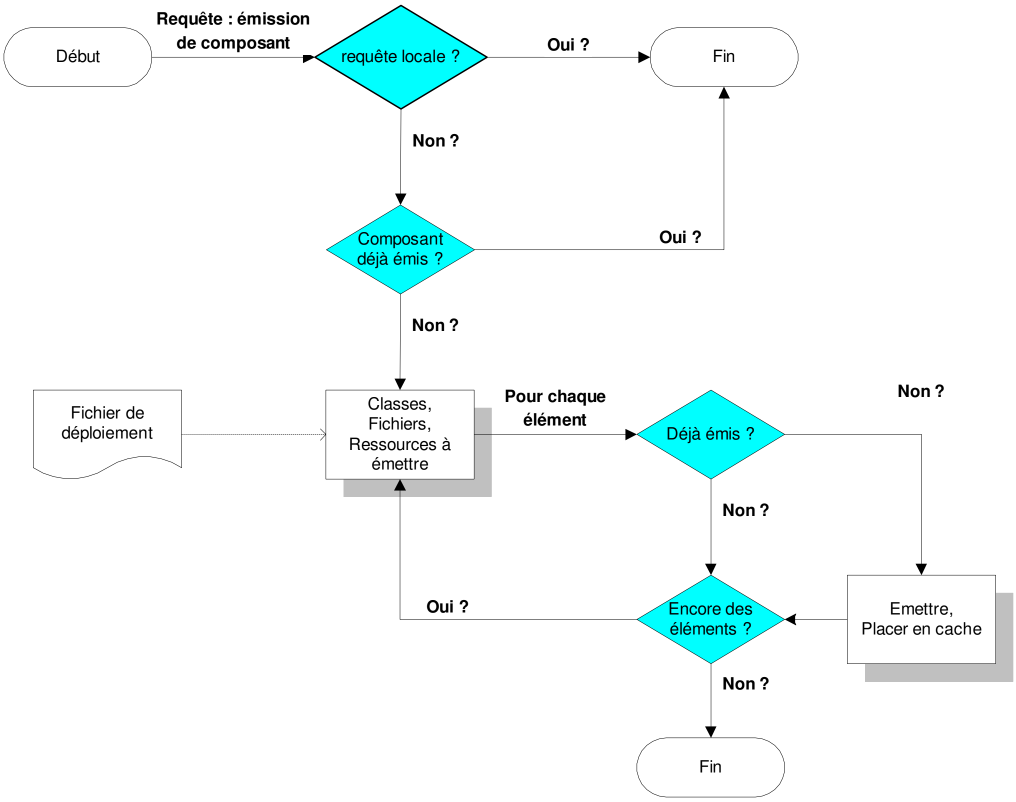


FIG. 10.8 – *Algorithme d'émission de composant : gestion de caches à deux niveaux*

Systèmes de cache dans Comet : côté Client

Trois systèmes de cache d'informations sont mis en œuvre pour supporter la face client des protocoles inter-démon dans le cadre des requêtes d'émission de code et de données :

1. Un cache de niveau 1 enregistre les émissions de composants «entiers» sur un démon donné. Ce cache est remis à jour à intervalles très longs (de l'ordre de l'heure, sauf requête exceptionnelle).
2. Un cache de niveau 2 enregistre individuellement les classes et fichiers expédiés sur un site donné. Ce cache de granularité plus fine est mis à jour plus régulièrement (ainsi, les nouvelles versions de classes peuvent être émises).
3. Deux systèmes de cache sont également mis en œuvre pour limiter au maximum les accès disques. Ainsi, les définitions de composants et événements sont maintenues en mémoire centrale pendant une durée assez longue.

Les paramètres de ces caches (sous-classes de VolatileCache se regroupent en deux catégories :

1. Le délai de renouvellement des informations du cache (remise à jour, effacement).
2. La taille maximale du cache.

Caches côté Serveur

Les définitions de classes, fichiers et ressources reçus sont placées dans des bases de données associées à des caches dédiés et *persistants*. Lorsque la taille maximale de ces caches est dépassée, une copie de la ressource «la plus ancienne» est effectuée sur disque. Les informations contenues dans ces «entrepôts de données» sont effacées à la demande, lorsque toutes les instances locales d'un composant donné ont été détruites. Deux options sont alors disponibles : récupération manuelle ou automatique, selon deux classes opérationnelles de démons (propriété système).

Chargement dynamique de classes

Les programmes écrits dans le langage Java sont compilés en un bytecode dédié. L'exécution de ce code cible portable est supportée par la machine virtuelle java (ou JVM) qui offre l'ensemble des fonctionnalités de l'environnement proposé par la société SUN Microsystems.

Une des fonctionnalités les plus avancées de cette environnement concerne le chargement de classe pendant l'exécution des applications. En réifiant au niveau «classe» un code compilé lui-même portable, le chargement et le lien dynamique de code n'est plus alors du domaine du système d'exploitation.

Le chargement dynamique de classe est pris en charge en Java par une descendante de la classe abstraite ClassLoader. Du point de vue de ce chargeur/lieur, une référence de classe à charger correspond tout d'abord à la donnée d'un nom («chemin» complet de la classe) et d'une définition (bytecode) associée. Cette définition peut être donnée directement sous la forme d'un tableau d'octets ou cherchée dans l'environnement local (classes se trouvant dans le chemin de recherche ou classes système). Le dernier type d'information nécessaire au chargement de la classe concerne l'ensemble de ses dépendances :

- la super-classe qu'elle étend,
- les interfaces implémentée,
- les classes et interfaces utilisées dans l'implémentation (attributs, paramètres de méthodes, variables locales etc.), et
- récursivement, toutes les classes et interfaces dont dépendent les «dépendants» directs de la classe à charger.

Notons également la gestion relativement poussée des aspects liés à la sécurité de l'opération de chargement.

Dans le cadre de Comet, le chargement des classes se fait sous la responsabilité de la classe DaemonClassLoader. La principale «nouveauté» de cette variante concerne le chargement de classes dont la définition provient du réseau, via les requêtes de protocoles inter-démons Comet.

10.4.3 Connexions

Le protocole probablement le plus complexe mis en œuvre au niveau de base du système concerne l'établissement des connexions.

Catégories de canaux

Le pendant opérationnel des connexions «logiques» mises en œuvre par la plateforme Comet se nomment des *canaux*, supports des communications effectives. Après établissement, chaque connexion sera affectée à un canal spécifique.

En comparaison du modèle formel, nous apportons au niveau implémentation un raffinement «système» dont le but est d'optimiser les communications entre composants. Ce raffinement consiste en l'introduction de plusieurs catégories de canaux :

- les *canaux publics* : partagés par plusieurs connexions, les événements «transportés» sur ces canaux sont donc multiplexés.
- les *canaux privés* : deux composants peuvent être reliés directement, pour un type effectif de connexion donné, par un lien «physique» (au sens du protocole TCP).

Le premier cas de figure, employé par défaut, est idéal pour le support de transport d'événements peu fréquents (la majorité des cas). Il existe exactement un canal «physique» public entre deux démons, à partir du moment ou au moins une connexion entre les deux démons a été demandée[14].

Les canaux privés, pour leur part, sont gérés directement par les composants concernés. Ils sont utilisés pour la transmission d'événements en mode rafale (*burst mode*) ou, plus généralement, pour les communications dont le critère de performance l'emporte sur les ressources nécessaires à leur support.

14. Au démarrage du système, il n'est donc pas utile de relier les différents démons, l'environnement lui-même est dynamique dans notre approche.

Par abus de langage, nous parlerons de *connexions publiques* et de *connexions privées* pour désigner des connexions établies pour des canaux respectivement publics et privés.

Ces catégories de canaux/connexions se subdivisent en deux sous-ensembles : canaux de type distant ou local. Lors d'une requête de connexion, le premier test effectué concerne la localisation relative des composants source et destination. S'ils se trouvent sur le même démon, alors la connexion est établie localement et la couche réseau n'est pas employée. Envoyer un événement localement revient donc, en gros, à un appel de méthode dans une activité[15].

La figure 10.9 présente une configuration de démons/composants/connexions composée de connexions supportées par des canaux publics/privés et locaux/distants. La configuration considérée comporte trois sites (machines «physiques») et quatre démons Comet. Les différents types de canaux possibles sont représentés sur ce schéma. Bien sûr, ces variations sont purement opérationnelles ; d'un point de vue logique, toutes les connexions disposent des mêmes propriétés, au sens du modèle formel qui les caractérise.

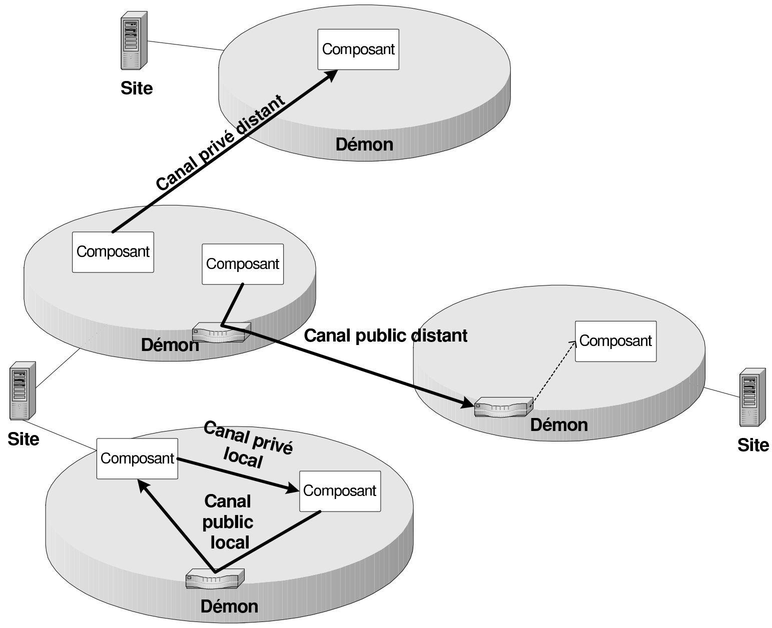


Fig. 10.9 – *Catégories de canaux de communication*

Notons également que l'architecture proposée pour l'implantation du protocole de connexion - central dans notre approche - fournit une base assez générale pour la spécification de paramètres liés à la qualité de service des ressources systèmes allouées pour la communication entre composants. Des négociations point-à-point de natures variées peuvent être menées comme raffinement du protocole de résolution des contraintes de typage. Ces négociations peuvent concerner l'allocation de ressources système mais également servir de base à des protocoles de sécurité (authentification, signature, etc.)[16].

Protocole de connexion

Quels que soient les types de canaux de communications employés, la structure générale du protocole d'établissement de connexion correspond au diagramme 10.10 page suivante.

Ce diagramme présente trois entités : le démon effectuant la requête de connexion (au milieu), ainsi que les démons source (à gauche) et destination (à droite). Le test de localité, effectué en premier lieu, permet de définir si ces entités sont identiques (test validé) ou différentes (test invalidé). Ceci influe sur la nature de la requête : distribuée ou locale, pour la source et/ou pour la destination[17].

Dans un premier temps, la requête de connexion est transmise au composant source (via une requête spécifique) pour inférer le ou les types sortants pouvant être associés à la connexion demandée. Une fois cette inférence

15. Puisque l'émission d'un événement est de nature asynchrone, une simple invocation de méthode n'est pas suffisante pour opérationnaliser ce mécanisme localement.

16. Pour passer à un niveau «production», des services «minimaux» de gestion de sécurité devraient être implantés. Le prototype actuel est, de fait, relativement ouvert.

