

# Behavioural Contracts for Components

Cyril Carrez

▶ **To cite this version:**

Cyril Carrez. Behavioural Contracts for Components. domain_other. Télécom ParisTech, 2003. English. NNT : . pastel-00000798

**HAL Id: pastel-00000798**
**https://pastel.hal.science/pastel-00000798**

Submitted on 6 Sep 2004

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.




# Thèse

**présentée pour obtenir le grade de docteur de**
**l'École Nationale Supérieure des Télécommunications**

Spécialité : Informatique et Réseaux


## Cyril Carrez


---

# Contrats Comportementaux pour Composants

---

**Soutenue le 15 Décembre 2003, devant le jury composé de :**

| | |
|---|---|
| Jean-Bernard Stefani | Président |
| Jean-Pierre Courtiat | Rapporteurs |
| Laurence Duchien | |
| Antoine Beugnard | Examinateurs |
| Marc Pantel | |
| Sylvie Vignes | |
| Elie Najm | Directeur de Thèse |

'Cos the only thing misplaced was direction
And I found direction.

Fish/Marillion
(Misplaced Childhood, 1985)

Celui qui te devance d'une nuit te devance d'une idée
El Nimour, 2003

# Résumé

La conception basée composants est une nouvelle méthode de construction d'applications et de systèmes distribués. Les composants sont des pièces de puzzle réutilisables que le concepteur d'applications assemble pour créer son logiciel. Les spécifications de ces composants sont alors utilisées lors de la vérification compositionelle de l'assemblage, pour assurer le bon déroulement de l'exécution de l'application résultante.

Cette conception par composition pose cependant plusieurs problèmes. Les services non uniformes impliquent que les messages échangés entre les composants le soient selon un dialogue bien précis, qui doit être respecté. Les liens d'interconnection entre les composants évoluent au gré de leur comportement : création, suppression et modification de ces liens ne sont pas forcément détectables à l'assemblage des composants, donc a priori non vérifiables. Survient alors le problème des «messages non compris», qui a pour conséquence que certaines requêtes peuvent ne pas être traitées par les composants.

Notre but est de fournir un cadre formel pour la vérification compositionelle des systèmes à base de composants.

Nous définissons un langage de type d'interfaces comportementales. Ce langage s'inspire des types réguliers, selon une sémantique de messages. Nous introduisons également la notion d'actions permises et obligatoires, qui imposent des contraintes non seulement au composant, mais aussi à son environnement. Ce langage constitue un *contrat comportemental* à partir duquel nous définissons deux vérifications.

La première vérification impose que le composant respecte son contrat d'interface. Nous avons choisi d'avoir une sémantique de composant la plus abstraite possible, pour étudier les contraintes minimales imposées par le contrat. Notre abstraction inclut les notions de ports, références vers ces ports, et tâches d'exécution. La dynamique de notre modèle repose essentiellement sur les notions d'état du port (émission/réception), de son activité (actif/suspendu), et les relations entre les ports (par exemple attente de résultat sur un autre port).

La deuxième vérification est utilisée lors de l'assemblage des composants, ou lors de leur déploiement sur une plate-forme d'exécution. Les contrats sont utilisés pour vérifier que deux interfaces interconnectées sont compatibles : compatibilité des services modalisés respectifs, qui se propage récursivement aux arguments des messages échangés. Cette vérification est faite *sans prendre en compte le comportement interne du composant* ; elle s'avère donc précieuse dans le cadre des composants sur étagères, où le composant n'arbore que sa spécification, et non son calcul interne.

Cette double vérification nous assure que l'application évoluera sainement : tout message envoyé sera consommé, et il n'y aura pas d'interblocage externe entre les composants.

**Mots-clés** : typage comportemental, vérification, composition, composants.

# Abstract

Component based design is a new methodology for the construction of distributed systems and applications. Components are reusable puzzle parts the creator of applications assembles to create his software. Specifications of those components are then used during compositional verification of the assembly, to ensure the good course of the execution of the resulting application.

This conception by composition gives rise to several problems. Non uniform services implies that messages are exchanged between components according to a well-defined dialogue, which must be honoured. Interconnection links between components evolve at the liking of their behaviour : creation, deletion and modification of those links are not necessarily determined at the assembly of the components, thus not verifiable. Then, the "message not understood" problem occurs, which consequence is that some requests may not be processed by the components.

Our goal is to provide a formal framework for compositional verification of component based systems.

We define a behavioural interface type language. This language takes inspiration from regular types, with message semantics. We also bring in the concept of allowed and obligatory actions, which enforces constraints on both the component and its environment. This language forms a *behavioural contract* from which we define two verifications.

The first verification imposes the component honours its interface contract. We chose a component semantics which is as abstract as possible, to study minimal constraints demanded by the contract. This abstraction encompasses concepts of ports, references towards those ports, and threads. The dynamic of our model primarily uses concepts of port states (sending/receiving), its activity (active/suspended), and relations between ports (for instance, waiting a result on another port).

The second verification is used during the assembly of the components, or during their deployment on an executing platform. Contracts are used to check that two interconnected interfaces are compatible : compatibility of respective modal services, which recursively propagates to arguments of exchanged messages. This verification is made *without taking into account the internal behaviour of the component* ; hence, it is precious when used with components off-the-shelf, where each component shows only its specification, not its internal calculus.

This double verification ensures that the application will evolve soundly : every sent message will be consumed, and there will be no external deadlock between components.

**Keywords** : behavioural typing, verification, composition, components.

# Remerciements

Je tiens à remercier les personnes qui m'ont offert l'opportunité d'effectuer cette thèse dans les meilleurs conditions.

Je remercie tout d'abord Elie Najm pour m'avoir dirigé dans mes travaux de recherche tout au long de cette thèse. Merci pour tes conseils avisés, tant scientifiques que pédagogiques, et pour la patience et la gentillesse avec lesquels tu les as prodigués. Enfin, et surtout, merci de m'avoir fait profiter de ta passion pour ce travail.

Mes remerciements les plus chaleureux vont également à Laurence Duchien et Jean-Pierre Courtiat pour avoir accepté la lourde de tâche de rapporteur. Merci pour vos précieux conseils et remarques, sans lesquels mon rapport ne serait pas ce qu'il est.

Je remercie également Jean-Bernard Stefani pour m'avoir fait l'honneur de présider le jury de cette thèse. J'adresse aussi mes remerciements les plus sincères à Sylvie Vignes, Antoine Beugnard, et Marc Pantel qui m'ont fait l'honneur et le plaisir d'accepter de siéger à mon jury. Merci pour l'intérêt que vous avez porté à mon travail.

Je remercie également Alessandro Fantechi, avec qui j'ai eu un plaisir immense à travailler, mais surtout qui a vu les premiers déboires de ce travail.

Mes premiers remerciements affectueux vont à Christian et Marie-José, alias Papa & Maman. Merci Papa de m'avoir lancé dans la longue et difficile voie de la recherche, et fait entrevoir la noble tâche de l'enseignement. Maman, merci pour tout le soutient moral ; «c'est tout ce que je peux faire» mais ces petites choses sont au final immenses. Merci encore à tous les deux pour les discussions du petit-déjeuner de fin de thèse, la relecture attentive de mon rapport, et plein d'autres choses encore.

Je remercie également mes frères & belles-sœurs Stéphane & Catherine et Yann & Céline pour leur soutient, leur intérêt pour cette aventure, et leur présence à la soutenance, qu'elle soit physique ou morale.

Un merci particulièrement chaleureux à Monique, alias Marraine, qui est venue de Bordeaux pour assister à ma soutenance ; j'ai été très touché de ta présence, alors que le sujet était bien loin de tes préoccupations.

Du fond du cœur, je remercie Alice et An'So pour les Discussions Fondamentales de la Vie, de l'Univers, et du Reste, mais aussi pour les affinités et la spiritualité que j'ai trouvées et garderai, je l'espère, à jamais. Elles m'ont été d'un grand secours dans les pires moments.

Parmi les collègues, je tiens à remercier chaleureusement les occupants du bureau C234-4 : Mai Trang, Wessam, Rami, Dany, Habib, Alexandre, Axel, et bien entendu celui qui est l'âme même de ce bureau, le Président, Khaled.

Je remercie aussi les (ex)doctorants avec qui j'ai partagé la vie du département,

ce qui donne dans un désordre le plus complet et avec les oublis : Thomas, Nadine & Sam, Abbas, Philippe, Robert, Ouahiba et Rola.

Il en est avec qui j'ai eu plus d'affinités, notamment autour des fameuses bières du Vendredi soir qui débouchent fatalement sur un resto ; j'ai nommé Arnaud le Faux (devenu Petit Suisse), Alexandre (le même), Richard (à qui je dois un post-doc en Norvège), Nico & Isabelle, Strop & Gaëlle, Romain & Cécile, Pascal & Alda, Tatiana, Arnaud, Jean, Talel (si si, y'a 5 minutes), et Elie ; parmi ces zappeurs de l'ENST, il me reste à remercier chaudement Jean-Phi, Intermittent de la Recherche$^{TM}$ et précurseur à ses heures, non seulement pour me rappeler comment sont les Français, mais aussi pour élargir ma culture de la bière, et parce que j'aurai le plus grand plaisir à partager une bière chez lui, en Belgique. A ces amis s'ajoutent les illustres zappeurs, qui faisaient parti du même groupe, à savoir Arnaud le Vrai, Krimo, Fernando & Céline, Loutfi, Cyril & Sophie, Frédéric et Sonia & Vania.

Je remercie également les membres du département Informatique et Réseaux, pour l'ambiance de travail des plus agréable ; surtout Hayette, Jean-Louis et Sophie, pour les bonjours qui mettaient toujours de bonne humeur lors des passages (très chaleureux) au secrétariat.

D'autres personnes ont suivi d'un peu plus loin toute cette histoire, et se sont toujours demandé comment serait la fin. Merci au plongeurs Violaine, Audrey (parce que je lui ai promis), Laurent, Alain, Thierry et Eric, au plaisir de faire des bulles ensemble avec les poissons.

Et non, Will, je ne t'ai pas oublié, ni les remerciements qui vont suivre et que je te dois en partie. Pour leur support durant l'accouchement long et difficile de cette satanée thèse, je tiens à remercier, sans ordre particulier (mais je mets quand même Will en premier) : Will (squatteur officiel) pour le Cragganmore, la découverte du Whisky[Jac00] et l'Ecosse, Clems & Gwen pour le Macallan, Bozo pour le Bowmore, Pierrot & Séverine pour le Glenfarclas 105, Cédric & Gaëlle pour le Laphroaig, Sam & Audrey pour l'Ardbeg et ce voyage en Ecosse, Tof' & Fabienne pour avoir fait un pot whisky à leur mariage, ce qui 1) m'a donné une idée pour mon pot de thèse et 2) m'a donné une occasion de me mettre en valeur auprès de la gente féminine («Y'a les blondes, et le Bowmore»). Un merci particulier pour Kolo, alias binôme, pou les mot de tro let (une ver de cet thé en mot de tro let a été réa ave le pac 3L$^{E}$T). Merci aussi à Isabelle & Gwenolé et Valère, pour me rappeler qu'il ne faut pas trop boire de whisky. Je remercie également D2, Slimo, Frizouille, Stéphane !!, Big, Jean, et bien d'autres de Telecom avec qui j'ai partagé des pauses parfois trop longues sur le continuum test, le plus intéressant du forum. Je terminerai cette grande clique d'ingénieurs par celles et ceux que j'ai croisés trop rapidement aux détours d'un couloir des Catacombes, j'ai nommé Doc (merci pour la visite), Flore, Carole & Olivier.

Je remercie également Florence, Laurence et Sandrine, même si elles ne connaîtront peut-être jamais la fin de l'épisode ; vous avez été des amies inoubliables, malheureusement les chemins se croisent et se décroisent au gré des événements.

Enfin, je ne terminerai pas ces remerciements sans une pensée pour mon bien aimé : vous l'avez deviné, mon monocycle ; il m'a ramené à la maison après bien des journées de labeur acharné, et m'a prodigué la meilleure détente juste après le travail. Ma petite roue, merci du fond du cœur.

*Je dédie cette thèse*
*à tous ceux qui m'ont soutenu moralement*
*pendant la rédaction de cet ouvrage.*
*Plus particulièrement à Papa & Maman,*

# Table des matières


**1 Introduction** **1**
- 1.1 Généralités . . . 1
- 1.2 Problématique . . . 3
- 1.3 Notre approche . . . 5
- 1.4 Plan . . . 7

**2 Composants et Typage** **9**
- 2.1 Définitions : composants et connecteurs . . . 10
  - 2.1.1 Consensus général . . . 10
  - 2.1.2 Modèle de composant UML2.0 . . . 11
  - 2.1.3 Modèle de composant ACCORD . . . 12
- 2.2 Les langages de description d'architecture (ADL) . . . 13
  - 2.2.1 Caractéristiques des composants et des connecteurs . . . 14
  - 2.2.2 Caractéristiques des configurations . . . 16
  - 2.2.3 Quelques ADL existants . . . 16
  - 2.2.4 Wright . . . 16
  - 2.2.5 Darwin . . . 18
- 2.3 Approches à base de composants . . . 18
  - 2.3.1 Ce que décrivent les spécifications . . . 19
  - 2.3.2 Les composants . . . 20
  - 2.3.3 L'environnement d'exécution . . . 23
  - 2.3.4 Déploiement de composant . . . 30
  - 2.3.5 Conclusion . . . 32
- 2.4 Spécifications formelles pour composants . . . 33
  - 2.4.1 Automates d'interface et supposition/garantie . . . 33
  - 2.4.2 Typage comportemental . . . 34
  - 2.4.3 Contrats . . . 36
- 2.5 Conclusion . . . 39
- 2.6 Etude de cas . . . 40

**3 Types d'interface** **43**
- 3.1 Présentation informelle autour d'un compte en banque . . . 43
- 3.2 Syntaxe du langage d'interface . . . 45
- 3.3 Sous-typage . . . 47
- 3.4 Règles de compatibilité . . . 50
- 3.5 Propriétés des sous-types . . . 51
- 3.6 Dualité d'interfaces . . . 53
  - 3.6.1 Dualité et références serveurs . . . 54
- 3.7 Etude de Cas . . . 55

3.7.1 Spécifications du composant Rapporteur . . . . . 56
3.7.2 Spécifications du composant Gestion . . . . . 57
3.7.3 Spécifications du composant Article . . . . . 58
3.7.4 Relations entre les types de l'étude de cas . . . . . 59
3.8 Conclusion . . . . . 59

**4 Modèle de composant 61**
4.1 Présentation informelle . . . . . 61
4.2 Notations pour les Composants . . . . . 63
4.2.1 Ports et Références . . . . . 63
4.2.2 Tâches . . . . . 63
4.3 Médium de Communication . . . . . 65
4.4 Sémantique des composants . . . . . 66
4.4.1 Configuration de Composants . . . . . 68
4.5 Etude de cas . . . . . 68
4.5.1 Diagramme d'état du composant Rapporteur . . . . . 70
4.5.2 Diagramme d'état du composant Gestion . . . . . 72
4.5.3 Diagramme d'état du composant Article . . . . . 74
4.5.4 Cas d'un port suspendu à un autre . . . . . 74
4.6 Conclusion . . . . . 74

**5 Respect de contrat 77**
5.1 Contrats et observateurs . . . . . 77
5.1.1 Règles de respect de contrat : transitions valides . . . . . 78
5.1.2 Règles de respect de contrat : cas d'erreurs . . . . . 80
5.1.3 Composant honorant un contrat . . . . . 82
5.2 Assemblage de composants . . . . . 82
5.2.1 Assemblage sain . . . . . 83
5.3 Propriétés . . . . . 83
5.3.1 Préservation de la compatibilité et propriété de consommation des messages . . . . . 83
5.3.2 Type d'une référence dans la configuration . . . . . 84
5.3.3 Absence d'interblocage externe . . . . . 86
5.3.4 Propriété de vivacité (sous contraintes) . . . . . 92
5.4 Etude de cas . . . . . 94
5.4.1 Vérification du composant Rapporteur . . . . . 94
5.4.2 Vérification du composant Gestion . . . . . 96
5.4.3 Vérification du composant Article . . . . . 98
5.5 Conclusion . . . . . 100

**6 Intégration des serveurs non réentrants 101**
6.1 Syntaxe et présentation informelle pour les serveurs non réentrants . 102
6.1.1 Syntaxe . . . . . 102
6.1.2 Graphe de dépendance des ports . . . . . 103
6.1.3 Gestion des liens dynamiques . . . . . 104
6.1.4 Assemblage sain . . . . . 105
6.2 Formalisation . . . . . 107
6.2.1 Modification des règles de respect de contrat . . . . . 107

6.2.2 Gestion des liens dynamiques . . . . . 110
6.2.3 Assemblage sain avec ports non réentrants . . . . . 111
6.3 Propriété : absence d'interblocage externe . . . . . 112
6.3.1 Démonstrations intermédiaires . . . . . 113
6.3.2 Démonstration d'absence d'interblocage externe . . . . . 115
6.4 Exemple : le dîner de philosophes . . . . . 118
6.5 Conclusion . . . . . 120

**7 Conclusion** **123**
7.1 Bilan des travaux . . . . . 123
7.1.1 Travaux similaires . . . . . 124
7.2 Travaux futurs . . . . . 126
7.2.1 Vérification de contrats . . . . . 127
7.2.2 Composant composite . . . . . 127
7.2.3 Profil UML . . . . . 129
7.2.4 Intégration aux plates-formes de composants . . . . . 129
7.3 Applications . . . . . 130

# Table des figures


1.1 Comparaison des concepts . . . . . 1
1.2 Vérifications à l'assemblage . . . . . 3
1.3 Envoi de référence . . . . . 3
1.4 Erreur : réception impossible ou message non compris . . . . . 4
1.5 Obligations entre les interfaces . . . . . 4
1.6 Un composant respecte son contrat . . . . . 5
1.7 Les contrats assurent que les interfaces sont compatibles entre elles . 6

2.1 Un composant CORBA . . . . . 21
2.2 Architecture répartie à composant : modèle d'interaction avec le composant . . . . . 24
2.3 Architecture répartie à composant : appel de méthode . . . . . 25
2.4 Appel de méthode métier : architecture CCM. . . . . 26
2.5 Appel de méthode métier : architecture EJB . . . . . 26
2.6 Appel de méthode métier : architecture .NET . . . . . 26
2.7 Invocation de la méthode `onMessage` d'un Bean *Message-Driven* . . 28
2.8 Co- et Contra-variance : évolution des propriétés lors du sous-typage 39
2.9 Exemple d'interaction : accès accordé, et entrée de la critique du rapporteur . . . . . 42
2.10 Exemple d'interaction : accès pour le rapporteur refusé . . . . . 42

3.1 Composant rapporteur . . . . . 56
3.2 Composant Gestion . . . . . 57
3.3 Composant Article . . . . . 58

4.1 Notation graphique pour les composants . . . . . 62
4.2 Un exemple de configuration . . . . . 62
4.3 Evolution des observations sur les ports . . . . . 64
4.4 Composant `Rapporteur` à l'assemblage . . . . . 70
4.5 Diagramme d'état du composant `Rapporteur` . . . . . 71
4.6 Composant `Gestion`, à l'assemblage . . . . . 72
4.7 Diagramme d'état du composant `Gestion` . . . . . 73
4.8 Composant `Article`, à l'assemblage . . . . . 74
4.9 Diagramme d'état du composant `Article` . . . . . 75
4.10 Exemple d'un port $g'$ qui se suspend à un port $c$ . . . . . 76

5.1 Exemple d'interblocage externe . . . . . 87
5.2 Vérification du composant `Rapporteur` . . . . . 95
5.3 Vérification du composant `Gestion` . . . . . 97

5.4 Vérification du composant `Article` . . . . . 99

6.1 Implantation possible d'un serveur non réentrant . . . . . 101
6.2 Assemblage et graphes de dépendances : risque d'interblocage . . . . 104
6.3 Relations de dépendances pour l'échange de référence . . . . . 105
6.4 Assemblage de composant : cycle avec un port non réentrant . . . . . 106
6.5 Port $s$ à risque : $s$ peut être suspendu au client $c$ qui peut requérir les services de $s$ . . . . . 112
6.6 Diagramme d'état (partiel) pour le composant `Philosophe` . . . . . 119
6.7 Dîner de philosophes : liens statiques . . . . . 120
6.8 Dîner de philosophes : liens non connus à l'assemblage . . . . . 121

7.1 Composant composite : délégation simple ($T \preceq S$) et multiple ($(T_a, T_b) \preceq S$)128

# Liste des tableaux

2.1 Caractéristiques des composants et des connecteurs . . . . . 15

3.1 Règles de sous-typage . . . . . 49

4.1 Règles pour la sémantique de composants . . . . . 67
4.2 Règles pour la configuration de composants . . . . . 68

5.1 Règles de respect de contrat (transitions valides) . . . . . 79
5.2 Règles de respect de contrat (cas d'erreurs) . . . . . 81

6.1 Règles de compatibilité composant-interface, avec les serveurs non réentrants . . . . . 108
6.2 Règles de compatibilité composant-interface : cas d'erreurs, avec les serveurs non réentrants . . . . . 109

Chapitre 1

# Introduction

## 1.1 Généralités

Les besoins croissants des utilisateurs ont engendré le développement d'applications de plus en plus complexes. Le résultat est que les concepts manipulés par la communauté informatique sont de plus en plus abstraits, mais en contrepartie les mécanismes sous-jacents sont de plus en plus complexes.

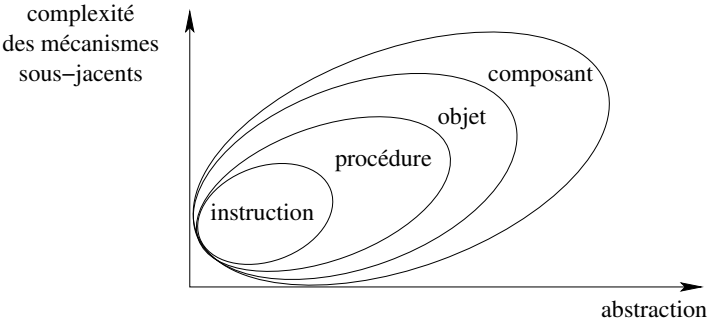



**Fig. 1.1:** *Comparaison des concepts*

La figure 1.1 résume l'évolution de ces concepts au cours des cinquante (ou plus) dernières années. L'instruction est le concept le plus simple et le plus proche de la machine. Affectations de valeurs à des variables, conditions d'exécutions et autres boucles se côtoient pour former l'algorithme du programme, la recette de cuisine qui donne le résultat attendu. Mais appréhender un algorithme d'un millier de lignes, en un seul coup d'œil, relève alors de l'exploit ; de plus la réutilisation est inexistante.

La procédure (ou fonction) permet de s'éloigner un peu plus du cœur de la machine : les instructions y sont regroupées selon une fonctionnalité précise. L'algorithme peut se réduire à quelques lignes, abstraction donnant les étapes principales de la recette de cuisine ; il n'est parfois même pas utile d'étudier les procédures pour comprendre la substance de l'algorithme. De plus, la portabilité est meilleure, car la procédure peut être réutilisée à d'autres endroits du programme, voire par d'autres programmes.

Toutefois, les procédures manipulent des données qui sont décrites à part ; les

programmes sont plus pensés en termes d'algorithmes que de données manipulées. Les objets, dans les années 80, ont recentré les concepts sur ces données : un objet est une représentation des données, avec des méthodes permettant de les manipuler. La conception des applications s'en est trouvée améliorée : un logiciel est décrit en termes de relations entre objets, non plus en termes d'opérations sur des données. Même si les concepts d'objets ont permis une abstraction supplémentaire et une réutilisation accrue, l'inconvénient est qu'ils sont plus complexes à mettre en œuvre que les concepts précédents. Des incohérences entre les différents langages ou compilateurs sont apparues[1]. L'avènement des objets a cependant permis de faire un bon en avant assez conséquent dans la conception des logiciels. D'une part, la réutilisation peut véritablement avoir lieu (même si concevoir dans le but de réutiliser demande encore un certain effort par rapport à concevoir dans le seul but d'utiliser). D'autre part, les objets ont facilité le développement de systèmes distribués.

Mais l'évolution dynamique du logiciel reste difficile : si le programmeur veut modifier un objet dans un exécutable, il doit soit recompiler entièrement ce dernier, soit mettre en œuvre des mécanismes complexes d'invocation dynamique d'interface (par exemple en utilisant la technologie CORBA). Ces dernières années, un pas supplémentaire a été franchi grâce aux composants. Les composants regroupent les objets dans des bouts de code (ou d'application), que l'on rassemble, connecte, pour donner le logiciel final. L'abstraction résultante n'a jamais été aussi élevée : l'application est vue comme des pièces d'un puzzle que l'on assemble. La réutilisation n'a jamais été aussi poussée : une application paramètre le composant pour s'en servir conformément à ses besoins spécifiques. Par contre, la complexité sous-jacente n'a jamais été aussi grande : apparaissent les notions de déploiement de composant, de conteneur, de services utilisés par le composant, etc. Même s'il était possible d'appréhender l'appel de méthode d'un objet dans ses moindres détails, il est plus difficile de connaître toutes les fonctionnalités engendrées lors de l'utilisation d'un service d'un composant (s'agit-il de composants CCM, EJB, .NET ? Sur quelle plate-forme des EJB : JOnAS ou JBoss ? ?). Mais l'avantage résultant est que l'application n'est plus conçue en termes de méthodes appliquées à un objet, mais véritablement de *services* fournis par le composant, pour les besoins de l'application. L'interaction avec un composant n'est pas à considérer comme un simple appel de procédure ou de méthode, mais plutôt comme un véritable dialogue avec le composant. En résumé, un modèle de composant s'approche plus du modèle mental du développeur que les autres.

L'application est alors issue de composants génériques, et d'un assemblage spécifique [Szy02]. En fait, dans le meilleur des mondes, le concepteur d'applications n'a plus qu'à choisir ses composants sur une étagère, régler les paramètres pour adapter les composants aux besoins de l'application, et les interconnecter. Mais cette conception par composition pose toutefois plusieurs problèmes, essentiellement dûs à l'évolution du système au cours de son exécution.

[1] Antoine Beugnard a comparé la sémantique de la liaison dynamique dans de nombreux langages objets, et a trouvé certaines disparités : `http://perso-info.enst-bretagne.fr/~beugnard/papiers/lb-sem.shtml`.

## 1.2 Problématique

La première chose à faire lors de l'assemblage est bien sûr de s'assurer que celui-ci est valide, c'est-à-dire que l'utilisation de chacun des composants se fait correctement selon leur spécification propre. Dans les autres concepts présentés précédemment, cela se faisait à l'aide d'un système de type : l'instruction ne peut additionner une chaîne de caractères et un entier, une procédure (ou une méthode d'un objet) ne porte que sur certains types de données ; éventuellement, des pré- et post-conditions renforcent les contraintes de type : l'utilisateur d'une procédure doit faire en sorte que la pré-condition soit vraie.

Concernant les composants, l'utilisation d'un service repose sur un protocole bien précis, qui peut être plus complexe qu'un simple appel de méthode suivi du retour contenant les résultats. Ce protocole d'interaction peut être spécifié à l'aide de pré- et post-conditions : par exemple «la méthode `open` doit avoir été appelée» est une pré-condition de la méthode `read`. Cependant, un protocole d'interaction peut tirer parti des types *comportementaux* : ces types permettent de spécifier des interfaces non-uniformes, c'est-à-dire qui peuvent changer durant l'exécution. Par exemple «la première méthode est `open`, puis suit une série de `read`, et `close` termine le dialogue» ; ce type est plus lisible qu'un ensemble de pré-conditions. De plus, la vision donnée par des pré- et post-conditions est en général restreinte à une seule opération ; les types comportementaux permettent de regrouper un ensemble d'opérations selon une logique qui est plus intuitive.

Un autre point important est que le typage comportemental facilite la vérification de la validité des liens à l'assemblage (par exemple le protocole d'un client est conforme à celui du serveur). Toutefois, un problème plus complexe survient lorsque l'application s'exécute : de nouvelles connections peuvent apparaître, sans qu'elles soient visibles à l'assemblage, et provoquer des erreurs non prévues.

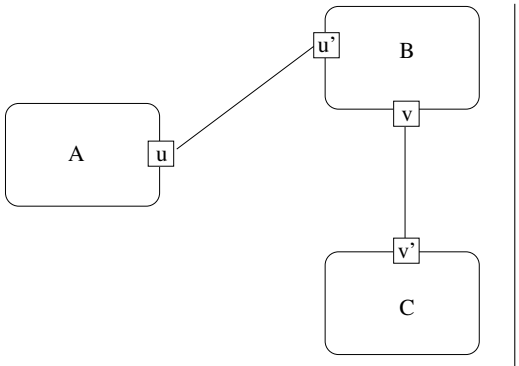



**Fig. 1.2:** *Vérifications à l'assemblage*

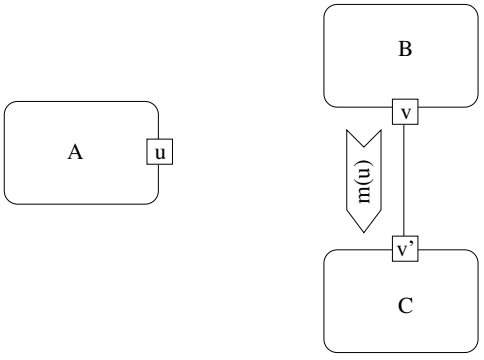



**Fig. 1.3:** *Envoi de référence*

Les figures 1.2 et 1.3 donnent l'exemple d'une application de trois composants $A, B, C$. La figure 1.2 représente l'application lors de son assemblage. Le composant $A$ contient le port[2] $u$, le composant $B$ a les ports $u'$ et $v$, et le composant $C$ a le port $v'$. Les ports $u$ et $u'$ sont reliés entre eux, de même que $v$ et $v'$ ; les vérifications de compatibilité de type sont faites uniquement à l'assemblage[3], et assurent que les

[2] Un port est un point d'accès à certains services du composant.

[3] Le type du port $u$ est compatible avec le type du port $u'$.

ports $u$ et $u'$ se comprendront sans risque d'erreur, de même que $v$ et $v'$. L'application s'exécute jusqu'à la figure 1.3 où le composant $B$ délègue au composant $C$ son interaction avec le composant $A$ : le port $v$ envoie au port $v'$ le message $m(u)$ qui contient la référence vers le port $u$. Le composant $C$ peut alors dialoguer avec le composant $A$, mais cela n'est plus possible pour $B$.

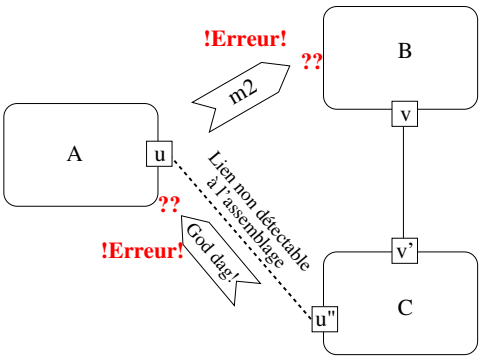



**Fig. 1.4:** *Erreur : réception impossible ou message non compris*

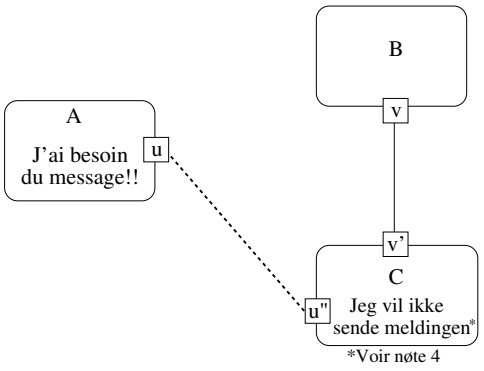



**Fig. 1.5:** *Obligations entre les interfaces*

Les figures 1.4 et 1.5 donnent deux exemples d'erreurs qui peuvent survenir suite à la création du lien entre $A$ et $C$ :

- $A$ ne peut plus envoyer de message à $B$ (il n'y a aucun port pour la réception de ce message) ;
- $C$ doit envoyer des messages qui peuvent être reçus par $A$, c'est-à-dire des messages que $A$ peut comprendre, et qui seront consommés.

Nous ne nous intéressons pas seulement au problème «message non consommé». La dernière figure (1.5) présente le problème «message non reçu» : le composant $A$ a besoin, pour son calcul interne, que $C$ envoie un message[4]. Nous désirons garantir cette réception, au moins du point de vue de la compatibilité des interfaces.

Plusieurs difficultés se présentent quant aux problèmes exposés :

- lorsque le concepteur crée une application, il prend des COTS (*Component off the Shelf*, composants sur étagère) et les assemble. Malheureusement, il ne connaît pas forcément le comportement interne de ces composants (comme le composant $C$ de l'exemple ci-dessus). Par contre, leurs spécifications lui permet de s'assurer que son assemblage est correct, et que son application ne fera pas d'erreurs ;
- nous ne voulons effectuer qu'une seule vérification : à l'assemblage. Ensuite, chaque composant s'exécute selon certaines règles, sans pour autant provoquer des erreurs. Dans cette optique, l'envoi de référence vers un port est une difficulté supplémentaire : il provoque une modification de la configuration des composants, au travers de liens qui ne sont pas visibles à l'assemblage donc non vérifiables. Ces liens sont modifiés soit pour *débuter* un dialogue entre deux ports, mais aussi pour *continuer* un dialogue comme dans l'exemple ci-

[4] J'ai dû me mettre au norvégien( !) pour comprendre le comportement de $C$ et vérifier cet exemple : *God dag* veut dire «Bonjour» et *Jeg vil ikke sende meldingen* veut dire «Je n'enverrai pas le message». Le composant $C$ ne respecte pas la contrainte imposée par $A$.

dessus. Cette dynamique des liens est classique en $\pi$-calcul, mais nous semble assez originale dans le monde des composants ;
- la vérification doit être automatique, et rapide : cela nous permettra de déployer des composants alors que l'application est en cours d'exécution.

Les modèles de composants existants, ainsi que les Langages de Description d'Architecture, proposent des outils de formalisation de l'assemblage de composants. Cependant, jusqu'à maintenant, le typage des interfaces du composant se faisait en donnant le nom des méthodes et leurs arguments. Le typage comportemental ajoute la mise en séquence des méthodes, ou même des messages échangés. Il permet alors de spécifier les protocoles de communication, mais il a actuellement des limites. Par exemple, peu d'approches comportementales prennent en compte les modalités des actions (en termes d'action permise ou obligatoire) ; ces modalités ont l'avantage d'imposer des contraintes à la fois sur le composant et sur son environnement, ce qui rend ainsi la vérification *compositionelle*. De plus, la vérification de composition prend souvent en compte les relations entre les ports : nous cherchons à vérifier uniquement la compatibilité de deux ports communicants, sans connaître les dépendances internes entre les ports. Enfin, la gestion des liens dynamiques (création, modification) n'est pas toujours présente.

## 1.3 Notre approche

Nous définissons un langage d'interface qui s'inspire des types comportementaux de [NNS99a] et de la logique déontique [von51, MW93] (avec la notion d'actions permises ou obligatoires). Ce langage spécifie le protocole d'interaction des interfaces du composant, créant ainsi un contrat comportemental.

Ce contrat est doublement utilisé :
- le composant doit respecter le contrat qui lui est associé (voir figure 1.6). Nous définissons une sémantique de composant et des règles permettant de vérifier, à partir de cette sémantique et de notre langage d'interface, que le composant respecte son contrat. Cette vérification est statique, et peut se faire à la compilation du composant.

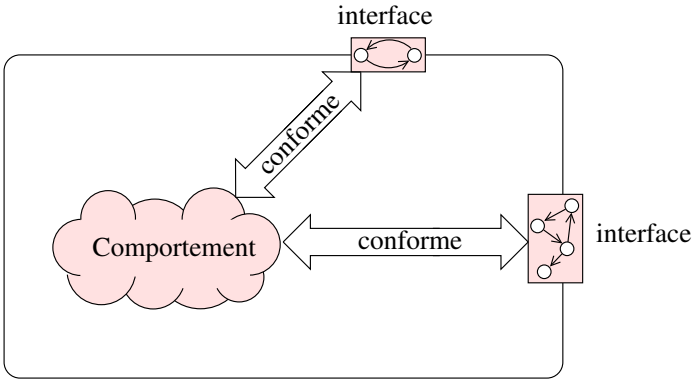



**Fig. 1.6:** *Un composant respecte son contrat*

Concernant le comportement interne du composant, nous n'avons pas de langage à proprement parler : nous avons une sémantique qui se veut suffisamment abstraite pour vérifier le respect de contrat. Elle se borne essentiellement aux envois et réceptions de messages sur les interfaces, et aux liaisons internes entre interfaces (par exemple une interface attend la réception d'un message sur une deuxième interface appartenant au même composant) ;

– l'assemblage est tel que les interfaces communicantes sont compatibles, comme le montre la figure 1.7. Chaque contrat définissant le type de chacune des interfaces du composant, la vérification revient à montrer que deux interfaces communicantes ont des types compatibles : les émissions de l'un correspondent aux réceptions de l'autre. Cette vérification peut se faire lors du déploiement du composant. Le comportement du composant n'entre pas du tout en ligne de compte lors de cette vérification.

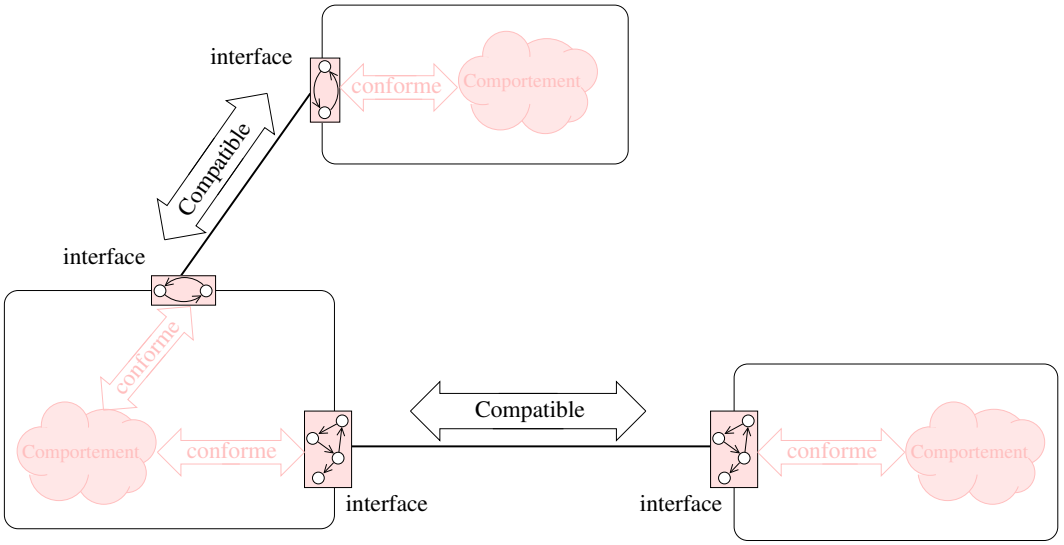



**Fig. 1.7:** *Les contrats assurent que les interfaces sont compatibles entre elles*

Cette double vérification permet à l'application d'évoluer sainement : tout message envoyé sera consommé, et il n'y aura pas d'interblocage entre les composants. Notre méthode est bien entendu dans la mouvance des vérifications compositionnelles de Abadi et Lamport [AL93, AL95].

---

Une partie de notre travail a été publiée aux conférences FORTE 2003 [CFN03a] et CFIP 2003 [CFN03b].

Cette thèse a été financée par le projet RNRT Accord et le projet européen Mikado. Le projet Accord (Assemblage de Composants par Contrats en environnement Ouvert et Réparti[5]) propose un cadre d'analyse et de conception dédié à l'assemblage de composants permettant la construction fiable d'applications. Le projet

[5] `http://www.infres.enst.fr/projets/accord/`.

MIKADO (*Mobile Calculi based on Domains*[6]) construit un modèle formel de programmation basé sur la notion de domaines, qui supporte la mobilité de programmes dans un environnement fortement distribué. Il fourni les bases mathématiques à une norme pour l'informatique répartie dans les systèmes ouverts.

## 1.4 Plan

Le chapitre 2 donne un état de l'art selon deux axes :

- nous donnons la définition généralement admise d'un composant (au travers de UML2.0 et du projet ACCORD), puis nous abordons les Langages de Description d'Architecture (ADL). Nous passons également en revue les trois approches à base de composant les plus connues : CCM, EJB et .Net ;
- le deuxième axe porte sur les spécifications formelles du composant. Dans un premier temps, nous abordons les automates d'interfaces sous l'angle de suppositions/garanties (suppositions faites sur l'environnement, garanties que l'on donne à l'environnement), puis le typage comportemental avec types réguliers (pour typer les ports) et algèbres de processus (pour typer les processus). Dans un deuxième temps, nous abordons la spécification par contrat.

La fin du chapitre décrit informellement l'étude de cas qui servira d'illustration pour les chapitres suivants. Cette étude de cas concerne la relecture des articles soumis à une conférence.

Le chapitre 3 définit le langage d'interface que nous adoptons pour typer les ports d'un composant. Il est basé sur les types réguliers, une sémantique de messages, et les notions d'actions permises (**may**) et obligatoires (**must**). Ce chapitre définit également une relation de sous-typage, et une relation de compatibilité de types. La deuxième relation est telle qu'un message envoyé est obligatoirement reçu par le type compatible. Nous étudions également les propriétés intéressantes relativement à ces deux relations.

Le chapitre 4 porte sur la sémantique de composant. Y sont définies les notions de ports, références vers ces ports, et tâches d'exécution. La dynamique du composant porte essentiellement sur une abstraction de l'état du port (émission/réception, actif/suspendu), et les relations entre les ports d'un même composant.

Les relations entre le langage d'interface et la sémantique du composant sont présentées dans le chapitre 5. Des règles permettent de vérifier que le composant respecte le contrat écrit dans le langage du chapitre 3 page 43. Nous définissons alors ce qu'est un assemblage sain, et prouvons les propriétés issues de cet assemblage.

Enfin, le chapitre 6 propose une extension qui porte sur les serveurs non réentrants. La syntaxe de notre langage y est légèrement modifiée, de même que les règles de respect de contrat. Tout comme le chapitre 5, nous définissons l'assemblage sain et les propriétés résultantes. L'abstraction de cet assemblage est cependant plus faible : nous avons besoin de raffiner la spécification du composant ; les dépendances entre les ports deviennent visibles, et les mouvements de références à l'intérieur du composant sont extrapolées à partir de la structure des types des ports de ce composant.

Le chapitre 7 conclut ce manuscrit et dresse les principales perspectives autour de travaux futurs et d'applications possibles.

[6] `http://mikado.di.fc.ul.pt/`.

Chapitre 2

# Composants et Typage

Les principales difficultés rencontrées lors de la conception de logiciel, comme la gestion de complexité, de la conformité ou de l'interopérabilité, ne trouvent que des réponses partielles dans la programmation orientée objet ou les intergiciels (*middleware*). Pour mieux surmonter ces difficultés, une réponse possible est de définir le plan de construction du logiciel, correspondant à la définition d'une architecture du système. La conception selon une approche orientée composant [Szy02] consiste alors à voir une application comme une collection d'unités de programmation indépendantes, interconnectées à l'aide d'une plate-forme d'exécution. Il est dans ce cas plus facile de gérer les éventuelles évolutions : la modification de l'application revient à modifier son plan de construction.

Plusieurs notions sont mises en jeu dans cette conception :

- les unités de programmation que sont les composants. Principalement deux vues existent : le comportement interne du composant (son code), et une description sur la façon dont il doit être utilisé (sa spécification) ;
- la manière dont les composants sont interconnectés, à savoir quel composant interagit avec quel autre, ce qui pose des problèmes de vérification de compatibilité entre composants. Cette interconnexion peut aller du simple trait à l'utilisation de connecteurs qui permettent d'adapter les interfaces communicantes ;
- le support d'exécution, c'est-à-dire le lieu où le composant va s'exécuter.

Ce chapitre pose des bases pour le reste du manuscrit. La première section donne un état de l'art rapide sur la notion de composant logiciel telle qu'elle est perçue à travers les méthodes de modélisation. Ensuite sont abordés les langages de description d'architecture, puis les plates-formes de composant les plus en vogue. La section suivante concerne les différentes solutions pour formaliser la spécification du composant : typage (classique, comportemental, à base de suppositions/garanties) et contrats. La dernière section décrit de manière informelle l'exemple qui illustrera les notations introduites dans les chapitres suivants.

## 2.1 Définitions : composants et connecteurs

### 2.1.1 Consensus général

Selon Clemens Szyperski [Szy02], «un composant logiciel est une unité de composition avec des interfaces spécifiées contractuellement et des dépendances de contexte explicites. Un composant logiciel peut être déployé indépendamment et est sujet à la composition par des tiers».

Cette définition intrinsèque du composant est à rapprocher de celle plus extrinsèque de Bertrand Meyer [Mey00], qui définit les composants comme des logiciels «orientés clients». Un composant est un élément de programme qui peut être utilisé par d'autres éléments de programme, qui sont alors clients du composant ; il s'agit ici du concept de composition logicielle. De plus, les clients n'ont pas besoin d'être connus par les auteurs du composant : celui-ci doit être d'un intérêt suffisamment général pour adresser un large rayon de «clients» qui ne sont pas en contact direct avec les auteurs de ce composant. Un composant est donc développé indépendemment de la configuration du produit final.

Concrètement, le consensus général est qu'un composant comporte au moins deux parties :

- le code exécutable du composant, qui met en œuvre les services proposés ;
- une spécification, qui décrit les interfaces (avec leur type) et les propriétés du composant (contraintes, propriétés non fonctionnelles).

La spécification est utilisée au niveau modélisation pour construire l'application ; elle sert aussi lors du déploiement composant (la mise en place du code du composant sur le support d'exécution) : par exemple un descripteur de déploiement est une spécification qui indique la façon dont le composant doit être déployé (les services dont il a besoin, son cycle de vie, ...).

L'interconnexion entre les composants peut se faire par l'intermédiaire de connecteurs. Ces derniers modélisent de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui gouvernent ces interactions. Par exemple, un appel de procédure, l'accès à une variable partagée, ou un protocole d'accès à des bases de données avec un système de gestion de transactions. Tout comme le composant, il y a deux descriptions pour le connecteur : son interface et son implantation.