17. Bien que d'apparence triviale, le critère de localité conduit à 16 configurations différentes.

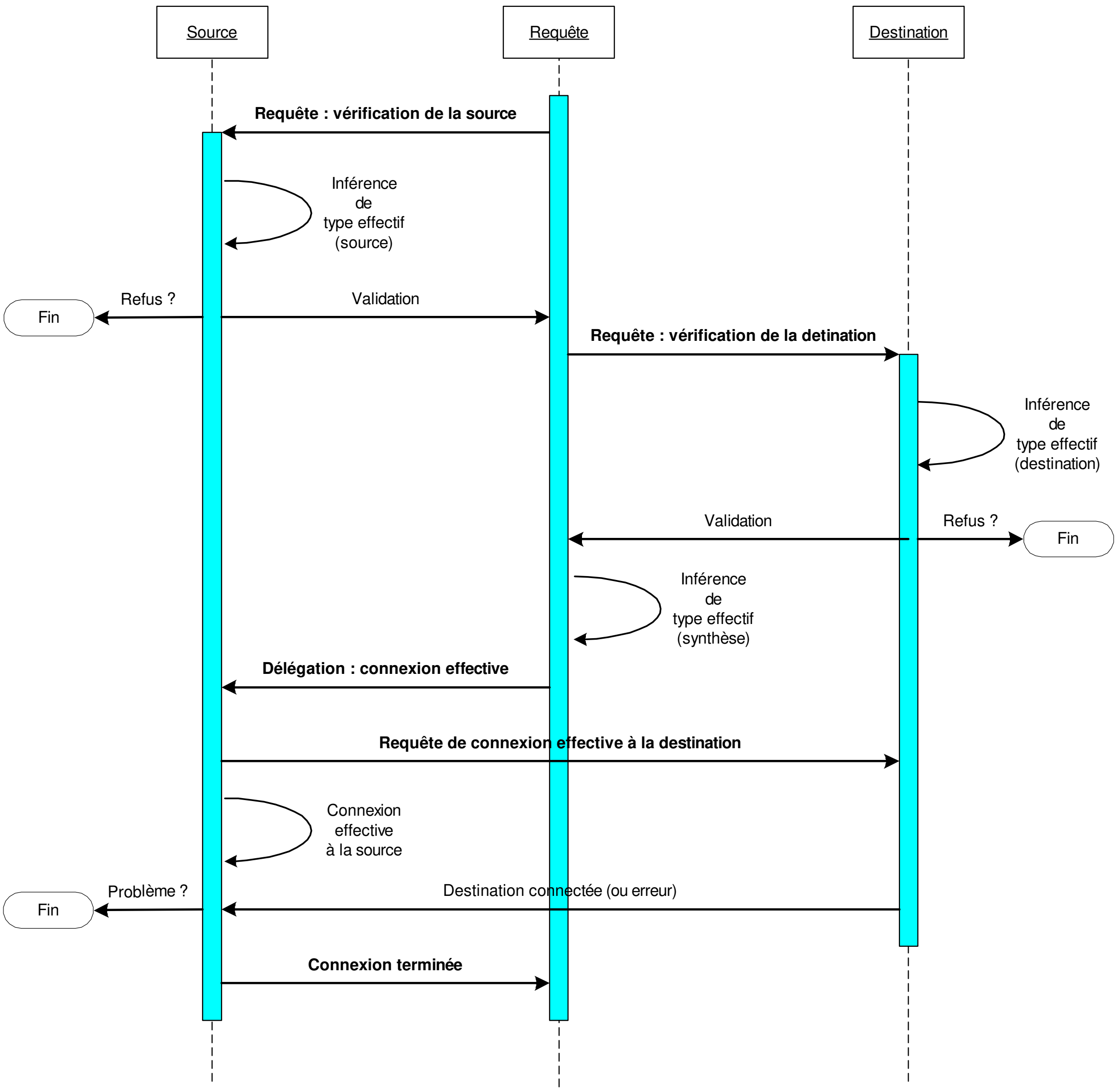


FIG. 10.10 – *Protocole d'établissement de connexions opérationnelles*

effectuée, la requête est validée ou refusée côté source. Le même traitement se produit côté destination. Rappelons que le choix des types entrant à la destination est dépendant des types sortant choisis initialement; ceci explique la sérialisation temporelle de ces deux requêtes. Finalement, le type effectif est synthétisé au niveau de l'émetteur de la requête (au centre sur notre diagramme).

A partir de cet état du protocole (contraintes de typage satisfaites), le contrôle de la requête est délégué au démon source. Ce dernier effectue les traitements qui lui sont spécifiques pour opérationnaliser la connexion. Ces traitements varient en fonction de la caractéristique publique ou privée du canal de connexion (cf. section suivante). La source émet de plus une requête de connexion effective à la destination qui réalise alors - et en parallèle - des traitements spécifiques similaires.

Finalement, le démon source «répond» à l'émetteur de la requête selon la nature de sa terminaison : correcte ou interrompue.

Les points sensibles qu'il nous reste à éclaircir concernent les phases de connexion effective pour les composants source et destination. Nous nous intéressons plus précisément aux canaux distants[18] de catégorie publique ou privée.

Etablissement effectif des connexions

La phase préliminaire d'établissement effectif de connexion concerne l'allocation de la ressource système nécessaire à la connexion : le canal de communication. Dans le cadre d'une allocation publique, s'il n'existe pas de canal «physique» reliant les démons source et destination concernés, un nouveau canal public doit être alloué pour ce couple au niveau des démons. Dans le cadre d'une liaison privée, le canal est géré (et donc créé) par le composant destination. Dans les deux cas, la source ne se préoccupe pas directement de l'allocation (si ce n'est pour la négociation de la qualité de service associée).

Le type effectif est communiqué au composant source pour réaliser l'association entre le type d'événement considéré et le canal de communication concerné. Cette relation est conservée dans le méta-objet MetaSend associé.

En parallèle, le composant destination enregistre - selon un procédé identique - les associations entre, d'une part, le type effectif de connexion et le canal de communication utilisé pour l'émission (qu'il alloue si nécessaire) et, d'autre part, la «meilleure» fonctionnalité (bloc when) liée au type d'événement concerné.

10.4.4 Autres protocoles

En supplément des protocoles - de nature relativement complexe - étudiés ci-dessus, d'autres interactions de moindre ampleur peuvent être initiées entre deux démons Comet.

Instanciation de composants

Une fois téléchargé sur un démon donné, un composant peut être instancié par l'intermédiaire d'une requête adaptée. Bien sûr, la requête d'instanciation peut, comme nous l'avons vu, être locale comme de coutume dans les environnements standard ou provenir de n'importe quel démon/site Comet en cours de fonctionnement.

La requête d'instanciation est prise en charge par une méthode de la classe InstantiateComponentClient, en communication avec le site distant et la classe InstantiateComponentServer. La seule information circulant sur le réseau concerne le nom complet de la classe (nommage Java) à l'aller et l'identificateur du composant créé en retour. La classe InstantiateComponentCore prend en charge l'instanciation effective (en fait, un simple new sur la classe du composant), qu'elle soit locale ou distante.

Requêtes de transferts de classes

Dans le cadre de l'émission d'un événement ou de l'assignation dynamique d'un rôle, des instances d'éléments précédemment téléchargés sont créées. Cette demande d'instanciation locale peut échouer en raison de l'absence d'une classe nécessaire.

Nous avons également vu que durant le téléchargement de classe, l'ensemble des classes associées devaient être disponibles du côté serveur en fin de requête.

18. Pour les canaux locaux, il s'agit du même protocole mais en version simplifiée, sans allocation de ressource.

Un protocole de «récupération» est tenté à chaque fois qu'une classe est signalée comme manquante par le système. Cette récupération peut être initiée côté client (requête d'émission de classe) ou côté serveur (requête de réception de classe).

Assignations de rôles

Les assignations de rôles correspondent, en gros, à une version «simultanée» des protocoles de téléchargement et d'instanciation à distance. Dans un premier temps, le code nécessaire à la création d'une instance du rôle doit être téléchargée sur le démon qui gère le composant sur lequel sera assigné le rôle. Une requête d'instanciation est ensuite réalisé puis le rôle est ajouté, selon le procédé décrit au chapitre 8, au sein du composant, puis activé lorsque toutes les étapes de vérifications ont été «franchies» avec succès. Rappelons que ces vérifications se font selon deux points de vue :

1. point de vue structurel : le type du rôle doit correspondre à celui du composant auquel il est assigné (filtres entrant et crochets du rôles en accord avec le type entrant du composant, filtres sortant du rôle en accord avec le type sortant du composant, etc.).
2. point de vue comportemental :
   - Première phase : l'assignation est vérifiée par le composant, qui peut demander une authentification ou placer le rôle dans un «bac à sable» (*sandbox*), etc.
   - Deuxième phase : le rôle peut demander des informations fines (méthodes,attributs,modèle de comportement employé,etc.) sur le composant qui doit le «recevoir».

Ces vérifications critiques, nous l'avons déjà dit, sont les deux points d'entrée fondamentaux du mécanisme d'assignation dynamique de rôles intrusifs en termes opérationnels.

Requêtes d'information

De nature fortement introspective, le noyau du système propose un protocole de requête permettant de s'informer sur le fonctionnement d'un démon Comet en particulier. Ces requêtes permettent de répondre aux questions du genre :

- Quels sont les composants actuellement instanciés sur le démon ?
- Comment sont reliés ces composants entre eux et avec d'autres composants situés sur d'autres démons ?
- Quelles classes sont disponibles dans le système de cache ?
- Quels rôles sont assignés à tel ou tel composant ?
- Quelle est la charge mémoire ou CPU du noyau ?
- etc.

Ce protocole est actuellement en voie de «Cometisation», c'est-à-dire que des composants Comet spécialisés sont en cours de conception pour proposer ces services, avant de réaliser l'amorçage complet du système dans un avenir proche.

Il existe beaucoup d'autres détails intéressants concernant l'implantation du noyau de la plateforme Comet. Par exemple, les émissions d'événement sont optimisées pour éviter toute recopie inutile d'événement. En cas de diffusion multi-point vers des composants se trouvant sur un même démon, alors une seule copie de l'événement est émise et c'est le démon *destinataire* qui réalise la diffusion proprement dite.

L'existence du modèle formel sous-jacent de l'implantation décrite ici, malgré leur relative indépendance (puisque la seconde ne raffine qu'informellement le premier), permet de poser clairement la frontière entre les aspects qui doivent être «figés», sous peine de s'écarter de leur sémantique formelle, de ceux qui peuvent être ouverts comme par exemple le déploiement du code des applications ou encore la séparations en versions publiques et privées des canaux de communications «physiques». De notre point de vue, les nombreux schémas Z que nous avons établis, ainsi que leur description informelle, représentent en quelque sorte la documentation «optimale» de notre plateforme interstitielle (à comparer aux spécifications OMG, Sun ou Microsoft...).

« Un langage est universel s'il a la puissance d'une machine de Turing, ce qui, compte tenu de la rudimentarité de celle-ci (...) n'est pas difficile à obtenir; il est probablement plus délicat de concevoir un langage utile qui ne soit pas universel »

Christian Queinnec

Conclusion

Nous proposons dans cette thèse une refondation des systèmes interstitiels sous l'angle d'un modèle formel qui la rend assez précise et relativement indépendante du contexte opérationnel sous-jacent. Ce versant formel de notre travail ne prétend pas à la complétude mais fournit simplement une explication du sens, selon nous très précis, des concepts introduits.

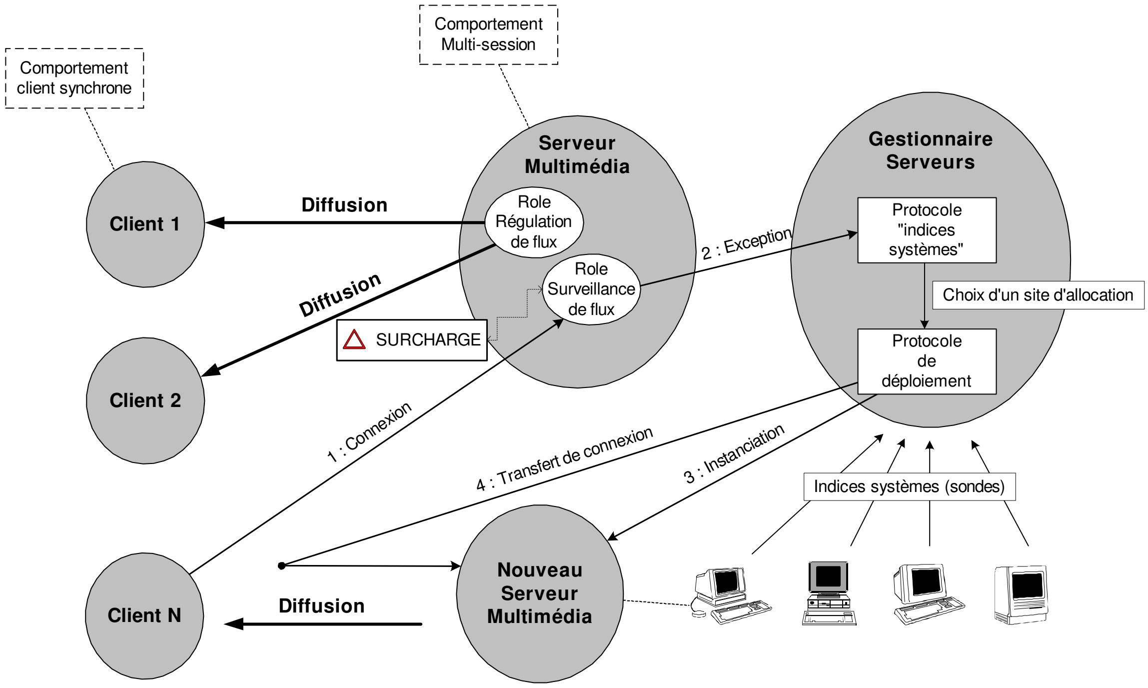
Malgré son intérêt et sa clarté, notre approche, prise globalement, peut sembler quelque peu «radicale» ; peut-être même trop pour être industriellement «satisfaisante». Mais il nous apparaît comme plutôt évident que sur les problématiques précises de composition et d'adaptation dynamiques, les concepts et outils proposés sont «puissants» et pourraient être réintégrés dans des contextes moins «intransigeants» en matière de principes de conception.

En l'occurrence, nous avons décrit tout au long de notre débat des services adaptatifs pour systèmes distribués qui, s'ils sont exprimables de façon très concise en Scope/Comet, n'en sont pas pour autant relégués au statut de simples «jouets». Reprenons certains des services et applications évoqués ou détaillés sous l'auspice de dénominations évocatrices :

- Au chapitre 6 : une infrastructure, certes simple mais générique, pour la composition dynamique d'architectures à flots de données typées,
- Au chapitre 7 :
  - un service adaptable d'analyse et de notification d'indices systèmes fondé sur le principe du *publish/subscribe* avec filtrage implicite en fonction du type des données publiées, et
  - un modèle comportemental générique pour composants multi-sessions.
- Au chapitre 8 :
  - un service de filtrage de contenu générique et «enfichable» dynamiquement,
  - un protocole dynamique de rediffusion ordonnée d'événement, pouvant entre autre servir de base à des services de réplication pour la tolérance aux pannes, et
  - un mécanisme adaptatif de traces génériques et conditionnelles, utilisé dans le cadre de l'environnement Halley.
- Au chapitre 9 :
  - un service de nommage synchrone/bidirectionnel,
  - Un moteur «minimal» - et quasi-implicite - pour la validation de transactions, et
  - deux protocoles adaptatifs «fondamentaux» pour la synchronisation de flux de données côté serveur.
- Au chapitre 10 :
  - un modèle de comportement autorisant la gestion de co-routines, et
  - un environnement interactif pour le déploiement, le contrôle et le déverminage des applications Comet.

Ces applications, quoique de natures variées, reposent toutes sur un nombre réduit de concepts et outils, souvent réutilisés. En conséquence, il semble pour nous assez clair que notre plateforme, plus que simplement conceptuelle et formelle, permet de décrire des systèmes réels ; ces derniers exploitant au mieux les propriétés adaptatives de notre intersticiel et notamment son principal trait caractéristique : le mode de diffusion multi-point implicite en fonction du type. La primitives sendreceive, décrite au chapitre 9, est selon nous emblématique des gains offerts par la reconsidération des systèmes mêmes «traditionnels» sous l'angle des concepts et outils proposés ici.

Il est désormais temps de reprendre (enfin !) l'exemple de système multimedia adaptatif proposé au chapitre 1 pour le reconsidérer sous la bannière de notre plateforme désormais opérationnelle. Comme le montre la figure ci-dessous, nous pouvons faire apparaître, à l'aide de notre nouveau vocabulaire né de la construction et de l'étude de notre modèle/plateforme, des solutions assez précises concernant l'approche des problèmes posés d'adaptations structurelles et comportentales du système de diffusion.



Ainsi, l'allocation dynamique de serveur peut être simplement fournie par l'intégration de notre service de notification système (cf. chapitre 7) et d'une heuristique de choix de site d'allocation à partir des indices notifiés. Avec notre support linguistique pour le déploiement, l'instantiation et les interconnexions entre composants (cf. chapitre 8), toute adaptation structurelle comme l'allocation d'un nouveau serveur s'envisage de manière directe et à un haut-niveau d'abstraction.

Nous avons également montré, dans la «chirurgicale» décomposition des mécanismes d'interactions client/serveur du chapitre 9, que nous pouvions aller dans Comet jusqu'à intervenir sur les cadences de traitement des événements au sein des composants. Ces interventions étant le fait de rôles assignables à des composants en cours de fonctionnement, l'adaptation comportementale de grain fin est maintenant à notre portée ; nul besoin d'entrer dans le noyau du système - ou d'attendre que quelqu'un le fasse à notre place - pour aborder ces questions sensibles et trop souvent posées comme annexes. Sur notre exemple, la réutilisation du rôle de surveillance de flux décrit au chapitre 9 nous permet de détecter les phénomènes de surcharge du côté du serveur. Nous pouvons également employer la régulation de flux pour contrôler la diffusion multimédia proprement dite. Toutes ces fonctionnalités sont fournies dynamiquement, sans perturber le fonctionnement «normal» des composants mis en jeu...

Durant notre étude, nous avons pu passer - et c'est selon nous le plus grand intérêt de ce long processus de recherche que représente la thèse scientifique - d'une intuition forte concernant l'adéquation des concepts liés à la composition et l'adaptation dynamique des systèmes distribués et de «ce qu'il faudrait faire pour les approcher» à une véritable solution, pragmatique et imparfaite certes, mais relativement «propre», c'est-à-dire une solution dont les principes premiers sont formels.

Selon nous, ces concepts ont maintenant un véritable sens et s'il ne s'agira plus dans l'avenir proche de les reconsidérer, leur mise en œuvre pratique n'est pas encore complètement finalisée. Même s'il s'agit avant tout d'une plateforme expérimentale, l'environnement Comet implanté en Java n'est plus tout à fait un prototype ne pouvant servir qu'à valider expérimentalement les idées présentées dans cette thèse. Nous envisageons ainsi, à relativement court terme, de pouvoir comparer cette infrastructure avec d'autres plateformes interstitielles et notamment le célèbre Java/RMI et les *Enterprise JavaBeans* qui se fondent sur ce derniers. Nous pensons effectuer ces comparaisons sur des aspects assez techniques comme par exemple les critères de performance. Notre intuition est que malgré la focalisation sur les aspects de composition et d'adaptation dynamiques, nous pensons pouvoir obtenir des gains, qualitatifs et quantitatifs, sur des aspects plus généraux comme bien sûr les interactions asynchrones multi-point mais également, et cela pourrait devenir un résultat intéressant, sur les échanges client/serveur plus «traditionnels». Dans le même temps, ces comparaisons nous permettrons de réfléchir en termes d'interopérabilité (et pourquoi pas un service Commun Corba pour la composition et l'adaptation dynamique ?).

Au niveau des fonctionnalités expérimentales, nos travaux s'orientent à court terme vers le déverminage distribué adaptatif : traces enfichables (i.e. *pluggables*), mode pas à pas, points d'arrêts conditionnels, etc. Nous avons vu (par exemple pour ce qui concerne la trace) que les rôles filtres et crochets semblent particulièrement «puissants» pour mettre en œuvre ces mécanismes. Cela pourrait permettre, dans une certaine mesure, de reconsidérer la difficile problématique de déverminage des systèmes distribués en cours d'exécution.

La *sécurité* et la *migration profonde* représentent deux autres aspects importants des systèmes distribués ouverts. L'approche de ces problématiques dans l'optique de l'adaptation dynamique ouvre, selon nous, un territoire assez nouveau de recherche et de développement. Ces aspects sortent du cadre de l'infrastructure Comet mais leur introduction dans cette dernière nous semble un point de départ intéressant pour permettre leur considération sous l'angle de l'adaptation dynamique. Un premier stage de maîtrise nous a donné quelques indices satisfaisant concernant la faisabilité de ces «extensions». Il s'agit également du point de départ de notre projet de rejoindre l'équipe du Professeur Akinori Yonezawa à l'Université de Tokyo, dans le cadre du programme AMO (*Autonomous Mobile Objects*).

En partant de problématiques plutôt «système» (puisqu'il s'agissait de mon thème de DEA), j'ai au cours de ma thèse (très) lentement mais inexorablement «évolué» vers des considérations de plus en plus linguistiques. D'une vision assez naïve des types, nous avons pu ainsi nous diriger vers un modèle certes assez éloigné des travaux actuels dans le domaine mais dont nous avons, selon nous, «prouvé» la valeur, en tout cas dans le cadre de notre interstitiel Comet.

Dans cette perspective, la conception et la validation du langage de composition Scope représente selon moi le réel «tournant» dans mon travail de thèse. Il est intéressant de noter que malgré son ciblage pour l'environnement Java, c'est avant tout l'implantation de ce langage dans un environnement «lispien» (et autrement plus ouvert) qui nous a permis de délimiter précisément ce qui faisait la différence entre le langage de composition et le langage d'opérationalisation de ces mêmes compositions.

Cette délimitation revêt à mes yeux une importance capitale puisqu'elle fait la différence entre *langages informatiques universels* et *langages spécifiques*. Malgré le faible nombre de constructions proposées, le langage Scope (et ses extensions pour Comet) dénote une sémantique dont la complexité est tangible au vu des nombreux schémas Z qu'il faut produire pour la caractériser, ne serait-ce que partiellement ! Ainsi, l'on vérifie en quelque sorte le «pronostic» de Christian Queinnec : un langage utile (comme, nous l'espérons en tout cas, notre langage de composition) et non-universel (nous sommes loin, dans Scope, de la «puissance» des machines de Türing) n'est pas, pour autant, un langage simple à concevoir. Mais il ne faut pas non plus penser que le choix du langage universel sous-jacent est arbitraire. Notre expérience double Lisp/Java nous a enseigné que ces deux environnements ne nous conduisaient absolument pas vers les mêmes gammes de solutions; et ce même pour ce qui concerne la nature des constructions linguistiques des langages spécifiques envisagés[19] !

En parallèle des développements liés à l'infrastructure Comet, nous avons en conséquence démarré une réflexion, encore assez abstraite, concernant cette problématique de *conception* - et, de façon encore plus importante, de *composition* - de langages spécifiques «utiles». Ceci nous a conduit à une première proposition qui nous semble pleine d'espoir, sous la forme d'un prototype de méta-outil pour la conception et la composition de langages spécifiques nommés *Jargons* [Pes01]. Il est intéressant de signaler que cet outil correspond en gros à une «extrapolation» des procédés mis en œuvre pour concevoir le langage de composition Scope décrit dans cette thèse. Le résultat, construit au dessus d'un cœur hybride fonctionnel/logique, propose une vision assez radicale de la réflexion - c'est-à-dire au sens du système Maciste de Jacques Pitrat [Pit96] - avec règles (fonctionnelles), méta-règles (ordre supérieur, contrôle) et méta-méta-règles (amorçage, monitoring,etc.). Ceci nous dirige irrémédiablement - à très long terme sans doute - vers l'étude de systèmes plus *autonomes* (toujours au sens de Maciste), mais il s'agit d'un autres débat ...

19. Lisp et Scheme proposent un avantage certain : la possibilité d'étendre le langage dans les termes mêmes du langage. De sorte rétrospectivement, il fut «salutaire» selon nous de passer de Lisp à Java et non l'inverse.

Bibliographie

[A^{+}93] Gul Agha et al. *Abstraction and Modularity Mechanisms for Concurrent Computing.* In Gul Agha, editor, *Research directions in Concurrent Object-Oriented Programming*, pages 3–21. MIT Press, 1993.

[Abr96] Jean-Raymond Abrial. *The B Book. Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Agh86] Gul Agha. *Actors : A Model of Concurrent Computation in Distributed Systems.* Series in Artificial Intelligence. MIT Press, 1986.

[Ast99] Mark Astley. *Customization and Composition of Distributed Objects : Policy Management in Distributed Software Architectures.* PhD thesis, University of Illinois at Urbana-Champaign, May 1999.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and Distribution in Object-Oriented Programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[Bri89] Jean-Pierre Briot. *Actalk : a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment.* In S. Cook, editor, *Proceedings of ECOOP'89*, pages 109–129. Cambridge University Press, July 1989.

[Bri96] Jean-Pierre Briot. *An Experiment in Classification and Specialization of Synchronization Schemes.* In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, LNCS 1049, pages 227–249. Springer Verlag, March 1996.

[Bri97] Jean-Pierre Briot. *Actalk : A Framework for Object-Oriented Concurrent Programming - Design and Experience.* unpublished, 1997.

[BY89] Jean-Pierre Briot and Akinori Yonezawa. *Inheritance and Synchronization in Object-Oriented Concurrent Programming.* In *ABCL: An Object-Oriented Concurrent System* [Y^{+}89].

[BZB^{+}97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp) version 1 functional specifications. IETF RFC 2205 Proposed Standard, 1997.

[CN02] Brian Carpenter and Kathleen Nichols. Differentiated services. IETF Working group, 2002.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[Dic] Grand dictionnaire terminologie. `http://www.granddictionnaire.com`.

[DMN68] O.-J. Dahl, B. Myrhaug, and K. Nygaard. SIMULA 67 Common Base. Norwegian Computer Center, Oslo, Norway, 1968.

[DVM95] David C. Luckham, James Vera, and Sigurd Meldal. *Three Concepts of System Architecture.* Technical report, University of Standford, July 1995.

[ED99] Huw Evans and Peter Dickman. Zones, Contracts and Absorbing Changes: An Approach to Software Evolution. In *Proceedings of OOPSLA'99*. ACM Press, November 1999.

[Fit99] Tom Fitzpatrick. *Open Component-Oriented Multimedia Middleware for Adaptive Distributed Applications.* PhD thesis, Computing Department, Lancaster University, September 1999.

[FR89] Brian Foote and Ralph E. Johnson. *Reflective Facilities in Smalltalk-80.* In *Proceedings of OOPSLA'89*, volume 24, pages 327–335. ACM Press, October 1989.

[Frø96] Svend Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996.

[GGM97] Rachid Guerraoui, Benoît Garbinato, and Karim Mazouni. *GARF: A Tool for Programming Reliable Distributed Applications. IEEE concurrency*, 5(4):32–39, December 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[GS93] David Garlan and Mary Shaw. *An Introduction to Software Architecture.* In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.

[Gue99] Rachid Guerraoui. *What Object-Oriented Distributed Programming may Be - and What it does Not have to Be.* White paper, 1999.

[Han99] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, 1999.

[HB77] C. E. Hewitt and H. Baker. *Actors and Continuous Functionals.* In J. Heuhold, editor, *Proceedings IFIP Working Conference on Formal Description of Programming Concepts*, pages 367–387, August 1977.

[HKPM97] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. *The Coq Proof Assistant : A Tutorial : Version 6.1.* INRIA, Rocquencourt, France, 1997.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[Hoa99] C.A.R. Hoare. Theories of programming: top-down, bottom-up and meeting in the middle. Published online, 1999.

[IMY92] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. *RbCL: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel.* In *Proceedings of IMSA'92 International Workshop on Reflection and Meta-Level Architecture*, 1992.

[Ish91] Yutaka Ishikawa. *Reflection Facilities and Realistic Programming.* *ACM SIGPLAN Notices*, 26(8):101–110, August 1991.