Le reste de cette section décrit deux modèles de composants. Le modèle de composant UML2.0, et le modèle de composant du projet RNTL Accord. Le modèle de composant UML2.0 est très récent. Il est issu de UML (*Unified Modeling Language*) qui était déjà, dans sa version 1.5 [OMG01], un langage de modélisation largement utilisé dans l'industrie. De plus, la notation présente l'intérêt de pouvoir être modifiée pour intégrer des modèles abstraits spécifiques. C'est l'approche qui est en général effectuée dans les différents projets ayant besoin d'une modélisation particulière du problème traité. Un exemple est le projet RNTL Accord, dans lequel une partie de cette thèse a eu lieu, et qui a proposé un modèle de composants sur la base de UML2.0.

Il est à noter que parallèlement à la version 1.0 de UML, la méthodologie Catalysis [DW98] a vu le jour en 1998 ; ses auteurs, Desmond D'Souza et Alan Wills,

ont d'ailleurs participé à la formalisation de UML1.0. Catalysis est une méthode de conception d'applications à base de composants abstraits ; elle fournit un processus de développement d'applications plus complet, et plus précis que UML. La méthodologie est basée sur le principe de raffinement, de types (pour spécifier la sémantique d'un objet), et de collaborations entre objets (pour spécifier les interactions entre ces objets) ; l'utilisation de canevas (ou *frameworks*) facilite la conception par patron d'interaction. La méthode Catalysis permet de typer les interfaces des composants, mais est beaucoup plus riche, car elle propose tout un modèle de conception. Nous nous intéressons plus à la vérification des assemblages de composants, qui est plus à notre portée de complexité. Le processus de développement de Catalysis (avec les raffinements et les collaborations entre composants) pourra permettre d'approfondir notre travail d'un point de vue modélisation.

### 2.1.2 Modèle de composant UML2.0

La notation UML est formalisée par l'OMG (*Object Management Group*). Elle est basée sur le principe du MOF (*Meta-Object Facility* [OMG02b]) : un méta-modèle est fait à l'aide de diagrammes de classes, et décrit les éléments manipulés par le modèle (classes, relations, états...). Quatre plans existent : instance, modèle, méta-modèle et méta-méta-modèle ; chacun est décrit par rapport au suivant (le modèle décrit les instances ; le méta-modèle décrit les concepts utilisés dans le modèle), et le méta-méta-modèle se décrit lui-même, en plus de décrire le méta-modèle. Il est alors possible d'adapter la notation UML pour ses propres besoins, à l'aide de profils UML qui indiquent les modifications apportées (en général) au méta-modèle.

Les composants étant un aspect de plus en plus important, notamment avec les dernières plates-formes réparties (voir section 2.3 page 18), l'OMG a doté la nouvelle version de son langage de modélisation d'un modèle de composant. Le méta-modèle de composant de UML 2.0 [OMG03b] permet de définir les spécifications des composants, ainsi que l'architecture de l'application que l'on désire développer.

UML2.0 spécifie un composant comme étant une unité modulaire, réutilisable, qui interagit avec son environnement par l'intermédiaire de points d'interactions appelés ports. Les ports sont typés par les interfaces : celles-ci contiennent un ensemble d'opérations et de contraintes ; les ports (et par conséquent les interfaces) peuvent être fournis ou requis. Le comportement interne du composant ne doit être ni visible, ni accessible autrement que par ses ports. Enfin, le composant est voué à être déployé un certain nombre de fois, dans un environnement a priori non déterminé lors de la conception (excepté au travers des ports requis).

Il existe deux types de modélisation de composants dans UML2.0 : le composant basique et le composant composite (ou *packagé*). La première catégorie définie le composant comme un élément exécutable du système. La deuxième catégorie étend la première en définissant le composant comme un ensemble cohérent de parties (appelées *parts*, chacune représentant une instance d'un autre composant).

La connexion entre les ports requis et les ports fournis se fait au moyen de connecteurs. Deux types de connecteurs existent : le connecteur de délégation et le connecteur d'assemblage. Le premier est utilisé pour lier un port du composant composite vers un port d'un composant situé à l'intérieur du composant composite (donc pour relier par exemple un port requis à un autre port requis). Le deuxième type de

connecteur est utilisé pour les liens d'assemblage (donc relier un port requis à un port fourni). Les connecteurs sont vus comme des moyens d'assemblage et d'adaptation ; en réalisant cette connexion entre les ports (peut-être incompatibles), le connecteur offre l'avantage d'effectuer les assemblages sans modification des composants.

Plusieurs projets basent leur modèle de composant sur UML2.0. la section suivante décrit le modèle du projet dans lequel une partie de cette thèse a eu lieu.

### 2.1.3 Modèle de composant ACCORD

Nous reprenons dans cette section les définitions du projet RNTL ACCORD (Assemblage de Composants par Contrats en environnement Ouvert et Réparti[1]). Ce projet propose un cadre d'analyse et de conception dédié à l'assemblage de composants permettant la construction fiable d'applications. Nous utiliserons une partie des concepts, essentiellement la notion de composant et de contrat sémantique.

Le projet ACCORD définit le composant comme «une entité logicielle qui réalise des interactions avec d'autres composants via des connecteurs», décrite selon trois niveaux [ACC02] :

**le type du composant,** caractérisé par un nom, un ensemble d'interfaces regroupant des opérations, un ensemble de valeurs et un ensemble de propriétés ;

**la classe de composant** qui correspond à l'implantation de ce dernier. Un composant possède des ports auxquels sont attachées des interfaces du service du type de composant. Un port (et aussi les interfaces associées) peut être offert ou requis ;

**l'instance de composant** qui est le résultat de l'activation du composant.

Les connecteurs réalisent une interaction entre composants, et sont utilisés pour connecter les ports des composants entre eux. Les points d'accès aux connecteurs sont de deux types :

**un port** (optionnel) permet d'accéder au connecteur comme s'il s'agissait d'un composant, notamment pour le configurer (par exemple pour affiner des paramètres de qualité de service). Le projet ACCORD parle alors de «connecteur complexe» ;

**une prise** sera reliée à un port d'un composant ou d'un connecteur, et utilisée pour assurer l'interaction entre les composants. Elle dispose d'interfaces (comme le port), mais qui ne sont pas nécessairement explicitées[2] ; la prise peut alors être vue comme un port générique ou abstrait, qui sera spécialisé lorsqu'il sera relié à un port.

La différence entre un connecteur et un port se situe essentiellement dans la fonctionnalité du connecteur ; cependant, la frontière entre les deux reste assez floue : une entité de gestion de vote peut aussi bien être considérée comme un composant à part entière que comme un connecteur réalisant une interaction particulière. Le projet ACCORD définit plus formellement un connecteur comme ayant au moins une prise.

[1] `http://www.infres.enst.fr/projets/accord/`

[2] Par exemple, «squelette» et «souche» sont des prises.

Composants et connecteurs évoluent dans une structure d'accueil (ou un environnement d'exécution). Le composant y est d'abord installé et configuré ; les instances s'y exécuteront par la suite.

Le projet ACCORD définit les contrats [FLA03] qui décrivent les propriétés de ces éléments (composants, connecteurs, interfaces, ports, prises et opérations). Ils sont organisés en trois axes, ou *dimensions* :

**le contrat syntaxique** définit les expressions bien formées qui sont utilisées pour accéder à un composant (par exemple, la signature d'une méthode) ;

**le contrat sémantique** définit la signification de ces expressions. Par exemple des expressions OCL (*Object Contraint Language* [OMG03a], langage de contraintes de UML), ou des contraintes de qualité de service, etc ;

**le contrat pragmatique** définit les propriétés de l'environnement du composant (par exemple ordre des invocations, ou propriétés épistémiques).

Comme le composant est défini selon trois niveaux (type, classe, instance), les trois dimensions de contrats se retrouvent à chacun de ces niveaux (respectivement contrat abstrait, d'implantation et d'exécution). La compatibilité des différents types de contrats permet d'assurer l'assemblage correct des composants et des connecteurs. Un composant résultant d'un tel assemblage est un composant composite.

La granularité des définitions précédentes est trop forte dans le cadre de notre étude. Par exemple, le connecteur permettant de faire interagir les composants est trop spécifique pour une première approche de notre travail. Nous préférons nous concentrer sur la vérification du bon fonctionnement d'un assemblage de composants, notamment du point de vue des contrats sémantiques. Nous ne garderons de ces définitions que la notion de composant[3], port, contrat[4] et assemblage.

## 2.2 Les langages de description d'architecture (ADL)

Les premiers travaux pour répondre aux besoins de description d'architecture logicielle ont été menés dès le début des années 90. Sont apparus alors les langages de description d'architecture, ou ADL (*Architecture Description Language* [SG96]). L'idée est de fournir une structure de haut niveau de l'application plutôt que l'implantation dans un code source spécifique [Ves93]. Un ADL est donc une notation pour la conception architecturale des applications.

De nombreux ADL ont émergé ces dix dernières années, comme par exemple Wright [AG96, All97], Rapide [Tea97] ou Darwin [MDEK95, MDK94, NK95]. Chacun propose une architecture à sa manière, les uns privilégiant plutôt les éléments de l'architecture et leur assemblage structurel, d'autres s'orientant plutôt vers la configuration de l'architecture et la dynamique du système. Face à cet amalgame de solutions, et ce manque de consensus sur la définition précise d'un ADL, Medvidovic et Taylor en ont donné une classification [MT00]. Ils définissent tout d'abord

[3] Notre modèle de composant couvre le type de composant (description de ses interfaces) et la classe du composant (pour la vérification des contrats).

[4] Toutefois nous ne distinguerons pas la notion de contrat pragmatique de celle de contrat sémantique.

les trois concepts inhérents aux ADL : composants, connecteurs et configuration ; ils comparent ensuite une dizaine d'ADL existants. Nous reprenons dans cette section une grande partie de leur article. Une étude plus approfondie des ADL est aussi donnée par [SD02] dans le cadre du projet ACCORD ; on y trouvera notamment un exemple détaillé pour chaque ADL.

Les trois concepts des ADL que définissent Medvidovic et Taylor sont les suivants :

**le composant** est une unité de calcul ou de stockage. Il peut être simple ou composé[5], et sa fonctionnalité peut aller de la simple procédure à une application complète. Le composant est en fait considéré comme un couple spécification-code : la spécification donne les interfaces, les propriétés du composant ; le code correspond à la mise en œuvre de la spécification par le composant ;

**le connecteur** modélise un ensemble d'interactions entre composants. Cette interaction peut aller du simple appel de procédure distante aux protocoles de communication (par exemple un système de vote dans lequel le connecteur récupère les votes et diffuse le résultat). Tout comme le composant, le connecteur est un couple spécification-code : la spécification décrit les rôles des participants à une interaction ; le code correspond à l'implantation du connecteur. Cependant, la différence avec le composant est que le connecteur ne correspond pas à une unique, mais éventuellement à plusieurs unités de programmation[6] ;

**la configuration** de l'architecture défini les propriétés topologiques de l'application : connections entre composants et connecteurs, mais aussi, selon les ADL, des propriétés de concurrence, de répartition, de sécurité, etc. La topologie peut être dynamique, auquel cas la configuration décrit la topologie ainsi que son évolution (liens entre composants et connecteurs, propriétés...).

Plusieurs caractéristiques agrémentent ces trois concepts, que nous développons dans les sections suivantes.

Un quatrième concept est toutefois abordé dans [MT00], il s'agit des outils associés aux ADL. Bien que nous n'en donnions pas les détails, nous pouvons indiquer que leurs caractéristiques permettent au concepteur d'application de travailler sur la base des trois concepts précédents. Les outils fournissent donc des aides à la conception, des vues multiples de l'architecture, et des fonctionnalités d'analyse, de raffinement ou de compilation.

### 2.2.1 Caractéristiques des composants et des connecteurs

Composants et connecteurs partagent six caractéristiques globales : l'interface, le type, la sémantique, les contraintes, l'évolution (du moins les possibilités fournies par l'ADL), et les propriétés non fonctionnelles. Ces caractéristiques sont définies dans le tableau 2.1 page suivante, avec la signification dans le cas précis des composants ou des connecteurs.

[5] On parle alors de composant *composite*.

[6] Typiquement, le connecteur peut représenter une communication à travers le réseau : la souche client, le squelette serveur, et la couche transport.

| caractéristique : | signification des caractéristiques pour : **composant** | **connecteur** |
|---|---|---|
| **interface**<br>ensemble des points d'interaction avec l'extérieur. | services fonctionnels du composant : les services offerts et requis par celui-ci. | mécanismes de connexion entre composants. Certains ADL décrivent ces interactions comme des rôles. |
| **type**<br>abstraction des fonctionnalités, en vue de leur réutilisation. | abstraction des fonctionnalités fournies par le composant. | abstraction des mécanismes de communication, de coordination ou de médiation entre composants. |
| **sémantique**<br>abstraction du comportement, qui permet de spécifier les aspects dynamiques et les contraintes de l'architecture. Le modèle sémantique assure aussi des projections cohérentes entre les niveaux d'abstraction de l'architecture. | abstraction des fonctionnalités du composant (et qui seront utilisées par l'application). | abstraction des protocoles d'interaction. |
| **contraintes**<br>propriétés ou assertions qui doivent être vérifiées sous peine d'incohérence du système (qui devient alors inacceptable). | limites d'utilisation du composant et des dépendances intra composants. | limites d'utilisation du protocole d'interaction mis en œuvre par le connecteur et des dépendances intra connecteurs. |
| **évolution**<br>un ADL doit permettre l'évolution des éléments de l'architecture, donc permettre la modification de leurs propriétés. En général l'évolution est assurée par des techniques de sous-typage ou de raffinement. | évolution de l'interface, du comportement, de l'implantation du composant. | évolution de l'interface et du comportement du connecteur. Permet notamment de faire évoluer les protocoles d'interaction. |
| **propriétés non fonctionnelles**<br>les propriétés fonctionnelles permettent de spécifier les aspects non fonctionnels liés à la sécurité, la performance, la portabilité, etc. La signification ne diffère pas entre les propriétés fonctionnelles du composant et celles du connecteur. | | |

**Tab. 2.1:** *Caractéristiques des composants et des connecteurs*

### 2.2.2 Caractéristiques des configurations

Un ADL doit fournir les possibilités de configuration suivantes :

**des spécifications compréhensibles,** c'est-à-dire que la syntaxe du modèle topologique doit être simple et intuitive. La structure de l'application peut dans ce cas se comprendre à partir de sa configuration, sans rentrer dans les détails des composants et connecteurs. De plus, cela facilite la communication entre les différents partenaires d'un projet ;

**la composition,** ou plus précisément la possibilité de décrire les architectures à différents niveaux de détail de composition. On parle alors de composition hiérarchique, dans laquelle le composant *primitif* est une unité non décomposable, et le composant *composite* est composé de composants (composites ou primitifs). La composition hiérarchique est utilisée dans les approches descendantes par raffinements successifs ;

**le raffinement et la traçabilité** fournissent un cadre sûr pour la composition (une description de plus en plus détaillée). La traçabilité permet de garder trace des changements successifs entre les différents niveaux d'abstraction ;

**l'hétérogénéité** est une caractéristique importante dans le cadre du développement de grands systèmes et de la réutilisation de l'existant. Un ADL doit donc être capable de spécifier une architecture indépendamment des supports utilisés (langage de programmation et de modélisation, système d'exploitation) ;

**la mise à l'échelle** permet la réalisation d'applications complexes et dont la taille peut devenir importante ;

**l'évolution** de la configuration pour qu'elle puisse prendre en compte de nouvelles fonctionnalités et faire évoluer l'application sous-jacente. Cela se traduira essentiellement par la possibilité d'ajouter, de retirer ou de remplacer des composants ou des connecteurs ;

**l'aspect dynamique** de l'application concerne les modifications *en cours d'exécution*, au contraire de l'évolution où les changements sont effectués *en atelier* (*off-line*) ;

**les contraintes** décrivent les dépendances dans la configuration, et s'ajoutent à celles des composants et connecteurs ;

**les propriétés non fonctionnelles** que l'on ne peut exprimer individuellement sur un composant ou connecteur.

### 2.2.3 Quelques ADL existants

Parmi les ADL qui existent, les plus proches de notre travail sont les ADL Wright [AG96, All97] et Darwin [MDEK95, MDK94, NK95].

### 2.2.4 Wright

Wright est un langage de spécification d'architecture issu de la thèse de Robert J. Allen [All97]. Il repose sur les trois concepts de la classification des ADL (composant, connecteur et configuration), ainsi qu'un concept de style. La notion de style

décrit des propriétés communes à un ensemble de configurations (cette notion est à rapprocher de celle des *frameworks* de Catalysis).

Les descriptions des composants et connecteurs contiennent deux parties :

- l'interface, c'est-à-dire les ports du composant (déclarés avec **port**), et les 'prises' des connecteurs, qui sont considérés comme des rôles (déclarés avec **role**) ;
- le calcul, c'est-à-dire le comportement du composant. Cette partie est déclarée avec **computation** pour le composant, et **glue** pour le connecteur.

Ces deux parties sont décrites en CSP [BHR84], et fournissent l'enchaînement des événements émis (par exemple «response**!**r») ou reçus («request**?**u»). Un composant *proxy* peut se décrire comme suit :

**Component** Proxy
**Port** s = request**?**u → response**!**r → s □§
**Port** c = request**!**u → response**?**r → c ⊓ §
**Computation** = s.request**?**u → c.request**!**u → c.response**?**r→ s.response**!**r
→ **Computation** □§

Ce composant possède deux ports : *s* et *c*. Le port *s* reçoit les requêtes et envoie les réponses à un client. Le port *c* est utilisé pour déléguer la requête à un autre composant. La partie **Computation** rend la délégation explicite (requête reçue sur *s*, envoi d'une requête sur *c*, attente de la réponse, puis envoi de cette réponse sur *s*). Un point faible cependant concerne les parties «⊓§» et «□§» de la description des comportements. «request**!**u → ... ⊓ §» représente le choix non déterministe, ou interne : le composant choisit entre le comportement qui envoie la requête, et le comportement «§» qui représente le processus qui termine sans erreur. «request**?**u → ... □§» représente le choix déterministe, ou externe : l'environnement décide l'action à exécuter. Ces spécifications sont obligatoires lorsque le port ne sera pas attaché. Nous pensons qu'elles alourdissent la syntaxe, et rendent la lecture du type plus difficile pour le programmeur débutant.

Les connecteurs sont toujours utilisés pour connecter deux ports de deux composants. Plusieurs tests sont effectués pour vérifier la validité de la configuration. Wright utilise la spécification CSP des ports et des rôles pour vérifier entre autres leur compatibilité.

La spécification des ports des composants est très proche de notre travail. Cependant, les modalités ne sont pas présentes, ou complexes à utiliser : la syntaxe «request**!**u → ... ⊓ §» indique par exemple que l'envoi de la requête n'est pas obligatoire. Nous proposons une syntaxe qui est plus intuitive.

De plus, lors de l'assemblage de composants, nous supprimons la partie comportementale : la spécification de nos composants ne comporte pas les parties **Computation** ou **Glue** de Wright. Ce langage les utilise en fait pour vérifier le comportement de configurations de composants : par exemple absence d'interblocage. Nous proposons plutôt de ne vérifier que la compatibilité entre les ports connectés, mais en contre partie nous introduisons des contraintes au niveau du comportement du composant pour assurer l'absence d'interblocage. Nous pensons que cette approche permet une vérification plus rapide que celle de Wright.

### 2.2.5 Darwin

Darwin est un langage de configuration : il favorise la description de la configuration de l'architecture au travers des interactions entre les composants. Son support d'exécution est appelé Regis [MDK94], et la sémantique des composants est basée sur le $\pi$-calcul [Mil93].

Un composant Darwin est décrit par une interface qui contient les services fournis et requis, en termes de messages émis et reçus. La déclaration du composant *proxy* est donnée par :

```
component proxy {
    provide   s <entry request, response>
    require   c <port client>
}
```

*s* est une interface fournie qui accepte des appels entrants avec une requête de type *request* et une réponse de type *response*. Le port *c* est quant à lui un port requis.

Darwin permet l'analyse du comportement d'un assemblage de composants. L'architecture utilise un analyseur de systèmes de transitions étiquetées (LTSA, *Labelled Transition System Analyser*) et une sémantique de processus à états finis (FSP, *Finite State Processes*). Une implantation et des outils de démonstration sont disponibles à `http://www-dse.doc.ic.ac.uk/concurrency/`.

Par exemple, le comportement du composant proxy est définit comme suit :

$$\text{Proxy} = \text{s.request} \rightarrow \text{c.request} \rightarrow \text{c.response} \rightarrow \text{s.response} \rightarrow \text{Proxy}$$

Tout comme Wright, nous désirons éviter d'avoir recours au comportement du composant pour assurer que l'exécution d'une configuration de composants se déroulera sans erreurs.

Une des points positifs de Darwin est qu'il permet l'instanciation de composant. Elle se fait selon deux modes : l'instanciation dynamique crée le composant avec des paramètres d'initialisation, tandis que l'instanciation paresseuse crée le composant lors du premier appel vers son instance. Nous ne nous sommes pas intéressés à cette dynamique de l'assemblage ; il serait intéressant d'approfondir cette voie en s'inspirant de l'architecture Darwin. Par contre, les composants Darwin ne sont pas multi-tâches, au contraire de notre modèle. De plus, nous considérons des liens dynamiques (avec échange de références sur les ports), qui ne sont pas présents dans les ADL.

## 2.3 Approches à base de composants

Dans cette section, nous passons en revue les trois approches à base de composant du moment, à savoir les composants Corba (CCM) de l'OMG, les composants EJB de Sun, et les composants .NET de Microsoft ; ces approches sont souvent appelées *serveurs d'applications*. Les points communs sont la distinction de différentes catégories de composant (par exemple avec ou sans état), l'utilisation de fabrique de composants et de délégation, des services prédéfinis, et enfin un modèle de déploiement utilisant des descripteurs XML.

### 2.3.1 Ce que décrivent les spécifications

La version 3.0 de CORBA [OMG02a] comporte un modèle de composant, le CORBA *Component Model* (CCM). La spécification[7] CCM date de Août 2002, et est proposée par l'OMG (*Object Management Group*). Quatre modèles de composant (ainsi qu'un méta-modèle que nous ne présenterons pas) existent pour les composants CCM :

- le modèle abstrait permet de décrire la spécification du composant : le composant et sa maison (qui fournira un accès au composant) sont décrits en IDL (*Interface Definition Language*) ;
- le modèle de programmation permet de définir la coopération entre le composant et sa structure d'accueil ; il définit les relations entre l'implantation du composant (les exécuteurs) et les spécifications du modèle abstrait. Quatre catégories de composant et l'environnement de programmation y sont également décrits ;
- le modèle de déploiement définit un processus d'installation de composants sur différents sites ;
- le modèle d'exécution définit l'environnement d'exécution des composants (avec les services associés, comme la transaction ou la persistance).

Différents rôles d'utilisateurs dans le CCM sont également définis (comme le concepteur de composant, ou l'implanteur). Étant basée sur CORBA, cette architecture de composant permet d'utiliser n'importe quel langage d'implantation.

La spécification EJB (*Enterprise JavaBeans*$^{TM}$) est proposée par Sun Microsystems, et décrit une architecture pour composants distribués développés en Java. La version 2.0 a vu le jour en Août 2001 [SUN01] ; elle couvre trois types de composants (*Entity*, *Session* et *Message-Driven*), avec les cycles de vie, modèles d'appels et descripteur de déploiement correspondant, ainsi que les services de base que le serveur d'applications EJB doit fournir (transaction, sécurité et persistance). L'interopérabilité entre serveurs d'applications est basée sur CORBA/IIOP, ce qui autorise l'accès à un composant EJB à partir d'un client CORBA. Enfin, la spécification décrit les différents rôles impliqués dans le développement d'applications reposant sur les EJB (comme la conception de composant, leur assemblage...).

L'architecture .NET a été proposée par Microsoft en 2000 [`http://www.microsoft.com/net/`]. Il est difficile de décrire exactement ce qu'est .NET : de premier abord, il s'agit d'un regroupement d'applications Microsoft, peut-être mis en place pour contrer les autres plates-formes de composant. Cependant, il est plus juste de penser l'architecture .NET comme une stratégie orientée selon deux points :

- une évolution du modèle de composant COM+ ; cependant le concept de «composant .NET» n'est pas clairement défini ;
- fournir une plate-forme de développement Windows, qui soit ouverte à tout type de langage.

L'architecture de Microsoft a donné le jour à deux spécifications à l'organisation de standardisation internationale ECMA (*European Computer Manufacturers Association*) : le langage C♯ (ECMA-334 [ECM02b]) et l'infrastructure CLI (*Common*

[7] On préférera plutôt [MMC02], qui présente le modèle de composant CORBA de manière plus pédagogique.

*Language Infrastructure*, ECMA-335 [ECM02a]). Ces spécifications ont ensuite été soumises à l'ISO/IEC JTC 1[8] qui a sorti en Avril 2003 les normes ISO/IEC 23270 (C♯) et ISO/IEC 23271 (CLI).

L'architecture .NET est une plate-forme multi-langage (les applications peuvent être écrites en n'importe quel langage, comme avec CORBA), et peut-être multi-environnement (il existe des implémentations libres sous Linux, comme Mono [Xim], ou sous FreeBSD, comme Rotor [Mic]). .NET s'articule autour de quatre concepts :

**CTS (*Common Type System*),** qui est un système de type. Il comprend les types et leurs constructions autorisées, mais aussi les liaisons inter-langages (comme l'héritage) ;

**CLS (*Common Language Specification*)** qui définit les propriétés qu'un langage doit vérifier pour inter-opérer avec la plate-forme .NET (c'est-à-dire pour développer avec le langage choisi, sur la plate-forme ou en inter-opérant avec d'autres langages) ;

**CLR[9] (*Common Language Runtime*).** C'est l'environnement d'exécution de la plate-forme .NET. Il permet de mettre en œuvre le «code géré» (*Managed Code*) : chargement du code, vérifications de sécurité, *garbage collecting*... Une bibliothèque de classes est définie, et correspond à un ensemble d'interfaces de programmation (API Windows) ;

**des méta-données** pour décrire les types et les référencer.

Le modèle de composant dans .NET n'est pas aussi bien décrit dans les spécifications de l'ECMA que les EJB ou CCM. Cependant, divers articles provenant du site Web de Microsoft sont plus lisibles (*An Introduction to Microsoft .NET Remoting Framework* et *Microsoft .NET Remoting : A Technical Overview*, accessibles à partir de la page `http://msdn.microsoft.com/netframework/using/understanding/networking/remoting/default.aspx`), ou encore les livres spécialisés dans le domaine [Sma03].

### 2.3.2 Les composants

La norme CCM est la seule à avoir un modèle abstrait de composant, avec une représentation claire et précise des relations entre le composant et son environnement (avec notamment la notion des services que fournit ou requiert le composant). Dans la norme EJB, un composant correspond à un *Bean*, pour lequel seuls les services fournis sont déclarés sous un regroupement de méthodes ; les services requis d'un EJB ne sont pas définis aussi clairement que pour un composant CCM. L'architecture .NET de Microsoft est bien plus floue quant à la notion de composant : aucune norme n'existe concernant la définition d'un «composant .NET» ; d'ailleurs, la littérature technique parle plutôt d'«Objet Distant» (*Remote Object*), même si cet objet peut être considéré par l'environnement d'exécution comme du code binaire.

Les spécifications décrivent plusieurs types de composants. Généralement, le type de composant résulte d'une combinaison de deux axes : persistance du composant et son état – avec ou sans état. **Avec Etat** indique que l'état du composant est mémorisé

[8] *International Organization for Standardization, Internatiuonal Electronical Commission, Joint Technical Committee.*

[9] ou VES (*Virtual Execution System*) dans le standard CLI.

entre deux appels de méthodes. La **Persistance** rend compte que le composant est sauvegardé sur disque pour pouvoir le récupérer après un redémarrage du serveur d'application ; la persistance est en général liée à la notion de **Clé primaire**, qui est l'identité du composant : elle est obligatoire si le client veut récupérer un composant spécifique, et n'a d'utilité que si le composant est persistant.

Les sections suivantes portent sur la spécification des composants et leurs différents types. Pour qu'un composant soit de tel ou tel type, sa classe d'implantation doit dériver d'une classe abstraite décrivant le type de composant recherché.

Les sections 2.3.3 page 23 et 2.3.4 page 30 se consacreront à l'architecture d'accueil et au déploiement des composants (plus précisément à leur descripteur de déploiement).

#### 2.3.2.1 Composant CCM

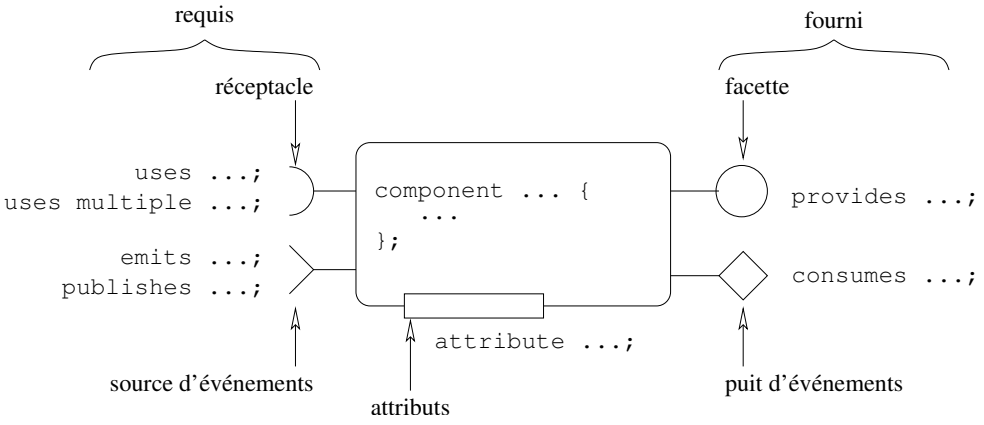



**Fig. 2.1:** *Un composant* Corba

Un composant Corba est communément représenté selon la figure 2.1. Il est décrit en IDL3 (*Interface Definition Language* pour la version 3 de Corba), et comporte plusieurs types de ports que l'on peut interconnecter entre eux :

| | fourni | requis |
|---|---|---|
| synchrone (`interface`) | **facette** ensemble de méthodes implantées par le composant. | **réceptacle** ensemble de méthodes requises par le composant. Deux types existent : les réceptacles simples sont connectés à une seule référence de facette, tandis que les réceptacles multiples peuvent être connectés à plusieurs références de facette. |
| asynchrone (`eventtype`) | **puits d'événement** consomme des événements. | **source d'événement** produit des événements. Deux types existent : diffusion un vers un ou un vers $n$. |

Chaque port fourni du composant correspond à l'implantation d'une interface, et a sa propre référence (référence de facette/de puits). Il est de plus possible d'accéder aux attributs du composant pour le configurer.

Trois interfaces (facettes) sont implantées par tous les composants, et permettent d'accéder aux différents ports proposés par le composant :

- `Components::Navigation` permet d'accéder aux facettes ;
- `Components::Receptacles` permet de (dé)connecter les composants entre eux de façon dynamique ;
- `Components::Events` permet d'accéder aux sources et puits d'événements (avec notamment des opérations de (dés)abonnement à une source d'événements).

Quatre catégories de composants existent, selon que l'on considère le mode d'accès (modèle d'utilisation CORBA), et l'existence d'une clé primaire :

| **Modèle d'usage** | **Persistance** | **Avec Etat** | **Clé primaire** | **Catégorie de composant** |
|---|---|---|---|---|
| Sans état | Non | Non | Non | Service |
| Conversationnel | | Oui | | Session |
| Durable | Oui | Oui | Non | Processus |
| | | | Oui | Entité |

Les modèles d'usage sont des patrons d'interactions définis par la norme ; «Durable» veut dire que le composant est persistant. La différence entre les catégories «Processus» et «Entité» est que la persistance de la première est transparente pour le client.

Le type du composant est décrit dans le descripteur de déploiement correspondant. La mise en œuvre des composants se fait dans n'importe quel langage de programmation, pourvu que la projection de IDL3 vers celui-ci soit définie.

#### 2.3.2.2 Composant EJB

Les composants de cette section sont appelés *Enterprise Java Beans* ou plus communément *Beans*. Chaque composant est en fait une classe Java proposant un ensemble de méthodes. Ainsi, contrairement à CCM, le composant EJB n'a pas de notion de ports ou d'interfaces asynchrones (événements). Les interfaces requises ne sont pas déclarées explicitement, mais si la classe Java fait appel à un autre composant EJB, elle doit importer la spécification des interfaces d'accès à ce composant (par le mot-clé `import` – c'est-à-dire que la classe Java utilise les méthodes d'autres classes, ce que l'on pourrait considérer à la rigueur comme une déclaration d'interface requise).

La spécification des EJB permet d'avoir au total quatre catégories de composants. Ils sont regroupés en trois types :

***Entity Bean*** (Entité). Composant EJB utilisé essentiellement pour l'accès à une base de données. Le Bean Entité peut être utilisé par plusieurs clients à la fois (mais il n'y a pas d'appels en parallèle : le conteneur doit les mettre en séquence). Ce bean possède une identité (ou clé) qui lui offre des possibilités de persistance ;

***Session Bean*** (Session). Composant EJB s'exécutant à la demande d'un unique client (qui ouvre une *session* avec le composant). Le Bean Session peut utiliser les services de Transaction du serveur EJB, mais n'est pas persistant. Il existe deux types de ce Bean :

**Statefull** (avec état), pour lequel un seul client à la fois peut avoir une session ouverte ;

**Stateless** (sans état), pour lequel plusieurs clients peuvent se connecter, mais de façon séquentielle.

**Message-Driven Bean.** Composant EJB fonctionnant comme consommateur d'événements asynchrones. Il peut utiliser les services de Transaction, et n'est pas persistant. Il ne possède pas d'état. Le Bean Orienté-Messages est le seul à avoir un mode d'appel différent des autres composants ; ce mode d'appel sera détaillé dans la section 2.3.3[10].

#### 2.3.2.3 Composant .NET

Les composants .NET sont appelés Objets Distants (*Remote Objects*). Ils ont plusieurs interfaces spécifiques, et peuvent être décrits dans n'importe quel langage. Un mécanisme d'introspection permet de naviguer parmi les services proposés par le composant (ce mécanisme est mis en œuvre par la bibliothèque `System.Reflection` de *C♯*). Trois types de composants existent, suivant le mode d'appel et les cas avec/sans état :

***Client Activated Objects.*** Ce composant est activé à l'initiative du client (une seule instance par client). Le composant a un état, et un contexte gérés par l'environnement d'exécution. Le cycle de vie du composant est géré par un système de bail (*lease*) ;

***Server Activated Object.*** Ce composant est activé à l'initiative du serveur. Une instance peut être partagée entre plusieurs clients. Contrairement au *Client Activated Object*, il n'y a pas de contexte lié au composant.

Deux types existent :

**Single Call,** où le composant n'a pas d'état, et ne sert qu'à une seule requête ;

**Singleton,** où le composant a un état. Comme plusieurs clients accèdent à la même instance, cet état est partagé entre les clients ; ce composant est donc utile pour le partage de données entre clients.

Contrairement aux composants CCM et EJB, il n'y a pas de notion de composant persistant. Il n'est pas possible d'avoir une persistance gérée par l'environnement (à la EJB ou CCM) ; mais il est possible, par du code adéquat[11], de retrouver cette spécificité.

### 2.3.3 L'environnement d'exécution

Dans cette section nous passons en revue les environnements d'exécution définis dans les architectures EJB, CCM et .NET. La première partie décrit les principes généraux, les suivantes visent chacune une architecture particulière.

[10] La gestion des messages se fait par JMS (Java Message Service).

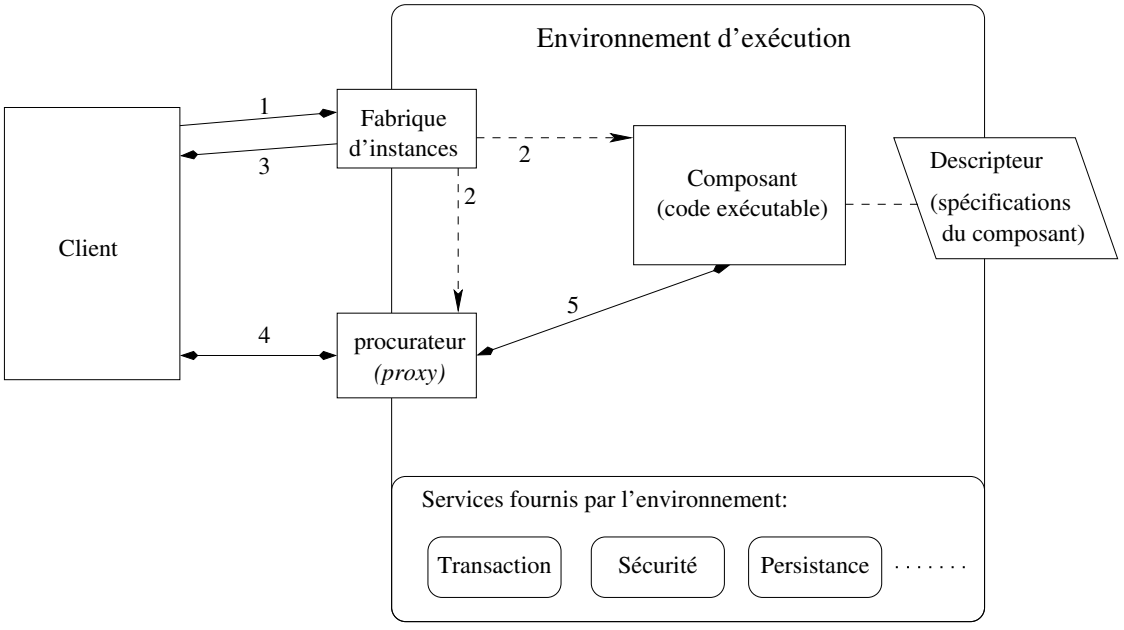
[11] En fait un mécanisme de sérialisation et d'écriture dans un fichier ou une base de données.

Les composants sont placés dans un environnement d'exécution (en général appelé conteneur[12]). Cet environnement va garantir au composant certaines propriétés ou services (de persistance, de restriction d'accès...). Pour cela, un contexte est éventuellement associé au composant ; ce contexte rassemble les caractéristiques du composant pour les propriétés recherchées, et fournit un accès, pour le composant, aux services demandés.

L'environnement d'exécution va aussi permettre l'accès aux méthodes du composant, selon deux principes :

**la fabrique d'instance** : permet de créer des instances du composant demandé ; en général il s'agit d'une version simplifiée du patron *Abstract Factory* de [GHJV94] ;

**la délégation** , ou *Proxy* [GHJV94] : les requêtes ne sont pas envoyées directement au composant, mais à une entité qui les dirigera vers le composant adéquat.




1. demande d'interaction avec le composant
2. gestion des instances
3. envoi de la référence vers le composant
4. utilisation des services du composant
5. délégation

**Fig. 2.2:** *Architecture répartie à composant : modèle d'interaction avec le composant*

La figure 2.2 donne le modèle de programmation. Le composant a été déployé dans un environnement d'exécution, selon les données présentes dans le descripteur associé au composant ; cet environnement fournit de plus les services auquel le composant a souscrit : service de transaction, de sécurité, ou autre. Lorsqu'un client veut utiliser les services d'un composant particulier, il contacte tout d'abord la fabrique

[12]Seul .NET utilise un nom différent : le Domaine d'Application.

d'instance correspondante. Celle-ci gère les instances du composant demandé : création d'une nouvelle instance ou accès à un cache (*pool* d'instances) ; un procurateur (ou *proxy*) est également créé, et fait office de point d'accès aux services du composant. Enfin, une référence vers le procurateur est envoyée au client. Le client n'accède pas directement aux service du composant : les requêtes sont envoyées au procurateur qui les transmet au composant.

Les requêtes sont envoyées à travers le réseau selon les méthodes classiques de souche et de squelette représentant des convertisseurs (*Marshalling*, qui encaspule l'appel de méthode dans un flot de données, qui sera désencapsulé à l'arrivée). La réception de la requête et la gestion des services souscrits est géré comme le montre la figure 2.3 : la requête traverse une série d'intercepteurs qui, selon le cas, font appel aux services de l'environnement ; par exemple, un service de sécurité bloquera l'appel en début de chaîne. L'appel à la méthode proprement dite du composant se fait en dernier. La réponse suivra le chemin inverse.

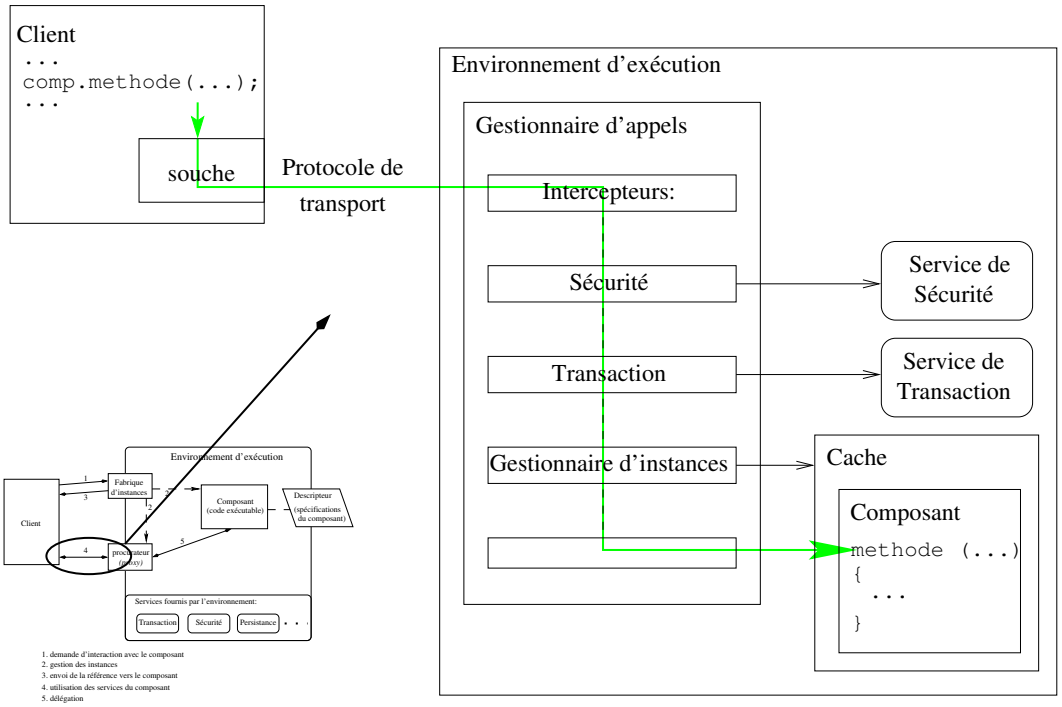



**Fig. 2.3:** *Architecture répartie à composant : appel de méthode*

#### 2.3.3.1 Environnement CCM

L'environnement CCM s'appuie sur l'architecture Corba. Cette architecture est construite autour d'un bus logiciel (l'ORB, *Object Request Broker*), qui s'occupe de faire inter-opérer les objets ou composants entre eux, et propose divers services (localisation, transaction, etc.). Au dessus de ce bus logiciel, des conteneurs prennent en charge l'accueil des composants CCM. La figure 2.4 page suivante donne l'architecture résultante. Celle-ci est assez complexe, aussi nous ne rentrerons pas dans les

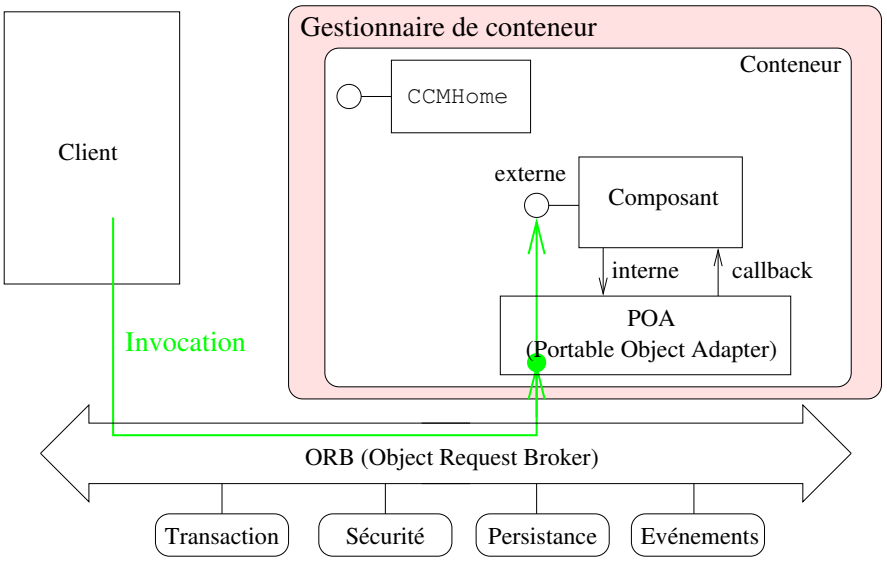



**Fig. 2.4:** *Appel de méthode métier : architecture CCM.*

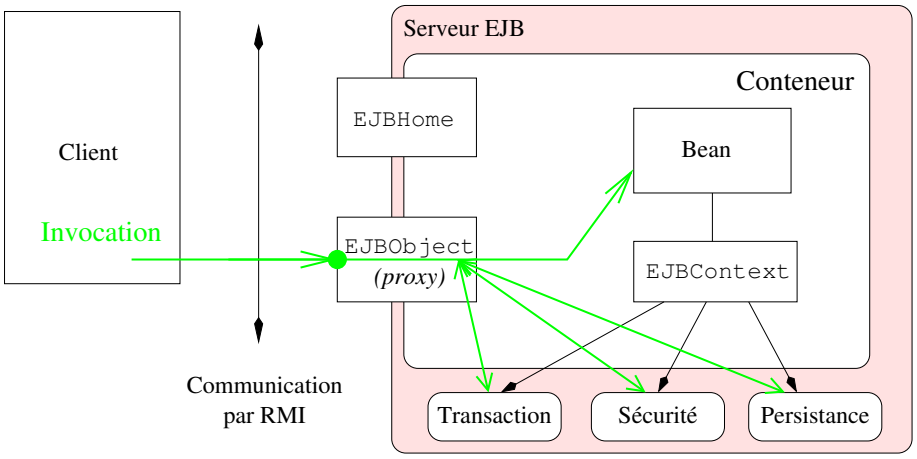



**Fig. 2.5:** *Appel de méthode métier : architecture EJB*

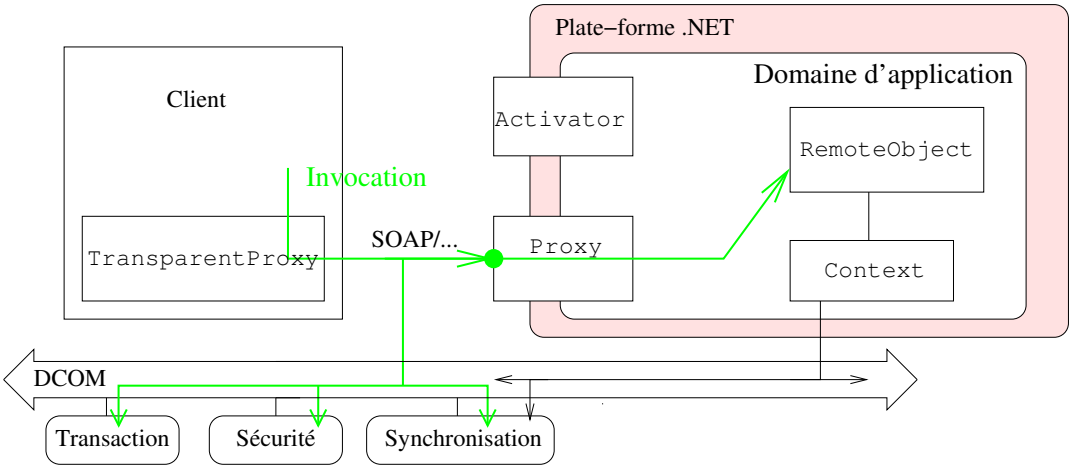



**Fig. 2.6:** *Appel de méthode métier : architecture .NET (cas d'un composant* Client-Activated*)*
Les appels aux services de transaction, sécurité... se font au travers du bus DCOM, qui ne fait pas partie de .NET

détails lorsqu'il s'agit des appels à travers l'ORB. On y remarque toutefois que les principes généraux y sont représentés : les composants sont mis en place dans des conteneurs (eux-mêmes gérés par un serveur de conteneur[13]), la fabrique de composant est représentée par une classe de type `CCMHome`, et la délégation est effectuée par le POA (*Protable Object Adapter*) et l'ORB.