[IWC+88] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik, a visual programming environment. In *Proceedings of OOPSLA'88.* ACM Press, November 1988.

[K+97] Gregor Kiczales et al. *Aspect-Oriented Programming.* In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241. Springer Verlag, June 1997.

[KBH+01] R. Koster, A. P. Black, J. Huang, J. Walpole, and C. Pu. Thread transparency in information flow middleware. In *Proceedings of Middleware'01 (IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing).* Springer Verlag, November 2001.

[Led98] Thomas Ledoux. *Réflexion dans les systèmes répartis : application à CORBA et Smalltalk.* PhD thesis, Département informatique de l'école des mines de Nantes, 1998.

[Luc96] David C. Luckham. *Rapide : A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. DIMACS Partial Order Methods Workshop*, IV, July 1996.

[LY00] X. Liu and H. Yang, editors. *International Symposium on Principles of Software Evolution (ISPSE 2000), Kanazawa, Japan.* IEEE computer society, November 2000.

[Mae87] Pattie Maes. *Concepts and Experiments in Computational Reflection.* In *Proceedings of OOSPLA'87*, volume 22, pages 147–155, October 1987.

[McA95a] Jeff McAffer. *Metalevel Programming with Coda.* In *Proceedings of the European Conference on Object-Oriented Computing (ECOOP)*, LNCS 952, pages 190–214. Springer Verlag, August 1995.

[McA95b] Jeff McAffer. *A Meta-Level Architecture for Prototyping Object Systems.* PhD thesis, University of Tokyo, September 1995.

[Mey97] B. Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[Mey00] R. Meyer. State of the art of distributed event models. Technical report, University of Dublin, Trinity College, 2000.

[Mic97] Microsoft. *Component Object Model extensions (COM+).* `http://www.microsoft.com/com`, 1997.

[Mig99] Frédéric Migeon. *Etude et implantation de mécanismes réflexifs dans un langage concurrent.* PhD thesis, Université Paul Sabatier, Toulouse, 1999.

[Mil99] Robin Milner. *Communicating and Mobible Systems: The PI-Calculus.* Cambridge University Press, 1999.

[MW91] Satoshi Matsuoka and Takuo Watanabe. *Hybrid Group Reflective Architecture for Object-Oriented Reflective Programming.* In *Proceedings of ECOOP'91*, volume LNCS 512, pages 231–250. Springer Verlag, 1991.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages.* In Gul Agha, editor, *Research directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[Obj01] Object Management Group. *Common Object Request Broker Architecture (CORBA) 2.6 Specifications.* `http://www.corba.org`, 2001.

[OMG99] Object Management Group. *Corba CCM specification (draft).* `http://www.omg.org`, 1999.

[Pes01] Frédéric Peschanski. Jargons: Experimenting composable domain-specific languages. In *Proceedings of ICFP 2001 Scheme Workshop*, September 2001.

[Pit96] Jacques Pitrat. Implementation of a reflective system. *Future Generation Computer Systems*, 12:235–242, 1996.

[PQB02] Frédéric Peschanski, Christian Queinnec, and Jean-Pierre Briot. A typeful composition model for dynamic software architectures. Technical report, University of Paris VI - Pierre et Marie Curie, 2002.

[RSBP95] N. Revault, H.A. Sahraoui, G. Blain, and J.-F. Perrot. A metamodeling technique: The métagen system. In *Tools 16: Tools Europe'95*, pages 127–139, Versailles, France, Mar. 1995. Prentice Hall.

[S+98] Marc Shapiro et al. *PerDiS : design, implementation, and use of a PERsistent DIstributed Store.* Technical Report No 3525, Inria, October 1998.

[Spi01] J. M. Spivey. *The Z Notation : a reference manual.* `http://spivey.oriel.ox.ac.uk/ mike/zrm/`, 2001.

[Stu96] Daniel C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[Sun98] Sun Microsystems. *JavaBeans 1.01 Specifications.* `http://java.sun.com/products/javabeans`, 1998.

[Sun00] Sun Microsystems. *Enterprise JavaBeans 1.2 specification.* `http://java.sun.com/ejb`, 2000.

[Sun01] Sun Microsystems. *Java Remove Method Invocation (RMI) 1.4 Specifications.* `http://java.sun.com/rmi`, 2001.

[Szy98] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, 1998.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. In *Proceedings of OOPSLA 2001.* ACM Press, October 2001.

[Van01] Gilles Vanwormhoudt. *Java Remote Method Invocation.* `http://lifl.lifl.fr/ vanwormh/Web/rmi.pdf`, 2001.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages, an annotated bibilography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[Y+89] Akinori Yonezawa et al. *ABCL, An Object-Oriented Concurrent System.* Series in Computer Systems. MIT Press, 1989.

[Zel96] Gregory Zelesnik. The unicon language reference manual. Technical report, Carnegie Mellon University, 1996.

Quatrième partie

Annexes

«Faites attention au code ci-dessus, j'ai juste prouvé qu'il était correct, je ne l'ai pas essayé»

Donald E. Knuth

A
Preuves

A.1 Système de type

A.1.1 Preuves des sous-types et super-types directs

Propriétés à prouver :

$$(1) \forall x, y_1, y_2 : TYPE \bullet (y_1 \neq y_2) \wedge (y_1 \succ_{dir} x) \wedge (y_2 \succ_{dir} x) \Rightarrow y_1 \not\sim y_2$$
$$(2) \forall x, y_1, y_2 : TYPE \bullet (y_1 \neq y_2) \wedge (y_1 \prec_{dir} x) \wedge (y_2 \prec_{dir} x) \Rightarrow y_1 \not\sim y_2$$

Preuves : Commençons par la propriété (1)

$$(3) y_1 \succ_{dir} x \Rightarrow \forall z : TYPE \bullet z \succ x \Rightarrow (z \not\sim y_1) \vee (z \succeq y_1) \qquad [\text{def de } \succ_{dir}]$$
$$(4) y_2 \succ_{dir} x \Rightarrow (y_2 \not\sim y_1) \vee (y_2 \succeq y_1) \qquad [(3) \text{ et identification de } z \text{ à } y_2 \text{ et } (\succ_{dir} \subset \succ)]$$
$$\Rightarrow (y_2 \not\sim y_1) \vee (y_2 \succ y_1) \qquad [y_1 \neq y_2 \text{ par hypothèse dans (1)}]$$
$$(5) y_2 \succ_{dir} x \Rightarrow (y_1 \not\sim y_2) \vee (y_1 \succ y_2) \qquad [\text{idem (3) et (4) pour } y_2]$$
$$(4) \wedge (5) \Rightarrow (y_1 \not\sim y_2) \vee [(y_2 \succ y_1) \wedge (y_1 \succ y_2)]$$
$$\Rightarrow (y_1 \not\sim y_2) \vee \mathit{false} \Rightarrow y_1 \not\sim y_2 \qquad [\text{propriété (1)}]$$

La propriété (3) correspond à la définition partielle de la relation de super-type direct pour le type y_1. En identifiant y_2, élément de $TYPE$ à z, nous écrivons en (4) que y_2 n'est super-type strict de x que s'il n'est pas relié à y_1 ou s'il lui est "supérieur". De la définition de la relation super-type direct, nous pouvons également remarquer qu'elle correspond à un sous-ensemble de la relation super-type strict. Ce résultat est pris en compte en dans la propriété (4). La propriété (5) est identique mais en utilisant la définition de la relation super-type direct pour y_1. En combinant (4) et (5), nous obtenons la propriété (1) que nous devions prouver □

La propriété (2) peut être prouvée selon le même procédé □

A.1.2 Théorème d'initialisation du schéma *InitTypeSystem*

Propriété à prouver :

$$\exists\ TypeSystem' \bullet InitTypeSystem$$

avec :

– Schéma *TypeSystem* défini page 63
– Schéma *InitTypeSystem* défini page 64

Preuve :

$$
\begin{array}{ll}
\exists\, TypeSystem' \bullet InitTypeSystem & \text{[théorème d'initialisation]} \\
\Leftrightarrow \exists\, TypeSystem' \bullet [\, TypeSystem' \mid & \\
\quad types' = \{Event\} \wedge & \\
\quad super' = \{Event \mapsto Event\} \,] & \text{[def de } InitTypeSystem\text{]} \\
\Leftrightarrow \exists\, TypeSystem' \bullet & \\
\quad types' = \{Event\} \wedge & \\
\quad super' = \{Event \mapsto Event\} & \text{[quantification de schéma]} \\
\Leftrightarrow \exists\, types' : \mathbb{P}_1\, TYPE \bullet & \\
\quad \exists\, super' : TYPE \leftrightarrow TYPE \bullet & \\
\qquad \forall\, t_1 : types';\, t_2 : types' \bullet \{t_1 \mapsto t_2\} \in super' \Leftrightarrow t_1 \succeq t_2 \wedge & \\
\qquad \mathrm{dom}\, super' = \mathrm{ran}\, super' = types' \wedge & \\
\qquad \exists_1\, r : types' \bullet \forall\, t : types' \bullet r \succeq t \wedge & \\
\qquad types' = \{Event\} \wedge & \\
\qquad super' = \{Event \mapsto Event\} & \text{[def de } TypeSystem\text{]} \\
\Leftrightarrow \exists\, types' : \mathbb{P}_1\, TYPE \bullet & \\
\quad \{Event \mapsto Event\} \in TYPE \leftrightarrow TYPE \wedge & \\
\quad \forall\, t_1, t_2 : types' \bullet \{t_1 \mapsto t_2\} \in \{Event \mapsto Event\} \Leftrightarrow t_1 \succeq t_2 \wedge & \\
\quad \mathrm{dom}\{Event \mapsto Event\} = \mathrm{ran}\{Event \mapsto Event\} = types' \wedge & \\
\quad \exists_1\, r : types' \bullet \forall\, t : types' \bullet r \succeq t \wedge & \\
\quad types' = \{Event\} & \text{[regle du point unique pour } super'\text{]} \\
\Leftrightarrow & \\
\quad (1) \qquad \{Event\} \in \mathbb{P}_1\, TYPE \wedge & \\
\quad (2) \qquad \{Event \mapsto Event\} \in TYPE \leftrightarrow TYPE \wedge & \\
\quad (3) \qquad \{Event \mapsto Event\} \in \{Event \mapsto Event\} \Leftrightarrow Event \succeq Event \wedge & \\
\quad (4) \qquad \mathrm{dom}\{Event \mapsto Event\} = \mathrm{ran}\{Event \mapsto Event\} = \{Event\} \wedge & \\
\quad (5) \qquad \exists_1\, r : \{Event\} \bullet \forall\, t : \{Event\} \bullet r \succeq t & \text{[point unique pour } types'\text{]}
\end{array}
$$

Pour arriver au propriétés à prouver (1) à (5), nous réécrivons plusieurs fois le théorème d'initialisation, tout d'abord en remplaçant le nom de schéma *InitTypeSystem* par sa définition (première ligne). Ensuite, puisque ce schéma déclare un sous-schéma *TypeSystem'*, nous pouvons utiliser celui introduit par le quantificateur existentiel global pour simplifier notre théorème (deuxième ligne). Finalement, nous remplaçons le schéma *TypeSystem'* par sa définition et appliquons deux fois la règle du point unique qui permet de remplacer les quantificateurs existentiels pour les variables *types'* et *super'* qui y sont déclarées.

La propriété (1) ne peut être prouvée et doit être posée : le type *Event* est un type valide pour le système. Dans le chapitre suivant, nous donnerons des proriétés supplémentaires à ce type racine mais pour tout ce qui concerne l'aspect structurel du système, son nom et le fait qu'il soit valide seront deux conditions nécessaires et suffisantes.

La deuxième propriété (2) découle simplement de la première : la relation minimale *super'* est bien une relation entre types si *Event* est un type valide.

De la propriété (3) qui restreint la relation de sous-typage *super*, nous pouvons déduire : $Event \succeq Event$.

Nous pouvons finalement vérifier trivialement les propriétés (4) et (5) qui correspondent aux deux propriétés fondamentales de l'invariant du système *TypeSystem*.

En conclusion, le système de type initial *InitTypeSystem* vérifie le théorème d'initialisation et est donc un état initial *cohérent* pour le système de type *TypeSystem* □

A.1.3 Preuve de l'opération *AddType*

Propriété à prouver :

pre *AddType*

où le schéma *AddType* est défini page 65.