La partie implantation du composant vise à distinguer les parties fonctionnelles et non fonctionnelles du composant. L'assemblage se fait dans un *framework* appelé CIF (*Component Implementation Framework*), dans lequel le langage CIDL (*Component Implementation Definition Language*) permet de décrire la structure de l'implantation d'un composant. Cela permet de donner les relations entre les déclarations de spécification en IDL3 et les exécuteurs. Ces derniers correspondent aussi bien aux objets implantant les fabriques d'instances qu'aux objets mettant en œuvre les facettes des composants. Un composant peut ainsi être composé de plusieurs objets, appelés segments, chacun implantant un ou plusieurs ports du composant. C'est en CIDL également que l'on déclare à quelle catégorie le composant appartient, et les notions de persistance. Le code proprement dit du composant est difficilement présentable en quelques lignes ; on retiendra que le(s) objet(s) mettant en œuvre le composant doivent implanter les interfaces décrivant chacune une facette, ainsi qu'une interface de *callback* (permettant au conteneur de gérer par exemple la persistance du composant[14]).

Les fabriques d'instances sont appelées maisons de composants, et font partie du modèle abstrait de composant CCM[15]. La maison de composant doit être déclarée en IDL3 par la syntaxe `home MaMaison manages MonComposant {...}`[16]. L'objet résultant dérive alors soit du type de base `Components::CCMHome`, soit de `Components::KeylessCCMHome`.

Enfin, la délégation des appels de méthode se fait au travers du POA et de l'ORB. Le *Object Request Broker* non seulement localise le conteneur du composant, mais met aussi en œuvre divers services (comme la persistance ou la sécurité) qui sont appelés soit directement par le conteneur, soit par le composant. Le POA s'occupe de gérer les instances de composant, et de rediriger les méthodes métier vers le composant correspondant. Les principes de convertisseur et d'intercepteurs se retrouvent dans les concepts de base de l'ORB et du POA.

#### 2.3.3.2 Environnement EJB

L'architecture EJB est la première architecture distribuée de composants : Sun a mis en application les patrons fabrique et délégation, et le principe de composant déployable sur une plate-forme ; un serveur JNDI (*Java Naming and Directory Interface*) fournit le mécanisme d'introspection des composants. La figure 2.5 page 26

[13] En fait, un conteneur ne gère qu'une seule catégorie de composant.

[14] Deux choix sont possibles : `SessionComponent` pour les composants non persitants, ou `EntityComponent` pour les autres.

[15] Cette partie n'est pas intégrée dans la section Composant CCM, car notre travail porte sur un composant générique, pour lequel la notion de maison n'est pas indispensable. De plus, cela permet de mieux comparer les différentes architectures.

[16] Nous ne cherchons pas à donner exactement la façon dont les composants CCM sont utilsés, mais plutôt donner les premiers indices pour pouvoir comparer rapidement les différentes architectures. Le lecteur se référera à [MMC02] pour des informations plus précises.

donne une vue succincte d'un appel de méthode d'un composant EJB de type Session ou Entité (l'Orienté Message est implanté différemment, comme nous verrons plus loin).

On retrouve la fabrique de composant et la classe de délégation[17]. La première est une classe qui dérive de `EJBHome` ; plusieurs méthodes de type `create` et `remove` permettent de créer ou supprimer une instance. La délégation est assurée par une classe qui dérive de `EJBObject`. Le bean doit implanter l'interface convenant à sa catégorie (`SessionBean` ou `EntityBean`) ; ces interfaces définissent des méthodes de *callback* qui permettent au conteneur de gérer les instances du bean. Par exemple, les méthodes `ejbCreate` ou `ejbRemove` doivent être implantées.

L'accès au bean Orienté-Messages est différent, et représenté figure 2.7. Il n'y a pas de fabrique de composant, ni de délégation. Ce bean est en fait un consommateur de messages JMS (*Java Message Service*), et l'accès au bean se fait au travers de JMS, en envoyant les messages vers la destination JMS (il s'agit d'une file d'attente). La classe Java du bean doit implanter les interfaces `MessageDrivenBean` et `MessageListener` ; la méthode `onMessage(Message)` doit être implantée par le bean : elle interprétera le message passé en argument.

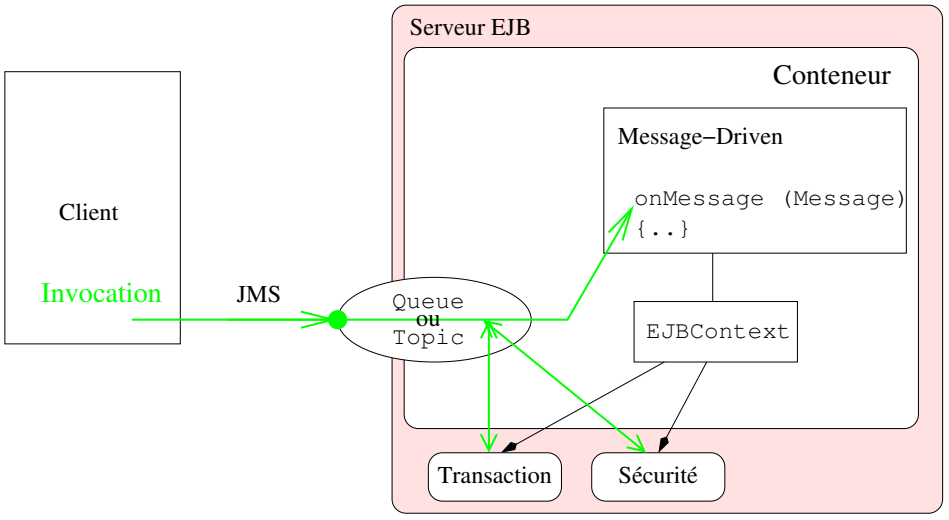



**Fig. 2.7:** *Invocation de la méthode `onMessage` d'un Bean* Message-Driven

Enfin, les services sont implantés au niveau du serveur EJB ; ils sont accessibles par le bean au travers du contexte de l'EJB (`EJBContext`), et accédés lors de l'appel de méthode de l'`EJBObject` par un mécanisme d'intercepteur.

La communication entre le client et le serveur EJB peut se faire de deux manières :
- sur le même site (nouveauté de la version 2.0) : on parlera alors de bean *local*, et l'accès se fait par simple appel de méthode locale. Cependant, la fabrique et la classe de délégation doivent dériver respectivement de `EJBLocalHome` et `EJBLocalObject` [18] (bien entendu, rien n'oblige d'utiliser absolument ce type

[17] Ces classes sont en général générées par l'outil de déploiement.

[18] Nous n'avons pas explicité ce point de détail pour avoir une vue la plus simple possible de l'architecture.

d'accès : un client local peut utiliser le bean selon l'accès expliqué ci-après) ;
- sur des sites distants : la communication se fait par RMI (*Remote Method Invocation*). Un mécanisme classique de souches et de squelettes est alors mis en place.

#### 2.3.3.3 Environnement .NET

.NET est un environnement d'exécution (décrit par le CLR) et une bibliothèque de classes de base, dont le rôle est de charger, exécuter et gérer des types .NET. L'architecture distribuée de composant, en général dénommée *remoting .NET*, représente une petite partie de cet environnement.

Les composants doivent dériver de la classe `MarshalByRefObject`. Ils sont gérés par un domaine d'application qui fait office de conteneur. Chaque composant doit s'enregistrer auprès de ce domaine d'application ; cet enregistrement peut se faire de manière programmatique, ou à l'aide d'un fichier de configuration (le descripteur de déploiement, voir section 2.3.4 page suivante). Le mécanisme d'introspection est mis en œuvre dans une bibliothèque `System.Reflection` de *C♯*.

Deux modes d'instanciation du composant sont possibles : soit par le constructeur `new` (ou la méthode `GetObject`), soit par la fabrique d'instance. La première méthode est utilisée pour l'activation serveur des composants, c'est-à-dire que les composants ne seront activés que lors du premier appel métier vers ceux-ci. Dans la deuxième méthode, le client utilise la fabrique d'instance, appelée `Activator`, et pour laquelle le composant est créé immédiatement (voir figure 2.6 page 26). Les deux méthodes retournent une référence vers le composant qui contiendra les informations de localisation et d'accès aux méthodes du composant[19]. Ces appels méthode passeront par des *proxys*, expliqués plus loin.

La gestion du cycle de vie du composant .NET est différente des autres architectures, et dépend de la catégorie du composant :

***Client-Activated***
: Le cycle de vie est géré par le client ou le composant : activation au plus tôt[20] et gestion par un système de bail (*lease*) implanté sur le domaine d'application. Un bail peut être renouvelé à chaque appel vers le composant, ou au travers d'une chaîne de *sponsors* qui décident ou non du renouvellement ;

***Server-Activated***
: Le cycle de vie est géré par le domaine d'application : activation ou au plus tard, soit lors d'un appel métier.

Un mécanisme de *garbage collecting* détruira le composant du serveur dès qu'il ne sera plus utilisé.

Les *proxys* sont au nombre de deux, comme le montre la figure 2.6 page 26. Un *proxy* local (qui se trouve dans le même domaine d'application que le client), appelé `TransparentProxy` et créé lors de l'activation, est toujours utilisé pour les accès aux méthodes métiers. Ce *proxy* transmet les requêtes à un *proxy* distant appelé `RealProxy` ; ce dernier a accès aux méthodes du composant, mais aussi aux

[19] Il s'agit d'une référence *à la CORBA*.
[20] Soit dès que le client demande une référence vers le composant.

informations sur les instances. Un point intéressant pour .NET est que les références vers les `TransparentProxy` peuvent être envoyés à travers le réseau, le domaine d'application recevant la référence fabriquera un *proxy* lors d'un accès aux méthodes du composant.

La communication se fait à l'aide d'un mécanisme de souche et de squelette qui fonctionne à l'aide de chaînes (*Channels*) : les requêtes (et réponses) sont converties en messages `IMessage` ; un message envoyé passe d'abord au travers d'une série d'intercepteurs enregistrés sur le domaine d'application, est transporté sur le réseau, puis traverse en sens inverse la même chaîne d'intercepteurs, pour être ensuite converti en requête vers le composant (voir figure 2.3 page 25). Il est possible de mettre en œuvre ses propres intercepteurs, ce qui permet d'ajouter des services de sécurité, de chiffrement, etc. Le dernier intercepteur s'occupe du transport du message sur le réseau ; cela peut se faire par SOAP sur HTTP, directement en binaire sur TCP, ou avec un autre protocole de communication définit par l'utilisateur.

Enfin, le contexte est un ensemble de propriétés d'exécution partagées entre les composants (par exemple le synchronisme, les relations entre tâches (*threads*), ...).

### 2.3.4 Déploiement de composant

Le composant est défini comme un ensemble de code déployable sur une plateforme d'exécution. L'unité du composant est en général un fichier représentant une archive, contenant principalement le code du composant, et un descripteur de déploiement indiquant comment effectuer ce déploiement. Déployer un composant revient à utiliser le descripteur de déploiement pour :

- connecter le composant avec l'architecture de composant existante (notamment les ports requis) ;
- intégrer le code du composant dans l'environnement ;
- mettre en place les mécanismes concernant le cycle de vie du composant (selon le type du composant) ;
- mettre en place les services requis par le composant, si ces services sont gérés par l'environnement d'exécution.

Les descripteurs de déploiement sont écrits en XML (*eXtended Markup Language*). Ils contiennent toutes les caractéristiques nécessaires au bon déploiement du composant (catégorie de composant, interfaces, services requis, ...).

#### 2.3.4.1 CCM

Les descripteurs des composants CORBA sont écrits en OSD (*Open Software Descriptor*), un dérivé de XML.

Les différents éléments permettant le déploiement du composant CCM sont regroupés dans deux fichiers «ZIP» correspondant à deux paquetages différents :

**le paquetage du composant** donne les informations pour le déploiement d'un unique composant. Cette archive contient :

- les implantations du composant ;
- la spécification IDL correspondante ;
- le descripteur de composant (CCD, CORBA *Component Descriptor*) qui donne les services que le conteneur doit fournir ;

- un descripteur de propriétés (CPF, *Component Property File*), qui définit les valeur par défaut des attributs du composant ;
- un descripteur de paquetage (CSD, CORBA *Software Descriptor*), donnant des informations diverses sur le composant (auteur, licence, compilateur utilisé...).

**le paquetage d'un assemblage de composant** donne les informations pour le déploiement d'un assemblage de composant. On y trouve :

- les paquetages des composants utilisés pour l'assemblage ;
- un descripteur de propriétés (CPF) ;
- un descripteur d'assemblage (CAD, *Component Assembly Descriptor*), qui donne les caractéristiques de l'assemblage (fabriques, instances de composant et connections à créer).

#### 2.3.4.2 EJB

Les composants EJB sont empaquetés dans un fichier JAR (*Java Archive* [SUN03]) appelé *ejb-jar*. Ce fichier contient :

- les classes Java mettant en œuvre le composant : le bean, la fabrique et le *proxy* ;
- le descripteur de déploiement, qui donne les spécifications du composant : catégorie, services du conteneur utilisés, références vers d'autres composants EJB, etc ;
- les composants EJB utilisés (par inclusion ou par référence vers un ejb-jar).

#### 2.3.4.3 .NET

Dans l'architecture .NET, le fichier de déploiement est un fichier de librairie Windows (.DLL), avec éventuellement un exécutable (.EXE) [21]. Ces fichiers contiennent les informations pour l'assemblage du composant :

- nom du composant, sa version ;
- la spécification des interfaces du composant ;
- les ressources (textes, images, sons, ...) ;
- les composants ou assemblages requis ;
- les permissions de sécurité ;
- les méta-données (*metadata*) qui donnent des informations sur les types et/ou classes fournies par l'assemblage. Elles sont utilisées pour creer les accès vers les composants (objets *proxy*). Ces méta-données peuvent être utilisées à la compilation du composant client, ou à partir d'une description WSDL (*Web Services Description Language*) qui décrit le composant et ses méthodes.

Un composant .NET peut être hébergé sur la plate-forme de plusieurs façons :

- le composant est un code exécutable classique (le plus simple) ;
- hébergement par le serveur IIS (*Internet Information Server*) ; les composants .NET sont alors vus comme des *Web Services* ;

[21] La compilation d'un composant .NET fournit soit le fichier DLL, soit l'exécutable et un fichier DLL.

– hébergement par une infrastructure de composants .NET, qui permet d'avoir les avantage des services COM+ : transaction, compilation JIT (*Just In Time*), cache d'objet, ...

La configuration du composant dans son environnement peut se faire de manière programmatique, ou textuelle (par un fichier de configuration en XML). La configuration permet de spécifier les divers informations sur le composant : le nom et la catégorie des composants, les chaînes d'intercepteurs (`Channels`), les différents baux et sponsors, etc.

### 2.3.5 Conclusion

Nous venons de voir trois plates-formes de composants, qui suivent plus ou moins les caractéristiques des composants présentées en début de chapitre. L'architecture la plus utilisée, et la plus ancienne est sans conteste EJB. Cependant, la plus complète reste la spécification CCM, où nous avons véritablement la notion qu'un composant peut présenter plusieurs ports ; le modèle du composant reste plat, même s'il peut être «segmenté» en plusieurs classes d'implantations, cette segmentation étant visible de l'extérieur ; les EJB et .NET n'offrent pas cette transparence. Cependant, CCM et EJB offrent la possibilité de déployer un assemblage de composants, mais ce n'est pas aussi complet qu'un composant composite.

Un point négatif pour EJB est que la spécification contraint le langage de programmation ; ce n'est pas le cas pour CCM, ni pour .NET. Enfin, l'architecture .NET est plus floue quant à la spécification du composant, et nous la situons plutôt entre EJB et CCM : l'objet distant .NET peut être issu d'une composition, mais n'a pas la notion de ports comme les CCM. Bien qu'étant plus complète, la spécification CCM est complexe comparée aux EJB : plusieurs vues du composant, plusieurs descripteurs, et une technologie basée sur le bus CORBA qui est parfois considéré comme assez lourd. De plus, beaucoup moins d'implantation existent comparé aux EJB.

Le choix de telle ou telle architecture doit se faire par rapport aux besoins de l'application. Pour une application dédiée au système d'exploitation de Microsoft, .NET est le choix évident. Si un composant doit obligatoirement présenter plusieurs interfaces, les efforts devront s'orienter vers une implantation CCM. Si le composant doit être déployé sur plusieurs machines différentes (ou se déplacer entre ces machines), EJB remporte haut la main. On remarquera surtout qu'il existe, dans les spécifications CCM et EJB, des possibilités d'interaction entre les deux architectures. Nous ne connaissons pas, à ce jour, de volonté de la part de Microsoft d'orienter l'architecture .NET vers l'interopérabilité entre les plates-formes de composants. Enfin, il faut souligner que le projet ACCORD fournit des transformations de son modèle de composant vers une plate-forme spécifique CCM [CM03] ou EJB [CGT03] ; il n'est donc pas obligatoire de faire le choix de la plate-forme d'exécution au moment de l'assemblage.

Un point négatif pour ces trois architectures reste qu'elles ne prennent pas en compte les connecteurs. Il y a là une difficulté supplémentaire, qui concerne le déploiement du connecteur, différent ce celui du composant. Ce point est soulevé par Medvidovic et Taylor dans leur classification, et aussi par E. Cariou dans sa thèse [Car03]. Dans le cas où le connecteur relie deux composants distants, une partie du connecteur doit être déployée sur la plate-forme du premier composant, tandis

que l'autre partie doit être déployée avec le deuxième composant. E. Cariou propose un processus de raffinement en trois étapes qui permettent de répartir ces différentes parties du connecteur selon les besoins (par exemple gestion des données centralisée ou distribuée).

Pour finir, nous pensons qu'il est primordial de voir la programmation d'un composant comme une entité rendant un service *de manière interactive*. Pour que la technologie composant se démarque du paradigme des objets, il faut abandonner le modèle d'interaction de type procédural ; il est préférable d'opter pour un modèle d'interaction de type message, qui est beaucoup plus souple, mais beaucoup plus difficile à construire. De plus, un appel de méthode peut être vu comme un échange de message particulier de type requête/réponse. Un compromis est de penser l'interaction entre composants (donc leur spécification) en termes de messages, mais d'implanter le composant en utilisant les procédures ou méthodes objets.

## 2.4 Spécifications formelles pour composants

Les travaux donnant des spécifications formelles pour les composants sont orientés selon trois axes : ceux issus d'une sémantique à base d'automates à états finis, ceux dérivant du $\pi$-calcul, et les contrats dérivant des pré- et post-conditions.

### 2.4.1 Automates d'interface et supposition/garantie

Les automates d'interfaces [dAH01a, dAH01b, CdAHM02] ont été proposés par L. de Alfaro et T. Henzinger. Ptolemy II [CdAH$^+$02, LX01] est une implantation de leur approche. Elles sont définies en termes d'automates I/O [LT87], qui sont des automates dont les transitions indiquent les entrées ou sorties effectuées ; ainsi, ces automates donnent les séquences d'entrées/sorties autorisées sur l'interface du composant.

Une relation de raffinement est utilisée pour vérifier qu'un composant implante la spécification donnée par l'automate d'interface ; cette vérification est faite en termes de simulation [AHKV98] : un automate $P$ raffine un automate $Q$ si toutes les entrées de $Q$ peuvent être simulées par les entrées de $P$ (ou : un message reçu par $Q$ peut être reçu par $P$), et si toutes les sorties de $P$ peuvent être simulées par les sorties de $Q$ (si $P$ envoie un message, $Q$ peut envoyer le même message). Cette relation est aussi donnée en sémantique du jeu [AM98] : l'environnement doit pousser le composant à la faute (ne pas respecter les sorties de l'automate d'interface), mais toujours en respectant les entrées définies par l'automate d'interface.

La compatibilité entre interfaces est obtenue en composant les automates correspondant de telle sorte que les actions duales (une entrée correspondant à une sortie) deviennent des transitions internes. Les interfaces sont considérées comme compatibles tant qu'il existe un environnement qui peut interagir avec l'automate résultant de la composition. Notre modèle est plus contraignant, en ce sens qu'une sortie non prise en compte par le partenaire est considérée comme une erreur de l'émetteur.

Dans [dAH01b], les auteurs vont plus loin en donnant une algèbre d'interface caractérisant l'automate (entrée/sortie, supposition/garantie, dépendance de port – une entrée de l'un peu influencer la sortie de l'autre), et une algèbre de composant ca-

ractérisant la manière dont les contraintes sur les entrées/sorties sont écrites (suivant qu'il existe une relation entre les contraintes des entrées et celles des sorties).

La sémantique des automates d'interface est similaire aux suppositions/garanties (*assume/guarantee*) : les entrées correspondent aux suppositions quant aux actions de l'environnement, et les sorties sont les garanties que le composant doit respecter. Les premiers à parler de suppositions/garanties sont Misra et Chandy [MC81]. Ils appliquent l'idée des pré- et post-conditions aux processus : $r[h]s$ est la spécification interne du processus $h$, et est définie sur les traces de $h$, telle que si $r$ est vraie au pas $k$, alors $s$ est vraie au pas $k+1$. Le processus $h$ suppose en fait que $r$ est vraie, et garantit à son tour $s$.

Les notions de suppositions/garanties sont d'ailleurs à la base d'un raisonnement pour la composition de spécification [AL95, AL93]. La compatibilité des spécifications de deux processus doit montrer que les garanties de l'un assurent les suppositions de l'autre. K. L. McMillan propose une technique compositionnelle dans le cas où le raisonnement est circulaire [McM99] : la propriété de supposition est considérée vraie *jusqu'au* temps $t-1$ pour pouvoir prouver la propriété de garantie au temps $t$, et vice-versa. Henzinger et al. proposent également une méthodologie pour la décomposition des preuves [HQR00, HQR98].

Il est à noter que le paradigme suppositions/garanties mène naturellement à la spécification d'un contrat, comme nous le verrons dans la section 2.4.3 page 36 où le concept de contrat est apparu avec les pré- et post-conditions. C'est d'ailleurs l'approche que nous adoptons, tout comme l'ont fait par exemple Février, Najm et Stefani [FNS97].

### 2.4.2 Typage comportemental

Un type comportemental est une abstraction du comportement de l'entité considérée (objet, composant ou autre). Le service proposé par le type est alors appelé non uniforme, car il varie selon le contexte. Le typage comportemental permet donc de prendre en compte ces aspects dynamiques. La syntaxe des types est en général inspirée des algèbres de processus comme CCS [Mil83] ou le $\pi$-calcul [Mil93].

A notre connaissance, la première notion de comportement liée aux types se trouve dans [TJ92], où Talpin et Jouvelot associent un *effet* aux types de données, c'est-à-dire un ensemble d'interactions défini pour le type en question. Les auteurs définissent tout d'abord les régions qui représentent des structures de données : la région est une abstraction de l'emplacement mémoire alloué à ces données. Ensuite, le comportement d'une expression est exprimé par un *effet*, symbolisé en termes d'initialisation, de lectures et d'écritures sur les régions. Enfin, le type décrit les effets possibles sur les régions concernées, ainsi que leur évolution.

Mais le typage comportemental tel qu'on l'entend aujourd'hui a d'abord été caractérisé par O. Nierstrasz, dans ses types réguliers [Nie95] (c'est-à-dire qu'ils spécifient des processus représentables par une machine à état finis). Par exemple, le type `Buf=(b1, {b1=put.b2, b2=get.b1})` spécifie le comportement d'un buffer à une place : à l'initialisation, le buffer est de type `b1`, et accepte la seule méthode `put` ; une fois cette méthode activée, le type change en `b2` où la seule méthode accessible est `get`. La plupart des langages de type comportementaux suivent le même principe. Nierstrasz défini aussi une relation de sous-typage, ou de substitution des

requêtes : le sous-type doit proposer au moins les mêmes services que le super-type, avec moins d'erreurs. La relation est donnée en termes de transition, et un équivalent en termes de traces (les traces du super-type sont inclues dans celle du sous-type, et les traces échecs du sous-type sont inclues dans celles du super-type).

Il existe deux écoles quant au typage comportemental. L'une dérive des types réguliers de Nierstrasz, en général pour typer les ports ; notre approche appartient à cette école. L'autre utilise les algèbres de processus pour typer les processus eux-mêmes.

Les travaux découlant des types réguliers sont assez proches de notre proposition. Nous passons en revue rapidement les principaux ; la section 7.1.1 page 124 du chapitre 7 donne plus de détails.

Colaço, Colin, Dagnat, Pantel et Sallé travaillent sur l'inférence de type pour le langage d'acteur CAP [CTP03, Col02, DPCS00]. Ils s'intéressent tout comme nous au problème de la consommation des messages : assurer qu'un message envoyé sera finalement consommé. Contrairement à notre approche, les auteurs ont des files d'attente qui ne sont pas ordonnées. Concernant le type des ports du composant, nous pensons qu'il est préférable de se baser sur une sémantique qui conserve l'ordre des messages. Mais cela soulève le problème d'interblocage entre les composants.

Il est à noter que notre travail est la poursuite des travaux de Najm et Nimour [NNS99b, NNS99a, NN97], qui proposent un calcul d'objet pour interfaces non uniformes. Leur système de type assure que tout message envoyé sera compris (et consommé) par le récepteur. Ce système de type ne convient pas pour les composants, car d'une part la communication est synchrone, et d'autre part les types ne permettent pas de mélanger émissions et réceptions. Nous avons cependant gardé les notions d'interfaces publiques (accessibles par plusieurs clients), et privées (accessibles par un seul client). Un autre travail très proche de celui de Najm et Nimour, est celui de Ravara et Vasconcelos[RRV02, RV00]. Les auteurs s'intéressent aussi aux erreurs de type «message non compris», mais leur notion d'erreur est trop lâche : il n'y a pas d'erreur si le message est consommé ou si l'objet auquel il était destiné est éphémère. Ce dernier point nous semble être un point faible pour appliquer leur travail directement sur les composants.

Enfin, récemment, Gay et Vasconcelos, dans [GH99, VVR02, GH03] ont repris l'idée des types de session (*session types*) de Honda et al. [THK94, HVK98]. Les types de session sont associés aux canaux de communication, et spécifient le protocole d'interaction qui a lieu sur ce canal. Mais les auteurs ne démontrent pas l'absence d'interblocage, et n'ont pas de modalités sur les actions.

La deuxième école des types comportementaux s'intéresse aux processus. La plupart des travaux proposent un système de type pour le $\pi$-calcul, principalement parce que ce langage a un pouvoir d'expression assez élevé.

La proposition de Gérard Boudol [Bou97a] est ciblée sur le calcul bleu [Bou97b] (une variante du $\pi$-calcul), en s'inspirant de la logique linéraire ; son approche permet de détecter les interblocages, mais d'une part sa notion est trop large (des processus ne sont pas typables), et il ne détecte qu'une partie des messages non compris.

Puntigam propose dans [PP99, Pun97] un modèle de type basé sur le modèle d'acteur [AMST93]. Les types sont décrits à partir d'un ensemble de messages (signatures, conditions, changements d'état), et d'états. Le type définit donc les séquences

de messages qu'un objet est capable de traiter. Puntigam étend son système de type dans [Pun99] pour autoriser les séquences de messages non réguliers. Les états définissent des jetons qui peuvent être déposés ou retirés par les processus lors d'une interaction. Le contrôle se fait à la fois sur le nombre de jetons restant, mais aussi sur le nombre de processus pouvant y accéder.

Les types proposés par Pierce et Sangiorgi [PS93] permettent de restreindre, selon le contexte, les canaux de communications aux émissions, réceptions, ou les deux. Ce travail a été repris par Kobayashi et al. [KPT99, IK01] qui proposent un système de type dénotant les séquences d'interaction. Ils assurent que les processus communicants n'interfèrent pas avec les autres. Kobayashi a étendu ce travail pour pouvoir prouver l'absence de blocage d'un processus [Kob02]. Le type des canaux de communications sont estampillés de capacités et d'obligations (en termes de nombre de transition). Deux types sont compatibles si les capacités de l'un correspondent aux capacités de l'autre. L'inconvénient du système de type de Kobayashi est qu'il est difficile d'établir les obligations sans être trop restrictif. Le travail de Kobayashi est très proche du notre ; la section 7.1.1 page 124 du chapitre 7 approfondi la comparaison des deux travaux.

Les types comportementaux que nous venons de voir présentent quelques limites en regard du problème qui nous intéresse. D'une part, les modalités sur les actions sont très peu présentes. Nous pensons qu'elles sont primordiales pour imposer des contraintes sur l'environnement du composant, selon le concept de supposition/garanties. D'autre part, les travaux ne se positionnent pas vis-à-vis d'un assemblage de composant. Enfin, peu d'approches distinguent la notion de liaison point-à-point et celle de client/serveur (1-à-n).

### 2.4.3 Contrats

Le premier concept de contrat est en général attribué à C. A. R. Hoare qui, en 1969, a proposé d'ajouter aux instructions de programme des pré-conditions et des post-conditions [Hoa69]. Ces *triplets de Hoare*, notés $P\{Q\}R$, ont la sémantique suivante : *Si la pré-condition $P$ est vraie avant le début du programme $Q$, alors la post-condition $R$ sera vraie à la terminaison de ce programme.* Cette notion de pré- et post-condition permet de réduire l'utilisation des techniques de programmation dites défensives, où l'on vérifie constamment les valeurs des données manipulées. Elle a été mise en œuvre dans le langage Eiffel [Mey91] de B. Meyer, et est informellement utilisée dans les spécifications des bibliothèques des langages.

B. Meyer va plus loin, avec la conception par contrat (*Design by contract* [Mey92]). Un contrat est alors un ensemble d'*obligations* et de *bénéfices* : si un client respecte ses obligations (une lettre suffisamment affranchie), alors il bénéficie d'un service (la lettre arrive à destination). Ces obligations/bénéfices sont spécifiés à l'aide de pré- et post-conditions, ainsi que des invariants.

Mais cette notation n'est pas suffisante dans la plupart des cas. Aussi Beugnard et al. ont proposé une classification de contrats [BJPW99] :

**le contrat syntaxique** donne la signature des opérations fournies par un composant : nom, paramètres d'entrée/sortie, exceptions. Par exemple, une description en IDL d'un composant CORBA est un contrat syntaxique ;

**le contrat comportemental** spécifie les pré- et post-conditions des opérations, ainsi que les invariants des objets ou composants manipulés. Ce contrat est une abstraction du comportement de l'opération en question ;

**le contrat de synchronisation** spécifie les dépendances entre les services proposés par un composant : mise en séquence, parallélisme... Ce contrat est utile par exemple pour un accès multiple au service proposé, ou des opérations mutuellement exclusives. Il peut être spécifié à l'aide de d'expressions de chemin (*path-expressions*[22] [CH74]) ;

**le contrat de qualité de service** permet de quantifier l'offre du service proposé, par exemple le délai de réponse, la qualité (ou précision) du résultat... Ce contrat offre de plus la possibilité de négocier les différentes valeurs définies.

La vérification des contrats peut se faire statiquement (du moins pour les deux premiers niveaux), mais la plupart du temps la vérification se fera à l'exécution[23]. La violation du contrat provoque alors quatre actions possibles : ignorer la violation, rejeter la demande, bloquer le client (jusqu'à ce que le contrat soit valide) ou négocier. Ces différentes actions sont applicables selon les niveaux de contrat : les possibilités de négociation, par exemple, augmentent avec le niveau de contrat.

La spécification du composant par contrat trouve son application essentiellement dans la conception par composition, et dans la réutilisation des composants. En effet, dans ces paradigmes de conception, il est primordial de comprendre les différents rôles joués par les composants, de savoir comment ils collaborent. Les contrats permettent de spécifier les rôles du composant, et le contexte dans lequel ils doivent s'exécuter.

Ainsi, dans [HHG90], Helm et al. présentent plusieurs opérations sur les contrats. Le raffinement correspond à l'implantation du contrat ; l'inclusion est utilisée pour la composition des contrats. Enfin, la compatibilité de contrat (ou *conformance*) décrit comment le composant voit ses participants, du point de vue des méthodes déclarées. En fait, les contrats sont vu comme une abstraction de l'algorithme interne au composant, non comme une description de la façon dont il faut l'utiliser.

De même, plusieurs implantations de contrats ont vu le jour ces dernières années. Par exemple, dans [FLF02] Findler et al. proposent des contrats pour Java : le programme à vérifier est réécrit sur la base des pré- et post-conditions spécifiées en commentaire. Cette méthode présente l'inconvénient de ne pas être modulaire : un changement du contrat provoque une recompilation de toute l'application.

L'approche de ces annotations pré- et post-conditions dans des commentaires est aussi utilisée dans des bibliothèques de contrat comme iContract[24], qui propose un outil pour implanter les contrats en Java. La spécification du contrat se fait par des étiquettes comme `@pre` ou `@post` indiquées en commentaires (tout comme le fait Findler), et dont les expressions sont un sous-ensemble de OCL [OMG03a]. Ces étiquettes seront transformées en assertions. La vérification du contrat est donc dynamique, à l'exécution ; nous cherchons plutôt une vérification statique.

Le concept de conception par contrat de Meyer a aussi inspiré JML (*Java Mo-*

[22] Sortes d'expressions régulières décrivant les relations en processus : exécution en parallèle, en séquence, synchronisation...

[23] Soit du coté du client, soit à l'aide d'un moniteur de contrat.

[24] `http://www.reliable-systems.com/tools/iContract/iContract.htm`

*deling Language*[25] [BCC+03]), qui est un langage de spécification d'interfaces comportementales. Le comportement de l'interface est ici à prendre encore en termes de pré-conditions (`requires`), post-conditions (`ensures`) et d'invariants (`invariant`), spécifiés en commentaires. Les auteurs ajoutent cependant la notion de post-condition exceptionnelle, qui porte sur les exceptions. Par exemple, pour le comportement d'une méthode `square_root (x)` :

```
/*@ signal (IllegalArgumentException e)
  @           e.getMessage() != null && !(x > 0.0)
  @*/
```

indique que si l'exception `IllegalArgumentException` est levée, alors le message qu'elle contient ne doit pas être vide, et l'argument `x` doit être négatif. Tout comme iContract, un compilateur ajoute des assertions conformément aux contrats spécifiés ; par contre des outils existent pour vérifier les contrats statiquement. Il serait intéressant de transposer notre langage d'interfaces comportementales en pré- et post-condition pour JML, et d'effectuer la vérification du contrat statiquement.

Une autre approche tout aussi intéressante pour l'application de notre définition du respect de contrat est Jassda(*Java with Assertions Debug Architecture*[26] [Möl02, BM02]), développé par Mark Brörkens. Jassda permet de vérifier un comportement décrit en CSP. Les auteurs n'ont besoin que du *Byte-code* Java du programme à vérifier ; ils modifient légèrement ce dernier pour l'exécuter au travers de JDI (*Java Debug Interface*) qui récupère les traces des appels. Ces traces sont ensuite comparées au process CSP représentant le contrat. Notre langage se traduit assez facilement en CSP, et tirerait alors profit de l'architecture proposée par Jassda. Cependant, vérifier le bon enchaînement des actions ne suffit pas : nous imposons des contraintes sur le comportement interne du composant qui ne sont pas vérifiables avec Jassda. Par exemple, et de manière informelle, nous avons la notion d'attachement : deux ports attachés dialoguent entre eux ; cet attachement peut être dynamique, mais doit vérifier que les types de ces deux ports sont compatibles. Cette vérification n'est pas possible avec Jassda.

Toutes ces méthodes de conception par contrat son orientées vers la vérification que le composant respecte le contrat qui lui est associé. Une autre vision des contrats et de les utiliser dans le cadre d'une collaboration.

Le modèle proposé par L.F. Andrade et J.L. Fiadeiro peut être utilisé en ce sens. Les auteurs proposent un modèle objet pour la vérification de contrats [AF99] où la classe auquel s'applique le contrat n'est pas modifiée : le principe est d'encapsuler les appels méthode dans une classe abstraite qui vérifie les termes du ou des contrats qui s'appliquent. Le langage de contrat utilisé ressemble plus à une collaboration entre l'élément concerné et son environnement. Cette approche a été utilisée par Fontaine et al. pour la composition de services [FNR02], et traiter les problèmes liés aux interactions de service. Cependant, cette vision du contrat est à la frontière entre la classification de Beugnard et al. et les langages de coordination.

Pour finir, il est intéressant de voir la relation entre la spécification de contrat

[25] `http://www.jmlspecs.org/`
[26] `http://jassda.sourceforge.net/`

et le sous-typage d'une opération, résumé dans la figure 2.8. Le sous-typage d'une opération se résume en deux points :

- le sous-type doit accepter de la part de l'environnement tout ce que le super-type acceptait de la part de ce même environnement. Les contraintes sur les entrées de l'opération sous-typée doivent donc être égales ou relaxées (contra-variance) ;
- le sous-type doit donner à l'environnement un sous ensemble de ce que donnait le super-type. Les contraintes sur les sorties de l'opération doivent donc être égales ou renforcées (co-variance).

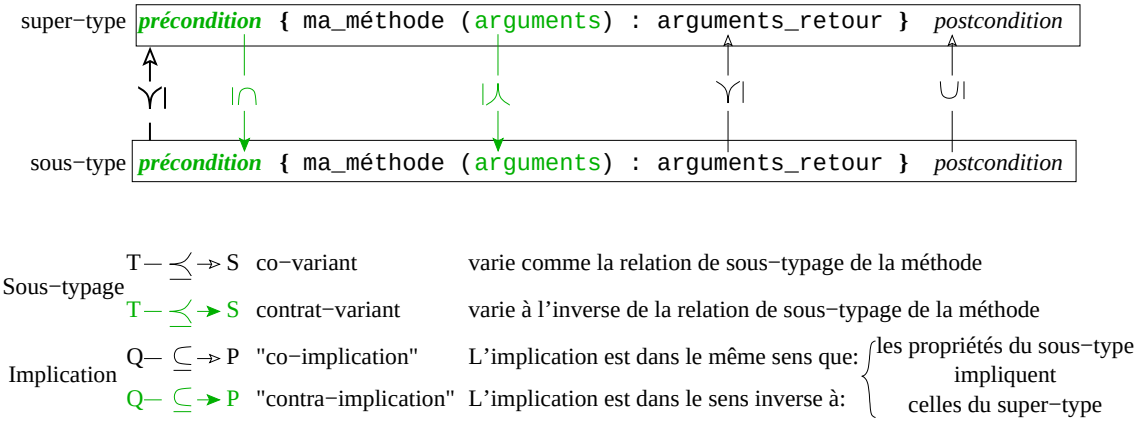



**Fig. 2.8:** *Co- et Contra-variance : évolution des propriétés lors du sous-typage*

Cette relation a été remarquée en premier par Liskov et Wing [LW94]. Elle est appuyée par le fait que le sous-type doit pouvoir être utilisé à la place du super-type. Contra- et co-variances ont été appliqués aux contrats pour les composants par Legond-Aubry, Enselme et Florin [LAEF03a, LAEF03b, FLA03].

## 2.5 Conclusion

Un composant est vu comme un couple spécification/code. La spécification du composant définit les ports du composant, qui sont les points d'accès vers les services qu'il propose ; elle décrit aussi des propriétés comme des contraintes sur l'utilisation des ports, ou des propriétés non fonctionnelles comme des services de persistance ou de transaction dont le composant aura besoin. Cette spécification est utilisée à la fois lors de la modélisation de l'application, mais aussi lors du déploiement du composant sur des plates-formes spécifiques comme EJB, CCM ou .NET.

Les Langages de Description d'Architecture (ADL) que nous avons présentés, Darwin et Wright, offrent un cadre formel mais qui ne convient pas pour le problème que nous voulons résoudre : les liens dynamiques ne sont pas pris en compte, et, surtout, le comportement du composant est nécessaire pour assurer que la configuration de composants ne provoquera pas d'erreurs.

Enfin, peu de systèmes de types sont proposés spécifiquement pour les composants EJB ou CCM, bien que ces systèmes y soient adaptables (après tout, un acteur

ou un processus peut être considéré comme un composant). Les contrats sont, quant à eux, plus orientés vers les composants. Un inconvénient est que la vérification des contrats, ou des types comportementaux, n'est pas prise en compte par les architectures de composants. L'assemblage par contrat concerne un outil de développement d'application (par exemple avec un atelier logiciel) dans lequel la compatibilité des contrats peut être vérifiée, mais reste statique. La compatibilité dynamique requiert qu'elle soit faite au déploiement du composant, ce qui pose des problèmes de performance. Notre approche est plutôt dans le sens d'une telle vérification. Même si plusieurs outils de vérification de contrats existent, ils ne présentent à chaque fois qu'une facette de la solution que l'on recherche (vérification à l'exécution, ou vérification uniquement du séquencement des messages).

Pour finir, la signification d'un contrat attaché à un connecteur complexe (qui n'est pas un simple médium de communication) est assez floue en ce qui concerne le comportement de ce connecteur, en terme de messages échangés[27]. Par exemple, dans le projet ACCORD, un connecteur est connecté aux composants par des prises ; ces prises s'adapteront au type du port auquel elles sont attachées (cette sémantique suit d'ailleurs celle proposée par Medvidovic et Taylor [MT00]). Il est donc difficile de vérifier, avant assemblage, si un connecteur respecte un type décrivant une interaction. Avoir un type comportemental pour ces prises ne nous semble donc pas judicieux.

Le connecteur doit en fait être vu selon différents niveaux d'abstraction. Au niveau le plus élevé, il est utilisé pour connecter des composants dont les ports ne sont pas compatibles, ou pour modéliser un médium de communication bien particulier (comme un système de vote). Par contre, à un niveau plus concret, les fonctionnalités du connecteur doivent être implantées, et réparties entre les différents points de connexion des composants mis en jeu. Cette approche par raffinement est préconisée par Eric Cariou et Antoine Beugnard [CB02, CBJ02, Car03]. Ils proposent de voir les connecteurs comme des composants de communication (ou médiums) qui réifient une abstraction de communication. Cette abstraction peut aller des protocoles de communication aux systèmes d'interaction et de collaboration (elle est d'ailleurs représentée en partie par un diagramme de collaboration UML).

Pour ces raisons de complexité du connecteur, notre travail ne porte que sur les composants. La notion de prise qui s'adapte au port est en dehors de notre cadre d'étude, mais nous proposons une solution dans le chapitre 7 page 123 qui conclut ce manuscrit.

## 2.6 Etude de cas

Nous illustrerons notre travail par une étude de cas. Nous nous intéresserons à un logiciel de gestion d'articles pour une conférence. Typiquement, plusieurs phases ont lieu entre la soumission des articles et la date de la conférence :

- soumission des articles par les auteurs. Les informations à fournir sont le titre de l'article, le titre, les auteurs (nom et affiliations), le contact (pour le résultat des rapports), et enfin l'article en lui même (fichier PDF ou PS) ;

[27] Des contrats de qualité de service, comme «transmettre un message en 10ms» peuvent bien entendu être attachés aux connecteurs.

- répartition des articles parmi les rapporteurs. En général effectuée à partir des préférences émises par ces rapporteurs (choix d'un ensemble d'articles, ou de domaines de compétence) ;
- relecture par les rapporteurs des articles assignés. Au moins 3 ou 4 rapporteurs lisent un article ; l'ensemble des rapports émis permettront de décider l'acceptation ou le rejet du papier ;
- notification aux auteurs des résultats ;
- envoi de la version finale des articles par les auteurs ;
- inscriptions des participants ;
- présentations des articles le(s) jour(s) J.

Nous nous intéresserons particulièrement à la phase de relecture. Nous proposons une architecture en trois composants :

**Rapporteur**
: Composant représentant le rapporteur : sa fonctionnalité principale est de permettre au rapporteur (personne physique) de remplir le rapport (ou d'avoir un accès aux autres rapports, mais nous ne détaillerons pas cette fonctionnalité) ;

**Article**
: Composant permettant de gérer un article (informations fournies par les auteurs, mais aussi les rapports ; plusieurs soumissions de rapports sont autorisées). Nous ne donnerons qu'un accès pour les rapporteurs, mais on peut aussi imaginer un accès pour les auteurs ;

**Gestion** de la conférence
: Gère les accès aux autres composants de l'application. Pour notre cas particulier, le composant `Rapporteur` doit demander au composant `Gestion` un accès au composant `Article`.

  Un autre exemple (non traité) concerne les auteurs : un composant `Auteur` peut demander un accès à un composant `Article` (qui peut être nouveau).

Le but de cet exemple n'est pas de définir une architecture complète, mais plutôt un exemple simple, et qui mette en jeu le maximum des possibilités que nous offrons.

Plusieurs implantations sont possibles. La première, classique, est celle où le composant `Rapporteur` doit demander au composant `Gestion` un accès à `Article`, pour pouvoir rentrer ou modifier un rapport sur l'article en question. `Gestion` enverra une référence vers un port de `Article` ; le `Rapporteur` utilisera cette référence pour envoyer la critique avec un deuxième port.

Nous proposons un protocole plus élégant, décrit par les figures 2.9 et 2.10 page suivante. Le `Rapporteur` utilisera le même port pour demander l'accès et envoyer la critique. Le composant `Gestion` identifie dans un premier temps le rapporteur (figure 2.9 : le rapporteur est correctement identifié), puis envoie au composant `Article` la référence vers le port du `Rapporteur`. Le composant `Article` utilise la référence reçue pour indiquer au composant `Rapporteur` d'une part que l'accès est autorisé, et d'autre part que le dialogue se poursuit avec lui (chaque message porte la référence de l'émetteur, ce qui permet de notifier le changement d'interlocuteur[28]) ; à la réception du message `accordé(...)`, le composant `Rapporteur` prend en compte la référence de son nouvel interlocuteur, et lui envoie le rapport.

[28] Cette sémantique fait partie du modèle de composant exposé dans le chapitre 4 page 61.

La figure 2.10 présente le cas où le rapporteur a donné un mauvais mot de passe (ou ne peut pas rapporter sur l'article demandé) ; le composant `Gestion` refuse de continuer le dialogue.

Ce protocole présente l'avantage que l'ensemble (`Gestion` et `Article`) peut être implanté par un seul composant avec un unique port *sans qu'il y ait changement du composant* `Rapporteur`.

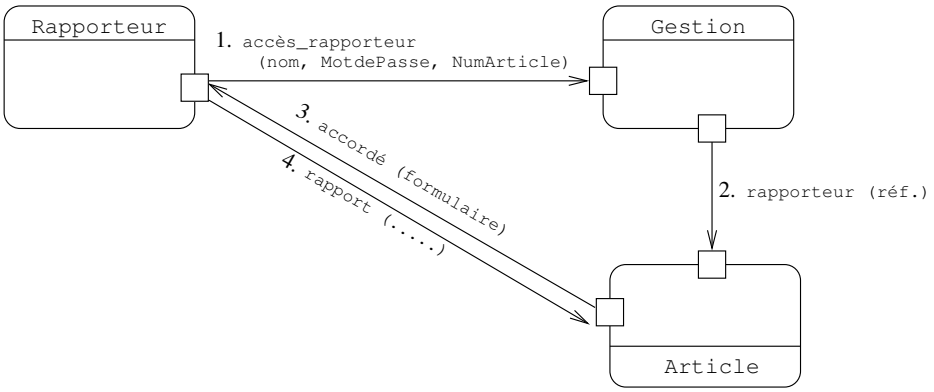



**Fig. 2.9:** *Exemple d'interaction : accès accordé, et entrée de la critique du rapporteur*

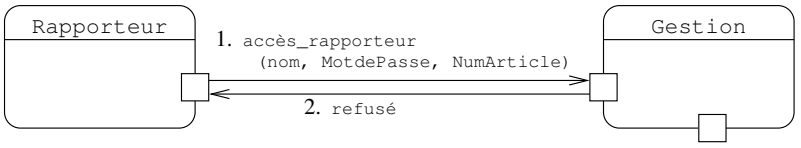



**Fig. 2.10:** *Exemple d'interaction : accès pour le rapporteur refusé (composant* `Article` *non représenté)*

Nous cherchons à pouvoir vérifier deux points :

- les liens entre les ports des composants sont valides à l'assemblage, c'est-à-dire que les types des ports interconnectés sont compatibles. Par exemple, le type du port du composant `Rapporteur` indiquera que sa première action est l'envoi du message `accès_rapporteur` ; ce port doit être mis en relation avec un port dont le type est tel que la première action est la réception de ce message `accès_rapporteur`. Le langage que nous proposons, ainsi que la relation de compatibilité correspondante, sont décrits dans le chapitre 3 page suivante ;
- le comportement du composant est conforme à la description donnée par le type comportemental de ses ports. Par exemple, le composant `Rapporteur` doit avoir le comportement décrit informellement par les figures ci-dessus : envoyer un message `accès_rapporteur`, attendre `accordé` avant de pouvoir envoyer `rapport`. Le chapitre 4 page 61 décrit la sémantique de notre modèle de composant ; le chapitre 5 page 77 permet de vérifier que le composant respecte le type de ses ports.

CHAPITRE 3

# Types d'interface

Dans ce chapitre nous décrivons le langage utilisé pour définir les interfaces. Un composant typé est un composant pour lequel chaque port initial ou créé en cours d'exécution a un type associé, et toute référence vers un port a aussi un type déclaré.

Nous adoptons un langage de type comportemental où le type d'un port décrit ses états possibles, et pour chaque état, les actions requises ou permises à travers ce port, et les états après l'exécution d'une action. Une des principales caractéristiques de ce langage est l'utilisation de modalités **may** et **must**, et la distinction entre interfaces point-à-point (pair, ou *peer*) et serveur.

En plus de la syntaxe du langage, ce chapitre présente une relation de sous-typage et une relation de compatibilité. Il se termine pas une étude de cas pour permettre de bien saisir les concepts présentés.

## 3.1 Présentation informelle autour d'un compte en banque

Notre langage de type décrit l'ensemble des messages échangés entre le port et l'environnement. Cette sémantique de message se rapproche des ADL Darwin [MDK94] (où les interfaces des composants exposent des services requis et fournis qui sont respectivement des messages envoyés et reçus) ou Wright [AG96], où le type d'un port est une expression CSP [BHR84].

Par exemple, l'accès à un compte en banque peut s'écrire, dans notre langage :

accès_compte = **may ? [** demande_accès (string, string) ;
**must ! [** autorisé (opérations_compte) ; **0**
+ refusé ; accès_compte **] ]**

Ce type se lit comme suit : le type *accès_ compte* spécifie la réception du message *demande_ accès* qui prend en paramètre deux chaînes de caractères (typiquement le nom de l'utilisateur et le mot de passe). Une fois ce message reçu, le type spécifie un envoi obligatoire (**must!**) à choisir parmi :

- envoi du message *autorisé* avec en paramètre une référence vers un port typé *opérations_ compte*. Une fois ce message envoyé, le type n'a plus d'actions (**0**) ;
- envoi du message *refusé*, et dans ce cas le type devient *accès_ compte* pour un nouvel essai d'authentification.

Le type *opérations_ compte* quant à lui, est déclaré en deux fois :

```
opérations_compte =
    may ? [   créditer (integer) ;  must ! [ solde (integer) ; opérations_compte ]
            + débiter (integer) ;
                        must ! [   solde (integer) ; opérations_compte
                                 + solde_négatif (integer) ; ops_compte_négatif ] ]

ops_compte_négatif =
    must ? [  créditer (integer) ;
                        must ! [   solde (integer) ; opérations_compte
                                 + solde_négatif (integer) ; ops_compte_négatif ]
            + débiter (integer) ; must ! [ solde_négatif (integer) ; ops_compte_négatif  ] ]
```

Un port typé *opérations_ compte* peut recevoir deux messages :

- *créditer* augmente le solde du compte du montant indiqué. Ce solde doit être ensuite envoyé par le message *solde(...)*, après quoi le port accepte de nouvelles opérations (type *opérations_ compte*) ;
- *débiter* retire du solde du compte le montant indiqué. Deux cas se présentent :
  - Si le compte reste positif, le solde est retourné et d'autres opérations peuvent être effectuées (message *solde(...)* suivi du type *opérations_ compte*) ;
  - Si le compte devient négatif, alors l'utilisateur en est informé par le message *solde_ négatif(...)*, et le type du port devient *ops_ compte_ négatif*. Ce type donne le comportement du port lorsque le compte est négatif.

Le type *ops_ compte_ négatif* est similaire au type *opérations_ compte*, à une différence fondamentale : la modalité qui porte sur la réception des messages *créditer* et *débiter*. Dans le cas où le compte est positif, cette modalité est **may?**, *peut* recevoir : elle indique qu'il est possible – mais pas obligatoire – d'effectuer des opérations de débit et de crédit sur le compte. Dans le cas où le compte est négatif, cette modalité est **must?**, *doit* recevoir : elle indique qu'il est obligatoire d'effectuer soit un débit, soit un crédit sur le compte.

La spécification *opérations_ compte* que nous venons de décrire autorise l'utilisateur du compte à effectuer des opérations qui potentiellement le mettent en négatif ; par contre, tant que le compte n'est pas suffisamment approvisionné, l'utilisateur doit effectuer des opérations sur ce compte[1].

Notre langage de type présente donc la particularité non seulement d'imposer des contraintes sur le comportement du composant, mais aussi d'imposer des contraintes sur son interlocuteur (soit : sur l'environnement du composant). L'introduction des modalités mène à un modèle sous-jacent qui est une sorte de Système de Transition Etiqueté Modal, dans lequel chaque état est soit **may** soit **must** [LSW95]. Cela a un impact assez fort sur les règles de compatibilité.

En outre, nous distinguons deux types de ports : les ports pairs (ou *peer*), et les ports serveurs. Les ports pairs sont engagés dans une communication de type point-à-point (1-à-1). Les ports serveurs sont engagés dans une communication de type client/serveur, au sens où ce sont les seuls à pouvoir servir plusieurs clients.

[1] un troisième type peut correspondre à la limite d'autorisation de découvert. Ce type interdira alors tout débit : **must? [** *créditer*(...) ; **must!** [...] ].

La raison principale de cette distinction est que le type comportemental permet une progression dans un dialogue qui doit être partagée entre les ports mis en relation (par exemple, l'émission de *autorisé(...)* doit coïncider avec la réception du même message) ; les ports pairs et les liaisons point-à-point nous permettent d'assurer facilement cette progression partagée. Nous ne désirons cependant pas nous restreindre à ces seuls types de communication ; un port serveur autorise les liaisons 1-à-n (ou client/serveur). Nous pensons que ces deux types de communication nous permettront de spécifier les autres modèles de communication (en utilisant un connecteur servant alors de médium de communication).

## 3.2 Syntaxe du langage d'interface

Le langage d'interface a la syntaxe suivante, que nous détaillons dans les paragraphes ci-après :

$$
\begin{array}{rcl}
\textit{type} & ::= & \textit{server_name} = \textit{mod receive_server} \\
 & | & \textit{peer_name} = (\textit{mod send} \mid \textit{mod receive}) \\
\textit{receive_server} & ::= & \textsf{? *[} \sum_i M_i \textsf{ ; } I_i \textsf{ ]} \\
\textit{receive} & ::= & \textsf{? [} \sum_i M_i \textsf{ ; } I_i \textsf{ ]} \\
\textit{send} & ::= & \textsf{! [} \sum_i M_i \textsf{ ; } I_i \textsf{ ]} \\
I & ::= & \textsf{0} \mid \textit{peer_name} \mid \textit{mod send} \mid \textit{mod receive} \\
\textit{mod} & ::= & \textsf{may} \mid \textsf{must} \\
M & ::= & \textit{name}[\ \textsf{(}\ \widetilde{\textit{args}}\ \textsf{)}\ ] \\
\textit{args} & ::= & \textit{basic_type} \mid \textit{peer_name} \mid \textit{server_name} \\
\textit{basic_type} & ::= & \textsf{boolean} \mid \textsf{integer} \mid \textsf{real} \mid \textsf{string.}
\end{array}
$$

La déclaration d'un *type* se fait en donnant un nom (*server_ name* ou *peer_ name*), suivi d'une description. Dans le cas d'un type serveur, la description du type est une modalité («**may**» ou «**must**», dont la signification précise est donnée page suivante), suivie des mots-clés «**?**» (réception) et «*****» (serveur), et d'un ensemble de messages en réception (que nous décrivons dans le paragraphe suivant). Un type pair est une modalité **may**/**must** suivie d'un ensemble de messages à envoyer (*send*) ou à recevoir (*receive*).

Les messages à envoyer (*send*) sont décrits par «**!**» suivi, entre crochets «**[**», «**]**», de la liste des messages en question. Les messages que le type peut recevoir (*receive*) sont décrits de la même manière, avec un «**?**».

La liste des messages est en fait un couple $(M_i, I_i)$ dont les éléments sont séparés par l'opérateur de séquence «**;**». Ces couples «$M_i$ **;**$I_i$» sont séparés par l'opérateur de choix «**+**». Une fois le message $M_i$ reçu (ou envoyé, selon le cas), le type modifie son comportement en $I_i$. Ce nouveau comportement peut être soit la terminaison («**0**» – zéro), soit le nom d'un autre type (*peer_ name*, ce type ne peut pas être un serveur), soit une autre action (modalité **may**/**must** suivie de messages à envoyer ou recevoir).

Enfin, les messages $M_i$ portent un nom, et des arguments $\widetilde{args}$. La notation $\tilde{v}$ indique que $v$ est un n-uplet. Dans la syntaxe du langage, les éléments de $\widetilde{args}$ sont séparés par des virgules. Un argument *args* d'un message est soit le nom d'un type (pair ou serveur), soit un type basique (*basic_ type*).

Les types peuvent être nommés, mais les relations définies dans ce chapitre ne s'intéressent qu'à la structure d'état des types.

La signification des modalités d'interface pour le port du composant et pour son partenaire (le port qui dialogue avec lui) est la suivante :

| | |
|---|---|
| **may ? [** $\Sigma M_i$ ; $I_i$ **]** | signifie "le port n'impose aucune contrainte d'envoi sur le partenaire, mais si le partenaire envoie un message $M_i$, alors le port garantit qu'il est prêt à le recevoir, et à exécuter ultérieurement l'action $I_i$". |
| **must ? [** $\Sigma M_i$ ; $I_i$ **]** | signifie "le port impose une contrainte d'envoi sur le partenaire, et si le partenaire envoie un message $M_i$, alors le port garantit qu'il est prêt à le recevoir, et à exécuter ultérieurement l'action $I_i$". |
| **may ! [** $\Sigma M_i$ ; $I_i$ **]** | signifie "le port peut envoyer au partenaire un des messages $M_i$ et exécuter ultérieurement l'action $I_i$ ; le partenaire doit être prêt à recevoir le message $M_i$". |
| **must ! [** $\Sigma M_i$ ; $I_i$ **]** | signifie "le port garantit qu'il enverra un des messages $M_i$ et exécutera ultérieurement l'action $I_i$ ; le partenaire doit être prêt à recevoir le message $M_i$". |

Les messages contiennent des arguments. Ainsi, aussi bien des données classiques (comme les entiers, les booléens...) que des références vers des ports (de type *peer_ name* ou *server_ name*), peuvent être envoyées dans les messages. Concernant les types basiques, nous ne prendrons pas en compte les structures de données complexes, mais leur ajout serait immédiat. L'envoi ou la réception de références suppose cependant quelques restrictions quant au comportement des composants impliqués :

| | |
|---|---|
| **!** m $(I)$ | signifie "le port envoie à son partenaire un message contenant une référence vers un port dont le comportement est décrit par le type $I$. De plus, la première action de ce port référencé doit être «**?**» (réception)". |
| **?** m $(I)$ | signifie "le port reçoit un message contenant une référence vers un autre port, dont le comportement est décrit par $I$. La première action de ce port référencé est «**?**» (réception)". |

La contrainte sur la première action de $I$ est inévitable. En effet, si cette action est «**!**» (envoi de message), alors le port référencé peut envoyer ce message à un troisième port, ce qui mènerai à des incompatibilités de comportement entre les composants.

Enfin, la construction «*****» permet la spécification d'un serveur. Celui-ci se duplique pour répondre à une requête et pour répondre à d'autres clients potentiels :

$I = mod$ **?** ***[** $m()$ **;** $I'$ **]** après la réception de $m$, un port dont le comportement est $I'$ est créé, tandis que le serveur se comporte à nouveau comme $I$. Le nouveau port interagira avec l'émetteur de la requête. La première action de $I'$ doit être **!** (envoi), pour indiquer à cet émetteur le port qui a été créé spécifiquement pour lui.

Plusieurs ports clients peuvent être connectés en même temps au même port serveur. Les ports pairs, quant à eux, ne sont connectés que par paires. Cette distinction permet les liaisons client/serveur et point-à-point.

Concernant le type serveur, on remarquera entre autres qu'un type (pair ou serveur) ne peut prendre le comportement d'un autre type déclaré en tant que serveur. D'une part, un port pair ne doit pas devenir un port serveur. D'autre part, un dialogue avec un serveur se fait souvent en mode connecté, les premiers messages permettant d'ouvrir un contexte dans lequel le dialogue aura lieu ; ouvrir un autre contexte dans le même dialogue (soit : avec le même couple (port, partenaire)) est une erreur de spécification. Par exemple, les déclarations suivantes sont interdites (le premier cas est celui où un port pair devient serveur, le deuxième où, dans un dialogue avec un serveur, un autre dialogue avec le même serveur est initié) :

$$\begin{aligned} I &= \textbf{may !}\ \ \textbf{[}\ M\ \textbf{;}\ \mathit{Server}\ \textbf{]} \\ \mathit{Server} &= \textbf{may ?}\ \ \textbf{*[}\ M\ \textbf{;}\ \mathit{Server}\ \textbf{]} \end{aligned}$$

## 3.3 Sous-typage

Dans cette section, nous définissons une relation de sous-typage $\preceq$. Nous désirons avoir la propriété classique des relations de sous-typage, à savoir un port instance d'un sous-type peut être utilisé là où un port instance du super-type est demandé, sans modification des composants du système. Nous détaillerons cette propriété une fois définie la compatibilité entre interfaces (section 3.4 page 50).

L'idée principale de «$T$ est sous-type de $S$», noté $T \preceq S$, est la suivante :

- Si $T$ et $S$ sont en réception, alors :
  - $T$ reçoit au moins les messages de $S$ ;
  - la relation de sous-typage est contra-variante, c'est-à-dire les arguments des messages de $S$ sont des sous-types des arguments des messages de $T$ ;
  - si $S$ a la modalité **may**, alors $T$ a la modalité **may**. (si $S$ a la modalité **must**, alors il n'y a aucune restriction sur la modalité de $T$).
- Si $T$ et $S$ sont en émission, alors :
  - $T$ envoie au plus les messages de $S$ ;
  - la relation de sous-typage est co-variante, c'est-à-dire les arguments des messages de $T$ sont des sous-types des arguments des messages de $S$ ;
  - si $S$ a la modalité **must**, alors $T$ a la modalité **must**. (si $S$ a la modalité **may**, il n'y a aucune restriction sur la modalité de $T$).

Par exemple, nous avons (avec une syntaxe allégée) :

$$\begin{aligned} \textbf{may ?}\ M_1 + M_2 + M_3 &\preceq \textbf{may ?}\ M_1 + M_2 \\ (\textbf{may}|\textbf{must})\ \textbf{?}\ M_1 + M_2 + M_3 &\preceq \textbf{must ?}\ M_1 + M_2 \\ (\textbf{may}|\textbf{must})\ \textbf{!}\ M_1 &\preceq \textbf{may !}\ M_1 + M_2 \\ \textbf{must !}\ M_1 &\preceq \textbf{must !}\ M_1 + M_2 \end{aligned}$$

Expliquons nos choix quant au sous-typage des modalités. Dans le cas d'une réception, si le super-type est **may?**, alors il n'impose aucune condition sur son partenaire. Le sous-type ne peut donc imposer de contrainte sur le partenaire, ce qui impose que le seul sous-type valable est **may?**. Un raisonnement analogue explique les autres relations de sous-typage.

La relation de sous-typage est telle qu'elle s'applique aussi sur les types des actions qui suivent les émissions et réceptions de message : si $I = \mathbf{may}\ \mathbf{!}\ [\ M_1\ ;I_1\ ]$ est sous-type de $J = \mathbf{may}\ \mathbf{!}\ [\ M_1\ ;J_1\ +\ M_2\ ;J_2\ ]$ alors $I_1$ est sous-type de $J_1$.

Nous montrerons dans la section 3.5 page 51 que cette relation donne bien, avec la compatibilité entre interfaces, la propriété attendue permettant de remplacer une instance de super-type par une instance de sous-type.

La relation de sous-typage est définie formellement par les règles du tableau 3.1 page ci-contre et le prédicat de sous-typage des modalités $\preceq_m$. Trois cas dans ce prédicat méritent quelques lignes d'attention.

**may?** $\preceq_m$ **0** et **0** $\preceq_m$ **may!** montrent bien la sémantique de la modalité **may** : un **may?** peut très bien remplacer une interface inactive **0** (aucun message ne sera envoyé vers l'interface **0**, donc aucun message ne sera envoyé vers l'interface **may?**). Une interface inactive **0** peut de même être considérée comme une interface pouvant envoyer des messages (mais sans obligation).

Les relations de sous-typage sont transitives. Nous avons donc une autre relation pour $\preceq_m$ : **may?** $\preceq_m$ **may!**. Cette relation surprend à la première lecture. Cependant, si l'on reprend la sémantique des modalités, nous avons un super-type qui *peut éventuellement* envoyer, et un sous-type **may?** qui n'enverra aucun message, donc qui respecte la sémantique de **may!**. Cette relation surprend surtout parce que l'on a l'habitude de penser les envois et réceptions de messages avec la modalité **must**.

Une fois que l'on a compris l'essence de la relation $\preceq_m$, les règles découlent des relations classiques de sous-typage. Les six premières règles concernent les types basiques et les interfaces inactives. Les règles ST-MAYRECV et ST-MAYSEND concernent le sous-typage mettant en jeu un type **0** (nous rappelons que $\tilde{I}$ est un n-uplet). La règle ST-RECVSEND concerne la relation **may?** $\preceq_m$ **may!** : on remarquera que la liste des messages est différente entre le sous-type et le super-type.

Les règles ST-ASSUMP, ST-NAME1 et ST-NAME2 prennent en compte les interfaces récursives, et les cas où l'interface est désignée par son nom («replace(*peer_name*)» remplace le nom *peer_name* par la définition du type correspondant). Les trois dernières règles donnent la définition classique des sous-types d'interface, à savoir le sous-type peut recevoir plus, et envoyer moins que son super-type. L'ordre des messages n'est pas considéré comme important.

Concernant les interfaces serveurs, un sous-type peut être serveur (alors que son super-type ne l'est pas), mais le cas inverse n'est pas autorisé, comme le montre la règle ST-RECV* qui est identique à ST-RECV, mais s'applique pour des interfaces serveurs (sous-type *et* super-type). Cette restriction s'explique simplement par le fait que les références serveur sont potentiellement connues de plusieurs ports : si le super-type $S\mathbf{*}$ est une référence serveur, alors le port correspondant peut répondre à plusieurs requêtes, ce qui ne sera pas le cas d'un port dont le sous-type $T \preceq S\mathbf{*}$ n'est pas de type serveur. Le cas contraire, $T\mathbf{*} \preceq S$ ne pose pas ce genre de problème.

On remarquera aisément que l'algorithme de décision de sous-typage se termine

$T_{\text{mod}} \preceq_m S_{\text{mod}}$

| $T_{\text{mod}}$ \ $S_{\text{mod}}$ | must? | may? | must! | may! | 0 |
|---|---|---|---|---|---|
| must? | $\surd$ | | | | |
| may? | $\surd$ | $\surd$ | | $\surd$ | $\surd$ |
| must! | | | $\surd$ | $\surd$ | |
| may! | | | | $\surd$ | |
| 0 | | | | $\surd$ | $\surd$ |

$$\text{ST-INT}\ \frac{}{\Gamma \vdash \textsf{integer} \preceq \textsf{integer}} \qquad \text{ST-BOOL}\ \frac{}{\Gamma \vdash \textsf{boolean} \preceq \textsf{boolean}}$$

$$\text{ST-INT2}\ \frac{}{\Gamma \vdash \textsf{integer} \preceq \textsf{real}} \qquad \text{ST-STRING}\ \frac{}{\Gamma \vdash \textsf{string} \preceq \textsf{string}}$$

$$\text{ST-REAL}\ \frac{}{\Gamma \vdash \textsf{real} \preceq \textsf{real}} \qquad \text{ST-ZERO}\ \frac{}{\Gamma \vdash \textsf{0} \preceq \textsf{0}}$$

$$\text{ST-MAYRECV}\ \frac{}{\Gamma \vdash \textsf{may? [*][ } \Sigma_i M_i(\tilde{I}_i)\textsf{ ; } T_i \textsf{ ]} \preceq \textsf{0}} \qquad \text{ST-ASSUMP}\ \frac{I \preceq J \in \Gamma}{\Gamma \vdash I \preceq J}$$

$$\text{ST-MAYSEND}\ \frac{}{\Gamma \vdash \textsf{0} \preceq \textsf{may ! [ } \Sigma_i M_i(\tilde{I}_i)\textsf{ ; } T_i \textsf{ ]}}$$

$$\text{ST-RECVSEND}\ \frac{}{\Gamma \vdash \textsf{may ? [*][ } \sum_{1 \leqslant i \leqslant n} N_i(\tilde{J}_i)\textsf{ ; } T_i \textsf{ ]} \preceq \textsf{may ! [ } \sum_{1 \leqslant i \leqslant m} M_i(\tilde{I}_i)\textsf{ ; } S_i \textsf{ ]}}$$

$$\text{ST-NAME1}\ \frac{\Gamma, \mathit{peer_name} \preceq I \vdash \text{replace}(\mathit{peer_name}) \preceq I}{\Gamma \vdash \mathit{peer_name} \preceq I}$$

$$\text{ST-NAME2}\ \frac{\Gamma, I \preceq \mathit{peer_name} \vdash I \preceq \text{replace}(\mathit{peer_name})}{\Gamma \vdash I \preceq \mathit{peer_name}}$$

$$\text{ST-RECV}\ \frac{\forall i \in \{1..m\} : \Gamma \vdash \tilde{I}_i \preceq \tilde{J}_i \ \wedge\ \Gamma \vdash T_i \preceq S_i}{\Gamma \vdash \mathit{mod}_T \textsf{ ? [*][ } \sum_{1 \leqslant i \leqslant n} M_i(\tilde{J}_i)\textsf{ ; } T_i \textsf{ ]} \preceq \mathit{mod}_S \textsf{ ? [ } \sum_{1 \leqslant i \leqslant m} M_i(\tilde{I}_i)\textsf{ ; } S_i \textsf{ ]}}\ \Box$$

$$\text{ST-RECV*}\ \frac{\forall i \in \{1..m\} : \Gamma \vdash \tilde{I}_i \preceq \tilde{J}_i \ \wedge\ \Gamma \vdash T_i \preceq S_i}{\Gamma \vdash \mathit{mod}_T \textsf{ ? *[ } \sum_{1 \leqslant i \leqslant n} M_i(\tilde{J}_i)\textsf{ ; } T_i \textsf{ ]} \preceq \mathit{mod}_S \textsf{ ? *[ } \sum_{1 \leqslant i \leqslant m} M_i(\tilde{I}_i)\textsf{ ; } S_i \textsf{ ]}}\ \Box$$

$$\text{ST-SEND}\ \frac{\forall i \in \{1..n\} : \Gamma \vdash \tilde{J}_i \preceq \tilde{I}_i \ \wedge\ \Gamma \vdash T_i \preceq S_i}{\Gamma \vdash \mathit{mod}_T \textsf{ ! [ } \sum_{1 \leqslant i \leqslant n} M_i(\tilde{J}_i)\textsf{ ; } T_i \textsf{ ]} \preceq \mathit{mod}_S \textsf{ ! [ } \sum_{1 \leqslant i \leqslant m} M_i(\tilde{I}_i)\textsf{ ; } S_i \textsf{ ]}}\ \Diamond$$

$$\Box \triangleq (\mathit{mod}_T \textsf{ ?} \preceq_m \mathit{mod}_S \textsf{ ?}) \ \wedge\ n \geqslant m$$
$$\Diamond \triangleq (\mathit{mod}_T \textsf{ !} \preceq_m \mathit{mod}_S \textsf{ !}) \ \wedge\ n \leqslant m$$
$$\tilde{I} \preceq \tilde{J} \triangleq \tilde{I} = (I_i)_{1..k}, \tilde{J} = (J_i)_{1..k} \ \wedge\ \forall i, I_i \preceq J_i$$

**Tab. 3.1:** *Règles de sous-typage*
[*] indique que la construction * peut être présente ou non

toujours, et est au pire quadratique en fonction du nombre d'état des interfaces.

**Propriété 1**
*La relation $\preceq$ est un préordre partiel.*

**Preuve.** La table de vérité de $\preceq_m$ assure la transitivité et la réflexivité parmi les modalités. Le reste de la démonstration se fait par induction structurelle. □

## 3.4 Règles de compatibilité

Dans cette section, nous définissons le prédicat symétrique $Comp(I, J)$ comme étant : "$I$ et $J$ sont des interfaces compatibles entre elles". La compatibilité entre les interfaces $I$ et $J$ est définie comme suit (de manière informelle) :

$I =$ **must ?** $m$ implique $J =$ **must !** $m$
$I =$ **may ?** $m$ implique $J =$ **must !** $m$ ou $J =$ **may !** $m$ ou $J =$ **0** ou $J =$ **may ?** $m'$
$I =$ **must !** $m$ implique $J =$ **must ?** $m$ ou $J =$ **may ?** $m$
$I =$ **may !** $m$ implique $J =$ **may ?** $m$
$I =$ **0** implique $J =$ **may ?** $m$ ou $J =$ **0**

On remarquera que le fait que **may?** soit compatible avec **may?** est entièrement dans la logique de la sémantique des modalités : aucune des deux interfaces ne recevra de message.

Les règles de compatibilité sont définies à partir de plusieurs relations élémentaires de compatibilité : entre modalités, messages et types.

Nous définissons tout d'abord la compatibilité entre les modalités comme la relation symétrique booléenne : $Comp_{\text{mod}}(mod_I\ [!|?], mod_J\ [!|?])$. Sa table de vérité est donnée ci-après :

| | $I$ | | | | |
|---|---|---|---|---|---|
| $J$ | **must?** | **may?** | **must!** | **may!** | **0** |
| **must?** | | | √ | | |
| **may?** | | √ | √ | √ | √ |
| **must!** | √ | √ | | | |
| **may!** | | √ | | | |
| **0** | | √ | | | √ |

Nous définissons aussi $Comp_{\text{msg}}$, une relation de compatibilité sur les types de message. Deux types de message sont compatibles si et seulement si ils ont le même nom et leurs arguments sont liés de telle sorte que les arguments des messages envoyés sont des sous-types des arguments des messages reçus. La définition formelle donne[2] :

$$\begin{aligned} Comp_{\text{msg}}(M_!, M_?) &\triangleq Comp_{\text{msg}}(M_!(I_1, \ldots, I_n), M_?(J_1, \ldots, J_n)) \\ &\triangleq M_! = M_? \ \wedge\ \forall i, I_i \preceq J_i \end{aligned}$$

[2]avec $M_!$ le message en émission et $M_?$ le message en réception. Cette distinction est nécessaire de part la relation de sous-typage imposée au niveau des arguments.

Nous pouvons dès lors définir la relation de compatibilité $Comp(I, J)$ entre deux interfaces comme étant la compatibilité entre les modalités et les messages, et telle que les transitions doivent mener à des interfaces compatibles. Ceci est formellement défini récursivement comme suit (avec $\rho \in \{\mathbf{?}, \mathbf{!}\}$, et où [*****] indique que la construction ***** peut être présente ou non) :

$$
\begin{array}{rcl}
Comp(I, J) & \triangleq & Comp(J, I) \\
Comp(\mathbf{0}, \mathbf{0}) & \triangleq & true \\
Comp(\mathbf{0}, mod_J\ \rho_J\ [\mathbf{*}][\ \Sigma_l\ M'_l\ ;\ J_l\ ]\ ) & \triangleq & Comp_{\text{mod}}(\mathbf{0}, mod_J\ \rho_J) \\
Comp(\mathbf{may?}\ [\mathbf{*}][\ \Sigma_l\ M_k\ ;\ I_k\ ], \\ \quad \mathbf{may?}\ [\mathbf{*}][\ \Sigma_l\ M'_l\ ;\ J_l\ ]) & \triangleq & true \\
Comp(mod_I\ \mathbf{!}\ [\ \Sigma_k\ M_k\ ;\ I_k\ ]\ , \\ \quad mod_J\ \mathbf{?}\ [\mathbf{*}][\ \Sigma_l\ M'_l\ ;\ J_l\ ]\ ) & \triangleq & Comp_{\text{mod}}(mod_I\ \mathbf{!}, mod_J\ \mathbf{?}) \\
 & & \wedge\ \big(\forall k, \exists l : Comp_{\text{msg}}(M_k, M'_l) \wedge Comp(I_k, J_l)\big)
\end{array}
$$

Cette définition récursive montre que la compatibilité d'une paire d'interfaces est une fonction booléenne de la compatibilité d'un ensemble fini de paires d'interfaces, qui correspondent aux interfaces résultant des transitions et des interfaces référencées dans les messages. Cela signifie que la vérification de la compatibilité est similaire à la vérification de l'équivalence d'automates finis, et par conséquent se termine toujours. Elle peut être effectuée par les techniques standards[3], avec une complexité quadratique selon le nombre d'interfaces (représentant les différents états des interfaces). Enfin, de par l'abstraction utilisée dans la définition des interfaces, ce nombre est présumé petit par rapport à la complexité du comportement du composant (c'est-à-dire par rapport au nombre d'état de ce dernier).

## 3.5 Propriétés des sous-types

Dans cette section, nous montrons la propriété classique des sous-types, à savoir qu'ils peuvent remplacer le super type, soit :

**Propriété 2**
*Si* $T \preceq S$ *Alors* $\forall I, Comp(I, S)$ *implique* $Comp(I, T)$

Cette propriété veut bien dire que le remplacement de $S$ par $T$ n'a aucune conséquence sur le partenaire du super-type. La relation de sous-typage peut alors être utilisée dans le cas où un serveur se spécialise, et dans les relations de délégation des composants composites.

**Lemme 1 (sous-typage et modalités)**
*Si* $T_{\text{mod}} \preceq_m S_{\text{mod}}$ *Alors* $\forall I_{\text{mod}}, Comp_{\text{mod}}(I_{\text{mod}}, S_{\text{mod}})$ *implique* $Comp_{\text{mod}}(I_{\text{mod}}, T_{\text{mod}})$

**Preuve.** Immédiat d'après les tables de vérité des relations $\preceq_m$ et $Comp_{\text{mod}}$ (en ne s'intéressant qu'aux cas où les modalités de $T$ et $S$ sont différentes, nous avons donc 5 cas). □

[3] par exemple les diagrammes de décision binaire [MKB98, Bry95].

**Preuve (propriété 2).**

Par induction sur la structure de $T$. Nous omettons les cas où $S$ est une référence serveur, mais la démonstration est identique.

Soient $T, S$ et $I$ tels que $T \preceq S$ et $Comp(I, S)$. Nous devons montrer que $Comp(I, T)$ est vraie. Trois cas sont possibles.

1. $S$ est en réception.
   Notons $S = mod_S\ \mathbf{?}\ [\ \sum_{1 \leqslant l \leqslant m} M_l^S(\tilde{J}_l^S)\ ;\ S_l\ ]$.
   Les super-types en réception ne sont concernés que par les règles ST-RECV et ST-RECV* pour lesquelles nous avons $T = mod_T\ \mathbf{?}\ [\mathbf{*}][\ \sum_{1 \leqslant l \leqslant n} M_l^T(\tilde{J}_l^T)\ ;\ T_l\ ]$ avec :

   $$n \geqslant m \tag{3.1}$$
   $$mod_T\ \mathbf{?} \preceq_m mod_S\ \mathbf{?} \tag{3.2}$$
   $$\text{et } \forall l \in \{1..m\} : \quad \tilde{J}_l^S \preceq \tilde{J}_l^T, \quad T_l \preceq S_l, \quad M_l^T = M_l^S \tag{3.3}$$

   La compatibilité entre $I$ et $S$ présente les cas suivants :

   (a) $I = \mathbf{0}$ ou $I = \mathbf{may?}\ [\mathbf{*}][\ \sum_k M_k^I(\tilde{J}_k^I)\ ;\ I_k\ ]$.
   Nous avons alors $Comp(I, T) = Comp_{\text{mod}}(mod_I, mod_T\ \mathbf{?})$, relation de compatibilité qui est vraie par le lemme 1 page précédente (en fait, $mod_S = \mathbf{may?} = mod_T$).

   (b) $I = mod_I\ \mathbf{!}\ [\ \sum_k M_k^I(\tilde{J}_k^I)\ ;\ I_k\ ]$. $Comp(I, S)$ s'écrit alors :

   $$Comp_{\text{mod}}(mod_I\ \mathbf{!}, mod_S\ \mathbf{?}) \tag{3.4}$$
   $$\wedge\ \ \forall k, \exists l : Comp_{\text{msg}}(M_k^I, M_l^S) \wedge Comp(I_k, S_l) \tag{3.5}$$

   D'après le lemme 1 page précédente, le terme 3.4 de la conjonction implique $Comp(mod_I\mathbf{!}, mod_T\mathbf{?})$.
   Quant au terme 3.5, nous avons $\forall k, \exists l : Comp_{\text{msg}}(M_k^I, M_l^S) \wedge Comp(I_k, S_l)$.
   Or par définition

   $$\begin{aligned} Comp_{\text{msg}}(M_k^I, M_l^S) &\triangleq M_k^I = M_l^S \wedge \tilde{J}_k^I \preceq \tilde{J}_l^S \\ &\triangleq M_k^I = M_l^T \wedge \tilde{J}_k^I \preceq \tilde{J}_l^T \quad \text{par 3.3 et la transitivité de } \preceq \end{aligned}$$

   De plus, comme $Comp(I_k, S_l)$ est vraie, nous avons, par induction ($T_l \preceq S_l$) et par le lemme 1 page précédente, $Comp(I_k, T_l)$.
   Nous avons donc bien la propriété :

   $$Comp(mod_I\mathbf{!}, mod_T\mathbf{?}) \wedge \big(\forall k, \exists l : Comp_{\text{msg}}(M_k^I, M_l^T) \wedge Comp(I_k, T_l)\big)$$

2. $S$ est en émission.
   Le raisonnement est similaire, aussi nous le détaillons avec moins de précision :
   Nous écartons le cas trivial où $S$ a la modalité **may!** et $T$ la modalité **may?** : dans ce cas $I$ a la modalité **may?**, donc $Comp(I, T)$ est vraie. Le cas $T = \mathbf{0}$ est lui aussi immédiat ($S$ a la modalité **may!**).
   Dans les autres cas de la démonstration, $T$ est obligatoirement en émission. Notons $S = mod_S\ \mathbf{!}\ [\sum_{1 \leqslant l \leqslant m} M_l^S(\tilde{J}_l^S)\ ; S_l]$ et $T = mod_T\ \mathbf{!}\ [\sum_{1 \leqslant l \leqslant n} M_l^T(\tilde{J}_l^T)\ ; T_l]$, avec $n \leqslant m$, $mod_T\ \mathbf{!} \preceq_m mod_S\ \mathbf{!}$ et $\forall i \in \{1..n\} : \tilde{J}_i^T \preceq \tilde{J}_i^S \wedge T_i \preceq S_i \wedge M_i^T = M_i^S$.

$I$ ne peut être qu'en réception : $I = mod_I$ **?** **[** $\sum_k M_k^I(\tilde{J}_k^I)$ **;** $I_k$ **]**. La compatibilité entre $I$ et $S$ donne :

$$Comp_{\text{mod}}(mod_S\ \mathbf{!}, mod_I\ \mathbf{?}) \wedge \forall k \in \{1..m\}, \exists l : Comp_{\text{msg}}(M_k^S, M_l^I) \wedge Comp(S_k, I_l)$$

Le deuxième terme implique (puisque $n \leqslant m$) $\forall k \leqslant n, \exists l : Comp_{\text{msg}}(M_k^S, M_l^I) \wedge Comp(S_k, I_l)$. Comme $M_k^T = M_k^S$, $\tilde{J}_k^S \preceq \tilde{J}_k^T$ et $T_k \preceq S_k$, nous sommes sûr que pour tout message de $T$, il existe un message (le même que pour $S$) de $I$ vérifiant $M_k^T = M_l^I \wedge \tilde{J}_k^T \preceq \tilde{J}_l^I \wedge Comp(T_k, I_l)$. Le lemme 1 page 51 permet enfin de conclure :

$$Comp_{\text{mod}}(mod_T\ \mathbf{!}, mod_I\ \mathbf{?}) \wedge \forall k \in \{1..n\}, \exists l : Comp_{\text{msg}}(M_k^T, M_l^I) \wedge Comp(T_k, I_l)$$

3. $S$ n'a pas d'action ($S = \mathbf{0}$).
   Alors $T$ a la modalité **may?**, et $I$ a soit la modalité **may?**, soit est inactif ($\mathbf{0}$). $Comp(I, T)$ est immédiat.

□

## 3.6 Dualité d'interfaces

La relation de compatibilité fait penser à une relation de dualité. D'ailleurs, Vallecillo et Al. [VVR02] donnent un système de dualité (qui correspond assez bien à notre relation de compatibilité), mais dans lequel le dual n'est pas unique. Nous définissons une relation de dualité, notée $.^{\mathcal{D}}$, définie pour les interfaces pairs, telle que $\mathrm{I}^{\mathcal{D}}$ soit unique et compatible avec $I$. Le dual de $I$ se calcule simplement en inversant les réceptions et émissions respectivement en émissions et en réceptions (Yoshida adopte la même relation de dualité pour les types des canaux de communication dans [Yos02]) :

$$\mathbf{0}^{\mathcal{D}} \triangleq \mathbf{0}$$
$$\left(mod\ \mathbf{!}\ \mathbf{[}\ \Sigma_i M_i(\tilde{J}_i)\ \mathbf{;}\ I_i\ \mathbf{]}\right)^{\mathcal{D}} \triangleq mod\ \mathbf{?}\ \mathbf{[}\ \Sigma_i M_i(\tilde{J}_i)\ \mathbf{;}\ {I_i}^{\mathcal{D}}\ \mathbf{]}$$
$$\left(mod\ \mathbf{?}\ \mathbf{[}\ \Sigma_i M_i(\tilde{J}_i)\ \mathbf{;}\ I_i\ \mathbf{]}\right)^{\mathcal{D}} \triangleq mod\ \mathbf{!}\ \mathbf{[}\ \Sigma_i M_i(\tilde{J}_i)\ \mathbf{;}\ {I_i}^{\mathcal{D}}\ \mathbf{]}$$

On remarque aisément les propriétés suivantes, dont les trois premières sont évidentes :

**Propriété 3 (dual)**

*3.1.* $I^{\mathcal{D}}$ *est unique ;*

*3.2.* $(I^{\mathcal{D}})^{\mathcal{D}} = I$ *;*

*3.3.* $Comp(I^{\mathcal{D}}, I)$ *;*

*3.4.* $Comp(I, J) \Leftrightarrow I \preceq J^{\mathcal{D}}$ *;*

*3.5.* $I \preceq J \Leftrightarrow J^{\mathcal{D}} \preceq I^{\mathcal{D}}$

La propriété 3.4 page précédente montre que, pour le type $J$, le plus grand super-type compatible avec $J$ est le dual $J^{\mathcal{D}}$.

**Preuve (propriété 3.4).**

**1.** $\Rightarrow$

La preuve est similaire à celle de la propriété 2 page 51, nous ne donnerons que les grandes lignes.

Nous prenons le cas où $I$ est en émission. Nous avons alors :

$$I = mod_I\mathbf{![}\Sigma_{1\leqslant k\leqslant n} M_k^I(\tilde{I}'_k)\,;\, I_k\,\mathbf{]}$$

$$J = mod_J\mathbf{?[}\Sigma_{1\leqslant l\leqslant m} M_l^J(\tilde{J}'_l)\,;\, J_l\,\mathbf{]} \qquad J^{\mathcal{D}} = mod_J\mathbf{![}\Sigma_{1\leqslant l\leqslant m} M_l^J(\tilde{J}'_l)\,;\, {J_l}^{\mathcal{D}}\,\mathbf{]}$$

La définition de la compatibilité donne :

$$Comp_{\text{mod}}(mod_I\mathbf{!}, mod_J\mathbf{?}) \tag{3.6}$$

$$\forall k \in \{1..n\}, \exists l : M_l^I = M_k^J \wedge \tilde{I}'_k \preceq \tilde{J}'_l \wedge Comp(I_k, J_l) \tag{3.7}$$

De 3.6 on déduit facilement par les tables de vérité $mod_I\mathbf{!} \preceq mod_J\mathbf{!}$. De 3.7 on déduit facilement $n \leqslant m$, et moyennant un réarrangement des messages nous avons bien :

$$mod_I\mathbf{![}\Sigma_{1\leqslant k\leqslant n} M_k^I(\tilde{I}'_k)\,;\, I_k\,\mathbf{]} \preceq mod_J\mathbf{![}\Sigma_{1\leqslant l\leqslant m} M_l^J(\tilde{J}'_l)\,;\, J_l\,\mathbf{]}$$

avec $\tilde{I}'_k \preceq \tilde{J}'_l, Comp(I_k, J_l)$ qui est vraie et l'hypothèse d'induction qui donne $I_k \preceq {J_l}^{\mathcal{D}}$.

Le cas $I = \mathbf{0}$ est semblable : $J$ a alors soit la forme $\mathbf{0}$, soit la forme $\mathbf{may\ ?\ [}\ldots\mathbf{]}$, dont le dual est bien super-type de $\mathbf{0}$.