Preuve :

$$
\begin{array}{l}
\text{pre } AddType \Leftrightarrow \exists\, TypeSystem' \bullet AddType \hfill [\text{définition de pre }] \\
\Leftrightarrow \exists\, TypeSystem' \bullet \\
\quad [\,TypeSystem;\ TypeSystem';\ type? : TYPE;\ super? : \mathbb{P}_1\ TYPE \mid \\
\qquad type? \notin types \land super? \subseteq types \land \\
\qquad \forall\, s_1, s_2 : super? \bullet (s_1 \sim s_2) \Rightarrow (s_1 = s_2) \land \\
\qquad types' = types \cup \{type?\} \land \\
\qquad super' = (super \cup \{s : super? \bullet s \mapsto type?\})^+ \\
\qquad\quad \cup \{type? \mapsto type?\}] \hfill [\text{def de } AddType \text{ et de } \Delta] \\
\Leftrightarrow [\,TypeSystem;\ type? : TYPE;\ super? : \mathbb{P}_1\ TYPE \mid \\
\quad \exists\, types' : \mathbb{P}_1\ TYPE \bullet \\
\qquad \exists\, super' : TYPE \leftrightarrow TYPE \bullet \\
\qquad\quad \forall\, t_1, t_2 : TYPE \bullet \{t_1 \mapsto t_2\} \in super' \Leftrightarrow t_1 \succeq t_2 \land \\
\qquad\quad \operatorname{dom} super' = \operatorname{ran} super' = types' \land \\
\qquad\quad \exists_1\, r : types' \bullet \forall\, t : types' \bullet r \succeq t \land \\
\qquad\quad types' = types \cup \{type?\} \land \\
\qquad\quad super' = (super \cup \{s : super? \bullet s \mapsto type?\})^+ \\
\qquad\qquad \cup \{type? \mapsto type?\}] \land \\
\qquad\quad type? \notin types \land super? \subseteq types \land \\
\qquad\quad \forall\, s_1, s_2 : super? \bullet (s_1 \sim s_2) \Rightarrow (s_1 = s_2)] \hfill [\text{def de } TypeSystem \text{ et quantification}] \\
\Leftrightarrow [\,types : \mathbb{P}_1\ TYPE;\ super : TYPE \leftrightarrow TYPE;\ type? : TYPE;\ super? : \mathbb{P}_1\ TYPE \mid \\
\quad \exists\, types' : \mathbb{P}_1\ TYPE \bullet \\
\qquad \exists\, super' : TYPE \leftrightarrow TYPE \bullet \\
\qquad\quad \forall\, t_1, t_2 : TYPE \bullet \{t_1 \mapsto t_2\} \in super' \Leftrightarrow t_1 \succeq t_2 \land \\
\qquad\quad \operatorname{dom} super' = \operatorname{ran} super' = types' \land \\
\qquad\quad \exists_1\, r : types' \bullet \forall\, t : types' \bullet r \succeq t \land \\
\qquad\quad types' = types \cup \{type?\} \land \\
\qquad\quad super' = (super \cup \{s : super? \bullet s \mapsto type?\})^+ \\
\qquad\qquad \cup \{type? \mapsto type?\}] \land \\
\qquad\quad type? \notin types \land super? \subseteq types \land \\
\qquad\quad \forall\, s_1, s_2 : super? \bullet (s_1 \sim s_2) \Rightarrow (s_1 = s_2) \land \\
\qquad\quad \forall\, t_1 : types;\ t_2 : types \bullet \{t_1 \mapsto t_2\} \in super \Leftrightarrow t_1 \succeq t_2 \land \\
\qquad\quad \operatorname{dom} super = \operatorname{ran} super = types \land \\
\qquad\quad \exists_1\, r : types \bullet \forall\, t : types \bullet r \succeq t] \hfill [\text{def de } TypeSystem] \\
\Leftrightarrow [\,TypeSystem;\ type? : TYPE;\ super? : \mathbb{P}_1\ TYPE \mid \\
\quad (P_1) types \cup \{type?\} \in \mathbb{P}_1\ TYPE \land \\
\quad (P_2) (super \cup \{s : super? \bullet s \mapsto type?\})^+ \cup \{type? \mapsto type?\} \in TYPE \leftrightarrow TYPE \land \\
\quad (P_3) \forall\, t_1, t_2 : TYPE \bullet \\
\qquad \{t_1 \mapsto t_2\} \in [(super \cup \{s : super? \bullet s \mapsto type?\})^+ \cup \{type? \mapsto type?\}] \Leftrightarrow t_1 \succeq t_2 \land \\
\quad (P_4) \operatorname{dom}[(super \cup \{s : super? \bullet s \mapsto type?\})^+ \cup \{type? \mapsto type?\}] \\
\qquad = \operatorname{ran}[(super \cup \{s : super? \bullet s \mapsto type?\})^+ \cup \{type? \mapsto type?\}] \\
\qquad = types' \cup \{type?\} \land \\
\quad (P_5) \exists_1\, r : types \cup \{type?\} \bullet \forall\, t : types \cup \{type?\} \bullet r \succeq t] \land \\
\quad (H_1) type? \notin types \land \\
\quad (H_2) super? \subseteq types \land
\end{array}
$$

$(H_3)\,\forall\, s_1, s_2 : super? \bullet (s_1 \sim s_2) \Rightarrow (s_1 = s_2) \wedge$
$(H_4)\,\forall\, t_1 : types;\, t_2 : types \bullet \{t_1 \mapsto t_2\} \in super \Leftrightarrow t_1 \succeq t_2 \wedge$
$(H_5)\,\mathrm{dom}\, super = \mathrm{ran}\, super = types \wedge$
$(H_6)\,\exists_1\, r : types \bullet \forall\, t : types \bullet r \succeq t\,]$ [point unique pour $types'$ et $super'$]

La première propriété (P_1) est vraie puisque *types*, par définition, est un sous-ensemble de $TYPE$ et le type ajouté, *type*? est également un type valide.

Pour prouver (P_2), cherchons une expression simplifiée de la relation $super'$:

$$(P_6) super' = (super \cup \{s : super? \bullet s \mapsto type?\})^+ \cup \{type? \mapsto type?\} \qquad \text{[postcondition de } AddType\text{]}$$

Notre principal problème ici est que la relation *super* est augmentée et fermée transitivement. Ecrivons cette relation en appliquant la fermeture transitive en trois étapes :

$$\begin{aligned}(P_6) \Leftrightarrow super' &= (super \cup \{s : super? \bullet s \mapsto type?\}^+ \\ &= super^+ \cup \{s : super? \bullet s \mapsto type?\} \\ &\quad \cup \{s_1 : types;\, s_2 : super? \mid s_1 \succeq s_2 \bullet s_1 \mapsto type?\}\end{aligned}$$

Tout d'abord, nous prenons la fermeture transitive de la relation *super* elle-même. Dans un deuxième temps, nous ajoutons l'ensemble des nouvelles assocations des types de *super*? vers le nouveau type ajouté *type*?. Enfin, nous fermons la relation en associant tous les super-types au sens large des éléments de *super*? vers *type*?.

Nous savons la relation $\succeq$ (et donc *super* qui possède les mêmes propriétés d'après (H_4)) transitive, ce qui induit : $super^+ = super$. Nous, pouvons donc réécrire (P_6) de la façon suivante :

$$\begin{aligned}(P_6) \Leftrightarrow super' &= (super \cup \{s : super? \bullet s \mapsto type?\}^+ \\ &= super \cup \{s : super? \bullet s \mapsto type?\} \\ &\quad \cup \{s_1 : types;\, s_2 : super? \mid s_1 \succeq s_2 \bullet s_1 \mapsto type?\}\end{aligned}$$

D'après le schéma *TypeSystem*, la relation *super* opère sur des types d'où $super \in TYPE \leftrightarrow TYPE$. On a aussi par définition, $super? \in TYPE$ et $type? \in TYPE$ donc

$$\{s : super? \bullet s \mapsto type?\} \in TYPE \leftrightarrow TYPE$$

Toujours pour les mêmes raisons, nous pouvons aisément montrer que

$$\{s_1 : types;\, s_2 : super? \mid s_1 \succeq s_2 \bullet s_1 \mapsto type?\} \in TYPE \leftrightarrow TYPE$$

Nous en concluons donc finalement la vérité de la propriété (P_2).

La propriété (P_3) correspond toujours à la définition de la restriction à $types'$ de $\succeq$ pour $super'$. Il faut en retenir le fait que le nouveau type ajouté *type*? est sous-type direct des types de *super*?.

Très importante, la propriété (P_4) relie entre elles les variables après changement : $super'$ et $types'$. Démontrons la partie de la propriété concernant le domaine de la relation $super'$ en partant de la propriété suivante :

$$\begin{aligned}(P_7)\,\mathrm{dom}\, super' &= \mathrm{dom}[(super \cup \{s : super? \bullet s \mapsto type?\})^+ \cup \{type? \mapsto type?\}] && \text{[def de } super'\text{]} \\ &= \mathrm{dom}(super \cup \{s : super? \bullet s \mapsto type?\})^+ \cup \mathrm{dom}\{type? \mapsto type?\} && \text{[propriété de dom]} \\ &= \mathrm{dom}(super \cup \{s : super? \bullet s \mapsto type?\})^+ \cup \{type?\} && \text{[def de dom]}\end{aligned}$$

D'après (H_2), $super? \subseteq types$ et, de façon triviale,

$$\mathrm{dom}\{s : super? \bullet s \mapsto type?\} = super?$$

D'où

$$\mathrm{dom}\{s : super? \bullet s \mapsto type?\} \subseteq types$$

Dans le même ordre d'idée, il est clair que

$$\mathrm{dom}\{s_1 : \mathit{types};\, s_2 : \mathit{super?} \mid s_1 \succeq s_2 \bullet s_1 \mapsto \mathit{type?}\} \subseteq \mathit{types}$$

puisque les associations composées se font avec l'antécédent s_1, élement de *types*.

D'après (H_5), dom *super* = *types*, ce qui nous permet d'écrire finalement :

$$\begin{aligned}
&(P_8)\, \mathrm{dom}(\mathit{super} \cup \{s : \mathit{super?} \bullet s \mapsto \mathit{type?}\})^+ \\
&\quad = \mathrm{dom}(\mathit{super} \cup \{s : \mathit{super?} \bullet s \mapsto \mathit{type?}\} \\
&\qquad \cup \{s_1 : \mathit{types};\, s_2 : \mathit{super?} \mid s_1 \succeq s_2 \bullet s_1 \mapsto \mathit{type?}\} && [\text{d'après } (P_6)] \\
&\quad = (\mathrm{dom}\,\mathit{super}) \cup (\mathrm{dom}\{s : \mathit{super?} \bullet s \mapsto \mathit{type?}\}) \\
&\qquad \cup (\mathrm{dom}\{s_1 : \mathit{types};\, s_2 : \mathit{super?} \mid s_1 \succeq s_2 \bullet s_1 \mapsto \mathit{type?}\}) && [\text{propriété de dom}] \\
&\quad = \mathit{types} \cup (T_1 \subseteq \mathit{types}) \cup (T_2 \subseteq \mathit{types}) && [\text{par hypothèse}] \\
&\quad = \mathit{types} && [\text{propriété de l'union}]
\end{aligned}$$

Nous avons désormais toutes les informations nécessaires pour donner à la propriété (P_7) une expression grandement simplifiée :

$$\begin{aligned}
&(P_7) \Leftrightarrow \mathrm{dom}\,\mathit{super}' = \mathrm{dom}(\mathit{super} \cup \{s : \mathit{super?} \bullet s \mapsto \mathit{type?}\})^+ \cup \{\mathit{type?}\} \\
&\quad \Leftrightarrow \mathrm{dom}\,\mathit{super}' = \mathit{types} \cup \{\mathit{type?}\} && [\text{d'après } (P_8)]
\end{aligned}$$

De cette réécriture, nous pouvons déduire dom $\mathit{super}' = \mathit{types}'$, ce qui correspond à la première "moitié" de la relation (P_4). Pour la deuxième partie, nous devons "calculer" le codomaine de la relation super' :

$$\begin{aligned}
&(P_9)\, \mathrm{ran}\,\mathit{super}' = \mathrm{ran}(\mathit{super} \cup \{s : \mathit{super?} \bullet s \mapsto \mathit{type?}\})^+ \cup \mathrm{ran}\{\mathit{type?} \mapsto \mathit{type?}\} \\
&\quad = (\mathrm{ran}\,\mathit{super}) \cup (\mathrm{ran}\{s : \mathit{super?} \bullet s \mapsto \mathit{type?}\}) \\
&\qquad \cup (\mathrm{ran}\{s_1 : \mathit{types};\, s_2 : \mathit{super?} \mid s_1 \succeq s_2 \bullet s_1 \mapsto \mathit{type?}\}) && [\text{d'après } (P_6) \text{ et prop de ran}] \\
&\qquad \cup \{\mathit{type?}\} && [\text{def de ran}] \\
&\quad = (\mathrm{ran}\,\mathit{super}) \cup \{\mathit{type?}\} \cup \{\mathit{type?}\} \cup \{\mathit{type?}\} && [\text{relations à image unique } \mathit{type?}] \\
&\quad = \mathit{types} \cup \{\mathit{type?}\} && [\text{d'après } (H_5) \text{ et prop de } \cup]
\end{aligned}$$

En conclusion, la propriété (P_4) est vraie, ce qui est un résultat important concernant la préservation de l'invariant du *TypeSystem* par l'opération *AddType*.

Pour prouver, selon (P_5) que le système de type admet toujours une racine unique, rappelons-nous que cette racine est posée comme étant le type *Event*. Il s'agit donc simplement de prouver que le type *type?*, une fois ajouté au système, est bien un sous-type de *Event* :

$$\mathit{Event} \succeq \mathit{type?}$$

D'après (H_6), nous savons que tous les éléments de *types* sont sous-types, au sens large, de *Event*. En particulier, ceci est vrai pour les éléments de *super?* puisque $\mathit{super?} \subseteq \mathit{types}$ d'après (H_2) :

$$\forall s : \mathit{super?} \bullet \mathit{Event} \succeq s$$

Puisque d'après (P_6),

$$\{s_1 : \mathit{types};\, s_2 : \mathit{super?} \mid s_1 \succeq s_2 \bullet s_1 \mapsto \mathit{type?}\} \subseteq \mathit{super}'$$

et en identifiant s_1 à *Event*, on peut en déduire

$$\mathit{Event} \succeq \mathit{type?}$$

d'où la vérification de la propriété (P_5).