Le cas $I$ en réception se traite comme le cas $I$ en émission : le cas $J$ en émission est identique. Dans le cas où $J$ a la modalité $\mathbf{may?}$, alors $I$ a la même modalité, et la relation de sous-typage à montrer est $\mathbf{may?} \preceq \mathbf{may!}$, qui est vraie. Pour le dernier cas, $J = \mathbf{0}$ et $I = \mathbf{may?[}\ldots\mathbf{]}$, nous avons bien $I \preceq J^{\mathcal{D}}$.

**2.** $\Leftarrow$

Nous avons $Comp(J^{\mathcal{D}}, J)$ (propriété 3.3 page précédente), d'où nous déduisons $Comp(I, J)$ (propriété 2 page 51). □

**Preuve (propriété 3.5 page précédente).**

$I \preceq J \Leftrightarrow Comp(I, J^{\mathcal{D}}) \Leftrightarrow J^{\mathcal{D}} \preceq I^{\mathcal{D}}$ □

### 3.6.1 Dualité et références serveurs

Notre fonction de dualité peut être étendue aux références serveurs de la manière suivante :

$$\left(mod\ \mathbf{?*[}\Sigma_i M_i(\tilde{J}_i)\,\mathbf{;}I_i\mathbf{]}\right)^{\mathcal{D}} \triangleq mod\ \mathbf{![}\Sigma_i M_i(\tilde{J}_i)\,\mathbf{;}{I_i}^{\mathcal{D}}\mathbf{]}$$

dans lequel le dual perd l'information donnée par le mot-clé «*****». Le dual est bien unique, mais nous ne pouvons affirmer pour une interface serveur $I$ que $(I^{\mathcal{D}})^{\mathcal{D}} = I$. Cependant, les autres propriétés sont conservées, notamment $Comp(I, J) \Leftrightarrow I \preceq J^{\mathcal{D}}$.

## 3.7 Etude de Cas

Pour simplifier, nous considérerons que les rapporteurs n'ont qu'un seul article à critiquer. De plus, les types pour les fichiers et les formulaires sont considérés comme des suites de chaîne de caractère, de type **strings**. Ce type n'est pas présent dans notre syntaxe, mais son ajout est évident.

Cet exemple illustre les principales fonctionnalités de notre langage d'interface, ainsi que les relations entre les types : les modalités **may** et **must**, la relation de sous-typage $\preceq$, la relation de dualité $\cdot^{\mathcal{D}}$, l'envoi de référence et la compatibilité des interfaces $Comp(\cdot, \cdot)$.

*Le reste de cette page est volontairement vide.*

### 3.7.1 Spécifications du composant Rapporteur

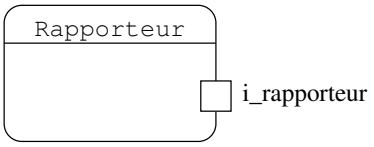



**Fig. 3.1:** *Composant rapporteur*

Le composant `Rapporteur` a un seul port, typé *i_ rapporteur*[4].

Le type *i_ rapporteur* ne peut envoyer qu'un seul message, *accès_ rapporteur*, prenant en paramètre le nom et le mot de passe du rapporteur (type *string*), et le numéro de l'article à critiquer (*integer*). Après l'envoi de ce message, l'interface attend deux messages ; *accordé* contient le formulaire (type *strings*) et permet à l'interface de passer au type *entrer_ rapport*, tandis que le message *refusé* rend l'interface inactive (**0**).

Le type *entrer_ rapport* permet d'interagir avec le composant `Article`, pour entrer la première version du rapport. Ce type peut envoyer deux messages. Le premier est *rapport*, et contient la critique du rapporteur (type *strings*) ; les messages attendus en retour sont *OK* (et l'interface est rendue inactive par **0**), et *erreur_ rapport* avec en argument la raison de l'erreur (dans ce cas le rapporteur doit à nouveau entrer la première version de son rapport). Le second message que *entrer_ rapport* peut envoyer est *demande_ article* ; la réponse attendue contient le fichier correspondant (type *strings*), et le type peut revenir à sa fonction initiale. Le message *demande_ article* est en fait optionnel, puisque le type revient à son état de départ.

```
i_rapporteur = must ! [ accès_rapporteur (string, string, integer) ;
                                must ? [   accordé (strings) ; entrer_rapport
                                         + refusé ; 0
                                       ]
                      ]
entrer_rapport =
      must ! [   rapport (strings) ; must ? [   Ok ; 0
                                              + erreur_rapport(string) ; entrer_rapport
                                            ]
               + demande_article ; must ? [ article (strings) ; entrer_rapport ]
             ]
```

[4] le «*i*» (comme interface) de *i_ rapporteur* permet de distinguer le nom du composant `Rapporteur` du nom du type de son unique port.

### 3.7.2 Spécifications du composant Gestion

Le composant `Gestion` s'occupe de gérer l'accès pour les rapporteurs et les auteurs. Il a deux interfaces : *gestion_ accès* pour les accès, et *envoi_ réf* pour l'envoi de référence vers le composant `Article`.

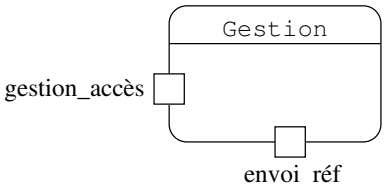



**Fig. 3.2:** *Composant Gestion*

Le type serveur *gestion_ accès* peut recevoir l'un des deux messages *accès_ rapporteur* ou *accès_ auteur*. Le premier message permet d'identifier le rapporteur et l'article qu'il veut critiquer ; l'interface doit ensuite soit envoyer le message *accordé* avec le formulaire et passer au type *gestion_ rapporteur*, soit le message *refusé* et s'arrêter. L'accès pour les auteurs renvoie vers le type *gestion_ auteur*. Les deux types résultant de la réception de l'un des deux messages sont décrits dans la sous-section suivante.

gestion_accès = **may ? *[** accès_rapporteur (string, string, integer) ;
**must ! [** accordé (strings) ; gestion_rapporteur
+ refusé ; **0**
**]**
+ accès_auteur ; gestion_auteur
**]**

Le composant `Gestion` ne s'occupe que de l'authentification des rôles rapporteur et auteur, et de distribuer les accès adéquats. Dans le cas où le rapporteur n'est pas correctement identifié, `Gestion` envoie le message *refusé*. Dans le cas où le rapporteur est correctement identifié, l'envoi du message *accordé* est délégué au composant `Article` : la référence du rapporteur est envoyée au composant `Article` correspondant à l'article à relire. Cela se fait par l'interface *envoi_ réf*[5]. Or, la référence envoyée est telle qu'elle attend soit un message *refusé*, soit un message *accordé* ; comme le rapporteur est correctement identifié, le type *rapporteur_ accepté* que l'on assigne à la référence envoyée ne prend pas en compte le message *refusé*. Nous avons là un exemple de sous-typage : le véritable type de la référence envoyée (**must ? [**accordé...+ refusé...**]**) est sous-type du type de l'argument du message *rapporteur* (**must ? [**accordé...**]**).

envoi_réf = **must ! [** rapporteur (rapporteur_accepté) ; envoi_réf **]**
rapporteur_accepté = **must ? [** accordé (strings) ; entrer_puis_changer_rapport **]**
entrer_puis_changer_rapport = gestion_rapporteur$^{\mathcal{D}}$

Le type *gestion_ rapporteur* est présenté dans la section suivante. La différence entre *entrer_ rapport* et *entrer_ puis_ changer_ rapport* est que le premier n'effectue qu'un seul envoi de rapport, tandis que le deuxième autorise un nombre infini de modification. Cette différence est dûe au fait que, pour le composant `Gestion`, le rapporteur peut modifier indéfiniment le rapport, ce qui est reflété dans le type que le composant `Gestion` associe au rapporteur.

[5] le type est récursif pour permettre plusieurs envois, donc plusieurs rapporteurs.

### 3.7.3 Spécifications du composant Article

Le composant `Article` a deux interfaces : l'une pour recevoir la référence du rapporteur (typée *réception_réf*), et l'autre pour interagir avec ce rapporteur. Le type de cette interface, *gestion_rapporteur_ok*, envoie le message de la dernière phase d'authentification (*accordé*) puis devient le type *gestion_rapporteur*. On remarquera que *gestion_rapporteur_ok* est le type dual de *rapporteur_accepté*.

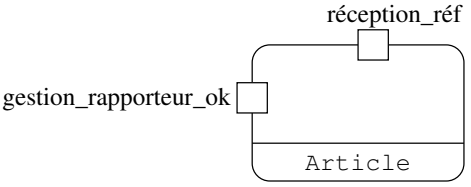



**Fig. 3.3:** *Composant Article*

Le type *gestion_rapporteur* est le pendant de *entrer_rapport*. La différence essentielle est l'action qui suit le message *OK* : lorsque ce message est envoyé, le type devient *gestion_rapport_chg*, qui est exactement comme le type *gestion_rapporteur*, excepté pour la modalité de la première action : **may ?** au lieu de **must ?**. Les types sont tels que le rapporteur est obligé d'envoyer son rapport, et peut le modifier par la suite en utilisant le même protocole.

```
réception_réf = may ? [ rapporteur (rapporteur_accepté) ;  réception_réf  ]
gestion_rapporteur_ok = must ! [ accordé (strings) ; gestion_rapporteur ]
gestion_rapporteur =
    must ? [   rapport (strings) ;
                  must ! [   Ok ; gestion_rapporteur_chg
                           + erreur_rapport(string) ; gestion_rapporteur
                         ]
             + demande_article ; must ! [ article (strings) ; gestion_rapporteur ]
           ]
gestion_rapporteur_chg =
    may ? [   rapport (strings) ;
                  must ! [   Ok ; gestion_rapporteur_chg
                           + erreur_rapport(string) ; gestion_rapporteur_chg
                         ]
             + demande_article ; must ! [ article (strings) ; gestion_rapporteur_chg ]
           ]
```

Nous donnons le type *gestion_auteur* pour information. On remarque qu'une fois que l'auteur envoie la version finale, il est dans l'obligation de s'inscrire.

```
gestion_auteur =
      may ? [ soumission (...) ;
                  must ! [   accepté (strings) ;
                                 must ? [ article_imprimable (strings) ;
                                              must ? [ inscription(...) ; 0 ] ]
                           + refusé (strings) ; 0
                         ]
            ]
```

### 3.7.4 Relations entre les types de l'étude de cas

Nous avons les relations suivantes entre les types :

| | |
|---|---|
| Compatibilités à vérifier à l'assemblage | $Comp$(i_rapporteur, gestion_accès)<br>$Comp$(envoi_réf, réception_réf) |
| Autres compatibilités | $Comp$(rapporteur_accepté, gestion_rapporteur_ok)<br>$Comp$(rapporteur_encours, gestion_rapporteur_ok)<br>$Comp$(entrer_rapport, gestion_rapporteur)<br>$Comp$(entrer_rapport, gestion_rapporteur_chg) |
| autres relations | rapporteur_encours $\preceq$ rapporteur_accepté<br>entrer_rapport $\preceq$ entrer_puis_changer_rapport<br>rapporteur_accepté = gestion_rapporteur_ok$^{\mathcal{D}}$ |

avec :
rapporteur_encours = **must ? [** accordé (strings) **;** entrer_rapport **+** refusé **; 0 ]**

## 3.8 Conclusion

Nous venons de donner un langage de type comportemental pour les interfaces d'un composant, et de définir une relation de sous-typage, et une relation de compatibilité entre les types. Ces relations ont été prototypées en Python (`http://www.python.org`). Notre langage permet de décrire de manière formelle les interactions avec le composant. Le projet SOFA [Viš02, PV02, AP03] adopte la même syntaxe mais ne propose pas l'envoi de références ; de plus, leur langage porte sur la séquence des méthodes que le composant exige. Notre langage, plus souple, permet la spécification d'une interface qui effectue plusieurs envois de messages les uns à la suite des autres, et permet de prendre en compte différents retours possibles pour une même méthode. On notera également qu'il est possible de spécifier les interfaces de type flux ou événements avec un type récursif. Dès lors, toutes les interfaces des composants EJB ou CCM peuvent être typées avec notre langage.

La notion de port requis/fourni est différente avec notre langage de type. Elle est valable au moment de l'assemblage, mais évolue au cours du temps. Ainsi, un port en envoi peut être défini comme requis, tandis que celui en réception peut être défini comme fourni.

Notre langage d'interface a toutefois quelques limitations : il est impossible d'envoyer et de recevoir des messages en même temps, et il n'est pas possible d'utiliser les deux modalités en même temps. Par exemple, $I =$ (**must?** $M$)+(**may!** $N$) n'est pas permis.

Chapitre 4

# Modèle de composant

Nous décrivons dans ce chapitre un modèle de composant se voulant le plus général possible. Les modèles de composants existants sont déjà trop spécifiques, ou manipulent trop de concepts (notamment, nous ne nous intéressons dans ce chapitre que au comportement interne du composant : la notion de type est donc logiquement absente). Nous sommes partis d'une simplification du modèle de composant UML2.0 : un composant interagit avec son environnement à l'aide de ports. Nous avons enrichi ces notions au minimum pour avoir une abstraction suffisante du comportement du composant. Cette abstraction est telle qu'elle nous permettra de vérifier par la suite que le composant respecte le type de ses ports.

Notre modèle abstrait de composant comporte les notions de ports, références de ports et tâches d'exécution ; les relations entre ces trois notions nous donnent une abstraction du comportement du composant et la façon dont il communique avec son environnement.

La première section de ce chapitre présente plus en détail, et de manière informelle, notre modèle de composant. Les deux sections suivantes introduisent les notations utilisées pour la sémantique des composants et pour la sémantique d'un medium de communication. La section 4.4 décrit la sémantique des composants. Un exemple termine ce chapitre.

## 4.1 Présentation informelle

Notre modèle décrit un système comme une configuration de composants communiquants. Chaque composant possède un ensemble de ports, un ensemble de références vers des ports (ports du même composant ou d'autres composants), et la communication se fait par envoi asynchrone de messages entre les ports. L'envoi de message ne peut avoir lieu que si le destinataire est connu de l'émetteur ; pour cela nous introduisons le concept de "port attaché à un autre port *partenaire*". Si un port est attaché, tout message envoyé par ce port est dirigé vers le port partenaire ; par contre, un port qui n'est pas attaché ne peut qu'effectuer des réceptions. Nous considérons de plus des configurations dynamiques : un composant peut créer de nouveaux ports, et dynamiquement attacher une référence d'un port partenaire à l'un de ses ports. Nous représentons graphiquement ces notions selon la figure 4.1 page suivante.

Dans notre environnement, les communications de type client/serveur et point-à-

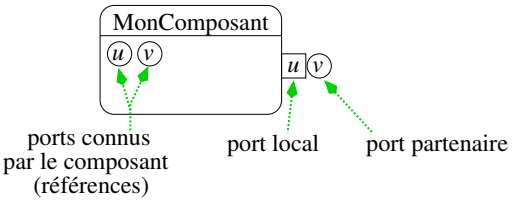



**FIG. 4.1:** *Notation graphique pour les composants*

point (ou *peer-to-peer*) peuvent être modélisées. Dans le premier cas, l'attachement est asymétrique : le port attaché à un autre est le client, et le port qui n'est pas attaché est le serveur. Plusieurs clients peuvent communiquer avec le port serveur (liaison 1-n). Dans les autres communications, les ports sont pairs (*peer*), et l'attachement est symétrique ; bien entendu, pour préserver la propriété point-à-point (ou liaison 1-1), une référence vers un port pair est considéré comme étant privée [NNS99a], c'est-à-dire qu'elle n'est connue que d'au plus deux composants (le composant propriétaire du port et celui qui va interagir avec ce port). De plus, les liaisons sont autorisées entre deux ports appartenant au même composant.

La figure 4.2 présente une configuration de trois composants, $C_1$, $C_2$ et $C_3$. On remarquera la liaison client/serveur entre le port $c$ de $C_1$ et le port $s$ de $C_2$ (l'étiquette «*» indique ici que la référence $s$ est un serveur), et les liaisons point-à-point entre les ports $x$ et $y$ (respectivement de $C_2$ et $C_3$), et entre les ports $u$ et $v$, tous deux appartenant à $C_1$.

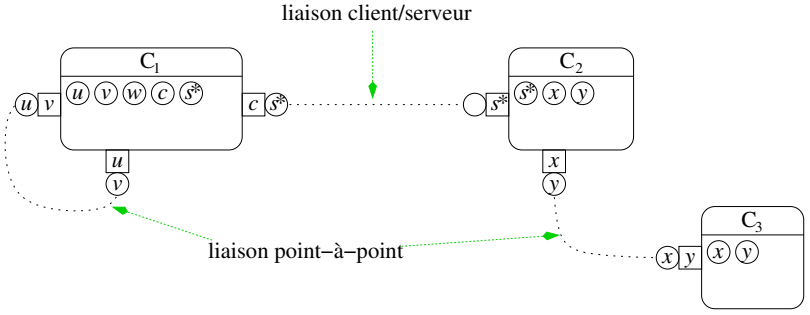



**FIG. 4.2:** *Un exemple de configuration*

Les composants sont multi-tâches. Nous considérons ici un modèle de tâches abstraites, en se concentrant uniquement sur les exécutions externes portant sur les ports, et les relations de dépendances entre les ports (un port attend un résultat sur un autre port). Ainsi, une tâche active est une chaîne dont la tête représente le port actif, et la queue une séquence ordonnée de ports suspendus. Cette chaîne peut varier dynamiquement : elle augmente lorsque le port en tête est suspendu et l'activité est donnée à un autre port ; elle diminue lorsque le port en tête est retiré de la chaîne (il se termine ou devient oisif) et le port précédent devient actif. De plus, les différentes tâches peuvent se supendre entre elles.

## 4.2 Notations pour les Composants

Comme précisé dans la section précédente, un composant comporte :
- un ensemble de *ports*. Ces ports représentent les interfaces du composant ;
- un ensemble de *références* vers des ports partenaires et vers les ports locaux ;
- une collection de *tâches* ;
- un *état*.

### 4.2.1 Ports et Références

L'ensemble des références est noté $\mathsf{R}$, ses éléments sont notés $u, v, w, s^{\textbf{*}}$. L'estampille «*****» est une indication comme quoi la référence est un port serveur. La notation ensembliste classique est utilisée pour les opérations sur $\mathsf{R}$ ; cependant, nous utilisons la notation abusive $\mathsf{R} \cup u$ pour l'insertion d'un élément sans qu'il n'y ait de doublons.

L'ensemble des ports locaux est noté $\mathsf{P}$. Nous considérerons $\mathsf{P}$ comme un ensemble de correspondances des ports vers les références de partenaires. Les éléments de $\mathsf{P}$ sont notés $(\cdot \multimap \cdot)$. Si un port $u$ est attaché à un partenaire $v$, alors nous aurons $(u \multimap v) \in \mathsf{P}$. Si le port $u$ n'est attaché à aucun partenaire, alors nous aurons $(u \multimap \bot) \in \mathsf{P}$. Nous écrirons aussi $u \in \mathsf{P}$ pour l'appartenance du port local $u$ à $\mathsf{P}$.

Par exemple, l'ensemble des ports et des références des composants de la figure 4.2 page ci-contre est décrit comme suit :

| | | |
|---|---|---|
| $\mathsf{R}_1 = \{u, v, w, c, s^*\}$ | $\mathsf{R}_2 = \{s^*, x, y\}$ | $\mathsf{R}_3 = \{x, y\}$ |
| $\mathsf{P}_1 = \{(u \multimap v), (v \multimap u), (c \multimap s^*)\}$ | $\mathsf{P}_2 = \{(s^* \multimap \bot), (x \multimap y)\}$ | $\mathsf{P}_3 = \{(y \multimap x)\}$ |

Les notations suivantes seront utilisées pour la manipulation des ports :

| | |
|---|---|
| $\mathsf{P}[u \multimap \bot]$ | le port $u$ est rajouté à $\mathsf{P}$. |
| $\mathsf{P}[u \multimap v]$ | attache le partenaire $v$ au port $u$. Efface le partenaire précédent. |
| $(u \multimap v) \in \mathsf{P}$ | le port $u$ est dans $\mathsf{P}$, et attaché à $v$. |
| $\mathsf{P} \backslash u$ | retire le port $u$ de $\mathsf{P}$. |

### 4.2.2 Tâches

Nous cherchons à avoir une abstraction de l'état d'un port. Pour cela, nous définissons l'ensemble des tâches du composant, noté $\mathsf{T}$. C'est un ensemble de tâches parallèles, reflétant les dépendances entre les ports du composant. Un port $u$ dépend d'un port $v$ lorsqu'une action de $u$ n'a pas été terminée, et que cette action a besoin que le port $v$ interagisse avec son partenaire ($u$ se *suspend* à $v$). Par exemple, dans un composant de type *proxy*, si un port $u$ reçoit une requête d'un client, $u$ va se suspendre au port $v$ qui va transmettre la requête au serveur et récupérer la réponse ; ensuite le port $u$ va devenir à nouveau actif pour répondre au client. L'état d'un port est une abstraction de son action (envoi, réception ou pas d'action) et de son activité (activé, suspendu ou oisif (*idle*)). Formellement, l'état d'un port $u$ est noté $u\rho^\sigma$, avec $\rho$ son action et $\sigma$ son activité, tels que :

$$\rho = \begin{cases} \mathsf{!} & u \text{ est dans un état d'envoi} \\ \mathsf{?} & u \text{ est dans un état de réception} \\ \mathsf{0} & u \text{ n'a pas d'action} \end{cases} \quad \text{et } \sigma = \begin{cases} \mathsf{a} & u \text{ est actif} \\ \mathsf{s} & u \text{ est suspendu (à un autre port)} \\ \mathsf{i} & u \text{ est oisif } (\textit{idle}) \end{cases}$$

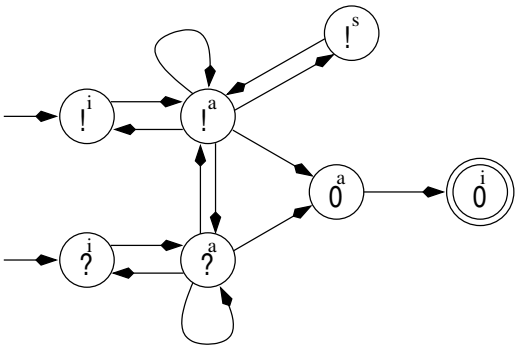



**FIG. 4.3:** *Evolution des observations sur les ports*

Par exemple, $u!^{\mathsf{a}}$ indique que le port $u$ est actif, et que sa prochaine action est un envoi. Il peut soit envoyer son message, soit être suspendu par un autre port. La figure 4.3 donne le diagramme d'état de l'activité d'un port. On remarquera les points suivants :

- lorsqu'un port est crée, il est dans un état oisif ;
- les seuls ports qui peuvent être suspendus sont les ports en émission. La combinaison $u?^{\mathsf{s}}$ est donc interdite : un port en réception est soit actif, soit oisif[1] ;
- $\mathbf{0}$ est l'état terminal pour un port (quelle que soit son activité $\sigma$. En fait, $u\mathbf{0}^{\mathsf{a}}$ doit rendre le contrôle de l'activité de la tâche, devenir $u\mathbf{0}^{\mathsf{i}}$ et disparaître) ;
- $u?^{\mathsf{a}}$ est un port actif en attente de message. Il peut soit recevoir le message en question et changer d'état, soit devenir oisif ;
- $u!^{\mathsf{a}}$ est un port actif en attente d'émission. Il peut soit envoyer un message, soit être suspendu par un autre port, soit devenir oisif.

Les états des ports seront notés $x, y$. Nous définissons aussi :

$x \rightarrowtail y$ qui indique que $x$ est suspendu par $y$. Cela veut dire que l'activité de $x$ est suspendue jusqu'à ce que $y$ termine ou devienne oisif.

Les tâches de $\mathsf{T}$ seront notées $t$. Une tâche $t$ est une séquence $x_1 \rightarrowtail x_2 \cdots \rightarrowtail x_n$. On notera $|t|$ la longueur de la tâche $t$, soit le nombre de ports qu'elle contient ($|t| = n$ dans l'exemple précédent). Cette séquence a les contraintes suivantes :

$x_i = u_i!^{\mathsf{s}}$ ssi $i < |t|$ (tous les ports, sauf le dernier, sont suspendus en émission)

Notre modèle autorise les dépendances entre les tâches. Il n'y a aucune restriction quant aux nombre de dépendances admises antre les tâches : une tâche peut être suspendue par plusieurs tâches, ou en suspendre plusieurs. Par exemple, $t_1 \rightarrowtail t_2$ indique que la tache $t_1$ est suspendue par $t_2$ : le port en tête de $t_1$ est suspendu par le port terminant la queue de $t_2$. Il est possible d'avoir en même temps $t_1 \rightarrowtail t_3$ :

$$\overbrace{u_1^{\mathsf{s}} \rightarrowtail \cdots \rightarrowtail u_n^{\mathsf{s}}}^{t_1} \nearrow \overbrace{w_1^{\mathsf{s}} \rightarrowtail \cdots \rightarrowtail w_n^{\mathsf{a}}}^{t_2}$$
$$\searrow \underbrace{v_1^{\mathsf{s}} \rightarrowtail \cdots \rightarrowtail v_m^{\mathsf{a}}}_{t_3}$$

[1] en fait, les règles de contrat du chapitre suivant imposent que le port actif en réception ne peut devenir oisif que s'il ne recevra plus de messages.

Les opérations sur $\mathsf{T}$ sont définies ci-après. Nous ajoutons à l'ensemble $\mathsf{T}$ un cache, ou *pool* de port oisifs. Dans les règles, les ports sont créés et supprimés à partir de ce cache. Chaque port et chaque tâche apparaît au plus une fois dans $\mathsf{T}$.

| | |
|---|---|
| $\mathsf{T} \mid u\rho^{\mathsf{i}}$ | ajoute un port dans le cache des ports oisifs. |
| $\mathsf{T}\backslash u$ | suppression du port $u$ de l'ensemble $\mathsf{T}$, et activation du port qui était éventuellement suspendu par $u$.<br>Cette opération n'est définie que si $u$ est la tête d'une tâche $t \in \mathsf{T}$.<br>Deux cas se présentent :<br>$\lvert t\rvert > 1$ : retire le port $u$ de $t$, et active le port suspendu par $u$.<br>$\lvert t\rvert = 1$ : la tâche $t$ est retirée de $\mathsf{T}$ en même temps que $u$. De plus, si $t$ est l'unique tâche qui suspendait une tâche $t'$, alors la tête de $t'$ devient active :<br>Si $(\exists t' = v_1^{\mathsf{s}} \rightarrowtail \cdots v_n^{\mathsf{s}} : t' \rightarrowtail t) \wedge (\nexists t'' \neq t : t' \rightarrowtail t'')$<br>alors $t' = v_1^{\mathsf{s}} \rightarrowtail \cdots v_n^{\mathsf{a}} \in \mathsf{T}\backslash u$. |
| $\mathsf{T}[u \rightarrowtail v]$ | suspend le port $u$ par le port $v$.<br>Cette opération n'est définie que si $u$ est la tête d'une tâche $t_1 \in \mathsf{T}$, et $v$ est dans une tâche singleton $t_2 = v\rho^{\mathsf{i}} \in \mathsf{T}$. Elle change l'état de $u$ à suspendu, ajoute $v\rho^{\mathsf{a}}$ comme nouvelle tête de $t_1$, et retire $t_2$ de $\mathsf{T}$. |
| $\mathsf{T}[t_1 \rightarrowtail t_2]$ | suspend la tâche $t_1$ par $t_2$ ; la tête de $t_1$ est alors suspendue. |
| $\mathsf{T}[t_1 \not\rightarrowtail t_2]$ | $t_1$ ne se suspend plus à $t_2$.<br>Cette opération n'est définie que si $t_1$ était suspendue à $t_2$. Elle inhibe la dépendance entre les deux tâches. Dans le cas où $t_1$ n'était suspendue que à $t_2$, le port en tête de $t_1$ devient actif. |
| $\mathsf{T}[u\rho'/u\rho]$ | modifie l'état d'un port $u$ : seul $\rho$ change en $\rho'$. |
| $\mathsf{T}[u\rho^{\mathsf{i}\rightarrow\mathsf{a}}]$ | active un port appartenant au cache des ports oisifs. Une nouvelle tâche est créé, et a son propre fil d'exécution (pas de dépendance). |
| $\mathsf{T}(u)$ | retourne $\rho^\sigma$ si $u\rho^\sigma \in \mathsf{T}$. |

## 4.3 Médium de Communication

Comme indiqué dans l'introduction, la communication entre les composants se fait par messages asynchrones. Ainsi, un message est d'abord déposé par l'émetteur dans un médium de communication, et, un peu plus tard, retiré de ce médium par le récepteur. La politique de la file d'attente que nous adoptons est FIFO (*First In, First Out*). Nous définissons $Com$, une abstraction du médium de communication, contenant une liste de files FIFO, une pour chaque référence de port contenue dans le composant : les messages sont écrits et lus à partir de $Com$. Nous adoptons les notations suivantes :

| | |
|---|---|
| $Com[\triangleleft u]$ | insère une file pour le port $u$. |
| $Com.u$ | la file pour le port $u$. C'est une suite ordonnée de messages «$v : M(\tilde{w})$» où $v$ est la référence du port émetteur, $M$ est le nom du message, et $\tilde{w}$ ses arguments. Si $u$ n'est pas encore défini, $Com.u = \bot$. |
| $Com\backslash u$ | la file du port $u$ est retirée de $Com$. |
| $Com[u\triangleright]$ | retire le message se trouvant en tête de la file du port $u$. |
| $Com[u \triangleleft v : M(\tilde{w})]$ | ajoute le message «$v : M(\tilde{w})$» en queue de la file du port $u$. |
| $Com.u_\triangleright$ | donne le prochain message (en tête de la file $u$) à être traité. |

## 4.4 Sémantique des composants

Cette section donne la sémantique des composants, avec les notations précédentes. Un composant est défini par son état et l'ensemble de ses ports, références et tâches :

$B(\mathsf{P},\mathsf{R},\mathsf{T})$ avec $B$ l'état du composant ;
$\mathsf{P}$, $\mathsf{R}$, $\mathsf{T}$ les ports, références et tâches comme définis plus haut.

Les règles du tableau 4.1 décrivent la sémantique des composants, montrant les transitions qu'un composant peut effectuer avec une abstraction de communication donnée. Une transition peut changer l'état du composant lui-même et/ou celui de l'abstraction de communication. Les actions du composant portent essentiellement sur la manipulation des ports (création/suppression), des références (attachement, suppression), des tâches (activation/suspension), et enfin l'envoi et la réception de messages. La signification précise des règles est la suivante :

**C-CREAT** permet la création d'un port du composant. La règle ajoute un port sans partenaire dont l'activité est oisive, une référence vers ce port (pour qu'elle puisse éventuellement être envoyée), et crée une file d'attente pour ce port.

**C-REMVPORT** supprime un port du composant. La suppression n'est possible que si le port est oisif, et la file d'attente est vide.

**C-REMVREF** retire une référence de l'ensemble $\mathsf{R}$. Cette référence ne doit être un port ($u \notin \mathsf{P}$), ni être attachée à un port ($u \notin CoDom(\mathsf{P})$).

**C-BIND** concerne l'attachement d'un port partenaire. Il n'est possible que si le port en question n'a pas de partenaire ($u \multimap \perp$). Seuls les ports en émission, et non suspendus peuvent attacher un port ($\mathsf{T}(u) = !^{\mathsf{a,i}}$) : attacher une référence à un port en réception n'a pas de sens (cette dernière sera écrasée lors de la réception – voir règle C-RECV) et un attachement alors que le port est suspendu peut être différé jusqu'à ce que le port soit actif. Si le partenaire en question est une référence pair, elle ne peut être attachée qu'à un seul port à la fois[2] : «peer($v$) $\Rightarrow v \notin CoDom(\mathsf{P})$» avec «peer($v$)» vrai si et seulement si $v$ est une référence pair. Le détachement d'un port partenaire n'est pas autorisé : cela permettrait à un port de changer de partenaire en cours de dialogue, et laisser l'ancien sur sa faim.

**C-ACTV** permet à un port $u$ de se suspendre à un port $v$. $u$ doit être actif en émission ($\mathsf{T}(u) = !^{\mathsf{a}}$), et $v$ oisif en émission ($\mathsf{T}(v) = !^{\mathsf{i}}$).

**C-ACTV2** active un port (provenant du cache des ports oisifs) sur sa propre tâche.

**C-DEACT** désactive un port : ce dernier doit être actif.

**C-THSUSP** est la suspension d'une tâche sur une autre. La condition (symbole $\diamond$ à droite de la règle) de C-THSUSP est telle que la tête de la tâche $t_1$ qui se suspend est en émission (le port peut être actif ou suspendu), qu'il n'y a pas d'interblocage entre les tâches d'un même composant (condition $t \neq t_1$), et que toute tâche $t$ sur laquelle $t_1$ se suspend indirectement (tâches $t$ telles que $t_2 \rightarrowtail^* t$), la tête de $t$ est en émission. C'est le pendant de la condition d'application de la règle C-ACTV ($\mathsf{T}(v) = !^{\mathsf{i}}$, le port $v$ est en émission). En résumé, la condition de C-THSUSP assure :

- qu'un port en réception ne peut devenir suspendu ;
- qu'un port ne peut se suspendre à un port en réception (plus précisément à une chaîne de suspension comprenant un port en réception) ;
- qu'il n'y a pas de création de cycle de dépendance entre les tâches internes.

[2] par contre, une référence vers un serveur peut être attachée à plusieurs ports.

$$\text{C-CREAT}\ \frac{\begin{array}{l}\mathsf{P}' = \mathsf{P}[u \multimap \bot] \qquad \mathsf{R}' = \mathsf{R} \cup u \qquad \mathsf{T}' = \mathsf{T} \mid u\rho^{\mathsf{i}} \\ Com' = Com[\triangleleft u]\end{array}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{creat}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\ u \notin \mathsf{P} \ \wedge\ Com.u = \bot$$

$$\text{C-REMVPORT}\ \frac{\begin{array}{l}\mathsf{P}' = \mathsf{P}\backslash u \qquad \mathsf{R}' = \mathsf{R}\backslash u \qquad \mathsf{T}' = \mathsf{T}\backslash u \\ Com' = Com\backslash u\end{array}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{remvport}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\ \mathsf{T}(u) = \rho^{\mathsf{i}} \ \wedge\ Com.u = \varnothing$$

$$\text{C-REMVREF}\ \frac{\mathsf{R}' = \mathsf{R}\backslash u}{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvref}(u)} B'(\mathsf{P},\mathsf{R}',\mathsf{T})}\ u \notin \mathsf{P} \cup CoDom(\mathsf{P})$$

$$\text{C-BIND}\ \frac{\mathsf{P}' = \mathsf{P}[u \multimap v]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{bind}(u \multimap v)} B'(\mathsf{P}',\mathsf{R},\mathsf{T}), Com}\ \square$$

$$\text{C-ACTV}\ \frac{\mathsf{T}' = \mathsf{T}[u \rightarrowtail v]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{actv}(u \rightarrowtail v)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \mathsf{T}(u) = !^{\mathsf{a}} \ \wedge\ \mathsf{T}(v) = !^{\mathsf{i}}$$

$$\text{C-ACTV2}\ \frac{\mathsf{T}' = \mathsf{T}[u\rho^{\mathsf{i}\to\mathsf{a}}]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{actv}(u)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \mathsf{T}(u) = \rho^{\mathsf{i}}$$

$$\text{C-DEACT}\ \frac{\mathsf{T}' = (\mathsf{T}\backslash u) \mid u\rho^{\mathsf{i}}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{deact}(u)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \mathsf{T}(u) = \rho^{\mathsf{a}}$$

$$\text{C-THSUSP}\ \frac{\mathsf{T}' = \mathsf{T}[t_1 \rightarrowtail t_2]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{susp}(t_1 \rightarrowtail t_2)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ \diamondsuit$$

$$\text{C-THRELAX}\ \frac{\mathsf{T}' = \mathsf{T}[t_1 \not\rightarrowtail t_2]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{\text{relax}(t_1 \not\rightarrowtail t_2)} B'(\mathsf{P},\mathsf{R},\mathsf{T}'), Com}\ t_1 \rightarrowtail t_2$$

$$\text{C-SEND}\ \frac{\mathsf{R}' = \mathsf{R} - \text{peer}(\tilde{v} \cup \{u\}) \qquad \mathsf{T}' = \mathsf{T}[u\rho/u!] \qquad Com' = Com[u' \triangleleft u : M(\tilde{v})]}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{u:u'!M(\tilde{v})} B'(\mathsf{P},\mathsf{R}',\mathsf{T}'), Com'}\ \triangle$$

$$\text{C-RECV}\ \frac{\begin{array}{ll}\mathsf{T}' = \mathsf{T}[u\rho/u?] & \mathsf{R}' = \mathsf{R} \cup \{\text{refs}(\tilde{v}), u''\} - \{u' | (u \multimap u') \wedge \text{peer}(u')\} \\ Com' = Com[u\triangleright] & \mathsf{P}' = \mathsf{P}[u \multimap u''] \text{ si peer}(u)\end{array}}{B(\mathsf{P},\mathsf{R},\mathsf{T}), Com \xrightarrow{u:u''?M(\tilde{v})} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'), Com'}\ \bigtriangledown$$

$\square \triangleq (u \multimap \bot) \ \wedge\ \mathsf{T}(u) = !^{\mathsf{a},\mathsf{i}} \ \wedge\ v \in \mathsf{R} \ \wedge\ \big(\text{peer}(v) \Rightarrow v \notin CoDom(\mathsf{P})\big)$

$\diamondsuit \triangleq t_1 \neq t_2 \ \wedge\ \text{head}(t_i) = !^{\mathsf{a},\mathsf{s}} \ \wedge\ \forall t, t_2 \rightarrowtail^* t : t \neq t_1 \ \wedge\ \text{head}(t) = !^{\mathsf{a},\mathsf{s}}$

$\triangle \triangleq \mathsf{T}(u) = !^{\mathsf{a}} \ \wedge\ (u \multimap u') \ \wedge\ u' \in Com \ \wedge\ \text{refs}(\tilde{v}) \subseteq \mathsf{R} \ \wedge\ \text{peer}(\tilde{v}) \cap CoDom(\mathsf{P}) = \varnothing$

$\bigtriangledown \triangleq \mathsf{T}(u) = ?^{\mathsf{a}} \ \wedge\ Com.u_{\triangleright} = u'' : M(\tilde{v})$

**Tab. 4.1:** *Règles pour la sémantique de composants*

**C-THRELAX** permet à $t_1$ de relâcher son attente sur $t_2$.
**C-SEND** régit les émissions du composant. Le port doit être actif en émission, attaché à une référence ($(u \multimap u')$ : le destinataire du message doit être connu). Les références pairs envoyées sont données par «peer($\tilde{v}$)» ; ces références ne doivent pas être attachées à un port du composant («peer($\tilde{v}$)$\cap CoDom(\mathsf{P}) = \varnothing$»), et d'autre part elles doivent être supprimées de l'ensemble $\mathsf{R}$ («$\mathsf{R}' = \mathsf{R} - \text{peer}(\tilde{v} \cup \{u\})$»). Cela assure le fait que ces références ne sont connues que de deux composants au maximum ; on remarquera que le port lui-même est aussi retiré de $\mathsf{R}$ pour les mêmes raisons. Une fois l'émission effectuée, le port passe à l'action suivante ($\mathsf{T}' = \mathsf{T}[u\rho/u!]$).
**C-RECV** régit la réception d'un message. L'émetteur est considéré comme le nouveau partenaire du port («$\mathsf{P}' = \mathsf{P}[u \multimap u'']$», sauf si le port $u$ est serveur, auquel cas l'attachement n'a pas lieu). L'ancien partenaire, s'il existait, est retiré de $\mathsf{R}$ (ce qui correspond à l'ensemble $\{u' | (u \multimap u')\}$ retiré de $\mathsf{R}$). Cela permet au partenaire de déléguer son comportement à un autre port. Les références reçues et le nouveau partenaire sont ajoutés à $\mathsf{R}$. Une fois la réception effectuée, le port passe à l'action suivante ($\mathsf{T}' = \mathsf{T}[u\rho/u?]$).

La différence entre les règles C-ACTV et C-THSUSP est la suivante : C-ACTV concerne la mise en séquence des instructions. Par exemple, dans la séquence «$v!\ ; u!\ ; \ldots$», le port $u$ est suspendu au port $v$. L'utilisation successive de la règle C-ACTV permet de réorganiser entièrement l'ordre des ports dans la tâche correspondante. La règle C-THSUSP est utilisée pour les synchronisations entre les fils d'exécutions ; la contrainte imposée ($\text{head}(t_i) = !^{\mathsf{a,s}}$) est plus difficile à maîtriser.

### 4.4.1 Configuration de Composants

Lorsque nous prenons en compte une configuration faite de plusieurs composants, nous considérons que le médium de communication *Com* est partagé entre tous les composants. Ainsi, les files d'attente sont partagées et les composants peuvent communiquer entre eux.

Le tableau 4.2 donne la règle de communication pour une configuration de deux composants. La communication n'est pas synchrone : les messages sont émis et retiré de *Com* selon les actions des composants, sans qu'il y ait synchronisation de ces actions. L'extension à une configuration à plus de deux composants est immédiate.

$$\text{CPAR}\ \frac{B_1(\mathsf{P}_1, \mathsf{R}_1, \mathsf{T}_1), Com \xrightarrow{\alpha} B_1'(\mathsf{P}_1', \mathsf{R}_1', \mathsf{T}_1'), Com'}{B_1(\mathsf{P}_1, \mathsf{R}_1, \mathsf{T}_1) \mid B_2(\mathsf{P}_2, \mathsf{R}_2, \mathsf{T}_2), Com \xrightarrow{\alpha} B_1'(\mathsf{P}_1', \mathsf{R}_1', \mathsf{T}_1') \mid B_2(\mathsf{P}_2, \mathsf{R}_2, \mathsf{T}_2), Com'}$$

**Tab. 4.2:** *Règles pour la configuration de composants*

## 4.5 Etude de cas

Nous présentons les diagrammes d'état des trois composants de notre étude de cas. Ces diagrammes sont inspirés des *Flowcharts*. On pourra les utiliser pour les vérifications de type présentés dans le chapitre suivant. (par exemple, Lamport et

Schneider donnent, dans [LS84], une extension de la représentation des programmes en selon la méthode de Floyd [Flo67]).

Sur les diagrammes, un rectangle représente l'état $B$ du composant. Ce rectangle contient les informations sur les ports, références et tâches comme définies dans ce chapitre. Les parties grisées indiquent les changements par rapport à l'état précédent. Les transitions portent le nom de l'action correspondante, il suffit de se reporter à la table 4.1 page 67 pour retrouver la règle correspondante. Enfin, chaque état 'rectangulaire' contient un diagramme d'état représentant le calcul interne du composant pour cet état : aucunes des transitions de ces diagrammes internes ne doivent pendre en compte une interaction avec l'extérieur (sous-entendu avec les ports ; une interaction avec l'utilisateur physique se fait bien entendu au travers de ces diagrammes).

Les noms des différents ports mis en jeux sont les suivant :

- $r$ pour le port du composant `Rapporteur` ;
- $g$ et $e$ pour le composant `Gestion` : $g$ interagit avec le rapporteur, $e$ enverra la référence du rapporteur ;
- $e'$ et $a$ pour le composant `Article` : $a$ interagit avec le rapporteur, $e'$ recevra la référence du rapporteur.

Les exemples ci-après montrent comment évoluent les ports : messages reçus et envoyés, création, suppression, (dé)activation de ports, attachement. L'envoi de référence est abordé. La suspension d'un port sur un autre est présentée à part.

*Le reste de cette page est volontairement vide.*

### 4.5.1 Diagramme d'état du composant Rapporteur

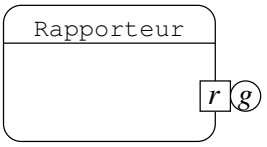



**Fig. 4.4:** *Composant* `Rapporteur` *à l'assemblage*

Le diagramme est présenté figure 4.5 page ci-contre. L'état initial devra être respecté lors de l'assemblage, c'est-à-dire que $r$ doit être attaché au port serveur $g$ à l'assemblage. Nous donnons deux exemples de diagrammes d'état imbriqués. Un pour l'état *Etat initial* qui demande au rapporteur les informations d'authentification ; pour l'étude de cas, nous supposons que le composant a déjà la liste des articles que le rapporteur devra critiquer. Un autre exemple est l'état *Envoi rapport* : selon le point d'entrée dans cet état, soit l'article est imprimé, soit les erreurs liées au rapport sont affichées ; dans les deux cas le rapport est ensuite demandé à l'utilisateur, puis envoyé.

Le diagramme d'état montre essentiellement comment évolue l'ensemble $\mathsf{T}$, et l'état du port $r$. La transition la plus intéressante est «$r$:**?**accordé(formulaire)» entre les états *Attente accès* et *Demande article*. Avant la transition, le port $r$ est attaché à $g$, et attend la réponse de l'authentification. Lorsqu'il reçoit le message *accordé*, le partenaire est modifié (soit $(r \multimap a)$ dans l'état *Demande article*), la référence $a$ est rajoutée à $\mathsf{R}$, et le port $r$ se met en émission. La modification de $\mathsf{R}$ est donnée dans la règle par «$\mathsf{R} \cup \{\text{refs}(\tilde{v}), u''\} - \{u' | (u \multimap u') \wedge \text{peer}(u')\}$», ce qui donne dans notre cas «$\mathsf{R} \cup \{a\} - \varnothing$» (le partenaire $g$ est une référence serveur, donc n'est pas concerné par la partie «$\{u' | (u \multimap u') \wedge \text{peer}(u')\}$»).

Une fois que le rapport est transmis sans erreurs (état *Rapport remis*, en bas du diagramme), le port $r$ devient oisif (état *Terminaison*), puis retiré du composant (état terminal *Fin*). Les références $g$ et $a$, quant à elles, ne sont pas effacées (c'est un choix arbitraire, mais rien ne l'empêche).