En conclusion, le schéma *AddType* est cohérent et l'opération qu'il décrit préserve bien l'invariant du système
□

A.2 Modèle de composition

A.2.1 Théorème d'initialisation pour *InitAbstractOperationalSystem*

Propriété à prouver :

$$\exists\, AbstractOperationalSystem' \bullet InitAbstractOperationalSystem'$$

Rappelons que nous avons déjà traité le problème d'initialisation du système de type. De plus, aucune contrainte n'est ajoutée dans l'état global que nous initialisons, il ne nous reste donc plus qu'à prouver :

$$\exists\, AbstractOperationalSystemPart' \bullet InitAbstractOperationalSystemPart'$$

avec :

- Schéma *AbstractOperationalSystemPart* défini page 71
- Schéma *InitAbstractOperationalSystemPart* défini page 73

Preuve :

$$\begin{array}{ll}
\exists\, AbstractOperationalSystemPart' \bullet InitAbstractOperationalSystemPart' & \\
\Leftrightarrow \exists\, AbstractOperationalSystemPart' \bullet & \\
\quad comps' = \emptyset \land & \\
\quad conxs' = \emptyset & \text{[def de schéma et quantification]} \\
\Leftrightarrow \exists\, comps' : \mathbb{P}\, AbstractComponent \bullet & \\
\quad \exists\, conxs' : \mathbb{P}\, EffectiveConnection \bullet & \\
\quad\quad comps' = \emptyset \land & \\
\quad\quad conxs' = \emptyset & \text{[def de schéma]} \\
\quad\quad \text{inv } AbstractOperationalSystemPart' &
\end{array}$$

La notation (non standard) inv *Schema* référence l'invariant du schéma considéré. Ici, l'invariant est trivialement respecté lorsque les variables $comps'$ et $conxs'$ sont associées à l'ensemble vide. En effet, les conditions de ces invariants sont toutes quantifiées sur ces deux variables ensemblistes[1] □

A.2.2 Preuve de cohérence de *NewComponent*

Propriété à prouver :

$$\text{pre } NewComponent$$

où *NewComponent* est définie page 74

Preuve :

$$\begin{array}{ll}
\text{pre } NewComponent \Leftrightarrow \exists\, AbstractOperationalSystemPart' \bullet NewComponent & \\
\Leftrightarrow [AbstractOperationalSystemPart;\ TypeSystem;\ c? : AbstractComponent \mid & \\
\quad \exists\, comps' : \mathbb{P}\, AbstractComponent \bullet & \\
\quad\quad \forall\, c_1, c_2 : comps' \bullet c_1.id = c_2.id \Rightarrow c_1 = c_2 \land & \\
\quad\quad c?.in_type \subseteq types \land & \\
\quad\quad c?.out_type \subseteq types \land & \\
\quad\quad \forall\, c : comps \bullet c.id \neq c?.id \land & \\
\quad\quad comps' = comps \cup \{c?\}\ ] & \text{[def et filtrage sur } comps\text{]}
\end{array}$$

1. Or, toute propriété est vraie sur une variable quantifiée sur l'ensemble vide.

$$
\begin{aligned}
&\Leftrightarrow [comps : \mathbb{P}\, AbstractComponent;\ TypeSystem;\ c? : AbstractComponent \mid \\
&\quad (P_1) comps \cup \{c?\} \subseteq \mathbb{P}\, AbstractComponent \wedge \\
&\quad (P_2) \forall\, c_1, c_2 : comps \cup \{c?\} \bullet c_1.id = c_2.id \Rightarrow c_1 = c_2 \wedge \\
&\quad (H_1) c?.in\_type \subseteq types \wedge \\
&\quad (H_2) c?.out\_type \subseteq types \wedge \\
&\quad (H_3) \forall\, c : comps \bullet c.id \neq c?.id \wedge \\
&\quad (H_4) \forall\, c_1, c_2 : comps \bullet c_1.id = c_2.id \Rightarrow c1 = c2\ ] && [\text{def et point unique sur } comps']
\end{aligned}
$$

Nous filtrons (à droite de la deuxième équivalence) la réécriture de l'obligation de preuve en ne retenant que les propriétés et hypothèses portant sur les variables $comps'$ et $comps$. Au final, nous obtenons deux propriétés à prouver : (P_1) et (P_2). La première est trivialement vraie puisque les éléments de $comps$ et la variable $c?$ sont de type $AbstractComponent$.

La propriété (P_2) permet de prouver l'unicité de l'identificateur $c?.id$. D'après (H_4), tous ces identificateurs sont déjà uniques. Puisque celui que l'on ajoute respecte l'hypothèse (H_3), on en déduit (P_2) trivialement □

A.2.3 Preuve de cohérence de l'opération *AddConnection*

Propriété à prouver :

pre $AddConnection$

où le schéma $AddConnection$ est défini page 76

Preuve :

$$
\begin{aligned}
&\text{pre } AddConnection \Leftrightarrow \exists\, AbstractOperationalSystem' \bullet AddConnection && [\text{définition de pre }] \\
&\Leftrightarrow [AbstractOperationalSystem;\ source?, dest? : AbstractComponent;\ T? : \mathbb{P}_1\, TYPE \mid \\
&\quad \exists\, conxs' : \mathbb{P}\, EffectiveConnection \bullet \\
&\qquad (\forall\, cx : conxs' \bullet \\
&\qquad\quad cx.source \in comps \wedge cx.dest \in comps \wedge \\
&\qquad\quad \forall\, t : types \bullet \exists\, t_1 : cx.source.out\_type \bullet t_1 \sim t \wedge \\
&\qquad\quad \forall\, t : types \bullet \exists\, t_2 : cx.dest.in\_type \bullet t_2 \succeq t) \wedge \\
&\qquad \exists\, InitEffectiveConnection \bullet conxs' = conxs \cup \{\Theta EffectiveConnection'\}\ ] \\
& && [\text{def et } conxs' \text{ unique variable modifiée}] \\
&\Leftrightarrow [AbstractOperationalSystem;\ source?, dest? : AbstractComponent;\ T? : \mathbb{P}_1\, TYPE \mid \\
&\quad \exists\, conxs' : \mathbb{P}\, EffectiveConnection \bullet \\
&\qquad (\forall\, cx : conxs' \bullet \\
&\qquad\quad cx.source \in comps \wedge cx.dest \in comps \wedge \\
&\qquad\quad \forall\, t : types \bullet \exists\, t_1 : cx.source.out\_type \bullet t_1 \sim t \wedge \\
&\qquad\quad \forall\, t : types \bullet \exists\, t_2 : cx.dest.in\_type \bullet t_2 \succeq t) \wedge \\
&\qquad source? \in comps \wedge dest? \in comps \wedge \\
&\qquad \Theta EffectiveConnection'.source = source? \wedge \\
&\qquad \Theta EffectiveConnection'.dest = dest? \wedge \\
&\qquad \Theta EffectiveConnection'.T = T? \wedge \\
&\qquad conxs' = conxs \cup \{\Theta EffectiveConnection'\}\ ] && [\text{def de } InitEffectiveConnection] \\
&\Leftrightarrow [AbstractOperationalSystem;\ source?, dest? : AbstractComponent;\ T? : \mathbb{P}_1\, TYPE \mid \\
&\quad (P_1) conxs \cup \{\Theta EffectiveConnection'\} \in \mathbb{P}\, EffectiveConnection \wedge \\
&\qquad (P_2)(\forall\, cx : conxs \cup \{\Theta EffectiveConnection'\} \bullet \\
&\qquad\quad (P_2 a) cx.source \in comps \wedge cx.dest \in comps \wedge \\
&\qquad\quad (P_2 b) \forall\, t : cx.type \bullet \exists\, t_1 : cx.source.out\_type \bullet t_1 \sim t \wedge
\end{aligned}
$$

$$
\begin{aligned}
&(P_2 c) \forall t : cx.type \bullet \exists t_2 : cx.dest.in\_type \bullet t_2 \succeq t) \wedge \\
&(H_1) source? \in comps \wedge dest? \in comps \wedge \\
&(H_2) \Theta EffectiveConnection'.source = source? \wedge \\
&(H_3) \Theta EffectiveConnection'.dest = dest? \wedge \\
&(H_4) \Theta EffectiveConnection'.T = T? \wedge \\
&(H_5)\ \mathrm{inv}\ AbstractOperationalSystem\ ]
\end{aligned}
$$

[point unique pour $conxs'$ et ajout de l'invariant principal du système]

Les réécritures successives décrites ci-dessus ne posent pas de gros problème, si ce n'est l'utilisation du symbole Θ que nous devons expliquer. En Z, $\Theta Schema$ donne l'élément caractéristique du schéma $Schema$ considéré. Il s'agit de l'unique moyen de déclarer une instance de schéma dans la définition même d'un autre schéma. Une fois déclarée, une condition de la forme $\Theta Schema.var = val$ donnera a la variable var de l'instance de $Schema$ la valeur val.

Dans notre schéma $AddConnection$, nous indiquons le lien entre le schéma local $EffectiveConnection'$ introduit par $InitEffectiveConnection$ et la variable $conxs'$ du schéma global $AbstractOperationalSystem'$ après ajout, et ceci en deux étapes. Premièrement, nous déclarons et définissons une instance caractéristique $\Theta EffectiveConnection'$. Dans un deuxième temps, la promotion au schéma global se fait par la condition :

$$conxs' = conxs \cup \{\Theta EffectiveConnection'\}$$

où nous indiquons que l'ensemble des connexions après ajout est adjoint de la nouvelle instance caractéristique créée pour l'occasion.

Finalement, nous obtenons deux propriétés (P_1) et (P_2) portant respectivement sur le typage de $conxs'$ ainsi que ses propriétés définies dans inv $AbstractOperationalSystem$.

La propriété (P_1) ne pose pas de problème puisque l'on sait (par (H_5)) que $conxs \in \mathbb{P}\, EffectiveConnecion$. De plus, par définition, toute instance caractéristique $\Theta Schema$ est du même type que $Schema$. Dans notre cas, cela donne à l'instance $\Theta EffectiveConnection'$ le type de $EffectiveConnection'$, lui-même identique au type de $EffectiveConnection$ (puisque le premier ne fait que décorer les variables du second).

La propriété (P_2) porte sur l'ensemble des connexions gérées par le système, c'est-à-dire les connexions initialement présentes $conxs$ auxquelles on ajoute notre nouvelle connexion $\Theta EffectiveConnection'$. D'après l'invariant du système (H_5), les connexions initialement présentes respectent toutes les trois sous-conditions $(P_2 a)$, $(P_2 b)$ et $(P_2 c)$.

Pour $(P_2 a)$, il nous reste donc à montrer que la connexion effective ajoutée vérifie bien :

$$\Theta EffectiveConnection'.source \in comps \wedge \Theta EffectiveConnection'.dest \in comps$$

ce qui est évident par (H_1).

Les conditions $(P_2 b)$ et $(P_2 c)$ (à prouver uniquement pour la nouvelle connexion ajoutée) concernent le typage des connexions. Pour les prouver, il nous faudrait disposer de plus d'informations sur le type effectif $T?$ de la nouvelle connexion; ce sera le rôle des schéma spécifiques associés aux primitives de connexion.

Ainsi, pour chaque primitive de connexion détaillée par la suite, nos obligations de preuve seront :

$$(P_I) \forall t : \Theta EffectiveConnection'.type \bullet \exists t_1 : \Theta EffectiveConnection'.source.out\_type \bullet t_1 \sim t$$

[restriction de $(P_2 b)$]

$$(P_{II}) \forall t : \Theta EffectiveConnection'.type \bullet \exists t_2 : \Theta EffectiveConnection'.dest.in\_type \bullet t_2 \succeq t)$$ [idem $(P_2 c)$]

$$(P_{III}) \forall t : \Theta EffectiveConnection'.type \bullet t \preceq type?$$ [contrat intentionnel]

Cette dernière condition (P_{III}) correspond à notre hypothèse de validité de contrat entre le type intentionnel et les types effectifs qui en sont inférés et qui doivent lui être "inférieures" dans la relation de sous-typage. Nous introduisons la variable $type?$ qui représentera, dans les primitives de connexion, le type intentionnel passé en paramètre.