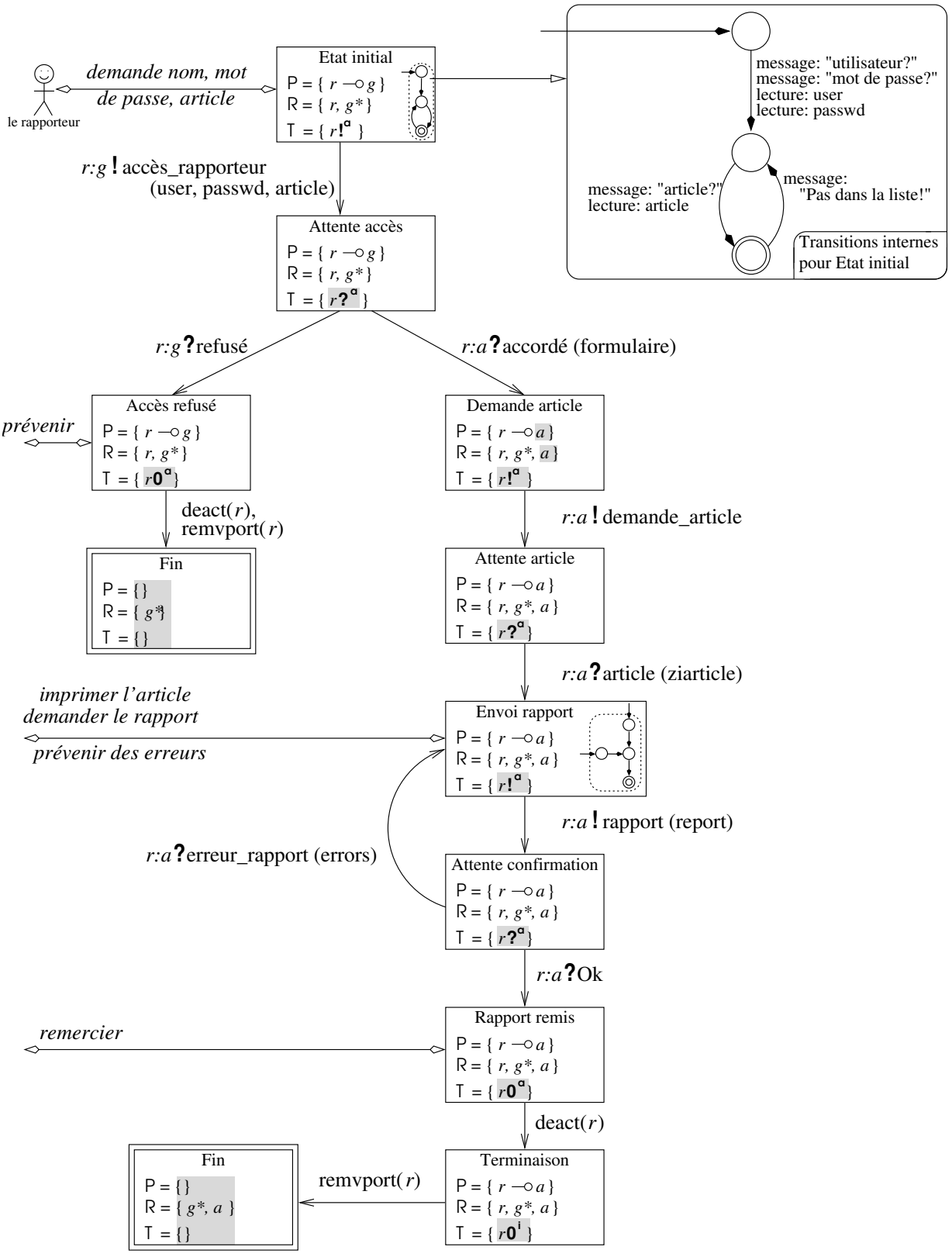



**FIG. 4.5:** *Diagramme d'état du composant* `Rapporteur`

### 4.5.2 Diagramme d'état du composant Gestion

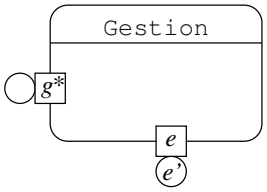



**Fig. 4.6:** *Composant `Gestion`, à l'assemblage*

Le diagramme d'état du composant `Gestion` est présenté figure 4.7 page suivante. Pour alléger le diagramme, les éléments peu importants des ensembles P, R et T sont représentés par des pointillés.

A l'assemblage, le port $g$ n'est pas attaché (et il restera sans partenaire, car c'est un port serveur), et le port $e$ est attaché à $e'$.

Sur réception du message *accès_rapporteur*, le composant doit créer un port qui aura sa propre tâche d'exécution (de par la sémantique du port serveur, présentée dans le chapitre précédent). La série d'états *Création fils* crée un tel port, $g'$, puis l'active avec la transition «actv($g'$)». Cette transition a deux effets :

- retour à l'état *Serveur accès* pour pouvoir traiter d'autres requêtes ;
- création d'une tâche d'exécution. Cette tâche s'occupe de vérifier l'accès au composant `Article`. Dans le cas où cet accès est refusé, le message *refusé* est envoyé, le port $g'$ et la référence $r$ du rapporteur supprimés. Dans le cas où cet accès est autorisé, le port $e$ est activé le temps d'envoyer la référence $r$ vers le composant `Article` (états *Envoi référence* et suivants). On notera deux points. Premièrement, la référence $r$ ne peut être envoyée si elle est attachée ($(g' \multimap r)$ dans notre cas) ; le port $g'$ doit être supprimé pour effectuer l'envoi. Deuxièmement, comme la référence $r$ est une référence pair, elle est retirée de R lorsqu'elle est envoyée.

La mise en œuvre du composant `Gestion` est critiquable : il ne sert à rien de créer le port $g'$ si le rapporteur peut accéder au composant `Article`. Cependant, le type de l'interface $g$ nous oblige à agir ainsi (voir chapitres 3 page 43 et 5 page 77).

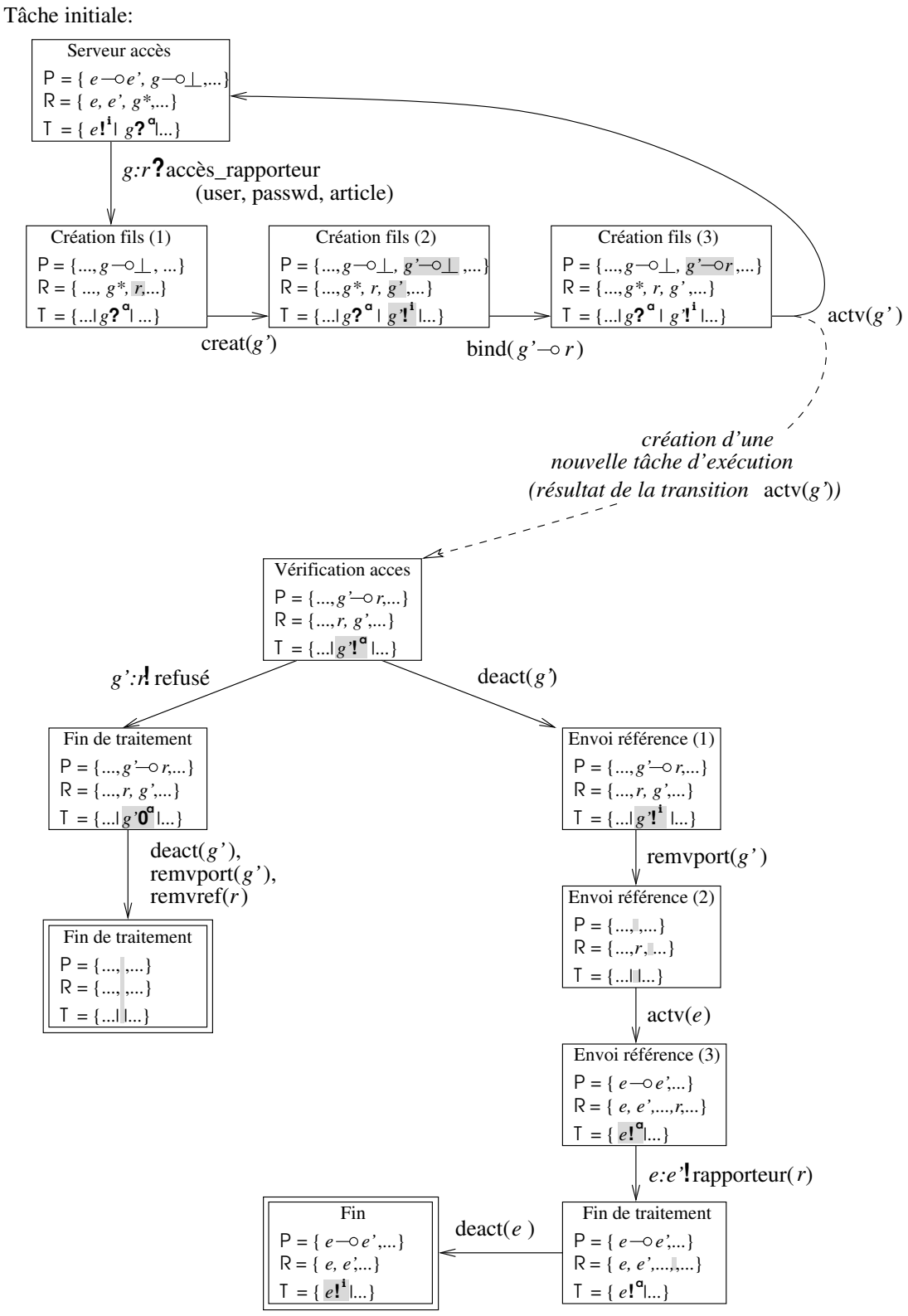



**FIG. 4.7:** *Diagramme d'état du composant* `Gestion`

### 4.5.3 Diagramme d'état du composant Article

A l'assemblage, le composant `Article` n'a qu'un seul port, $e'$, attaché à $e$, prêt à recevoir les références vers les rapporteurs. Le port $a$ sera créé en cours d'exécution, et n'apparaît donc pas à l'assemblage.

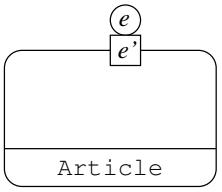



**Fig. 4.8:** *Composant `Article`, à l'assemblage*

Le diagramme d'état est donné figure 4.9 page ci-contre. La tâche initiale reste active en réception. A chaque réception du message *rapporteur*, une tâche est créée. Nous considérons cette création comme faisant partie du calcul interne du composant, aussi elle n'est pas présente dans notre sémantique de composant. Cette nouvelle tâche d'exécution créé le port $a$, l'attache à la référence $r$ du rapporteur, et active le port $a$ (états *Création port*). Ensuite viennent l'interaction avec le rapporteur. On remarquera que, tandis que le calcul du `Rapporteur` termine, celui de la tâche d'exécution ne se termine jamais. Cela est dû à la contrainte du **may ?** imposée par le type *gestion_rapporteur*.

### 4.5.4 Cas d'un port suspendu à un autre

Notre étude de cas ne présente pas de port qui se suspende à un autre. Nous donnons dans la figure 4.10 page 76 une extension de cette étude de cas, où la vérification du rapporteur se fait par un accès à une base de données : le port $c$ du composant `Gestion` envoie les requêtes d'interrogation au port serveur $s$.

Le port $g'$ se suspend alors au port $c$, qui était oisif en émission. $c$ envoie la requête (nous avons alors $g'!^{\mathsf{s}} \rightarrowtail c?^{\mathsf{a}}$). $c$ reçoit la réponse, puis se désactive pour que $g'$ puisse agir en conséquence.

## 4.6 Conclusion

Nous avons présenté dans ce chapitre notre sémantique abstraite de composant. Elle est basée sur une observation de l'état des ports : action (envoi **!**, réception **?**, inaction **0**) et activité (actif `a`, suspendu `s` ou oisif i). Les composants sont multi-tâches. Notre modèle de tâches se concentre uniquement sur l'état des ports. Une tâche est une chaîne dont la tête représente le port actif, et la queue une séquence ordonnée de ports suspendus.

Nous définissons la notion de partenaire et d'attachement de référence partenaire à un port : les messages envoyés le sont vers le partenaire.

Notre sémantique permet l'envoi de référence. Cela autorise les liaisons dynamiques : création mais aussi modification (changement de partenaire). Notre sémantique pour la modification d'un lien est assez originale puisqu'elle est à l'initiative du port émetteur, et que le port en réception ne modifie pas son comportement lors de ce changement. Deux types de ports existent : serveurs et pairs. Les ports serveurs ont la sémantique classique des serveurs (création d'un processus fils pour répondre au client) ; ils sont publics, c'est-à-dire connus de tous. Les ports pairs sont utilisés pour une interaction point-à-point, et sont privés, c'est-à-dire connus uniquement des deux intervenants de l'interaction.

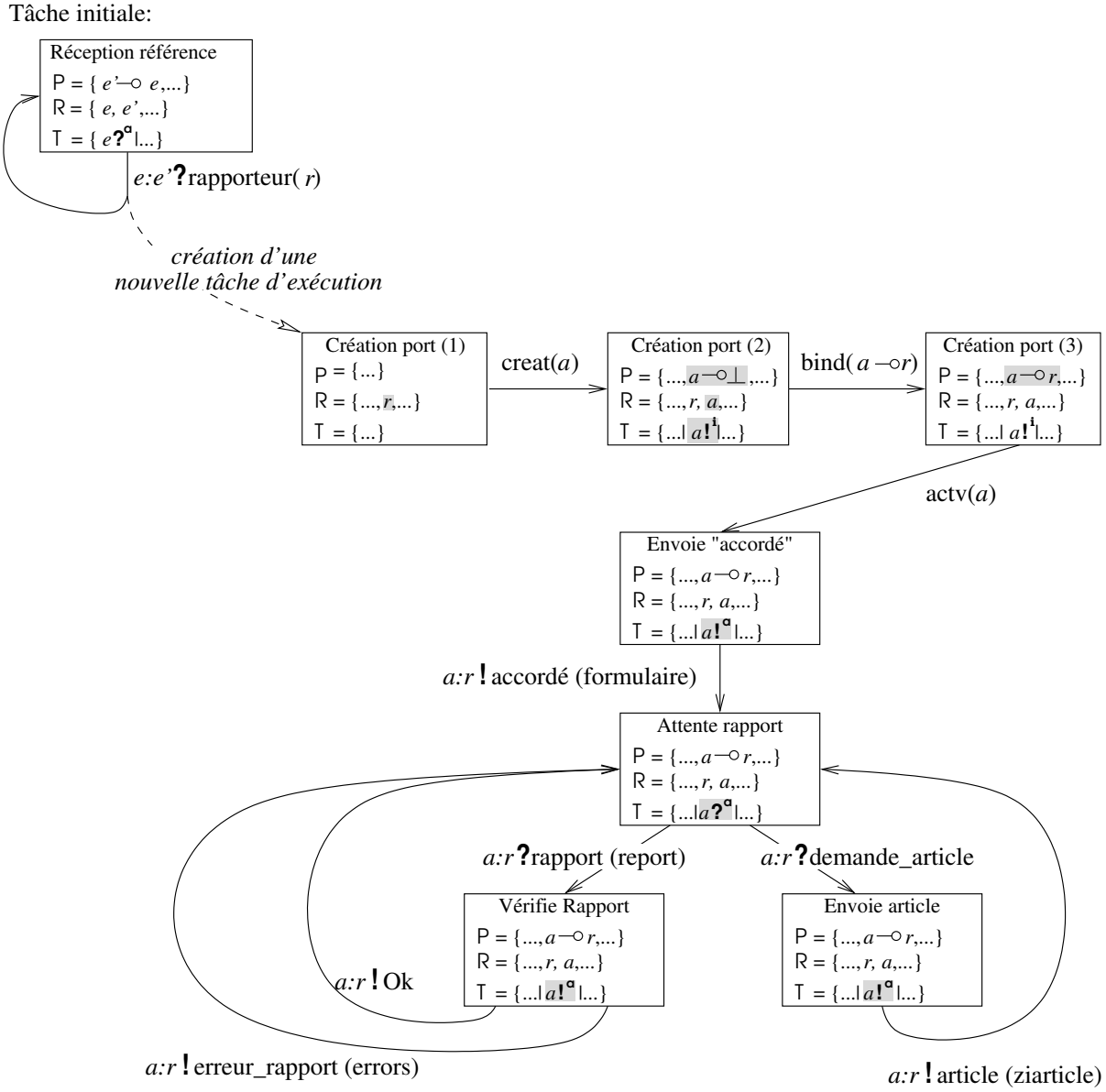



**FIG. 4.9:** *Diagramme d'état du composant* `Article`

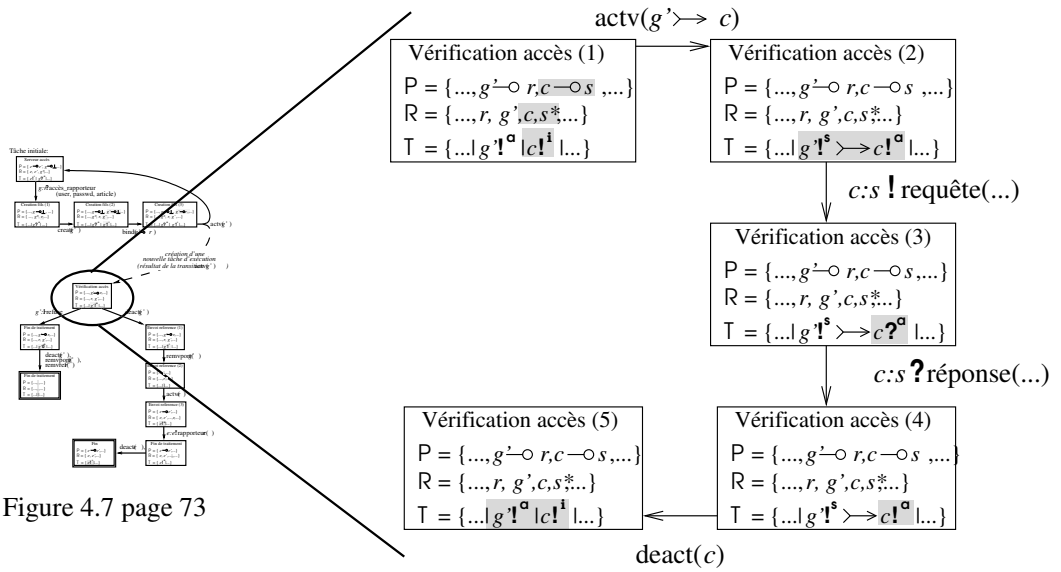


Figure 4.7 page 73

**FIG. 4.10:** *Exemple d'un port $g'$ qui se suspend à un port $c$*

Les contraintes de notre modèle de composant sont résumées ci-après. La raison principale de ces contraintes est bien entendue dûe aux propriétés que l'on veut démontrer (voir chapitre suivant).

- un port en réception ne peut faire d'attachement. En fait, cet attachement est inutile car le partenaire est modifié à chaque réception : l'émetteur du message est le nouveau partenaire ;
- un port suspendu en émission ($u!^{s}$) ne peut faire d'attachement. Cela ne présente pas de gêne particulière : l'attachement peut très bien se faire lorsque le port redeviendra actif ;
- un port ne peut se suspendre sur un port *en réception* (soit $v?^{a,s,i}$) Par contre, port peut se suspendre sur un autre port actif en émission qui peut être en réception par la suite : par exemple, si $u!^{s}$ est suspendu à $v!^{a}$, le port $v$ peut effectuer son action, et se mettre en réception. Nous aurons alors $u!^{s}$ suspendu à $v?^{a}$.

  Cette contrainte est la plus gênante. Cependant, lorsqu'un port se suspend à un autre, c'est pour demander un service : donc ce port doit envoyer un premier message avant de recevoir le résultat de la demande ;
- de détachement de partenaire n'est pas possible (pas de règle C-UNBIND). La raison est la suivante. Supposons que le port $u$ dialogue avec le port $v$. Supposons que $u$ détache $v$, pour attacher un troisième port $v'$ et lui envoyer[3] le message destiné à $v$. Nous avons alors les ports $v$ et $v'$ dans la configuration de composant qui ont interagit avec $u$ à un moment donné, et $v$ attend les envois de $u$ *parce que* $u$ *n'a pas terminé le dialogue avec lui, mais avec un autre port* $v'$.

[3] le detachement lorsque $u$ est en réception ne présente aucun intérêt dans notre sémantique.

Chapitre 5

# Respect de contrat

Le langage d'interface présenté dans le chapitre 3 page 43 impose des contraintes sur l'interface distante, par exemple en obligeant l'envoi de messages. Ces contraintes impliquent aussi des contraintes sur les composants ; elles constituent le contrat que le composant doit respecter. Dans ce chapitre, nous présentons la notion d'un composant respectant un contrat décrit dans notre langage. Cette notion est basée sur le principe d'observateurs : le contrat observe les actions du composant ; si l'action respecte le contrat, contrat et composant évoluent ensemble.

Ce chapitre est découpé en quatre sections : règles de respect de contrat, assemblage de composant, propriétés et étude de cas (nous reprendrons les diagrammes d'états de la section 4.5 page 68). Les propriétés sont :

- garantie qu'un message envoyé est reçu et compris, sous réserve d'équité entre les ports ;
- absence d'interblocage externe.

Moyennant deux contraintes supplémentaires exprimées en logique temporelle, nous pouvons garantir qu'un port typé **must?** recevra son message.

## 5.1 Contrats et observateurs

Les définitions du chapitre 4 sont étendues à la notion de contrat. Un composant a un contrat, représentant le type de chacun des ports et références du composant. Bien qu'il puisse apparaître une certaine ambiguïté entre la notation classique $T$ pour un type et notre ensemble $\mathsf{T}$ de tâches, ces ambiguïtés sont levées par le contexte dans lequel les notations sont utilisées. Nous adoptons, pour le reste du chapitre :

$u{:}T$ $u$ a le comportement $T$, qui est un *type* (selon le chapitre 3).

$u <: T$ $u$ a un comportement qui est un sous-type de $T$.

$(B, \tilde{C})$ à $B$ est associé le contrat $\tilde{C}$, un ensemble de $(u : T)$, tel que à chaque référence (port local ou partenaire) est associé un type. L'ajout d'un type est noté $\tilde{C} \Leftarrow u : T$ (si $u : T'$ est déjà dans $\tilde{C}$, l'ajout est sans effet). La suppression est notée $\tilde{C} \backslash u$. La modification est notée $\tilde{C}[u : T'/T]$ lorsque le type de $u$ change de $T$ en $T'$, et $\tilde{C}[u' : T'/u : T]$ lorsque la référence $u{:}T$ est remplacée par une autre référence $u'{:}T'$. Nous écrirons $u{:}T$ pour $u{:}T \in \tilde{C}$ lorsqu'il n'y a pas d'ambiguïté (notamment dans les règles).

La modification du type des références serveurs «*» est légèrement différente : le contrat fait évoluer le type de la référence, mais garde en même temps le type initial de la référence serveur. Par exemple, dans l'étude de cas, un client $(B, \tilde{C})$ utilise une référence $g^* : T$ en tant que serveur ; une fois que le serveur reçoit le premier message, $\tilde{C}$ contient $g^* : T$ et $g : T'$, avec $T'$ le type du port serveur après la première réception. Les règles ne mentionnent pas cette subtilité pour ne pas les surcharger.

### 5.1.1 Règles de respect de contrat : transitions valides

Les règles, exposées dans le tableau 5.1, sont basées sur celles du chapitre 4, et où *Com* est abstrait à partir de la structure d'état. Le prédicat Must(T) postule que tout port $u$ qui est typé **must!** n'est pas suspendu par un port $v$ qui est typé **may?**. Ce prédicat est à vérifier à chaque fois qu'un port peut modifier son action en réception, soit après un envoi ou une réception. Il n'est pas utile de vérifier le prédicat lorsqu'un port se suspend à un autre, car les règles de la sémantique du composant interdisent la suspension sur un port en réception.

Les différents contrats ne sont connus que localement : il n'existe pas d'ensemble global les regroupant. Lorsque nous parlons de type d'une référence $u$, il s'agit du type présent dans le contrat associé au composant ayant cette référence.

**CREAT** permet de créer un port. Cela ajoute le type du port à l'ensemble $\tilde{C}$.

**REMVPORT** concerne la suppression d'un port : son type est retiré de $\tilde{C}$. Aucune condition ne porte sur la règle, mais un port actif en réception qui peut recevoir un message ne peut être supprimé. En effet, la règle DEACT interdit qu'un port actif en réception devienne oisif, sauf dans un cas exceptionnel, où le port correspondant ne recevra plus de messages (la suppression est donc sans danger).

**REMVREF** supprime une référence de R uniquement. Cela n'est permis que si la référence n'est pas typée **must?** : le composant devra soit interagir avec cette référence, soit l'envoyer vers un autre port.

**BIND** est très importante, puisqu'elle concerne l'attachement d'une référence à un port. Le port et le partenaire doivent avoir des types compatibles. C'est ce qui nous permet d'avoir des liens dynamiques valides.

**DEACT** autorise la déactivation d'un port dans les cas suivants :

- $u$ n'est pas en réception : «$u : T \not\equiv mod$ **?** $[*]M_\Sigma$» ;
- le partenaire de $u$ est sous-type de **0** (dans ce cas, $u$ a le type **0** ou **may?**).

Concernant le deuxième point, le port $v$ n'enverra pas de messages : le corollaire 2 page 85 montre que le véritable type du port $v$ est sous-type du type de la *référence* de $v$ dans la configuration de composants, donc le *port* $v$ est sous-type de **0**, et n'est pas en envoi. La démonstration du corollaire se fait par induction sur les règles ; le raisonnement que l'on vient de donner peut donc paraître circulaire, mais le corollaire et cette mini démonstration peuvent se démontrer sans la règle DEACT.

**SEND** vérifie les envois de message. Lorsque $u$ envoie le message $M_k$, alors son type et celui de son partenaire $u'$ évoluent en conséquence (respectivement $T_k$ et $T'_k$ ; de par la relation de compatibilité, le type du partenaire est bien une réception, donc le type $T'_k$ existe bien) dans le contrat $\tilde{C}$ associé au composant. De plus, les références envoyées doivent être du bon type. Tout comme les références pairs qui sont envoyées sont retirées de R, nous devons les retirer du contrat $\tilde{C}$ associé au composant[1].

[1] peer($\tilde{v}_k \cap \overline{\mathsf{P}}$) représente les références pairs qui ne sont pas des ports locaux.

Par souci de compréhension, nous notons :
$M_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}_k)\,;\,T_k], M'_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}'_k)\,;\,T'_k] \quad m_k \triangleq M_k(\tilde{v}_k)$

$$\text{CREAT}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{creat}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{creat}(u)} (B'(\mathsf{P},\mathsf{R}',\mathsf{T}'),\tilde{C} \Leftarrow u{:}T)}$$

$$\text{REMVPORT}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvport}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{remvport}(u)} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}\backslash u)}$$

$$\text{REMVREF}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvref}(u)} B'(\mathsf{P},\mathsf{R}',\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{remvref}(u)} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}\backslash u)}\ (T \not\equiv \textsf{must?}\ M_\Sigma)$$

$$\text{BIND}\ \frac{u{:}T \qquad v{:}S \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{bind}(u \multimap v)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{bind}(u \multimap v)} (B'(\mathsf{P}',\mathsf{R},\mathsf{T}),\tilde{C})}\ Comp(T,S)$$

$$\text{DEACT}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{deact}(u)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{deact}(u)} (B'(\mathsf{P},\mathsf{R},\mathsf{T}'),\tilde{C})}\ \begin{pmatrix} u{:}T \not\equiv mod\ \textsf{?}\ [\textsf{*}]M_\Sigma \\ \vee\big((u \multimap v)\ \wedge\ v <: \mathbf{0}\big) \end{pmatrix}$$

$$\text{SEND}\ \frac{u{:}T \equiv mod\ \textsf{!}\ M_\Sigma \qquad u'{:}T' \equiv mod'\ \textsf{?}\ [\textsf{*}]M'_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'!m_k} B'(\mathsf{P},\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u:u'!m_k} (B'(\mathsf{P},\mathsf{R}',\mathsf{T}'),\tilde{C}[u{:}T_k/T, u'{:}T'_k/T']\backslash\{\text{peer}(\tilde{v}_k \cap \overline{\mathsf{P}})\}}\ \blacktriangleleft$$

$$\text{RECV}\ \frac{\begin{array}{c} u{:}T \equiv mod\ \textsf{?}\ M_\Sigma \\ u'{:}T' \equiv mod'\ \textsf{!}\ M'_\Sigma \end{array} \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u''?m_k} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u:u''?m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}[u{:}T_k/T, u''{:}T'_k/u'{:}T'] \Leftarrow \tilde{v}{:}\tilde{U}'_k)}\ \blacktriangle \wedge (u \multimap u') \in \mathsf{P}$$

$$\text{RECV-UN}\ \frac{u{:}T \equiv mod\ \textsf{?}\ M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'?m_k} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u:u'?m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}[u{:}T_k/T] \Leftarrow u'{:}{T_k}^{\mathcal{D}}, \tilde{v}{:}\tilde{U}_k)}\ \blacktriangle \wedge (u \multimap \bot) \in \mathsf{P}$$

$$\text{RECV*}\ \frac{u{:}T \equiv mod\ \textsf{?*}M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:w?m_k,\text{creat}(u'),\text{bind}(u' \multimap w),\text{actv}(u')} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u'/u^*:w?m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C} \Leftarrow u'{:}T_k, w{:}{T_k}^{\mathcal{D}}, \tilde{v}{:}\tilde{U}_k)}\ \blacktriangle$$

$$\text{OTHER}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\alpha} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\alpha} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C})}\ \alpha \in \{\text{actv}, \text{susp}, \text{relax}\}$$

$$\blacktriangleleft \triangleq \tilde{v}_k <: \tilde{U}_k \wedge \text{Must}(\mathsf{T}')$$
$$\blacktriangle \triangleq len(\tilde{v}) = len(\tilde{U}_k) \wedge \text{Must}(\mathsf{T}')$$
$$\text{Must}(\mathsf{T}') \triangleq \forall u \in \mathsf{T}', (u{:}\textsf{must!}M_\Sigma) \Rightarrow \forall v, u \rightarrowtail^* v : \neg(v{:}\textsf{may?}M_\Sigma)$$

**Tab. 5.1:** *Règles de respect de contrat (transitions valides)*

**RECV** vérifie les réceptions de message, lorsque le partenaire est connu. Notamment le type du partenaire est présent dans $\tilde{C}$ ($u' : T' \in \tilde{C}$), et, comme le message n'a pas été consommé, et que le partenaire doit respecter sont contrat (envoi autorisé), ce type est en émission ; donc le type $T'_k$ existe bien. Une fois la réception effectuée, le type du port et de son partenaire évoluent chacun en conséquence. Concernant le partenaire, la modification dans $\tilde{C}$ associe le contrat de l'ancien partenaire ($u'$, dont le type $T'$ devient $T'_k$ après la réception) au nouveau partenaire ($u''$ qui se retrouve typé $T'_k$). Le corollaire 2 page 85 assure que le véritable type de $u''$ est sous-type de $T'_k$.
**RECV-UN** concerne les réceptions pour lesquelles le partenaire est inconnu. Le type de ce partenaire est donc lui aussi inconnu. Nous choisissons de lui assigner le plus grand sous-type possible, qui est le type dual du port ayant reçu le message (ce plus grand super-type est assuré par la propriété 3 page 53). Les références sont enregistrées avec le type $\tilde{U}_k$, qui est obligatoirement un super-type du type réel des références.
**RECV*** vérifie le bon fonctionnement d'un port serveur : le composant doit créer immédiatement un nouveau port qui dialoguera avec le partenaire (ce qui correspond à la séquence d'actions «$u : w?m_k, \mathrm{creat}(u'), \mathrm{bind}(u' \multimap w), \mathrm{actv}(u')$»). Tout comme la réception avec un partenaire qui était inconnu, le type de ce dernier est le dual du port ayant reçu le message.
**OTHER** donne toutes les autres transitions valides, et dans lesquelles l'ensemble des contrats $\tilde{C}$ n'est pas modifié.

Concernant les réceptions de message nous ne vérifions pas le type des arguments, car le message a été envoyé selon le type de l'émetteur ; comme l'émetteur doit être compatible avec le récepteur, nous sommes sûrs que les arguments sont bien typés. Nous ne vérifions pas non plus si les variables qui enregistrent les arguments sont typées en conséquence : nous indiquons simplement que, après réception du message, ces variables sont typées selon les types des arguments donnés par l'interface. La méthode est adaptable pour vérifier que ces variables sont correctement typées.

Nous optons pour une vérification optimiste, où l'environnement doit respecter la sémantique des interfaces : un message reçu non compris dans le type n'est pas pris en compte.

Le cas des interactions entre les ports d'un même composant est lui aussi délicat. Soient $u$ et $v$ de tels ports. Le composant peut être programmé de telle façon que $u$ envoie un message à $v$ tout en sachant que le port $v$ est dans le même composant. Les conséquences d'une telle programmation obligent l'environnement à ne pas interagir avec le port $v$[2]. Tout comme l'environnement doit respecter l'interface qu'il simule, il doit en outre respecter le fait qu'il ne connaît pas la référence $v$ dans ce cas précis.

### 5.1.2 Règles de respect de contrat : cas d'erreurs

Les règles du tableau 5.2 page ci-contre donnent les cas où le composant ne respecte pas son contrat.
**REMVREF-ERR** lève une erreur lorsqu'une référence typée **must?** est retirée uniquement de R.

[2] si l'environnement envoie le message à $v$, alors $u$ n'est plus compatible et enverra un message qui ne sera pas compris.

**BIND-ERR** stipule que deux références ne peuvent être attachées si leurs types ne sont pas compatibles.
**DEACT-ERR** interdit qu'un port en réception devienne oisif ($u : mod\ \mathsf{?}\ldots$) s'il n'a pas de partenaire, ou si celui-ci peut éventuellement envoyer des messages ($v$ n'est pas sous-type de $\mathbf{0}$). La raison de cette interdiction est qu'elle nous assure le corollaire 1 page 84.
**SEND-ERR** concerne les erreurs lors d'envoi de messages, c'est-à-dire un envoi non autorisé. Deux cas se présentent : un message qui n'est pas dans la liste des envois autorisés, et lorsque le type demande une réception ou spécifie une interface inactive.
**RECV-ERR** concerne les réceptions de messages non effectuées.
**RECV*-ERR** correspond au cas où le composant, pour un port ayant un rôle serveur, ne crée pas immédiatement de port pour dialoguer avec le client.
**MUST-ERR** est utilisée lorsque le prédicat $\mathrm{Must}(\mathsf{T}')$ est faux (soit : un port typé **must!** est bloqué par un port typé **may?** ; la conséquence est que le premier port risque de ne pas honorer son contrat).

---

Par souci de compréhension, nous notons :
$M_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}_k)\ ;\ T_k], M'_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}'_k)\ ;\ T'_k] \quad m_k \triangleq M_k(\tilde{v}_k)$

$$\text{REMVREF-ERR}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{remvref}(u)} B'(\mathsf{P},\mathsf{R}',\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \to \mathit{Error}}\ T \equiv \mathbf{must?}M_\Sigma$$

$$\text{BIND-ERR}\ \frac{u{:}T \qquad v{:}S \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{bind}(u \multimap v)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \to \mathit{Error}}\ \neg \mathit{Comp}(T,S)$$

$$\text{DEACT-ERR}\ \frac{u{:}T \equiv mod\ \mathsf{?}\ [\mathbf{*}]M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{deact}(u)} B'(\mathsf{P}',\mathsf{R},\mathsf{T})}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \to \mathit{Error}}\ \begin{pmatrix} (u \multimap \bot) \\ \vee\big((u \multimap v) \wedge v \not<: \mathbf{0}\big) \end{pmatrix}$$

$$\text{SEND-ERR}\ \frac{u{:}T \equiv mod\ \rho\ [\mathbf{*}]M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'!m'} B'(\mathsf{P},\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \to \mathit{Error}}\ \neg m' : M_\Sigma \vee \rho = \mathsf{?}$$

$$\text{RECV-ERR}\ \frac{u{:}T \equiv mod\ \mathsf{?}[\mathbf{*}]M_\Sigma \qquad \forall k, B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'?m_k} \!\!\!\!\!\!/\ \ B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \to \mathit{Error}}$$

$$\text{RECV*-ERR}\ \frac{u{:}T \equiv mod\ \mathsf{?}\mathbf{*}M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:w?m_k,\mathrm{creat}(u'),\mathrm{bind}(u' \multimap w),\mathrm{actv}(u')} \!\!\!\!\!\!/\ \ B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \to \mathit{Error}}$$

$$\text{MUST-ERR}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \to B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \to \mathit{Error}}\ \neg\mathrm{Must}(\mathsf{T}')$$

$$\neg m' : M_\Sigma \triangleq m' = M'(\tilde{v}') \wedge \forall k, M' \neq M_k \vee \neg \tilde{v}_k : \tilde{U}'_k$$

---

**Tab. 5.2:** *Règles de respect de contrat (cas d'erreurs)*

### 5.1.3 Composant honorant un contrat

Un composant respectant un contrat, noté $B(\mathsf{P},\mathsf{R},\mathsf{T})\models\!\!\succ \tilde{C}$, est tel que les réductions successives ne mèneront jamais à *Error* :

$$B(\mathsf{P},\mathsf{R},\mathsf{T})\models\!\!\succ \tilde{C} \quad \text{ssi} \quad \forall B',\tilde{C}' \text{ tel que } (B,\tilde{C}) \rightarrow^* (B',\tilde{C}') : \quad (B',\tilde{C}') \nrightarrow \mathit{Error}$$

## 5.2 Assemblage de composants

Pour faire abstraction de la structure du composant et de son contrat, nous noterons, quand cela simplifie les relations :

| | |
|---|---|
| $B_u(\mathsf{P}_u,\mathsf{R}_u,\mathsf{T}_u),\tilde{C}_u$ | le composant (et son contrat $\tilde{C}_u$) contenant le port $u$, soit $u \in \mathsf{P}_u$. Bien entendu, il est possible d'avoir, pour deux références $u$ et $v$ distinctes : $B_u = B_v, \mathsf{P}_u = \mathsf{P}_v, \ldots$ |
| $u{:}T \in \mathsf{R}$ | Dans le composant tel que l'on ait $B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}$, nous avons $u \in \mathsf{R}$ et $u : T \in \tilde{C}$. Nous noterons aussi $u <: T \in \mathsf{P}$ pour indiquer que $u \in \mathsf{P}$ et a un type qui est sous-type de $T$. |
| $(u{:}T \multimap v{:}T')$ | Dans le composant $B_u$, $u$ de type $T$ est attaché à $v$ de type $T'$ *dans le contexte du contrat* $\tilde{C}_u$ : $(u \multimap v) \in \mathsf{P}_u$, $v \in \mathsf{R}_u$ et $u{:}T, v{:}T' \in \tilde{C}_u$. Il va de soit que $T'$ n'est pas obligatoirement le véritable type du port $v$ (il est possible d'avoir $v{:}T'' \in \tilde{C}_v$ avec $T'' \neq T'$). |

Nous définissons un assemblage de composants comme une configuration de composants ayant chacun leur contrat, et prêt à interagir via un médium de communication. Cet assemblage a quatre propriétés :

- la configuration est fermée du point de vue des références : chaque référence partenaire désigne un port d'un composant de la configuration ;
- tous les ports sont actifs, sur des tâches indépendantes ;
- les ports pairs en réception ont un unique partenaire dans toute la configuration ;
- un composant n'a connaissance d'aucune référence d'un port pair, excepté celles de ses propres ports et de leurs partenaires.

Les deux dernières conditions assurent qu'une référence pair est bien privée. Nous autorisons les deux types d'attachement : un serveur avec un port pair (liaison client/serveur), et un port pair avec un autre port pair (liaison point-à-point). La formalisation est la suivante :

$$\mathcal{A} = \{(B_1(\mathsf{P}_1,\mathsf{R}_1,\mathsf{T}_1),\tilde{C}_1),\ldots,(B_n(\mathsf{P}_n,\mathsf{R}_n,\mathsf{T}_n),\tilde{C}_n),\mathit{Com}\}$$

$$\text{avec les 4 propriétés} \begin{cases} \forall i,u : & u \in \mathsf{R}_i \Rightarrow \exists j \text{ tel que } u \in \mathsf{P}_j \\ \forall u \in \cup\mathsf{P}_i : & \mathsf{T}(u) = \rho^{\mathsf{a}} \\ \forall u \in \text{peer}(\cup\mathsf{P}_i), u?^{\mathsf{a}} : & \exists! v,j \text{ tel que } (v \multimap u) \in \mathsf{P}_j \\ \forall i : & u \in \mathsf{R}_i \Rightarrow u \in \mathsf{P}_i \ \vee \ \exists v : (v \multimap u) \in \mathsf{P}_i \end{cases}$$

### 5.2.1 Assemblage sain

Un assemblage sain (ou correct) est un assemblage où
- chaque composant satisfait les contrats de ses interfaces ;
- les ports reliés entre eux ont des interfaces compatibles entre elles (soit : un port est compatible avec son partenaire, et le type de ce partenaire est super-type du port le représentant).

**Définition 1 (Assemblage sain)**
*$\mathcal{A}$ est sain (noté $\models \mathcal{A}$) ssi*

$$\forall i : \left(B_i \models \tilde{C}_i\right) \wedge \left((u{:}T_u \multimap v{:}T_v) \in P_i \Rightarrow (Comp(T_u, T_v) \wedge \exists!\, j, T' \text{ tel que } v{<:}T' \in P_j)\right)$$

---

Jusqu'à présent, nous avons vu dans ce rapport les points suivant :
- un composant respecte le contrat (chapitre en cours) défini par le langage d'interface (chapitre 3 page 43) ;
- à l'assemblage, les ports interconnectés entre eux sont compatibles (chapitre 3 page 43).

Le reste de ce chapitre donne les propriétés résultant de cette double vérification.

## 5.3 Propriétés

Dans cette section, nous étudions les propriétés d'un assemblage de composants, et prouvons des propriétés de sûreté (aucune erreur n'a lieu, et pas d'interblocage entre les ports n'est possible), et des propriétés de vivacité (tous les messages seront finalement consommés).

### 5.3.1 Préservation de la compatibilité et propriété de consommation des messages

La relation de compatibilité est très importante dans notre étude. La propriété suivante assure que les ports attachés le sont toujours selon des types compatibles.

**Propriété 4 (Préservation de la compatibilité)**
*Si $\mathcal{A}$ est sain, alors $\forall \mathcal{C}, \mathcal{A} \rightarrow^* \mathcal{C}$, on a :*
*$\forall u, v \in \mathcal{C}$ tels que $(u{:}T_u \multimap v{:}T_v)$ Alors $Comp(T_u, T_v)$.*

**Preuve.** Par induction structurelle, et de par la définition de la règle de compatibilité (les interfaces évoluent vers des types compatibles) □

La propriété suivante, $P_{\text{sr}}$, d'un assemblage sain stipule simplement que l'assemblage restera sain tout au long de son évolution, soit "une configuration de composants ne mène jamais à *Error*" :

$$P_{\text{sr}} \triangleq \forall \mathcal{C} : \mathcal{A} \rightarrow^* \mathcal{C}, \mathcal{C} \not\rightarrow \mathit{Error}.$$

**Théorème 1 (Sûreté à l'exécution)**
*Si $\mathcal{A}$ est sain, alors $\mathcal{A} \vDash P_{\text{sr}}$.*

**Preuve.** Chaque composant de $\mathcal{A}$ respecte son contrat : $B_i \models\!\!\sim \tilde{C}_i$. La définition donnée dans la section 5.1.3 page 82 nous permet d'affirmer que les composants n'ont pas de transitions menant à *Error*, donc la configuration de composants ne ménera jamais à *Error*. □

Nous définissons aussi $P_{\text{mc}}$, qui stipule que "tous les messages envoyés sont finalement consommés".

$$P_{\text{mc}} \triangleq \forall u, v, i, M : (u \multimap v) \in \mathsf{P}_i,$$
$$(\mathcal{C} \xrightarrow{u:v!M} \mathcal{C}') \Rightarrow \exists \mathcal{C}'', \mathcal{C}''' \quad \text{tel que} \quad \mathcal{C}' \rightarrow^* \mathcal{C}'' \xrightarrow{v:u?M} \mathcal{C}''' \quad (5.1)$$

**Corollaire 1 (Consommation des messages)**
*Si $\mathcal{A}$ est sain, alors $\mathcal{A} \vDash P_{\text{mc}}$.*

**Preuve.** Ce corollaire est une conséquence du théorème 1, de l'utilisation des files d'attente FIFO et de la contrainte qu'un port en réception est actif. □

Le corollaire assure que le message sera consommé dans le futur, mais il est fort possible que la configuration de composants effectue infiniment des transitions, sans que la transition qui consomme le message soit exécutée. Cela est dû au fait que les règles de consommation de message rentrent en compétition les unes avec les autres : il se peut que le port récepteur ne voit jamais sa transition exécutée. Le problème que nous soulevons ici est un problème d'équité (*fairness* [LPS81]). Kesten et al. proposent dans [KPRS01] une méthode de vérification de modèles (*model-checking*) pour régler ce genre de situation, en formalisant les propriétés de vivacité en LTL (Logique Temporelle Linéaire).

### 5.3.2 Type d'une référence dans la configuration

Le lemme suivant donne les relations entre le type d'une référence dans la configuration et le type du port le représentant. Les files d'attente doivent être vides, sans quoi il n'est pas possible de comparer les deux types (l'un sera "en avance" sur l'autre). Le lemme ne s'applique que dans le cas où la référence $u$ et le port la représentant sont dans des composants différents ($u \notin \mathsf{R}_u$). Supposons que le composant qui détient la référence $u$ lui ait associé le type $T_{\text{part}}$[3]. Le lemme montre que :

- si le port $u$ est attaché à un partenaire, alors le type $T_{\text{part}}$ de la référence $u$ est super-type du dual du type du partenaire de $u$ ;
- si $u$ n'est pas attaché, alors le type $T_{\text{part}}$ de la référence $u$ est super-type du type de $u$.

Nous utiliserons cependant plutôt le corollaire qui découle de ce lemme, à savoir que le type $T_{\text{part}}$ de la référence $u$ est super-type du type du port $u$.

**Lemme 2**
*Soit $u$ tel que $u{:}T_{part} \in \mathsf{R}$, $u \notin \mathsf{R}_u$ et $Com.u = \varnothing$. Nous avons alors :*

$$(u{:}T \multimap v{:}T') \in \mathsf{P}_u \Rightarrow {T'}^{\mathcal{D}} \preceq T_{part}$$
$$(u{:}T \multimap \bot) \in \mathsf{P}_u \Rightarrow T \preceq T_{part}$$

[3] «part» comme partenaire.

**Corollaire 2 (relations entre le type d'une référence et le type du port associé)**
$\forall u$ *tel que* $Com.u = \varnothing$ :
$u{:}T_{part} \in \mathsf{R} \ \wedge \ u{:}T \in \mathsf{P}_u \Rightarrow T \preceq T_{part}$
*Si* $u \in \mathsf{R}_u$ *nous avons* $T = T_{part}$.

**Preuve.** Immédiat d'après le lemme 2 page précédente et la propriété 3.5 page 53 ($Comp(I,J) \Leftrightarrow I \preceq J^{\mathcal{D}}$). Le cas $u \in \mathsf{R}_u$ se décompose en deux possibilités :

- le port $u$ n'a pas été envoyé à l'extérieur du composant. L'égalité $T = T_{\text{part}}$ est évidente ;
- le port $u$ a été envoyé, et reçu ensuite par son composant. Lors de la réception de la référence $u{:}T'$, le type de cette référence est enregistré par $\tilde{C} \Leftarrow u{:}T'$ ; comme cette opération ne modifie pas le type de $u$ dans $\tilde{C}$, nous avons l'égalité $T = T_{\text{part}}$.

□

**Preuve (lemme 2).**

Par induction. Comme les files d'attentes doivent être vides, nous considérons, pour simplifier la démonstration, que les messages sont consommés immédiatement. Une démonstration plus rigoureuse étiquetterait les messages avec les types correspondant aux références (type de l'émetteur juste après l'envoi et type des arguments). Le lemme est vérifié à l'assemblage de part la propriété 3.5 page 53.

**1.** CREAT, REMVPORT, REMVREF, DEACT, OTHER
Immédiat.

**2.** BIND

On constate tout d'abord, en regardant les conditions de C-BIND (règle concernant les transitions du composant étiquetées bind($u \multimap v$), table 4.4 page 66), que l'on a $u$ en émission.

La règle BIND donne ($u{:}T \multimap v{:}T'_{\text{part}}$). Le cas de $v$ est trivial (les hypothèses pour $v$ n'étant pas modifiées lors de l'application de la règle). Concernant $u$, nous avions, avant l'attachement : ($u \multimap \perp$). Nécessairement, $u \in \mathsf{R}_u$ car la référence d'un port en émission ne peut être envoyée. Le lemme n'est pas concerné par ce cas.

**3.** SEND, RECV, RECV*, RECV-UN

Nous supposons, pour simplifier, qu'une réception a lieu immédiatement après un envoi. Nous différencions les cas suivants :

**références $\tilde{v}_k$ en argument de message**

Le raisonnement est assez trivial pour toute référence $w \in \tilde{v}_k$, car de part les relations de sous-typage et de compatibilité, $w$ envoyée avec un type $W_{\text{part}}$ ne peut être enregistrée par le récepteur qu'avec un type qui est super-type de $W_{\text{part}}$. La relation $\preceq$ étant transitive, le lemme reste donc vrai pour toute référence envoyée en argument de message.

**partenaire inconnu au moment de la réception** (règle SEND suivie de RECV-UN ou RECV*)

La démonstration fait utilisation de la règle duale définie sur les références pairs (section 3.6 page 53) ; on remarquera que, dans les règles, la relation de dualité est bien appliquée uniquement à ce type de référence (et non à des références serveurs).

Avant l'application des règles, nous avions (avec $u$ en envoi, $v$ en réception) :

$$(u{:}T \multimap v{:}T'_{\text{part}}) \qquad (v{:}T' \multimap \bot) \text{ avec par induction } T' \preceq T'_{\text{part}}$$

Nous notons $T_k$, $T'_{\text{part}_k}$ et $T'_k$ les types résultant de l'envoi ou la réception du message $M_k$. Lorsque $u$ applique SEND, et $v$ applique l'une des deux règles RECV, nous avons :

$$(u{:}T_k \multimap v{:}T'_{\text{part}_K}) \qquad (v{:}T'_k \multimap u{:}{T'_k}^{\mathcal{D}}) \text{ avec } T'_k \preceq T'_{\text{part}_k}$$

Concernant $u$, nous devons montrer $(T'_{\text{part}_k})^{\mathcal{D}} \preceq {T'_k}^{\mathcal{D}}$, qui est obtenue à partir de l'hypothèse d'induction $T'_k \preceq T'_{\text{part}_k}$ et de la propriété 3.5 page 53 ($I \preceq J \Leftrightarrow J^{\mathcal{D}} \preceq I^{\mathcal{D}}$).

Concernant $v$, nous devons montrer $({T'_k}^{\mathcal{D}})^{\mathcal{D}} \preceq T'_{\text{part}_k}$, qui est immédiat puisque ${T'_k}^{\mathcal{D}^{\mathcal{D}}} = T'_k$.

**partenaire connu au cours de la réception** (règle SEND suivie de RECV)

Avant l'application des règles, nous avions (avec $u$ en envoi, $v$ en réception attaché à $w$, avec $w$ égal ou différent de $u$) :

$$(u{:}T \multimap v{:}T'_{\text{part}}) \qquad (v{:}T' \multimap w{:}T'') \text{ avec } {T''}^{\mathcal{D}} \preceq T'_{\text{part}}$$

Nous noterons $T_k$, $T'_{\text{part}_k}$, $T'_k$ et $T''_k$ les types résultant de l'envoi ou la réception du message $M_k$. Bien entendu : ${T''_k}^{\mathcal{D}} \preceq T'_{\text{part}_k}$.
Lorsque $u$ applique SEND, et $v$ applique RECV, nous avons :

$$(u{:}T_k \multimap v{:}T'_{\text{part}_k}) \qquad (v{:}T'_k \multimap u{:}T''_k) \text{ avec } {T''_k}^{\mathcal{D}} \preceq T'_{\text{part}_k}$$

Concernant $v$, la propriété ${T''_k}^{\mathcal{D}} \preceq T'_{\text{part}_k}$ est vraie. Nous en déduisons facilement, pour le port $u$ : ${T'_{\text{part}_k}}^{\mathcal{D}} \preceq T''_k$ de par $({T''_k}^{\mathcal{D}})^{\mathcal{D}} = T''_k$ et la propriété 3.5 page 53.

□

### 5.3.3 Absence d'interblocage externe

#### 5.3.3.1 Exemple et formalisation de l'interblocage externe

L'interblocage externe représente une situation où un ensemble de ports sont bloqués à cause d'un cycle de dépendance. Le cas le plus simple d'interblocage externe est donné par la figure 5.1 page ci-contre : $u$ est bloqué en envoi par $v$ qui attend que $v'$ effectue son envoi, ce dernier étant bloqué par $u'$ qui attend que $u$ envoie son message.

Le cas général est plus complexe, et repose sur le principe des *Wait-For-Graph* : l'existence de cycles dans le graphe de dépendance construit au fur et à mesure de l'évolution du système. Si un cycle apparaît, alors il y a interblocage.

Nous introduisons une nouvelle relation de dépendance entre les ports, nommée dépendance externe, et notée $\dashrightarrow$, relatant les communications entre les ports distants. Par exemple, $u\mathsf{?^a} \dashrightarrow v\mathsf{!^s}$ veut dire que $u$ dépend de $v$, plus précisément que $u$ attend un envoi de $v$.

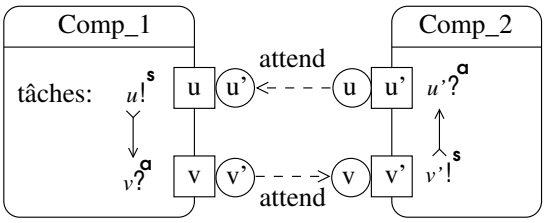



**FIG. 5.1:** *Exemple d'interblocage externe*
$(u \multimap u') \wedge (v \multimap v') \wedge (u!^{\mathsf{s}} \rightarrowtail v?^{\mathsf{a}}) \wedge (v'!^{\mathsf{s}} \rightarrowtail u'?^{\mathsf{a}})$

**Définition 2 (Dépendance externe $\dashrightarrow$)**
$u \dashrightarrow v$ *ssi* $u?^{\mathsf{a}} \ \wedge \ \big((v!^{\sigma} \ \wedge \ (v \multimap u)) \ \vee \ (v?^{\mathsf{a}} \ \wedge \ (u \multimap v) \ \wedge \ u : M(\ldots) \in Com.v)\big)$

**Remarque :** dans la relation de compatibilité, nous avons $Comp(\mathsf{may?}, \mathsf{may?})$. Dans ce cas, les files d'attente sont vides, et il n'y a pas de relation de dépendance. Dans les autres cas, le fait que $v$ puisse être en réception, soit $u?^{\mathsf{a}} \dashrightarrow v?^{\mathsf{a}}$, peut paraître surprenant. Cependant, ce cas particulier est dû au fait que les communications sont asynchrones ; il montre que $u$ vient d'envoyer un message $M$ et attend une réponse de $v$, tandis que $v$ n'a pas encore consommé le message $M$. Lorsque $v$ aura effectué son action, la relation de dépendance deviendra alors :

- $u?^{\mathsf{a}} \dashrightarrow v!^{\mathsf{a}}$ ;
- $u?^{\mathsf{a}} \dashrightarrow v?^{\mathsf{a}}$ s'il y a encore un message dans la file d'attente de $v$ ;
- $v$ n'a pas d'action ($v\mathbf{0}^{\mathsf{a}}$), dans ce cas le type de $u$ est obligatoirement **may?**, car sinon les types ne sont pas compatibles ; il n'y a pas de relation de dépendance entre $u$ et $v$ ;
- $v$ est en réception, mais sa file d'attente est vide. Dans ce cas, nous avons obligatoirement **may?** pour les types des deux ports (seule compatibilité possible si les ports sont tous les deux en réception). Dans ce cas, la dépendance est inexistante.

Nous définissons ensuite la relation $u\mathcal{S}v$ qui est vraie lorsque $u$ est suspendu à $v$ :

**Définition 3 ($\cdot\mathcal{S}\cdot$)**
$u\mathcal{S}v \triangleq u \rightarrowtail v \ \vee \ u \dashrightarrow v$
$u\mathcal{S}^*v \triangleq$ *fermeture transitive de* $\mathcal{S}$.

Un interblocage a lieu lorsqu'un cycle apparaît dans les relations de dépendance, ce qui s'écrit :

$$\begin{aligned}\text{Ext_deadlock}(\mathcal{C}) &\triangleq \exists u \in \mathcal{C} \text{ tel que } u\mathcal{S}^*u \\ &\triangleq \exists (u_i)_{1..n} \in \mathcal{C} \text{ telle que } \forall k < n : u_i\mathcal{S}u_{i+1} \wedge u_n\mathcal{S}u_1\end{aligned}$$

On remarque immédiatement que les ports oisifs ou actifs en émission ne sont pas concernés par un interblocage, car il ne dépendent d'aucun port.

**Théorème 2 (Absence d'interblocage externe)**
*Si* $\mathcal{A}$ *est sain, alors* $\mathcal{A} \vDash P_{\text{edf}}$
*avec* $P_{\text{edf}} \triangleq \forall \mathcal{C}, \mathcal{A} \rightarrow^* \mathcal{C} \Rightarrow \neg\text{Ext_deadlock}(\mathcal{C})$

La preuve du théorème 2 page précédente est fastidieuse. Même si les interfaces sont mutuellement compatibles, il n'est pas évident qu'un interblocage ne surviendra jamais entre les composants (les ports d'un composant peuvent être suspendus en attente de réception d'un message d'un autre port, ce qui mène à des dépendances potentielles entre les tâches, rajoutées aux dépendances internes entre les ports).

#### 5.3.3.2 Absence d'interblocage : démonstrations intermédiaires

Tous les lemmes se font sous la supposition que les interfaces sont compatibles, et que les composants respectent leurs contrats. Toutes les démonstrations se font par induction structurelle sur les règles de respect de contrat. Nous ne considérons que les transitions menant à des configurations stables, nous ignorerons donc les transitions menant à *Error*.

Le lemme suivant indique où se trouve une référence pair.

**Lemme 3**
*Soit $v$ une référence pair. Alors nous avons, de manière exclusive :*

1. *si $\exists B(\mathsf{P},\mathsf{R},\mathsf{T})$ tel que $v \in \mathsf{R}$, alors $B$ est unique ;*
2. *$\exists!(u, M)$ tel que $M(\ldots, v, \ldots) \in Com.u$ ; de plus $\nexists B(\mathsf{P}, \mathsf{R}, \mathsf{T})$ tel que $v \in \mathsf{R}$ ;*
3. *$\exists M(\ldots)$ tel que $v : M(\ldots) \in Com.u \ \wedge \ (u \not\multimap v)$ ; de plus $\nexists B(\mathsf{P}, \mathsf{R}, \mathsf{T})$ tel que $v \in \mathsf{R}$ ;*

**Preuve.** Par induction structurelle.

**1.** CREAT, REMVPORT, REMVREF, BIND, DEACT, OTHER
Immédiat.

**2.** SEND
$u$ envoie le message $M_k(\tilde{v})$ vers $u'$. Nous noterons $B_u(\mathsf{P}_u, \mathsf{R}_u, \mathsf{T}_u)$ le composant pour lequel $u \in \mathsf{P}_u$, et $B_{u'}$ de la même manière (il est possible que les deux composants soient les mêmes). Nous nous intéressons à trois types de références concernant le lemme à démontrer :

**une référence pair est dans un argument du message.**
Prenons une référence pair $v \in \tilde{v}$. Par induction, et sachant $v \in \mathsf{R}_u$, nous savons que $B_u$ est l'unique composant contenant $v$, et que les points 2. et 3. du lemme sont faux. Lorsque $v$ est envoyée, elle est retirée de $\mathsf{R}_u$ (voir la règle C-SEND du tableau 4.4 page 66) ; une fois la règle appliquée, nous avons donc $v$ qui n'est connue d'aucun composant, et il existe un message unique dans $Com$ dont les arguments contiennent cette référence (et le point 3. du lemme est faux).

**la référence $u'$ vers laquelle le message est envoyé.**
Le lemme reste vrai pour $u'$. Seul le point 1. est vrai, et nous avons $u' \in \mathsf{R}_u$.

**la référence $u$ ayant effectué l'envoi.**
Nous ne considérons pas le cas où $u'$ est attaché à $u$, car dans ce cas nous avons $u \in \mathsf{R}_{u'}$, et les points 2. et 3. sont faux.
Considérons le cas où $(u' \not\multimap u)$. Deux possibilités se présentent :

**$u$ effectue son premier envoi vers $u'$.** Intéressons-nous tout d'abord à la façon dont $u'$ a été attaché au port $u$. Cet attachement n'est possible que :

- par réception d'un message, ce qui est impossible puisque $(u' \not\multimap u)$ implique que $u'$ n'a pas pu envoyer de message vers $u$ ;
- par la régle BIND.
  Nous allons maintenant considérer l'action de $u$. Entre l'application de BIND et de SEND (l'envoi qui nous préoccupe), $u$ n'a pu faire de réception (auquel cas nous aurions $(u' \multimap u)$ pour que $u'$ puisse envoyer le message attendu) ni d'émission (puisque $u$ effectue son premier envoi).
  Entre la création de $u$ et l'application de BIND, $u$ n'a pu faire d'action.

Nous savons donc que, depuis que $u$ a été crée, $u$ est toujours resté en émission. Comme il s'agit de son premier envoi, nous avons $u \in \mathsf{R}_u$ et $\nexists M$ tel que $M(\ldots, u, \ldots) \in Com.w$ avant l'application de SEND (une référence en émission ne peut être envoyée). Comme cette règle retire $u$ de $\mathsf{R}_u$, nous avons bien le point 3. du lemme, et les deux autres sont faux.

**$u$ a déjà envoyé un message à $u'$.** Nous avions alors le point 3. du lemme qui était vrai par induction (rappelons que nous considérons ici $(u' \not\multimap u)$, donc le premier message envoyé par $u$ n'a pas été consommé par $u'$). Le lemme reste donc vrai après l'envoi du message.

**3.** RECV, RECV*, RECV-UN

Par le même raisonnement que SEND. Nous considérons le message $u' : M(\tilde{v})$ reçu par la référence $u$, avec toujours $B_u(\mathsf{P}_u, \ldots)$ tel que $u \in \mathsf{P}_u$. Nous considérons trois cas.

**les références contenues dans les arguments du message reçu.**

Par hypothèse d'induction, nous savons qu'il existe un unique message $u' : M(\ldots, v, \ldots)$ dans $Com.u$, et qu'aucun composant ne connaît $v$ dans l'ensemble de ses références $\mathsf{R}$. Donc, après réception, $B_u$ est le seul composant tel que $v \in \mathsf{R}_u$, et, le message ayant été retiré de la file de $u$, il n'existe aucun autre message dans $Com$ contenant $v$. La propriété est conservée.

**la référence $u'$ émettrice du message.**

Dans le cas où le message vient de la référence à laquelle $u$ était attaché, il n'y a pas de modification particulière (point 1. du lemme vrai).

Dans le cas où le message provient d'une nouvelle référence, alors avant réception du message nous avions le point 3. du lemme qui était vrai. Après réception, nous avons $u' \in \mathsf{R}_u$ et $B_u$ est bien le seul composant ayant connaissance de $u'$. Le point 1. du lemme est donc vrai, et, comme $(u \multimap u')$ est vrai après la réception, le point 3. est faux.

**la référence $u$ réceptrice.**

Pas de changements lors de l'application de la règle.

□

Le lemme suivant concerne les relations de dépendance externes entre ports pairs. Il stipule que pour ces ports, il n'existe qu'une seule relation de dépendance. C'est une conséquence du lemme 3.

**Lemme 4**

*Si $u$ et $v$ sont des ports pairs tels que $u \dashrightarrow v$, alors $u$ et $v$ sont uniques.*

**Preuve.** Prenons $u$ et $v$ tels que $u \dashrightarrow v$. Par définition, nous avons :

$$u\mathsf{?}^{\mathsf{a}} \ \wedge \ \big((v!^{\sigma} \ \wedge \ (v \multimap u)) \ \vee \ (v\mathsf{?}^{\mathsf{a}} \ \wedge \ (u \multimap v) \ \wedge \ u : M(\ldots) \in Com.v)\big)$$

Nous avons donc deux cas possibles (nous noterons $B_u(\mathsf{P}_u, \mathsf{R}_u, \mathsf{T}_u)$ le composant $B_u$ tel que $u \in \mathsf{P}_u$) :

$v$ **est en envoi :** $v!^{\sigma} \ \wedge \ (v \multimap u)$.

Comme $v$ ne peut avoir qu'un seul partenaire, il n'existe qu'un seul $u$ tel que $u \dashrightarrow v$.

Concernant l'unicité de $v$ : comme nous avons $(v \multimap u)$ alors nécessairement $u \in \mathsf{R}_v$ ; en effet d'un part les seules règles effectuant un attachement sont BIND et RECV, et ces règles prennent le partenaire de l'ensemble $\mathsf{R}$ ou l'y rajoutent, et d'autre part l'envoi d'une référence (et par conséquent sa suppression de l'ensemble $\mathsf{R}$) n'est possible que si elle n'est pas attachée. Supposons qu'il existe un composant $B_w(\mathsf{P}_w, \mathsf{R}_w, \mathsf{T}_w)$ dans lequel $w \neq v$ et $(w \multimap u)$ ; par le même raisonnement nous avons $u \in \mathsf{R}_w$. Par le lemme 3 nous avons $B_w = B_v$. Or il est impossible d'avoir dans un même composant une référence attachée à deux ports différents : la règle BIND interdit cela, et si l'attachement s'est effectué par la règle RECV, alors cela veut dire que $u$ a changé de partenaire entre temps, ce qui est impossible ($u$ ne peut modifier son partenaire que lors d'une réception, et dans notre cas le message reçu ne peut venir que de $v$). Nous avons donc l'unicité de $v$.

$v$ **est en réception :** $v\mathsf{?}^{\mathsf{a}} \ \wedge \ (u \multimap v) \ \wedge \ u : M(\ldots) \in Com.v$

Le même raisonnement s'applique. En supposant qu'il existe un $w \neq u$ ayant les mêmes propriétés que $u$, alors $w$ se trouve dans le même composant que $u$. Or il est impossible d'avoir une référence attachée à deux ports différents, donc $u$ est unique. $v$ est lui aussi unique car un port n'a qu'un seul partenaire.

□

#### 5.3.3.3 Absence d'interblocage : démonstration

Nous devons montrer qu'il n'y a pas d'interblocage, dont nous rappelons la définition :

$$\text{Ext_deadlock}(\mathcal{C}) \triangleq \exists (u_i)_{1..n} \in \mathcal{C} \text{ telle que } \forall k < n : u_i \mathcal{S} u_{i+1} \wedge u_n \mathcal{S} u_1$$

**Preuve (Absence d'interblocage externe).**

Par induction structurelle sur les règles de respect de contrat.

**1.** CREAT, REMVPORT, REMVREF, DEACT

Ces règles ne créent aucune nouvelle relation de dépendance.

**2.** BIND

Il y a création d'une relation de dépendance. Cependant comme seuls les ports actifs ou oisifs en émission peuvent faire un attachement, et que le prédicat Ext_deadlock($\mathcal{C}$) ne fait pas intervenir ce genre de ports, la propriété d'absence d'interblocage est maintenue.

**3.** SEND

Nous distinguons suivant la prochaine action $\rho$ du port $u$ :

$\rho = \mathsf{!}$ les dépendances restent inchangées ;

$\rho = \mathbf{0}$ la dépendance entre $u$ et son partenaire disparaît ;

$\rho = \mathsf{?}$ Nous avons ici le cas particulier d'une dépendance externe de type $u\mathsf{?}^{\mathsf{a}} \dashrightarrow v\mathsf{?}^{\mathsf{a}}$, avec $v$ partenaire de $u$ : $(u \multimap v)$.

Raisonnons par l'absurde et supposons (les calculs sur les indices se faisant modulo $n$) :

$$\exists (u_i)_{1..n} \in \mathcal{C} \text{ tel que } \forall k : u_i \mathcal{S} u_{i+1}$$

En particulier, nous avons $u$ et $v$ prenant part à cette circularité (puisque la règle a modifié une dépendance qui les concerne). Choisissons la suite $(u_i)$ telle que $u_1 = u$ et $u_2 = v$ ; nous avons bien $u \mathcal{S} v$.

Nous allons montrer que nécessairement si $v \mathcal{S} w$ alors $w = u$. Comme $v$ est actif, la seule relation de dépendance entre $v$ et $w$ est une dépendance externe : $v \dashrightarrow w$. Par le lemme 4 nous avons $w = u$.

Nous avons donc à la fois :

$$u \dashrightarrow v \text{ soit } v\mathsf{?}^{\mathsf{a}} \ \wedge \ (u \multimap v) \ \wedge \ u : M(\ldots) \in Com.v$$
$$v \dashrightarrow u \text{ soit } u\mathsf{?}^{\mathsf{a}} \ \wedge \ (v \multimap u) \ \wedge \ v : M'(\ldots) \in Com.u$$

En particulier, nous avons $v : M'(\ldots)$ dans la file d'attente de $u$, ce qui est contradictoire puisque $u$ vient d'effectuer un envoi, et que les types de $u$ et $v$ sont compatibles (donc $u$ aurait fait moins de réceptions que le nombre d'émission de $v$).

**4.** RECV, RECV-UN

Tout comme SEND, trois cas sont envisagés suivant la valeur de $\rho$ :

$\rho = \mathsf{?}$ nous éliminons le cas où la relation de dépendance est supprimée (dans le cas où le partenaire $v$ est sous-type de $\mathbf{0}$).

S'il y a une relation de dépendance externe, elle est identique à celle présente avant l'application de la règle. En effet, si $v$ le partenaire de $u$ est en réception, alors nous avions avant la réception du message : $v \dashrightarrow u$ soit $u\mathsf{?}^{\mathsf{a}} \ \wedge \ (v \multimap u) \ \wedge \ v : M(\ldots) \in Com.u$, relation qui est maintenue après la réception (il y a dépendance entre les deux ports, et la file d'attente de $v$ est forcément vide de part la compatibilité des types de $u$ et $v$). Si le partenaire $v$ est en émission, alors nous avions $u \dashrightarrow v$ soit $v\mathsf{!}^{\sigma} \ \wedge \ (v \multimap u)$, relation qui reste vraie après la réception. On remarquera de plus que le changement de partenaire pour $u$ lors de la réception du message ne modifie pas le raisonnement ;

$\rho = \mathbf{0}$ la dépendance entre $u$ et son partenaire disparaît ;

$\rho = \mathsf{!}$ nous avons donc $\mathsf{T}(u') = \mathsf{!}^{\mathsf{a}}$. Comme le prédicat d'interblocage externe ne prend pas en compte ce genre de ports, nous avons automatiquement $\neg \text{Ext_deadlock}(\mathcal{C}')$.

**5.** RECV*

Comme il y a création d'un port $u'$, nous avons une nouvelle relation de dépendance entre l'émetteur du message et ce nouveau port. Cependant, comme $u'$ a sa

propre tâche d'exécution, il est immédiat qu'il n'y a pas de création de cycle dans les relations de dépendances.

**6.** OTHER

Cette règle concerne toutes les autres transitions du composant, correspondant aux manipulations de tâches : il s'agit des règles C-ACTV, C-ACTV2, C-THSUSP et C-THRELAX.

La conclusion pour C-ACTV2, et C-THRELAX est assez immédiate : soit un port est activé mais il a sa propre tâche, soit une dépendance est supprimée. Il n'y a donc pas de création d'interblocage.

La règle C-ACTV ajoute la dépendance $u!^{\mathsf{s}} \rightarrowtail v!^{\mathsf{a}}$. Comme le prédicat d'interblocage externe ne concerne pas les ports actifs en émission, cette nouvelle relation ne crée pas de cycle de dépendance.

Le raisonnement pour la règle C-THSUSP est plus complexe. Raisonnons par l'absurde et supposons (les calculs sur les indices se faisant modulo $n$) :

$$\exists (u_i)_{1..n} \in \mathcal{C} \text{ tel que } \forall k : u_i \mathcal{S} u_{i+1}$$

Nécessairement, nous avons un port de $t_1$ et un port de $t_2$ prenant part à ce cycle de dépendance (la règle ayant créé la relation $t_1 \rightarrowtail t_2$). De part la définition d'une tâche, nous avons de plus tous les ports de $t_1$ et de $t_2$ qui participent au même cycle, en particulier les port en tête de chacune des tâches. Nous noterons $t_i \sqsubseteq (u_i)$ le fait que les ports de $t_i$ prennent part au cycle de dépendance (en fait, $t_i$ est une sous séquence de la suite $(u_i)$).

D'après la pré-condition de C-THSUSP, nous avons entre autres :

$$\text{head}(t_1) = !^{\mathsf{a,s}} \ \wedge \ \text{head}(t_2) = !^{\mathsf{a,s}} \ \wedge \ \forall t_i \text{ tel que } t_2 \rightarrowtail t_i : \text{head}(t_i) = !^{\mathsf{a,s}}$$

Or, le prédicat Ext_deadlock($\mathcal{C}$) ne prenant pas en compte les ports qui sont actifs en émission, nous pouvons écrire

$$\text{head}(t_1) = !^{\mathsf{s}} \ \wedge \ \text{head}(t_2) = !^{\mathsf{s}} \ \wedge \ \forall t_i \sqsubseteq (u_i) \text{ tel que } t_2 \rightarrowtail t_i : \text{head}(t_i) = !^{\mathsf{s}}$$
$$\text{Soit } \forall t_k \sqsubseteq (u_i), \text{head}(t_k) = !^{\mathsf{s}}$$

Nous déduisons donc : $\forall t_k \sqsubseteq (u_i), t_2 \rightarrowtail^* t_k$

En particulier : $t_2 \rightarrowtail^* t_1$. La règle C-THSUSP n'ayant rajouté que la dépendance $t_1 \rightarrowtail t_2$, nous pouvons dire qu'avant son application nous avions $t_2 \rightarrowtail^* t_1$, ce qui est contradictoire avec la précondition de C-THSUSP (en fait, tous les ports de la suite $(u_i)$ sont suspendus en émission ; cette suite représente un interblocage interne du composant, qui est proscrit par la règle C-THSUSP).

La propriété $\neg$Ext_deadlock($\mathcal{C}'$) est donc vraie. □

### 5.3.4 Propriété de vivacité (sous contraintes)

L'assemblage de composant a un problème de vivacité que nous avons soulevé lors du corollaire 1 page 84 : un port peut ne jamais voir sa transition exécutée. L'obligation de réception souffre d'un problème d'équité.

L'obligation d'envoi est plus particulière. La sémantique **must!** indique que le port *enverra* le message. Cependant, cela nous interdit le fait de pouvoir déléguer l'envoi à un autre composant. La sémantique doit être plus souple : l'obligation d'envoi est surtout (voire uniquement) utilisée pour assurer qu'un port typé **must?** recevra son message. Nous proposons donc de mettre de coté l'obligation d'envoi, et d'essayer plutôt d'assurer que toute référence dont la première action est typée **must?** sera finalement attachée à un port, et un message envoyé. La formule en logique temporelle est la suivante. Elle demande qu'infiniment souvent, il existe un port qui a une transition pour effectuer l'envoi, *et que ce port ne se suspendra pas ou ne deviendra pas oisif* tant qu'il n'aura pas effectué l'envoi. Concernant le port, la seule action possible est un envoi. Il nous reste un problème d'équité entre les ports, mais il est identique à celui de la consommation de messages.

$$P_{\textbf{must?}}(u) \triangleq \Box\Diamond\exists v, \mathcal{C}, \mathcal{C}' :$$
$$(v \multimap u) \ \wedge \ \mathcal{C} \xrightarrow{v:u!M_k} \mathcal{C}'$$
$$\wedge \ \forall \mathcal{C}'', \alpha : \mathcal{C} \xrightarrow{\alpha} \mathcal{C}'' \Rightarrow \alpha \notin \{\mathrm{actv}(v \rightarrowtail w), \mathrm{deact}(v), \mathrm{susp}(t_v \rightarrowtail t_w)\}$$

Cette propriété peut être vérifiée si, au niveau de chaque composant, il est possible de démontrer :

- si, lorsqu'un port est créé, sa première action est typée **must!**, alors il finira par être actif, et enverra ce premier message.
  La propriété à vérifier est donc (en faisant abstraction de P, R et T, et où $B_i, \tilde{C}_i$ est un état du composant et de son contrat) :

$$B, \tilde{C} \xrightarrow{\mathrm{creat}(u)} B', \tilde{C} \Leftarrow u{:}\textbf{must! [...]} \quad \text{implique,tant que } u : \textbf{must! [...]} :$$
$$\Box\Diamond\Big(B_1, \tilde{C}_1 \xrightarrow{u:u'!M_k} B_2, \tilde{C}_2$$
$$\wedge \forall C_3, \alpha : B_1, \tilde{C}_1 \xrightarrow{\alpha} B_3, \tilde{C}_3 \Rightarrow \alpha \notin \{\mathrm{actv}(u \rightarrowtail v), \mathrm{deact}(u), \mathrm{susp}(t_u \rightarrowtail t_v)\}\Big)$$

  Cette contrainte est vérifiable en vérification de modèles. Il est toutefois possible de modifier les règles de respect de contrat de la manière suivante :
  - ajouter un booléen *message_ envoyé* qui indique si l'action a été effectuée ou non. La création du port l'initialise à *faux*, les envois le modifient à *vrai*. Il est interdit de supprimer le port si ce booléen est faux ;
  - ajouter un compteur qui indique le nombre de transitions où le port est resté suspendu ou oisif. Il faut garantir que ce compteur est borné. Cette méthode s'inspire largement de la méthode de Kobayashi[Kob02] que nous avons présentée à la fin de la section 2.4.2 page 34 ;
  - vérifier qu'un tel port ne se suspend pas sur un port qui peut potentiellement avoir un dialogue sans fin avec son partenaire. Dans le cas des types récursifs, il faut prouver que le dialogue termine.
- La deuxième propriété à vérifier est qu'une référence typée **must?** et reçue par le composant est finalement attachée à un port de ce composant :

$$B, \tilde{C} \xrightarrow{v:v'?M(\ldots,u,\ldots)} B', \tilde{C} \Leftarrow u{:}\textbf{must? [...]}$$
$$\text{implique, tant que } \nexists w, (w \multimap u) : \Box\Diamond B_1, \tilde{C}_1 \xrightarrow{\mathrm{bind}(w \multimap u)} B_2, \tilde{C}_2$$

Cela peut se faire en obligeant l'attachement juste après la réception de la référence, ou en vérifiant que pour toute référence **must?** du composant, il existe une transition «bind($u \multimap v$)» (avec éventuellement «creat($u$)» juste avant l'attachement).

Nous pensons que la vérification de ces deux propriétés permet d'affirmer, pour une référence $v$ dont la première action est typée **must?**, que $v$ sera attachée à un port, et un envoi sera effectué. Dans notre proposition, il nous reste le problème d'équité entre les tâches d'exécution, mais nous pensons que c'est un problème à part (par exemple, il faut prouver que l'ordonnanceur des tâches est équitable : alors toute transition possible infiniment souvent sera finalement exécutée).

## 5.4 Etude de cas

Nous reprenons les diagrammes d'états donnés dans la section 4.5 page 68. Nous ajoutons à chaque état le contrat du composant, représenté par un parallélogramme. Il donne l'ensemble des types connus par le composant. Les parties grisées indiquent la vérification mise en jeu lors de telle ou telle transition.

### 5.4.1 Vérification du composant Rapporteur

La vérification du composant `Rapporteur` est présenté figure 5.2 page ci-contre. A l'*Etat initial*, le contrat $\tilde{C}$ contient le type de la référence serveur «$g^* : gestion_accès$» et celui de la référence du port local «$r : i_rapporteur$». Les types ont été présentés dans la section 3.7 page 55, nous les rappelons ici en partie :

```
i_rapporteur = must ! [ accès_rapporteur (...) ;
                              must ? [   accordé (...) ; entrer_rapport
                                       + refusé ; 0 ] ]
gestion_accès = may ? *[   accès_rapporteur (...) ;
                              must ! [   accordé (...) ; gestion_rapporteur
                                       + refusé ; 0 ]
                         + accès_auteur ; gestion_auteur]
```

A l'*Etat initial*, la règle SEND vérifie que la transition vers l'état suivant est autorisée. L'état du composant et les types peuvent évoluer alors en conséquence :

- le composant passe à l'état *Attente accès* ;
- le type de la référence $r$ évolue vers :

```
must ? [   accordé (...) ; entrer_rapport
         + refusé ; 0 ] ]
```

- le type initial de la référence serveur est conservé ($g^*$ : **may?*[...]**), tandis que le type de la référence $g$ évolue vers :

```
must ! [   accordé (...) ; gestion_rapporteur
         + refusé ; 0 ]
```

**Remarque :** Il n'est pas possible d'envoyer la référence $g$ avec le type **must![...]**. De plus, lorsque la réponse du serveur sera reçue, le partenaire $g$ de $r$ est écrasé par le nouveau partenaire, $a$. Avoir dans le contrat $\tilde{C}$ à la fois $g^*$ et $g$ qui représentent le même port, mais sous des types différents, ne pose donc pas de problèmes.

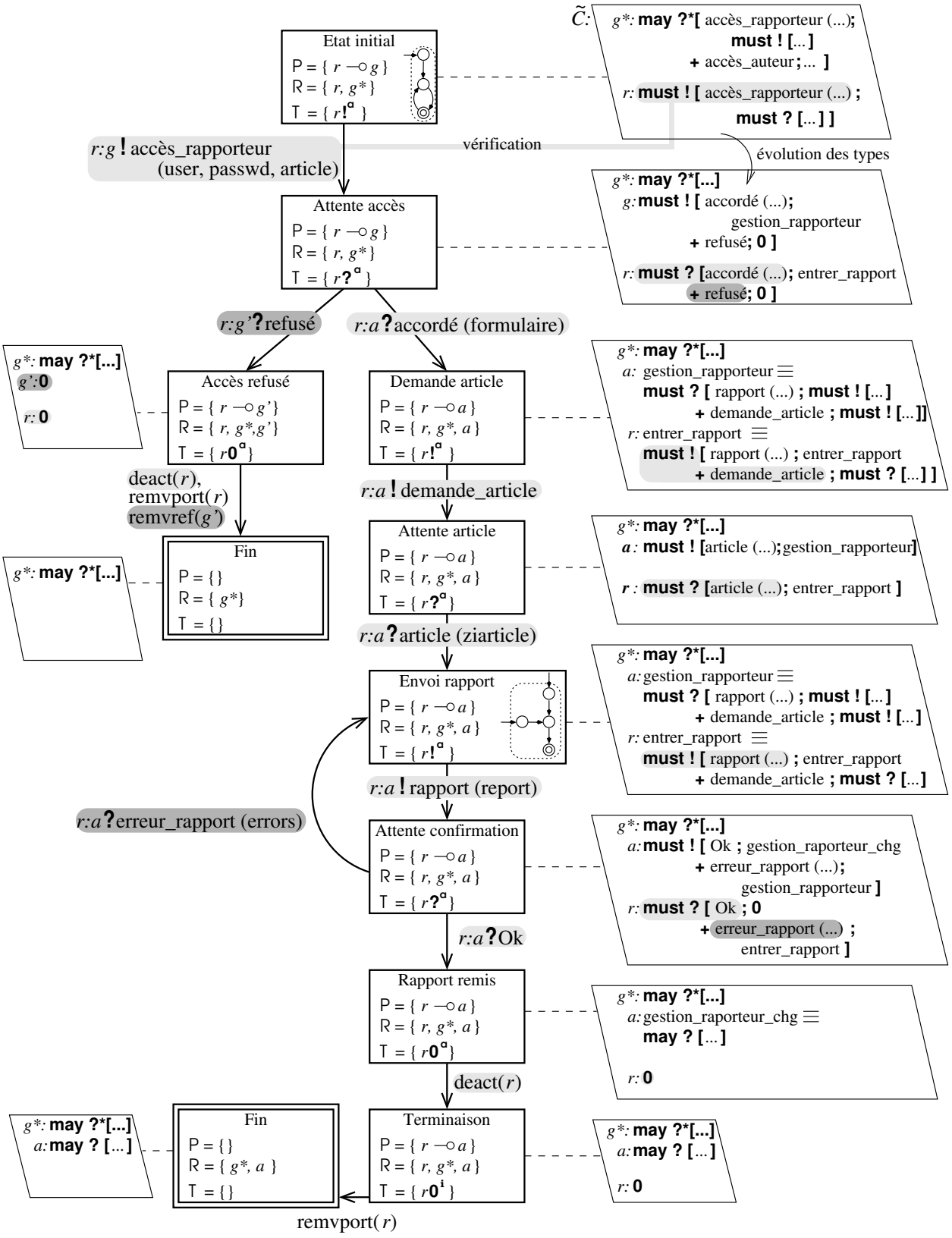



FIG. 5.2: *Vérification du composant* `Rapporteur`

Les transitions sortantes de l'état *Attente accès* présentent l'intérêt du changement de partenaire : $g'$ ou $a$ (selon le cas) sont tous deux typés suivant l'évolution du type associé à $g$ : «$g'$:**0**» et «$a$:*gestion_rapporteur*». On remarquera notamment que le message ne porte pas le type des références $g'$ et $a$ ; ce type est inféré à partir du type de l'ancien partenaire (ici, à partir du type de $g$).

Le diagramme se divise en deux après l'état *Attente accès*. La partie gauche concerne la suppression des ports et références lorsque l'accès est refusé : la transition «deact($r$)» est autorisée parce que $r$ n'est pas en réception, «remvport($r$)» ne présente pas de restriction, et «remvref($g'$)» est autorisée car $g'$ n'est pas typé **must?**.

La partie droite du diagramme présente une succession d'émissions et de réceptions. Elle se termine par la déactivation de $r$, qui est autorisée car le port n'est pas en réception.

On remarquera que le composant `Rapporteur` peut à son tour envoyer la référence $a$ à un autre composant (par exemple un co-rapporteur) pour modifier le rapport. Il peut aussi supprimer cette référence, car elle n'est pas typée **must?**.

### 5.4.2 Vérification du composant Gestion

La vérification du composant Gestion est donnée figure 5.3 page ci-contre. Nous rappelons le type du port serveur $g^*$ :

gestion_accès = **may ? *[** accès_rapporteur (...) ;
**must ! [** accordé (...) ; gestion_rapporteur
+ refusé ; **0 ]**
+ accès_auteur ; gestion_auteur**]**

La réception d'une requête *accès_rapporteur* demande à ce que les actions «creat($g'$)», «bind($g' \multimap r$)» et «actv($g'$)» soient effectuées juste après la réception. Le contrat contient alors :

- le type du port $g'$ créé pour répondre au client. Ce type est le suivant :

  **must ! [** accordé (...) ; gestion_rapporteur
  + refusé ; **0 ]**

- $r$ est enregistré avec le type dual de celui de $g'$, soit :

  **must ? [** accordé (...) ; entrer_puis_changer_rapport
  + refusé ; **0 ] ]**

  avec *entrer_puis_changer_rapport* le type permettant d'envoyer un rapport un nombre infini de fois. On remarquera ici une application du corollaire 2 page 85 : le type de $r$ dans le contrat du composant `Gestion` est sous-type du type réel de $r$.

Les autres vérifications se déduisent facilement à partir des règles. L'envoi de la référence $r$ est bien valide (voir état *Envoi référence (3)*, en bas du diagramme) : le message *rapporteur* qui est émis demande à ce que son argument soit typé *rapporteur_accepté*≡**must ? [** accordé(...) ; entrer_puis_changer_rapport **]**.
La référence $r$ est bien sous-type de *rapporteur_accepté*.

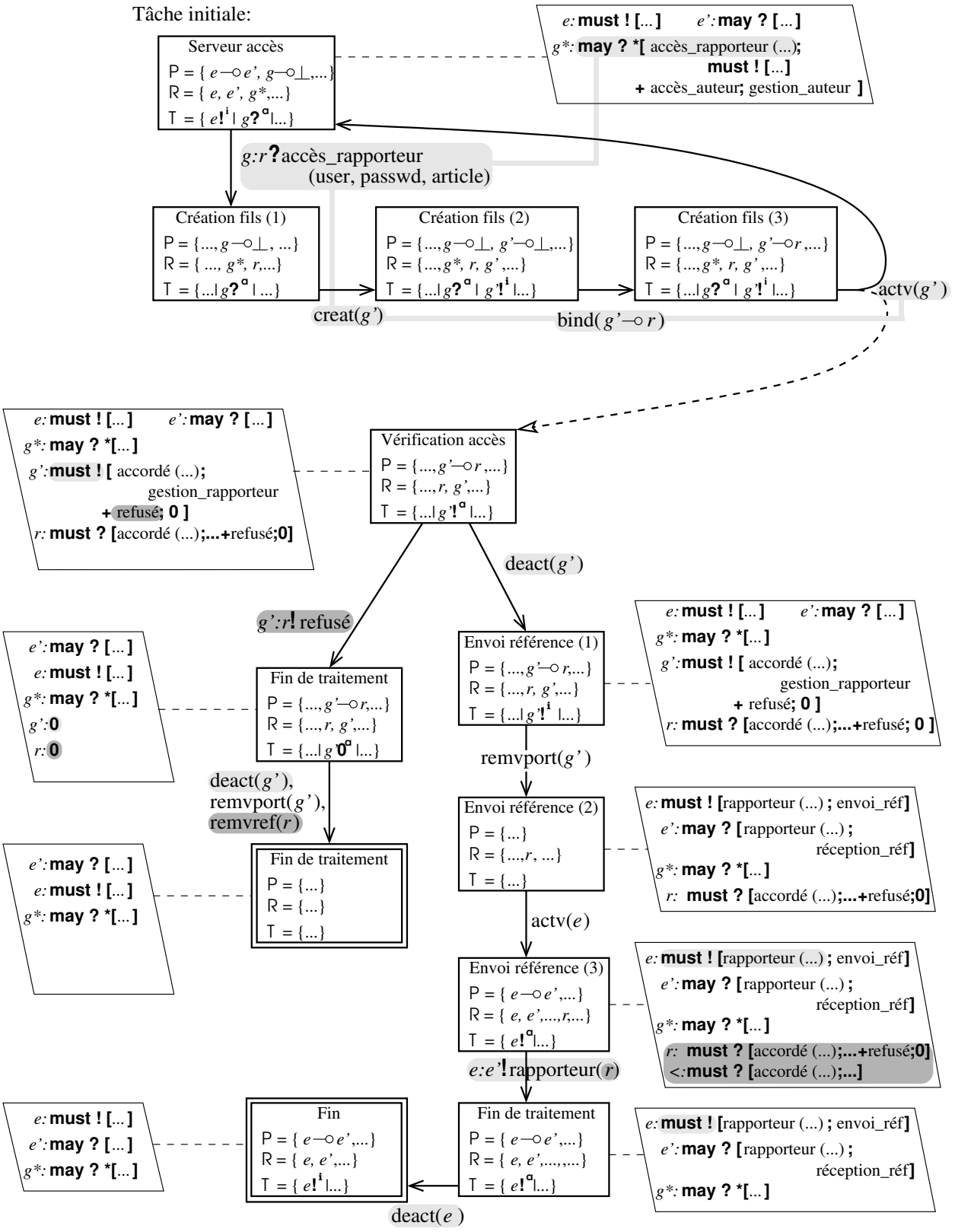



**FIG. 5.3:** *Vérification du composant* `Gestion`

### 5.4.3 Vérification du composant Article

La figure 5.4 page ci-contre donne la vérification du composant `Article`.

La référence $r$ est reçue avec le type *rapporteur_accepté*. Un port $a$ est créé, et son type rajouté au contrat. L'attachement $(a \multimap r)$ est valide car les types sont compatibles.