En conclusion, à la condition préalable que les propriétés (P_I), (P_{II}) et (P_{III}) soient satisfaites, le schéma de promotion $AddConnection$ est cohérent, tant du point de vue local (schéma $EffectiveConnection$) que du point de vue global (schéma $AbstractOperationalSystem$) $\square$

A.2.4 Preuve de cohérence de l'opération *AbsConnectSpecialize*

Propriétés à prouver : d'après preuve A.2.3 page 231)

$$(P_I)\,\forall\, t : T! \bullet \exists\, t_1 : c_1?.out\_type \bullet t_1 \sim t$$
$$(P_{II})\,\forall\, t : T! \bullet \exists\, t_2 : c_2?.in\_type \bullet t_2 \succeq t$$
$$(P_{III})\,\forall\, t : T! \bullet t \preceq type?$$

Preuve : La propriété (P_I)[2] est facile à prouver, en nous rappelant que $T!$ est défini comme un sous-ensemble de types de T_1 :

$$T! = \{t : T_1 \mid (\exists\, t' : T_2 \bullet t \preceq t')\})\qquad \text{[par définition]}$$
$$\Rightarrow T! \subseteq T_1$$
$$\Rightarrow \forall\, t : T! \bullet \exists\, t_1 : T_1 \bullet t_1 = t\qquad \text{[prop d'appartenance]}$$
$$\Rightarrow (1)\,\forall\, t : T! \bullet \exists\, t_1 : T_1 \bullet t_1 \sim t\qquad \text{[prop de }\sim]$$

$$T_1 = tmax(\{t_1 : c_1?.out\_type \mid t_1 \preceq type?\})\qquad \text{[par définition]}$$
$$\Rightarrow (2)\, T_1 \subseteq c_1?.out\_type$$

$$(1) \wedge (2) \Rightarrow \forall\, t : T! \bullet \exists\, t_1 : c_1?.out\_type \bullet t_1 \sim t \Leftrightarrow (P_I)\qquad \text{[cqfd }\square]$$

Pour (P_{II})[3], nous pouvons obtenir notre conclusion également de façon très simple :

$$T! = \{t : T_1 \mid (\exists\, t' : T_2 \bullet t \preceq t')\})\qquad \text{[par définition]}$$
$$\Rightarrow (1)\,\forall\, t : T! \bullet \exists\, t_2 : T_2 \bullet t_2 \succeq t$$

$$T_2 = tmax(\{t_2 : c_2?.in\_type \mid t_2 \preceq type?\})\qquad \text{[par définition]}$$
$$\Rightarrow (2)\, T_2 \subseteq c_2?.in\_type$$

$$(1) \wedge (2) \Rightarrow \forall\, t : T! \bullet \exists\, t_2 : c_2?.in\_type \bullet t_2 \succeq t \Leftrightarrow (P_{III})\qquad \text{[cqfd }\square]$$

De même, (P_{III})[4] peut être facilement établie grâce à la propriété $T! \subseteq T_1$:

$$(1)\,\forall\, t_1 : T_1 \bullet t_1 \preceq type?\qquad \text{[def de }T_1]$$

$$T! \subseteq T_1 \wedge (1) \Rightarrow \forall\, t : T! \bullet t \preceq type? \Leftrightarrow (P_{III})\qquad \text{[cqfd }\square]$$

En conclusion, d'après notre cahier des charges[5], la primitive de connexion par spécialisation est cohérente □

A.2.5 Preuve de cohérence de l'opération *AbsConnectGenSource*

Propriétés à prouver :

$$(P_I)\,\forall\, t : T! \bullet \exists\, t_1 : c_1?.out\_type \bullet t_1 \sim t$$
$$(P_{II})\,\forall\, t : T! \bullet \exists\, t_2 : c_2?.in\_type \bullet t_2 \succeq t$$
$$(P_{III})\,\forall\, t : T! \bullet t \preceq type?$$

2. Tout type effectif de connexion est relié par sous-typage avec au moins un type sortant choisi à la source.
3. Tout type effectif de connexion est sous-type d'au moins un type entrant choisi à la destination.
4. Le type effectif de connexion ne contient que des sous-types du type intentionnel déclaré.
5. Ce cahier des charges "contient" trois preuves spécifiques (P_I), (P_{II}) et (P_{III}) à conduire, ainsi que la preuves de cohérence du schéma de promotion lors de l'ajout effectif de connexion.

Preuve : Pour (P_I), la preuve donnée pour la spécialisation n'est plus valide puisque les types effectifs sont désormais sélectionnés dans l'ensemble $c_2?.in\_type$. Cependant, les éléments de l'ensemble résultant possède la propriété de posséder un type comparable dans l'ensemble $c_1?.out\_type$. Cette propriété correspond formellement à (P_I):

$$\forall t : T! \bullet \exists t_1 : c_1?.in\_type \bullet t_1 \sim t \qquad [\text{CQFD } \square]$$

Le cas de (P_{II}) est trivial puisque cette propriété porte exclusivement sur T_2, ensemble non modifié pour la généralisation à la source.

La preuve de (P_{III}) proposée précédemment n'est plus valide. Pour l'agrémenter, nous pouvons remarquer que $T!$ est maintenant sous-ensemble de T_2. Nous pouvons en déduire la preuve suivante :

$$\begin{aligned} & T! = \{t : T_2 \mid (\exists t' : T_1 \bullet t \preceq t')\} && [\text{par définition}] \\ & \quad \Rightarrow (1) \forall t : T! \bullet \exists t_2 \in T_2 \bullet t \preceq t_2 \end{aligned}$$

$$\begin{aligned} & T_2 = tmax(\{t_2 : c_2?.in\_type \mid t_2 \preceq type?\} && [\text{par définition}] \\ & \quad \Rightarrow (2) \forall t_2 : T_2 \bullet t_2 \preceq type? \end{aligned}$$

$$\begin{aligned} & (1) \wedge (2) \Rightarrow \forall t : T! \bullet \exists t_2 \in T_2 \bullet t \preceq t_2 \preceq type? \\ & \quad \Rightarrow \forall t : T! \bullet t \preceq type? \Leftrightarrow (P_{III}) && [\text{cqfd } \square] \end{aligned}$$

Ceci conclut nos obligations de preuve pour le schéma *AbsConnectGenSource* □

A.2.6 Preuve de cohérence de l'opération *AbsConnectGenDest*

Propriétés à prouver :

$$\begin{aligned} & (P_I) \forall t : T! \bullet \exists t_1 : c_1?.out\_type \bullet t_1 \sim t \\ & (P_{II}) \forall t : T! \bullet \exists t_2 : c_2?.in\_type \bullet t_2 \succeq t \\ & (P_{III}) \forall t : T! \bullet t \preceq type? \end{aligned}$$

Preuve : La propriété (P_1) est toujours valide puisque la définition de $T!$ reste inchangée.

Pour (P_2), le cas est similaire puisque nous pouvons simplement remarquer que T_2 reste un sous-ensemble des types entrant à la destination, ce qui laisse inchangée la preuve donnée pour le cas de la spécialisation :

$$\begin{aligned} & T_2 = tmax(\{t_2 : c_2?.in\_type \mid t_2 \preceq type?\}) \cup tmin(\{t_2 : c_2?.in\_type \mid t_2 \succ type?\}) && [\text{par définition}] \\ & \quad \Rightarrow T_2 \subseteq c_2?.in\_type \end{aligned}$$

Puisque T_1 est identique au cas de la spécialisation, la preuve pour (P_{III}) est également valide pour la généralisation à la destination □

A.3 Modèle comportemental

A.3.1 Preuve de cohérence de l'opération *DefaultMetaReceive*

Propriété à prouver :

pre *DefaultMetaReceive*

où *DefaultMetaReceive* est défini page 121

Preuve :

$$\text{pre } \mathit{DefaultMetaReceive} \Leftrightarrow \text{pre } (\mathit{GetCommChannel} \land \mathit{ChannelInPromote}) \quad [\text{def de } \mathit{DefaultMetaReceive}]$$
$$\Leftrightarrow \text{pre } \mathit{GetCommChannel} \land \text{pre } \mathit{ChannelInPromote} \quad [\text{promotion libre}]$$

Nous sommes en présence de promotion libre car l'état global *DefaultBehavior* et le schéma de promotion *ChannelInPromote* ne posent aucune contrainte sur l'état local *CommChannel*.

Commençons par calculer les préconditions de l'opération locale :

$$\begin{aligned}
&\text{pre } \mathit{GetCommChannel} \Leftrightarrow \exists\, \mathit{CommChannel}' \bullet \mathit{GetCommChannel} && [\text{def de pre }]\\
&\Leftrightarrow \exists\, \mathit{CommChannel}' \bullet [\mathit{CommChannel};\ \mathit{CommChannel}';\ \mathit{event}! : \mathit{Event} \mid\\
&\quad \mathit{events} \neq \langle\rangle \land\\
&\quad \mathit{event}! = \text{last } \mathit{events} \land\\
&\quad \mathit{events}' = \text{front } \mathit{events} \land\\
&\quad \mathit{type}' = \mathit{type} \land\\
&\quad \mathit{id}' = \mathit{id}] && [\text{def de } \mathit{GetCommChannel}]\\
&\Leftrightarrow \exists\, \mathit{type}, \mathit{type}' : \mathbb{P}\, \mathit{TYPE} \bullet\\
&\quad \exists\, \mathit{events}, \mathit{events}' : \text{seq } \mathit{Event} \bullet\\
&\qquad \forall\, \mathit{event} : (\text{ran } \mathit{events}') \bullet (\exists\, t : \mathit{type}' \bullet \mathit{event}.\mathit{type}' \preceq t) \land\\
&\qquad \forall\, \mathit{event} : (\text{ran } \mathit{events}) \bullet (\exists\, t : \mathit{type} \bullet \mathit{event}.\mathit{type} \preceq t) \land\\
&\qquad \mathit{events} \neq \langle\rangle \land\\
&\qquad \mathit{event}! = \text{last } \mathit{events} \land\\
&\qquad \mathit{events}' = \text{front } \mathit{events} \land\\
&\qquad \mathit{type}' = \mathit{type} && [\text{defs de schéma et quantification et } \mathit{id} \text{ n'intervient pas}]\\
&\Leftrightarrow \exists\, \mathit{type} : \mathbb{P}\, \mathit{TYPE} \bullet\\
&\quad \exists\, \mathit{events} : \text{seq } \mathit{Event} \bullet\\
&\qquad (H_1)\, \forall\, \mathit{event} : (\text{ran } \mathit{events}) \bullet (\exists\, t : \mathit{type} \bullet \mathit{event}.\mathit{type} \preceq t) \land\\
&\qquad (H_2)\, \mathit{events} \neq \langle\rangle \land\\
&\qquad (P_1)\ \text{front } \mathit{events} \in \text{seq } \mathit{Event} \land\\
&\qquad (P_2)\, \forall\, \mathit{event} : (\text{ran}(\text{ front } \mathit{events})) \bullet (\exists\, t : \mathit{type} \bullet \mathit{event}.\mathit{type} \preceq t)\\
&&& [\text{points uniques et simplifications}]
\end{aligned}$$

Puisque *events* est une variable du type seq *Event*, la sous-séquence front *events* est du même type, donc (P_1) est vraie.

De plus, front *events* est une sous-séquence de *events* d'où :

$$(H_3)\, \text{ran}(\text{ front } \mathit{events}) \subseteq \text{ran } \mathit{events}$$

On en conclut :

$$(H_1) \land (H_3) \Rightarrow (P_2)$$

Il nous reste à calculer les préconditions de la partie promotion :

$$\begin{aligned}
&\text{pre } \mathit{ChannelInPromote} \Leftrightarrow \exists\, \mathit{DefaultBehavior}';\ \mathit{CommChannel}' \bullet \mathit{ChannelInPromote} && [\text{def de } \mathit{pre}]\\
&\Leftrightarrow \exists\, \mathit{in_channels};\ \mathit{in_channels}' : \mathbb{P}\, \mathit{CommChannel} \bullet\\
&\quad \exists\, \mathit{CommChannel};\ \mathit{CommChannel}' \bullet\\
&\qquad \Theta\, \mathit{CommChannel} \in \mathit{in_channels} \land\\
&\qquad \mathit{in_channels}' = \mathit{in_channels} \setminus \{\Theta\, \mathit{CommChannel}\} \cup \{\Theta\, \mathit{CommChannel}'\}\\
&&& [\text{invariant vide pour } \mathit{DefaultBehavior}]\\
&\Leftrightarrow \exists\, \mathit{in_channels} : \mathbb{P}\, \mathit{CommChannel} \bullet
\end{aligned}$$

$$\exists\, CommChannel;\ CommChannel' \bullet$$
$$(H_1)\Theta\, CommChannel \in in\_channels \wedge$$
$$(P_1) in\_channels \setminus \{\Theta\, CommChannel\} \cup \{\Theta\, CommChannel'\} \in \mathbb{P}\, CommChannel$$

[point unique pour in_channels']

Tout d'abord, l'opération de retrait $in\_channels \setminus \{\Theta\, CommChannel\}$ est consistante grâce à l'hypothèse (H_1). Il nous faut maintenant prouver que l'ensemble obtenu est bien un élément de $\mathbb{P}\, CommChannel$. Dans ce but, Nous pouvons simplement poser :

$$\Theta\, CommChannel \in CommChannel \Rightarrow (H_2)\{\Theta\, CommChannel\} \in \mathbb{P}\, CommChannel$$

[prop de Θ et des ensembles]

$$\Theta\, CommChannel' \in CommChannel \Rightarrow (H_3)\{\Theta\, CommChannel'\} \in \mathbb{P}\, CommChannel$$

[idem]

On peut alors dériver la propriété :

$$(H_1) \wedge (H_2) \wedge (H_3) \Rightarrow (P_1)$$

D'où pre $DefaultMetaReceive \Leftrightarrow true$, ce qui termine notre preuve de cohérence □

A.3.2 Preuve de cohérence de l'opération $DefaultMetaInQueue$

Propriété à prouver :

pre $DefaultMetaInQueue$

où $DefaultMetaInQueue$ est défini page 121.

Preuve :

$$\text{pre } DefaultMetaInQueue \Leftrightarrow \exists\, DefaultBehavior' \bullet DefaultMetaInQueue \qquad \text{[def de pre]}$$
$$\Leftrightarrow \exists\, in\_queue, in\_queue' : \text{seq}\, Event \bullet$$
$$\exists\, event? : Event \bullet$$
$$in\_queue' = \langle event? \rangle \frown in\_queue \qquad \text{[def des schémas et simplifications]}$$
$$\Leftrightarrow \exists\, in\_queue : \text{seq}\, Event \bullet$$
$$\exists\, event? : Event \bullet$$
$$\langle event? \rangle \frown in\_queue \in \text{seq}\, Event \qquad \text{[point fixe pour } in\_queue'\text{]}$$

On peut conclure assez facilement :

$$event? \in Event \Rightarrow (H_1)\langle event? \rangle \in \text{seq}\, Event \qquad \text{[prop des séquences]}$$
$$(H_2)\, \forall\, s_1;\ s_2 : \text{seq}\, Event \bullet s_1 \frown s_2 \in \text{seq}\, Event \qquad \text{[prop de } \frown\text{]}$$
$$(H_1) \wedge in\_queue \in \text{seq}\, Event \wedge (H_2) \Rightarrow$$
$$\langle event? \rangle \frown in\_queue \in \text{seq}\, Event \qquad \text{[CQFD } \square\text{]}$$

A.3.3 Preuve d'activabilité parallèle de $DefaultMetaReceive$ et $DefaultMetaInFetch$

Propriété à prouver :

pre $DefaultMetaReceive \wedge DefaultMetaInFetch$

avec :

- $DefaultMetaReceive$ défini page 121,
- $DefaultMetaInFetch$ défini page 121.

Preuve : On peut remarquer que les deux fonctionnalités "travaillent" sur des parties indépendantes du schéma d'état *DefaultBehavior* (défini page 116). En effet, *DefaultMetaReceive* impose des conditions sur *in_channels* et défini $in\_channels'$ alors que *DefaultMetaInFetch* ne référence que *in_queue* et in_queue'. Or, il n'existe aucune relation entre ces variables[6] donc les deux schémas sont indépendants, d'où :

$$\begin{aligned}
&\text{pre } DefaultMetaReceive \wedge DefaultMetaInFetch \\
&\Leftrightarrow \text{pre } DefaultMetaReceive \wedge \text{pre } DefaultMetaInFetch && [\text{indépendance des schémas}]
\end{aligned}$$

Or nous avons déja prouvé la cohérence de *DefaultMetaReceive* (preuve A.3.1); il ne nous reste donc plus qu'à calculer les préconditions de *DefaultMetaInFetch* :

$$\begin{aligned}
&\text{pre } DefaultMetaReceive \wedge DefaultMetaInFetch \Leftrightarrow \text{pre } DefaultMetaInFetch && [\text{de ce qui précède}] \\
&\Leftrightarrow \exists\, DefaultBehavior' \bullet DefaultMetaInFetch \\
& && [\text{def de pre }] \\
&\Leftrightarrow \exists\, in_queue, in_queue' : \text{seq}\, Event \bullet \\
&\quad \exists\, event! : Event \bullet \\
&\quad\quad \#in_queue > 0 \wedge \\
&\quad\quad event! = \text{last } in_queue \wedge \\
&\quad\quad in_queue' = \text{front } in_queue \wedge && [\text{def des schémas et simplifications}] \\
&\Leftrightarrow \exists\, in_queue : \text{seq}\, Event \bullet \\
&\quad \exists\, event! : Event \bullet \\
&\quad\quad \#in_queue > 0 \wedge \\
&\quad\quad \text{front } in_queue \in \text{seq}\, Event && [\text{point unique pour } in_queue']
\end{aligned}$$

Cette dernière condition est valide si, et seulement si, *in_queue* n'est pas l'ensemble vide, ce qui est confirmé par $\#in\_queue > 0$. □

A.3.4 Cohérence du raffinement d'état pour *DefaultComponent*

Propriété à prouver :

$$\begin{aligned}
&\forall\, DefaultBehavior' \bullet \\
&\quad InitDefaultBehavior \Rightarrow (\exists\, AbstractComponent' \bullet InitAbstractComponent \wedge DefaultComponent')
\end{aligned}$$

avec :

- *DefaultBehavior* défini page 116,
- *InitDefaultBehavior* défini page 120,
- *AbstractComponent* défini page 69,
- *InitAbstractComponent* $\widehat{=} \emptyset$ (pas d'initialisation dans le modèle),
- *DefaultComponent* défini page 129.

Preuve : Commençons par écrire la deuxième partie (à droite) de la propriété à prouver :

$$\begin{aligned}
&(P_1) \Leftrightarrow \exists\, AbstractComponent' \bullet InitAbstractComponent \wedge DefaultComponent' \\
&\Leftrightarrow \exists\, id' : IDENTIFIER;\ in_type', out_type' : \mathbb{P}\, TYPE;\ in_channels', out_channels' : \mathbb{P}\, CommChannel; \\
&\quad functs' : \mathbb{P}\, Functionality \bullet \\
&\quad\quad \forall\, inct : \{c : in_channels' \bullet c.type\} \bullet (\forall\, it : inct \bullet \exists\, t : in_type' \bullet t \succeq it) \wedge \\
&\quad\quad \forall\, outct : \{c : out_channels' \bullet c.type\} \bullet (\forall\, ot : outct \bullet \exists\, t : out_type' \bullet t \succeq ot) \wedge \\
&\quad\quad \forall\, t : in_type \bullet \exists\, fs : functs \bullet t \preceq fs.type
\end{aligned}$$

6. Dans une première version du modèle, une telle relation existait mais imposait une limite importante : l'impossibilité de déconnecter un composant si la file en entrée n'était pas vide pour le type considéré. Ceci montre bien l'intérêt de ne pas "sur-formaliser" le modèle.

Nous pouvons voir que les propriétés qu'il nous faut démontrer, dans le cadre du raffinement, se résument aux conditions portant simultanément sur les variables du schéma abstrait (*AbstractComponent*) et sur celles du schéma raffiné (*DefaultBehavior*). Ces conditions sont résumées dans le schéma de raffinement (*DefaultComponent*). Nous pouvons maintenant nous intéresser à la partie «gauche» de notre proposition de preuve :

$$
\begin{aligned}
&(P_R) \Leftrightarrow \forall\, DefaultBehavior' \bullet InitDefaultBehavior \Rightarrow (P_1)\\
&\Leftrightarrow (\forall\, in\_channels', out\_channels' : \mathbb{P}\, CommChannel;\ in\_queue', out\_queue' : \mathrm{seq}\, Event;\\
&\quad funct s' : \mathbb{P}\, Functionality \bullet\\
&\qquad in\_channels' = out\_channels' = \emptyset \wedge\\
&\qquad in\_queue' = out\_queue' = \langle\rangle) \Rightarrow (P_1)
\end{aligned}
$$

Du fait de leur initialisation en tant qu'ensembles ou séquences vides, toutes les références aux variables de *DefaultBehavior* peuvent donc être «retirées» de (P_1), de sorte qu'il ne nous plus finalement qu'une seule condition à valider pour satisfaire le raffinement :

$$(P_R) \Leftrightarrow \forall\, t : in\_type \bullet \exists\, fs : functs' \bullet t \preceq fs.type$$

Or, le contrôle de l'adéquation entre les fonctionalités proposées par le composant concerné et les types entrant de ce dernier est, comme nous l'avons vu, du ressort du concepteur du composant. Cette condition peut être vérifiée automatiquement durant la phase de traduction de Scope vers Java. Mais quoiqu'il arrive, rien ne pourra perturber cette propriété puisque les types comme les fonctionnalités de base des composants sont intrinsèques. Ceci clot notre débat sur la validité formelle, au sens de la méthode Z, de notre proposition de raffinement □

«En informatique, les invariants sont éphémères.»
Alan Perlis

B
Glossaire de la notation Z

Remarque : ce glossaire est n'est pas exhaustif et ne couvre que la notation utilisée dans cette thèse. Les notations spécifiques aux multi-ensembles, par exemple, ne sont pas décrites.

Noms

$a,b \ldots z$	identificateurs de variables, fonctions, etc.
$A,B \ldots Z$	ensembles/schémas

Définitions

$a == x$	abbréviation
$a ::= b$	définition de type libre
$[a, \ldots]$	introduction d'un ensemble donné
$a\_$	opérateur préfixe
$\_a$	opérateur suffixe
$\_a\_$	opérateur infixe

Nombres

$\mathbb{Z}$	ensemble des entiers relatifs
$\mathbb{N}$	ensemble des entiers naturels
$\mathbb{N}_1$	ensemble des entiers naturels non-nuls
$m + n$	addition
$m - n$	soustraction
$m * n$	multiplication
$m \operatorname{div} n$	division entière
$m \operatorname{mod} n$	modulo arithmétique
$m \leq n$	inférieur ou égal à
$m < n$	inférieur à
$m \geq n$	supérieur ou égal à
$m > n$	supérieur à
$m \mathrel{..} n$	intervalle

Logique

$true$	constante vrai
$false$	constante faux
$\neg\, p$	négation logique
$p \wedge q$	conjonction logique
$p \vee q$	disjonction logique
$p \Rightarrow q$	implication logique
$p \equiv q$	équivalence logique
$\forall\, X \bullet p$	quantification universelle
$\exists\, X \bullet p$	quantification existentielle
$\exists_1\, X \bullet p$	quantification existentielle unique
$\mathbf{let}\ a == x;\ \ldots \bullet p$	définition locale

Expressions ensemblistes

$x = y$	Egalité entre expressions
$x \neq y$	Inégalité
$x \in A$	appartenance
$x \not in A$	non-appartenance
$\emptyset$	ensemble vide
$A \subseteq B$	inclusion ensembliste (au sens large)
$A \subset B$	inclusion stricte
$\{x, y, \ldots\}$	ensemble d'éléments
$\{X \mid p \bullet x\}$	ensemble défini en intention (tous les x de X tels que p)
$\lambda\, X \bullet x$	fonction d'ensemble, forme lambda
$\mu\, X \bullet x$	valeur unique x de l'ensemble X (x est une expression)
$\mathbf{if}\ p\ \mathbf{then}\ x\ \mathbf{else}\ y$	conditionnelle
$(x, y, \ldots)$	n-uplet ordonné
$A \times B \times \ldots$	produit cartésien
$\mathbb{P}\, A$	ensemble des parties de A (ensemble des sous-ensembles)
$\mathbb{P}_1\, A$	ensemble des parties *non-vides* de A
$\mathbb{F}\, A$	ensemble des sous-ensembles finis
$\mathbb{F}_1\, A$	ensemble des sous-ensembles finis non-vides
$A \cap B$	intersection d'ensembles
$A \cup B$	union d'ensembles
$A \setminus B$	différence d'ensembles
$\bigcap A$	intersection généralisée d'un ensemble d'ensembles
$\bigcup A$	union généralisée d'un ensemble d'ensembles
$first\ x$	premier élément d'un couple ordonné
$second\ x$	deuxième élément d'un couple ordonné
$\# A$	cardinal d'un ensemble fini

Relations

$A \leftrightarrow B$	Relation ($\mathbb{P}(A \times B)$)
$a \mapsto b$	association (élément d'ensemble relationnel)
$\mathrm{dom}\, R$	domaine de relation
$\mathrm{ran}\, R$	co-domaine de relation
$\mathrm{id}\, R$	identité relationnelle
$R^{\sim}$	relation inverse
R^{*}	relation réflexive et transitive
R^{+}	relation transitive et non-réflexive
$A \nrightarrow B$	fonction partielle
$A \rightarrow B$	fonction totale
$f\ x$	application de fonction (ou $f(x)$)

Séquences

$\mathrm{seq}\, A$	ensemble de séquences finies
$\mathrm{seq}_1\, A$	ensemble de séquences finies non-vides
$\langle\rangle$	séquence non-vide
$\langle x, y, \ldots\rangle$	séquence $\{1 \mapsto x, 2 \mapsto y, \ldots\}$
$s \frown t$	concaténation de séquences
$\frown/\, s$	concaténation distribuée
$\mathrm{head}\ s$	premier élément d'une séquence ($s(1)$)
$\mathrm{tail}\ s$	reste d'une séquence
$\mathrm{last}\ s$	dernier élément d'une séquence
$\mathrm{front}\ s$	tout sauf le dernier élément d'une séquence
$s \upharpoonright A$	extraction de sous-séquence à partir d'un ensemble filtre

Schémas

$S \mathrel{\widehat=} [d \mid p]$	définition du schéma S, avec déclarations d et invariant p
$z.a$	sélection de composant de schéma (z est une instance de schéma)
ΘS	n-uplet caractéristique des composants de S
$\mathrm{pre}\ S$	précondition(s) de S
$\mathrm{inv}\ S$	invariant de S (extension de la notation)
$S \setminus (a, \ldots)$	cacher les composants $(a, \ldots)$ dans S
$S \wedge T$	conjonction de schémas (union des déclarations, conjonction des invariants)
$S \vee T$	disjonction de schémas (union des déclarations, disjonction des invariants)
$S \mathbin{;} T$	composition séquentielle de schémas (i.e. S *puis* T)
$S \gg T$	composition «en tube» (S envoi ces sorties vers T)
$S[a/b, \ldots]$	renommage des composants de S (b devient a, etc.)

Conventions

$a?$	entrée d'une opération
$a!$	sortie d'une opération
a	variable d'état avant opération
a'	variable d'état après opération
ΔS	modification de schéma d'état
ΞS	changement d'état sans modification