La suite du diagramme est plus complexe quant à la vérification du contrat. Sur la figure, l'état *Attente rapport* est associé à deux types de contrats :

- le premier correspond au cas où *le rapport n'est pas encore remis*. Ce contrat contient les types suivants :

  entrer_puis_changer_rapport =
  **must ! [** rapport (...) ;
  **must ? [** Ok ; changer_rapport
  + erreur_rapport(...) ; entrer_puis_changer_rapport **]**
  + demande_article ; **must ? [** article (...) ; entrer_puis_changer_rapport **] ]**

  gestion_rapporteur =
  **must ? [** rapport (strings) ;
  **must ! [** Ok ; gestion_rapporteur_chg
  + erreur_rapport(string) ; gestion_rapporteur **]**
  + demande_article ; **must ! [** article (strings) ; gestion_rapporteur **] ]**

  Ce contrat est associé à l'état *Attente rapport* tant que le message *Ok* n'a pas été émis au moins une fois.
- le deuxième correspond au cas où *le rapport a été remis, et peut être modifié*. Ce contrat contient les types suivants (la seule différence est le changement de modalités **must** en modalités **may**) :

  changer_rapport =
  **may ! [** rapport (...) ;
  **must ? [** Ok ; changer_rapport
  + erreur_rapport(...) ; changer_rapport **]**
  + demande_article ; **must ? [** article (...) ; changer_rapport **] ]**

  gestion_rapporteur_chg =
  **may ? [** rapport (strings) ;
  **must ! [** Ok ; gestion_rapporteur_chg
  + erreur_rapport(string) ; gestion_rapporteur_chg **]**
  + demande_article ; **must ! [** article (strings) ; gestion_rapporteur_chg **] ]**

  Ce contrat sera toujours associé à l'état *Attente rapport* dès que le message *Ok* est émis.

On remarquera à nouveau l'application du corollaire 2 page 85 : *changer_rapport* est bien super type de **0**, qui est le type réel du port $r$.

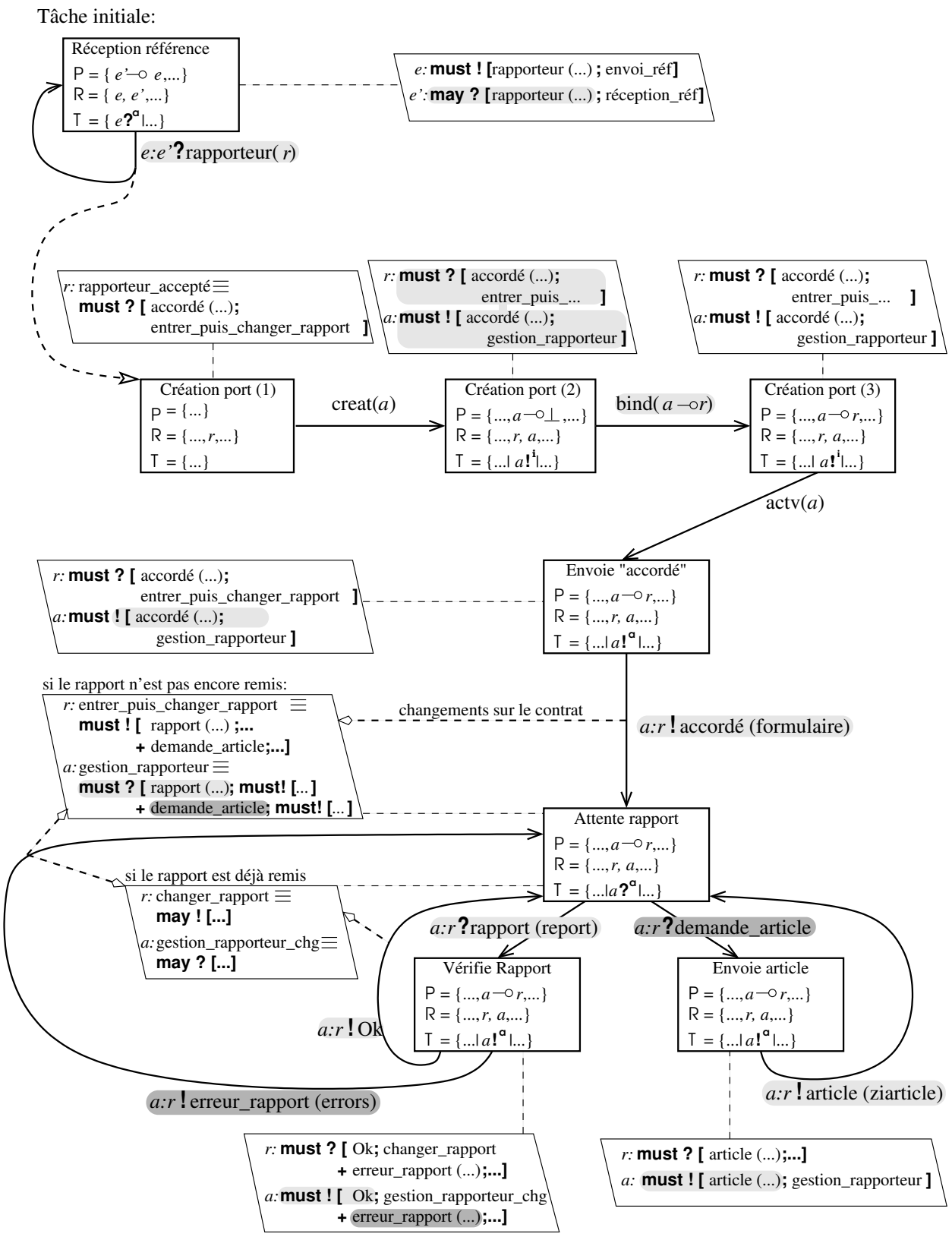



FIG. 5.4: *Vérification du composant* `Article`

## 5.5 Conclusion

Nous venons de présenter la relation conformité entre le composant et le contrat associant à chacun des ports du composant un type défini selon le chapitre 3 page 43.

Nous définissons un assemblage sain comme un ensemble de composants interconnectés tel que :

- les interfaces communicantes sont compatibles (au sens du chapitre 3) ;
- les seuls liens sont client/serveur et point-à-point ;
- chaque port a sa propre tâche d'exécution ;
- l'assemblage respecte la notion de référence privée (seuls les deux interlocuteurs connaissent la référence de leur partenaire).

Nous assurons alors qu'il n'y a pas d'interblocage externe entre composants, et que les messages envoyés sont compris et seront finalement consommés.

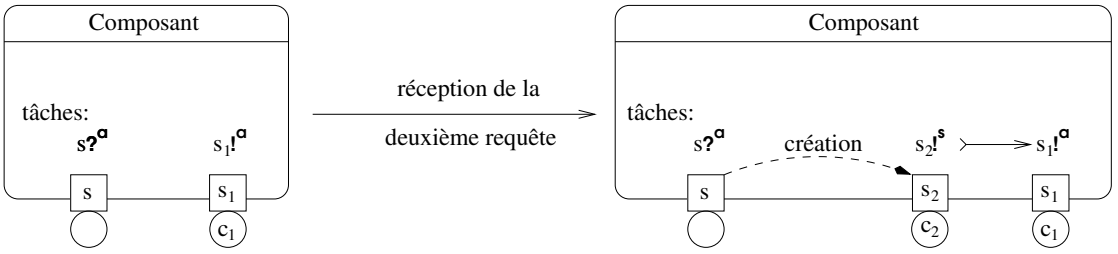
Il reste cependant un problème de vivacité, surtout concernant la sémantique du type **must?**, de par le fait que les références ayant un tel type peuvent circuler de composant en composant sans jamais être attachées à un port. Nous proposons d'avoir recours à deux formules de logique temporelle pour résoudre ce problème. Ces formules indiquent que, si le composant reçoit une référence typée **must?**, alors il finira par l'attacher à un port. La deuxième formule impose que, si la première action d'un port est typée **must!**, alors ce port finira par envoyer. Ces formules sont vérifiables par les techniques classiques de vérification de modèles. Cependant, les travaux futurs devront combler cette faiblesse de notre modèle.

Chapitre 6

# Intégration des serveurs non réentrants

Notre langage de type permet de spécifier les ports publics (ou serveurs) et des ports privés (clients, ou pairs). Les ports serveurs, dans notre système de type, sont réentrants : à chaque appel, un nouveau port est créé pour dialoguer avec le client, tandis que le port serveur est à nouveau actif pour répondre aux autres requêtes.

Certains types de serveurs non réentrants sont possibles, mais leur implantation n'est pas du tout intuitive. Cette implantation est présentée figure 6.1. La partie gauche de la figure montre la configuration du composant lorsque son port serveur $s$ est en train de répondre à un premier client $c_1$ : conformément à la spécification, le port $s_1$ a été créé pour envoyer la réponse. Lorsqu'un second client envoie une requête vers $s$, le composant crée le port $s_2$, et le suspend à $s_1$ ($s_1$ étant en émission, la suspension est valide). Le deuxième client n'accèdera donc au service de $s$ que lorsque la première requête sera traitée (soit $s_1$ envoie la réponse et disparaît). Cette implantation présente le double inconvénient de ne pas être intuitive, et d'être limitée : en effet, si le port $s_1$ est en réception, $s_2$ ne peut pas se suspendre sur $s_1$. Pour palier à ce problème, nous ajoutons la notion de serveur non réentrant. Cela signifie étendre la syntaxe des types, ainsi que le système de type associé.




Serveur non réentrant : $s$ = **may ? x[** requête (...) **;** **must ! [** réponse (...) **; 0 ] ]**

$s1$ = **must ! [** réponse (...) **; 0 ]**

$s2$ = **must ! [** réponse (...) **; 0 ]**

**Fig. 6.1:** *Implantation possible d'un serveur non réentrant*

Ce chapitre débute par une modification de la syntaxe du langage d'interface et une présentation informelle de l'intégration des serveurs non réentrants. Suivent deux sections concernant la formalisation et les propriétés résultantes. Un cas d'étude termine ce chapitre, sur le problème classique du dîner de philosophes.

## 6.1 Syntaxe et présentation informelle pour les serveurs non réentrants

### 6.1.1 Syntaxe

Tout comme le type serveur réentrant était estampillé «*****», nous choisissons d'estampiller les type serveur non réentrant par «**x**». La syntaxe de notre langage de type devient donc la suivante, où seul le non terminal *receive_server* est modifié :

$$
\begin{array}{rcl}
type & ::= & server_name = mod\ receive_server \\
 & | & peer_name = (mod\ send \mid mod\ receive) \\
receive_serveur & ::= & \mathbf{?}\ (\mathbf{*}|\mathbf{x})[\ \sum_i M_i\ ;\ I_i\ ] \\
receive & ::= & \mathbf{?}\ [\ \sum_i M_i\ ;\ I_i\ ] \\
send & ::= & \mathbf{!}\ [\ \sum_i M_i\ ;\ I_i\ ] \\
I & ::= & \mathbf{0} \mid peer_name \mid mod\ send \mid mod\ receive \\
mod & ::= & \mathbf{may} \mid \mathbf{must} \\
M & ::= & name[\ (\ \widetilde{args}\ )\ ] \\
args & ::= & basic_type \mid peer_name \mid server_name \\
basic_type & ::= & \mathbf{boolean} \mid \mathbf{integer} \mid \mathbf{real} \mid \mathbf{string}.
\end{array}
$$

Toutefois, deux contraintes doivent être appliquées aux serveurs non réentrants :

- toute réception, excepté la première, a la modalité **must**. En effet, comme le traitement doit finir pour servir d'autres clients, il faut forcer ces clients à envoyer les messages ;
- le traitement du service demandé par le client doit être fini, pour permettre au serveur de traiter les autres requêtes. Pour cela, nous imposons aux types estampillés «**x**» qu'il ne soient pas récursifs[1].

La conséquence du type non récursif est qu'il se termine nécessairement par **0**. Un serveur estampillé «**x**» a donc le comportement suivant :

1. attendre une requête d'un client ;
2. sur réception d'une requête, créer un port pour répondre au client. Le port du serveur se suspend ;

[1] cela peut se vérifier comme suit. Soit $D$ l'ensemble des équations définissant un type : $D = \{X_0 = T_0, \cdots, X_n = T_n\}$. Le type représenté par $D$ n'est pas récursif si (moyennant un réarrangement des indices) : $\forall k \in \{1..n\}, \forall i \in \{0..k-1\} : X_i \notin T_k$, soit $T_k$ n'a pas de référence vers l'un des types $(X_i)_{\{0..k-1\}}$ déjà définis. Le calcul de $D$ peut se faire à partir de la syntaxe des types.

3. le port serveur est suspendu jusqu'à ce que le port traitant avec le client se termine (son type est alors **0**). Le port serveur revient alors en 1.

La sémantique du serveur non réentrant, «le port serveur se suspend sur le port traitant la requête» présente un double avantage par rapport à la suivante : «le port serveur ne répond que aux requêtes du client courant». Tout d'abord, la sémantique choisie est celle qui provoque le moins de modification sur les règles existantes. Ensuite, elle permet au client d'envoyer la référence du port traitant avec lui vers un autre composant (le client peut donc changer au cours du traitement de la requête) ; il n'est pas possible d'avoir ce comportement avec la sémantique «le port serveur ne répond que aux requêtes du client courant».

Les relations de sous-typage et de compatibilité découlent trivialement pour cette nouvelle syntaxe, en étendant l'étoile «*****» à la possibilité d'avoir aussi l'estampille «**x**». Ainsi, dans les règles, «*****» est remplacée par «(*****|**x**)».

### 6.1.2 Graphe de dépendance des ports

Comme indiqué au début de cette section, la contrainte «un port ne peut se suspendre sur un port en émission» n'est pas viable dans le cas des serveurs non réentrants. Aussi, les propriétés découlant de cette contrainte (voir le chapitre 5 page 77 sur les contrats) ne sont plus valables. Notamment, un interblocage peut avoir lieu, comme dans l'exemple de la figure 5.1, section 5.3.3 page 86.

Pour assurer les mêmes propriétés d'absence d'interblocage externe, nous raffinons les interfaces du composant, en y ajoutant une abstraction des dépendances entre les différents ports, notée $\rightarrowtail^{\diamond}$ , qui indique les dépendances qui surviendront durant l'exécution du composant[2] :

$x \rightarrowtail^{\diamond} y$ Durant l'exécution du composant, le port $x$ dépendra du port $y$ (soit $x$ sera suspendu par $y$).

$x \leftrightarrow^{\diamond} y$ Durant l'exécution du composant, le port $x$ dépendra du port $y$, et inversement (dépendance réciproque).
La définition formelle donne : $x \leftrightarrow^{\diamond} y \Leftrightarrow (x \rightarrowtail^{\diamond} y) \wedge (y \rightarrowtail^{\diamond} x)$

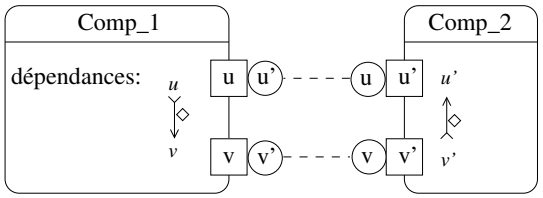
Nous obtenons donc un graphe orienté, que nous appelons graphe des dépendances. Nous imposons que ce graphe ne soit pas cyclique.

Pour illustrer l'utilité de ce graphe, reprenons la figure 5.1 sur un exemple d'interblocage (section 5.3.3 page 86). L'assemblage des deux composants mène à la configuration initiale de la figure 6.2 page suivante. Les liens d'assemblage représentent une dépendance externe potentielle entre deux ports ; cette dépendance n'est pas dirigée, car elle changera au cours du temps. L'assemblage des deux composants nous permet donc de déduire qu'un cycle de dépendance peut avoir lieu :

- $u$ peut dépendre de $v$ (relation $u \rightarrowtail^{\diamond} v$) ;
- $v$ peut dépendre de $v'$ (lien d'assemblage) ;
- $v'$ peut dépendre de $u'$ (relation $v' \rightarrowtail^{\diamond} u'$) ;
- $u'$ peut dépendre de $u$ (lien d'assemblage).

Notre proposition est d'interdire les assemblages où un tel cycle est réalisé. Bien entendu, comme les dépendances décrites au moment de l'assemblage représentent

[2] en fait, le symbole $\diamond$ est inspiré de la logique temporelle, et a la signification de «futur» (*eventually*).




| | Comp_1 : | Comp_2 : |
|---|---|---|
| Liens d'assemblage : | $(u \multimap u')$ | $(u' \multimap u)$ |
| | $(v \multimap v')$ | $(v' \multimap v)$ |
| Dépendances : | $u \overset{\diamond}{\rightarrowtail} v$ | $v' \overset{\diamond}{\rightarrowtail} u'$ |

**Fig. 6.2:** *Assemblage et graphes de dépendances : risque d'interblocage*

l'ensemble des dépendances futures, il se peut qu'un assemblage soit interdit alors qu'aucun cycle ne serait apparu durant l'exécution de cet assemblage. Cependant, notre but est de garantir des propriétés *dès l'assemblage* des composants, et ce en ayant le moins d'information possible sur le comportement des composants.

L'inconvénient de cette méthode est qu'il est nécessaire de connaître à l'assemblage les liens de communication entre les ports. La section suivante nous permet de prendre en compte la dynamique de ces liens.

### 6.1.3 Gestion des liens dynamiques

Les liens dynamiques entre ports sont des liens qui ne sont pas connus à l'assemblage, et qui apparaîtront en cours d'exécution. Les cas qui se présentent sont les suivants :

**création d'un lien entre deux ports**

Cette création a lieu lorsqu'une référence vers un port en réception vient d'être attachée à un port en émission (en tant que partenaire). Nécessairement, le partenaire doit être connu du composant : soit un autre port a reçu une référence vers ce partenaire, soit le partenaire était connu à l'assemblage ;

**suppression d'un lien**

Un lien est supprimé lorsque l'un des deux ports est supprimé ;

**modification d'un lien entre deux ports**

Un lien est modifié lorsque le récepteur change de partenaire. Cette modification correspond à la suppression du lien avec l'ancien partenaire, suivie, un peu plus tard, de la création du lien avec le nouveau partenaire.

A l'assemblage, les composants n'ont pas toutes les informations à leur disposition concernant les références partenaires. Il est donc impossible d'indiquer à l'initialisation les liens qui potentiellement seront créés. Cependant, à l'assemblage, chaque composant connaît l'ensemble de ses ports qui sont susceptibles d'être créés au cours de son exécution ; notamment les ports qui envoient et reçoivent des références vers d'autres ports.

Notre idée est d'inférer les relations entre ces ports. Nous pouvons ensuite les utiliser à l'assemblage pour détecter d'éventuels interblocages. Ces relations sont

notées $w \overset{\blacklozenge}{\leftrightarrow} v$, et sont interprétées de la façon suivante (on note $T_w$, $T_v$ les types respectifs de $w$ et $v$) :

**$w$ peut envoyer la référence $v$**

$T_w$ contient donc un message (en envoi) dont un argument est super-type de $T_v$ (on trouvera plus généralement la condition «est le type $T_v$»).

De plus, dans ce cas, soit $v$ est une référence serveur (et son type est le type initial du serveur, puisque les traitements sont effectués par d'autres ports), soit $v$ est une référence pair sans partenaire.

Par exemple, la figure 6.3 illustre la relation $w' \overset{\blacklozenge}{\leftrightarrow} v'$ (le port $w'$ enverra la référence $v'$).

**$w$ peut envoyer la référence partenaire de $v$**

Si $w$ envoie la référence partenaire de $v$, alors $T_w$ contient un message (en envoi) dont un argument est super-type du dual de $T_v$[3].

**$w$ peut recevoir le partenaire de $v$**

$T_w$ contient donc un message (en réception) dont un argument a un type qui est compatible avec $T_v$. Ce cas a lieu lorsque $v$ n'a pas de partenaire, et que sa première action est un envoi. Il est illustré par la figure 6.3, avec la relation $w \overset{\blacklozenge}{\leftrightarrow} v$.

**$w$ peut recevoir une référence qui sera envoyée par $v$**

$T_w$ contient donc un message (en réception) dont un argument est sous-type du type de la référence à envoyer.

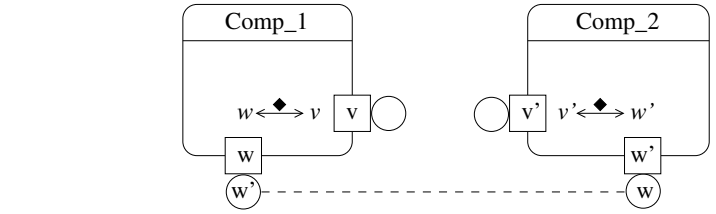



**Fig. 6.3:** *Relations de dépendances pour l'échange de référence*

On notera que ces relations peuvent être inférées uniquement à l'aide des types des ports du composant. Le comportement de celui-ci n'est pas mis en jeu.

### 6.1.4 Assemblage sain

En utilisant les deux sections précédentes, nous pouvons définir un assemblage sain en regard de l'extension que nous apportons.

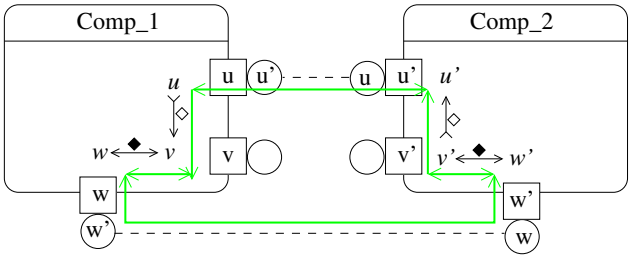
Les relations que nous venons de définir ( $\overset{\diamond}{\rightarrow}$ , $\overset{\diamond}{\leftrightarrow}$ et $\overset{\blacklozenge}{\leftrightarrow}$ ), et les attachements entre les ports, définissent, à l'assemblage, un graphe orienté. Les nœuds de ce graphe

[3] la formalisation prend en compte l'évolution des types, que nous ne détaillons pas ici pour plus de clarté.

sont les ports des composants, et les arêtes sont les relations entre ces ports[4]. Nous imposons que si un cycle est présent dans ce graphe, alors il ne contient aucun port représentant un serveur non réentrant.

Nous illustrons notre proposition par l'exemple suivant, où l'on a mis en commun les notions présentées dans les figures 6.2 page 104 et 6.3 page précédente. Nous avons deux composants, Comp_1 et Comp_2, avec les ports suivants :

| Comp_1 : | Comp_2 : |
|---|---|
| $u$ est un port non réentrant. | $u'$ est un port client de $u$. Le lien avec $u$ est connu à l'assemblage. |
| $v$ est un port dont $u$ va dépendre à un moment donné. La première action de $v$ est un envoi. A l'assemblage, $v$ ne connaît pas son partenaire. | $v'$ est le futur partenaire de $v$. La première action de $v'$ est une réception. |
| $w$ est un port qui doit recevoir $v'$, la référence partenaire de $v$. $w$ est lié à $w'$. | $w'$ est un port qui enverra la référence $v'$. $w'$ a comme partenaire le port $w$. |




$T_u = mod$ **?** **x[** $\ldots$ **]** non réentrant
$T_v = mod$ **!** **[** $\ldots$ **]**
$T_w = \ldots mod$ **?** **[** $M(T_{v'})$ **;** $\ldots$ **]** $\ldots$
$Comp(T_v, T_{v'})$

$T_{u'} = \ldots$ tel que $Comp(T_u, T_{u'})$
$T_{v'} = mod$ **?** **[** $\ldots$ **]**
$T_{w'} = \ldots mod$ **!** **[** $M(T_{v'})$ **;** $\ldots$ **]** $\ldots$

**Fig. 6.4:** *Assemblage de composant : cycle avec un port non réentrant*

Une exécution de cet assemblage pourrait être la suivante :
- Le port $u$ est en train de traiter avec un troisième composant. Aucun autre client ne peut donc accéder aux services de $u$ tant que ce traitement n'est pas terminé ;
- Pour effectuer ce traitement, le port $u$ a besoin du port $v$. Le composant récupère la référence $v'$ à l'aide du port $w$ ;
- Le port $v$ échange des messages avec son partenaire, $v'$ ;
- Pour traiter les messages de $v$, $v'$ a besoin d'utiliser les services du port $u$. Il se suspend donc à $u'$, le client créé pour l'occasion ;
- $u'$ envoie sa requête au port $u$. Cette requête ne peut être traitée, car $u$ n'est pas réentrant et est en relation avec un client.

[4] lorsque deux ports sont attachés, nous considérons que l'arête résultante est bidirectionnelle : elle symbolise le fait qu'en cours d'exécution, un des deux ports sera en réception et attendra le message de son partenaire.

Notre contrainte vise à éliminer un tel exemple d'interblocage. Les deux sections suivantes donnent la formalisation de ce qui vient d'être exposé, ainsi que la démonstration de la propriété d'absence d'interblocage.

## 6.2 Formalisation

L'ajout de serveurs non réentrant implique une modification dans les règles de la vérification de contrat. En effet, les règles de la sémantique de composant nous suffisent : la règle C-DEACT nous permettra de rendre le port serveur non réentrant dans l'état oisif. La modification des règles vise essentiellement à vérifier que toutes les dépendances potentielles entre les ports ($u\rho^\sigma \rightarrowtail v\rho^\sigma$) ont été déclarées au moyen de la relation $\overset{\diamond}{\rightarrowtail}$ .

### 6.2.1 Modification des règles de respect de contrat

Nous avons choisi d'intégrer les dépendances $\overset{\diamond}{\rightarrowtail}$ dans les contrats $\tilde{C}$. Ces dépendances sont statiques, aussi les règles les utilisant ne font que vérifier que la dépendance est présente : $u \overset{\diamond}{\rightarrowtail} v \in \tilde{C}$.

Les nouvelles règles (en gras) et les règles modifiées sont données dans le tableau 6.1 page suivante.

Dans les règles, nous notons $\mathit{instance}(s, u)$ le prédicat indiquant que le port $u$ a été créé pour répondre à un client du port serveur $s$ (le serveur $s$ peut être réentrant ou non). Nous noterons $\neg\mathit{instance}(s, u)$ pour indiquer que, $s$ (respectivement $u$) donné, il n'existe pas de $u$ (respectivement $s$) vérifiant $\mathit{instance}(s, u)$.

*Le reste de cette page est volontairement vide.*

Par souci de compréhension, nous notons :
$M_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}_k)\,;\,T_k], M'_\Sigma \triangleq [\Sigma_k M_k(\tilde{U}'_k)\,;\,T'_k] \quad m_k \triangleq M_k(\tilde{v}_k)$

$$\text{REMVPORT}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvport}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{remvport}(u)} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}\backslash u)}\ \neg\begin{pmatrix}\text{instance}(s,u)\\ \wedge\text{nonreentrant}(s)\end{pmatrix}$$

$$\textbf{REINIT}\ \frac{u{:}T \equiv \mathbf{0} \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\text{remvport}(u),\text{actv}(s)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\text{reinit}(s,u)} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C}\backslash u)}\ \text{instance}(s,u)$$

$$\textbf{RECVx}\ \frac{\begin{array}{c}u{:}T \equiv \mathit{mod}\ \mathsf{?x}M_\Sigma\\ B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:w\mathsf{?}m_k,\text{creat}(u'),\text{bind}(u'\multimap w),\text{actv}(u'),\text{deact}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')\end{array}}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{u'/u^{\mathsf{x}}:w\mathsf{?}m_k} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C} \Leftarrow u'{:}T_k, w{:}{T_k}^{\mathcal{D}}, \tilde{v}{:}\tilde{U}_k)}\ \blacktriangle$$

$$\textbf{SUSPEND}\ \frac{\text{parent}(u) \rightsquigarrow \text{parent}(v) \in \tilde{C} \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\alpha} B'(\mathsf{P},\mathsf{R},\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\alpha} (B'(\mathsf{P},\mathsf{R},\mathsf{T}'),\tilde{C})}\ \blacklozenge$$

$$\text{OTHER}\ \frac{B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\alpha} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \xrightarrow{\alpha} (B'(\mathsf{P}',\mathsf{R}',\mathsf{T}'),\tilde{C})}\ \begin{pmatrix}\alpha = \text{relax}(t_1 \not\rightsquigarrow t_2)\\ \vee\alpha \in \{\text{actv}(u),\text{deact}(u)\} \Rightarrow \neg\text{instance}(u,u')\end{pmatrix}$$

$$\blacktriangle \triangleq \mathit{len}(\tilde{v}') = \mathit{len}(\tilde{U}_k) \wedge \text{Must}(\mathsf{T}')$$

$$\text{Must}(\mathsf{T}') \triangleq \forall u \in \mathsf{T}', (u{:}\mathbf{must!}M_\Sigma) \Rightarrow \forall v, u \rightarrowtail^* v, \neg(v{:}\mathbf{may?}M_\Sigma)$$

$$\text{parent}(u) \triangleq \begin{cases} s \text{ si instance}(s,u)\\ u \text{ sinon}\end{cases}$$

$$\blacklozenge \triangleq \alpha = \text{actv}(u \rightarrowtail v)\ \vee\ \big(\alpha = \text{susp}(t_1 \rightarrowtail t_2)\ \wedge\ u = \text{head}(t_1)\ \wedge\ v = \text{last}(t_2)\big)$$

**Tab. 6.1:** *Règles de compatibilité composant-interface, avec les serveurs non réentrants*

**REMVPORT** n'est pas applicable sur un port $u$ effectuant un traitement non réentrant (*instance*$(s,u)$ et *nonreentrant*$(s)$ vrais).
**REINIT** réinitialise un serveur non réentrant : si le port serveur non réentrant $s$ a créé le port $u$ pour répondre à un client, la réinitialisation de $s$ ne peut avoir lieu que si le type de $u$ est $\mathbf{0}$. Dans ce cas, le port $u$ est supprimé, et $s$ est réactivé.
**RECVx** est utilisée pour les ports non réentrants, et lors de la réception du premier message. Elle fonctionne comme la règle RECV*, à la différence que le port serveur devient oisif.
**SUSPEND** vérifie que, lorsqu'un port se suspend à un autre, la dépendance créée est déclarée dans $\tilde{C}$. Si les ports en questions sont issus de ports serveurs (soit *instance*$(s,u)$) alors la dépendance déclarée prend en compte le port serveur correspondant (soit $s \rightsquigarrow v \in \tilde{C}$). Lorsqu'il s'agit d'une tâche $t_1$ qui se suspend à une tâche $t_2$, alors la dépendance «head$(t_1) \rightsquigarrow$ last$(t_2)$» doit être déclarée, avec $\mathit{last}(t) = \{u \in t \mid \forall v \in t, v \not\rightsquigarrow u\}$. $\mathit{last}(t)$ représente le port de la tâche $t$ qui dépend de tous les autres ports de cette tâche.

$$\textbf{REMVPORT-ERR}\ \frac{u{:}T \equiv mod\ \mathsf{?x}M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\mathrm{remvport}(u)} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}$$

$$\textbf{REINIT-ERR}\ \frac{u{:}T \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{\alpha} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \blacksquare$$

$$\text{SEND-ERR}\ \frac{u{:}T \equiv mod\ \rho\ [\mathsf{*}|\mathsf{x}]M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \xrightarrow{u:u'!m'} B'(\mathsf{P},\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \neg m'{:}M_\Sigma \vee \rho = \mathsf{?}$$

$$\textbf{RECVx-ERR}\ \frac{u{:}T \equiv mod\ \mathsf{?x}M_\Sigma \qquad B(\mathsf{P},\mathsf{R},\mathsf{T}) \not\xrightarrow{\alpha} B'(\mathsf{P}',\mathsf{R}',\mathsf{T}')}{(B(\mathsf{P},\mathsf{R},\mathsf{T}),\tilde{C}) \rightarrow \mathit{Error}}\ \blacktriangledown$$

$$\begin{aligned}
\blacksquare &\triangleq \mathrm{instance}(s,u) \wedge \big(T \equiv \mathsf{0} \Rightarrow \alpha \neq \mathrm{remvport}(u), \mathrm{actv}(s)\big) \\
&\qquad \wedge \big(T \not\equiv \mathsf{0} \Rightarrow (\alpha = \mathrm{actv}(s) \vee \alpha = \mathrm{remvport}(u))\big) \\
\neg m'{:}M_\Sigma &\triangleq m' = M'(\tilde{v}') \wedge \forall k, M' \neq M_k \vee \neg \tilde{v}_k{:}\tilde{T}'_k \\
\blacktriangledown &\triangleq \alpha = u : w\mathsf{?}m_k, \mathrm{creat}(u'), \mathrm{bind}(u' \multimap w), \mathrm{actv}(u'), \mathrm{deact}(u)
\end{aligned}$$

**Tab. 6.2:** *Règles de compatibilité composant-interface : cas d'erreurs, avec les serveurs non réentrants*

**OTHER** présente les autres actions permises. Mais un serveur non réentrant qui est en dialogue avec un client ne peut être activé ni désactivé de cette manière (s'il existe $u'$ tel que $\mathit{instance}(u,u')$).

Les règles d'erreur sont regroupées dans le tableau 6.2.
**REMVPORT-ERR** correspond à la suppression d'un port non réentrant se trouvant dans le cache des ports oisifs.
**REINIT-ERR** donne le cas d'erreur lorsque la réinitialisation du serveur non réentrant n'est pas correcte. Deux cas provoquent l'erreur :

- lorsque le traitement avec le client est terminé, le port $u$ n'est pas supprimé, et/ou le port serveur n'est pas réactivé ;
- le port effectuant un traitement non réentrant est supprimé avant la terminaison du traitement, ou lorsque le port serveur correspondant n'est pas réactivé ;
- le serveur est réactivé alors que le traitement avec le client n'est pas terminé.

**SEND-ERR** a été modifiée pour prendre en compte l'estampille «**x**».
**RECVx-ERR** est le cas où un port serveur non réentrant ne se comporte pas comme prévu.

### 6.2.2 Gestion des liens dynamiques

Cette section formalise la relation $\clubsuit\!\!\leftrightarrow$ .

Nous définissons pour cela les ensembles suivants :

$\mathcal{E}(T)$ est l'ensemble des types que $T$ est capable d'envoyer.

$$\mathcal{E}(T) \triangleq \begin{cases} \varnothing & \text{si } T \equiv \mathbf{0} \\ \bigcup_i \mathcal{E}(T_i) & \text{si } T \equiv \mathit{mod}\mathbf{?}\ [\sum_i M_i(\widetilde{\mathit{args}_i})\ ;T_i] \\ \left(\bigcup_i \mathcal{E}(T_i)\right) \cup \left(\bigcup_i \widetilde{\mathit{args}_i} \cap \{\neg \mathit{basic_type}\}\right) & \text{si } T \equiv \mathit{mod}\mathbf{!}\ [\sum_i M_i(\widetilde{\mathit{args}_i})\ ;T_i] \end{cases}$$

$\mathcal{R}(T)$ est l'ensemble des types que $T$ est capable de recevoir.

$$\mathcal{R}(T) \triangleq \begin{cases} \varnothing & \text{si } T \equiv \mathbf{0} \\ \bigcup_i \mathcal{R}(T_i) & \text{si } T \equiv \mathit{mod}\mathbf{!}\ [\sum_i M_i(\widetilde{\mathit{args}_i})\ ;T_i] \\ \left(\bigcup_i \mathcal{R}(T_i)\right) \cup \left(\bigcup_i \widetilde{\mathit{args}_i} \cap \{\neg \mathit{basic_type}\}\right) & \text{si } T \equiv \mathit{mod}\mathbf{?}\ [\sum_i M_i(\widetilde{\mathit{args}_i})\ ;T_i] \end{cases}$$

$\mathcal{S}(T)$ est l'ensemble des types nommés vers lesquels $T$ est susceptible d'évoluer. Par type nommé, nous entendons un type possédant un nom (*peer_name* ou *server_name*), c'est-à-dire déclaré avec la syntaxe «*nom* = ...».

$$\mathcal{S}(T) \triangleq \begin{cases} \varnothing & \text{si } T \equiv \mathbf{0} \\ T & \text{si } T \in \{\mathit{peer_name}, \mathit{server_name}\} \\ \bigcup_i \mathcal{S}(T_i) & \text{si } T \equiv \mathit{mod}[\mathbf{?}|\mathbf{!}]\ [\sum_i M_i(\widetilde{\mathit{args}_i})\ ;T_i] \end{cases}$$

La relation $u \clubsuit\!\!\leftrightarrow v$ est définie si l'une des quatre possibilités est vraie (en notant $T_u$ et $T_v$ les types respectifs de $u$ et $v$) :

1. $\exists T \in \mathcal{E}(T_u)$ tel que $T_v \preceq T$.
   C'est-à-dire que $u$ peut envoyer la référence $v$. Ce cas n'a lieu que lorsque la référence $v$ n'a pas de partenaire, et que sa première action est une réception ;
2. $\exists T \in \mathcal{E}(T_u), T'_v \in \mathcal{S}(T)$ tel que ${T'_v}^{\mathcal{D}} \preceq T$
   C'est-à-dire que $u$ peut envoyer la référence partenaire de $v$ ;
3. $\exists T \in \mathcal{R}(T_u)$ tel que $\mathit{Comp}(T, T_v)$
   C'est-à-dire que $u$ peut recevoir le partenaire de $v$. Ce cas a lieu lorsque $v$ n'a pas de partenaire, et que sa première action est un envoi ;
4. $\exists T'_u \in \mathcal{R}(T_u), T'_v \in \mathcal{E}(T_v)$ tel que $T'_u \preceq T'_v$
   C'est-à-dire que $u$ peut recevoir une référence qui sera envoyée par $v$ ;
5. $\exists T'_v \in \mathcal{S}(T_v)$ tel que $T_u \preceq T'_v$
   C'est un cas assez rare où le port $v$ est supprimé, et son partenaire est attaché au port $u$. Le partenaire est donc lié au même composant, mais à un port différent.

Les cinq possibilités de relations $u \clubsuit\!\!\leftrightarrow v$ ont chacune une complexité au pire quadratique en nombre d'état nommé des interfaces. Trouver toutes les relations possibles dans le composant est une complexité factorielle en nombre de ports. Nous pensons toutefois que notre niveau d'abstraction du composant est suffisamment élevé pour avoir un nombre de port et un nombre d'état par interface raisonnable en regard de cette complexité.

Cependant, si cette complexité se révélait trop importante, il faudrait que les relations $\overset{\clubsuit}{\leftrightarrow}$ soient produites par l'implanteur du composant, et vérifiées par la suite. Les propriétés démontrées plus loin ne seraient pas changées.

### 6.2.3 Assemblage sain avec ports non réentrants

Un assemblage sain qui contient des ports serveurs non réentrants est un assemblage où

- chaque composant satisfait les contrats de ses interfaces ;
- les ports reliés entre eux ont des interfaces compatibles entre elles (soit : un port est compatible avec son partenaire, et le type de ce partenaire est super-type du port le représentant) ;
- le graphe global des dépendances ( $\overset{\diamond}{\rightarrowtail}$ , $\overset{\clubsuit}{\leftrightarrow}$ et ports attachés entre eux) n'a pas de cycle qui contient à la fois un port serveur non réentrant et un de ses clients (la formalisation est cependant plus précise quant aux propriétés du cycle).

La définition de l'assemblage sain de la section 1 page 83 est en fait étendue au dernier point.

La définition suivante nous donne l'ensemble des ports non réentrants à risque, c'est-à-dire pour lesquels un interblocage peut avoir lieu. Elle formalise le cycle que l'on veut interdire, et représenté figure 6.5 page suivante.

**Définition 4 (Port non réentrant $u$ à risque)**
$P_{\text{risque}}(u) \triangleq \exists (u_i)_{1..n}$ *avec* $u = u_1$ *et telle que (les calculs sur les indices se font modulo n) :*

$$\begin{aligned} \exists k : \quad & u \xrightarrow{\mathcal{A}} {}^{*} u_{k-1} \overset{\diamond}{\rightarrowtail} u_k \\ & \wedge \forall i \in [k..n] : u_i \overset{\clubsuit}{\leftrightarrow} u_{i+1} \ \vee \ \text{partenaire}(u_i, u_{i+1}) \\ & \wedge \mathit{Comp}(T_u, T_{u_k}) \quad \textit{si } u : T_u \textit{ et } u_k : T_{u_k} \end{aligned}$$

*Avec* $\xrightarrow{\mathcal{A}}{}^{*}$ *la clôture transitive de* $\xrightarrow{\mathcal{A}}$ *et :*

$$\begin{aligned} \text{partenaire}(u, v) &\triangleq (u \multimap v) \ \vee \ (v \multimap u) \\ u \xrightarrow{\mathcal{A}} v &\triangleq u \overset{\diamond}{\rightarrowtail} v \ \vee \ u \overset{\clubsuit}{\leftrightarrow} v \ \vee \ \text{partenaire}(u, v) \end{aligned}$$

L'algorithme résultant de cette définition est assez complexe. Son implantation visera à trouver la suite $(u_i)$ dans l'ordre des conditions données. A savoir :

1. Partir d'un serveur non réentrant $s$ ;
2. Parcourir le graphe de dépendance des relations $\xrightarrow{\mathcal{A}}$ depuis $s$, jusqu'à trouver $c$ (le client potentiel) tel que $\cdot \overset{\diamond}{\rightarrowtail} c$ ;
3. Si $(c \multimap s)$, alors $P_{\text{risque}}(s)$ est vraie. Dans le cas où tous les liens client/serveur non réentrant sont statiques, l'algorithme termine ici ;
4. Sinon, parcourir le graphe de dépendance pour trouver les $(u_i)_{k..n}$ tels que $c = u_k, s = u_n$ et $u_i \overset{\clubsuit}{\leftrightarrow} u_{i+1} \vee \text{partenaire}(u_i, u_{i+1})$ ;
5. Si la suite $(u_i)_{k..n}$ est trouvée, vérifier la compatibilité des types de $s$ et $c$.

On préférera donc limiter au maximum le nombre de liens dynamiques client/serveur non réentrant, quitte à interdire de tels liens si l'algorithme s'avère trop coûteux. Nous préférons cependant avoir en théorie le moins de contraintes possibles sur l'assemblage, et éventuellement en ajouter dans la pratique.

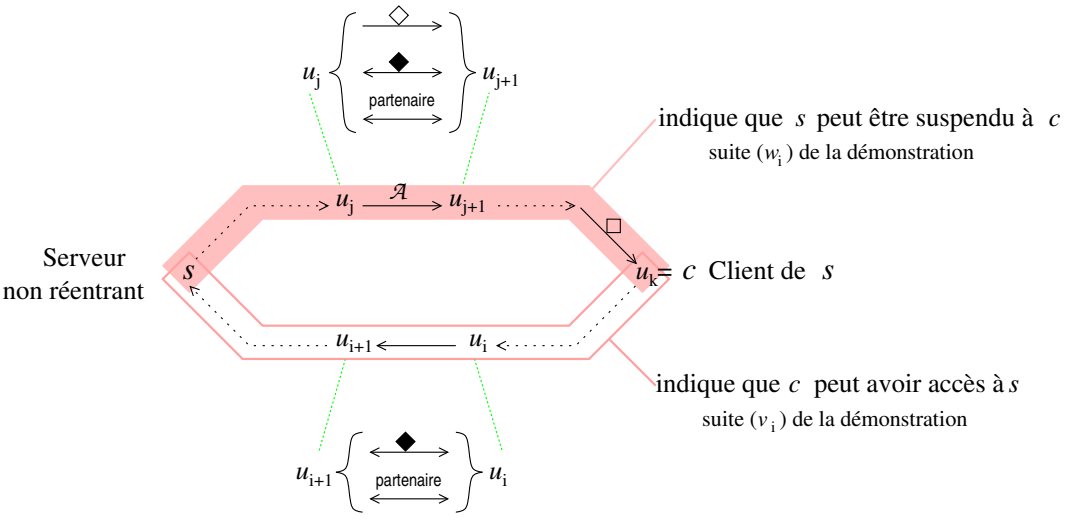



**FIG. 6.5:** *Port s à risque : s peut être suspendu au client c qui peut requérir les services de s*

Un assemblage contenant des ports serveurs non réentrants est sain (noté $\models\!\!\!\asymp_{\text{nr}}$) selon la définition suivante :

**Définition 5 (Assemblage sain avec ports non réentrants)**
$\models\!\!\!\asymp_{\text{nr}} \mathcal{A}$ *ssi* :

$$\models\!\!\!\asymp \mathcal{A} \text{ et } \forall u, \text{nonrentrant}(u) : \quad \neg P_{\text{risque}}(u)$$

## 6.3 Propriété : absence d'interblocage externe

Pour simplifier la formalisation du problème, nous étendons les relations de dépendance $\rightarrowtail$ et $\dashrightarrow$ aux ports serveurs non réentrants (les relations étendues de cette manière portent l'indice «nr») :

- $s?^{\text{i}} \rightarrowtail_{\text{nr}} u\rho^{\sigma}$ indique que le serveur non réentrant $s$ est suspendu au port $u$. Dans cette relation, le port $u$ est en fait le port créé pour traiter la requête du client. Nous avons : $s?^{\text{i}} \rightarrowtail u\rho^{\sigma}$ ssi instance$(s, u)$.
- $u \dashrightarrow_{\text{nr}} s$ indique que le port $u$ requiert les services du port non réentrant $s$, alors que celui-ci est oisif (en cours de traitement avec un autre client). Elle est formalisée[5] :
  $u!^{\sigma} \wedge (u \multimap s) \wedge \text{nonreentrant}(s) \wedge s^{\text{i}}$

Nous définissons ensuite $u\mathcal{S}_{\text{nr}}v$ qui est vraie lorsque $u$ est suspendu à $v$ :

**Définition 6 ($\mathcal{S}_{\text{nr}}$)**
$u\mathcal{S}_{\text{nr}}v \triangleq u\mathcal{S}v \vee u \rightarrowtail_{\text{nr}} v \vee u \dashrightarrow_{\text{nr}} v$
$u\mathcal{S}^{*}v \triangleq$ *fermeture transitive de* $\mathcal{S}$.

[5] $u$ est en émission car c'est un nouveau client de $s$.

Enfin, la définition de l'interblocage externe donne, avec les serveurs non réentrants :

$$\begin{aligned}\text{Ext_deadlock}_{\text{nr}}(\mathcal{C}) &\triangleq \exists u \in \mathcal{C} \text{ tel que nonreentrant}(u) \ \wedge \ u\mathcal{S}_{\text{nr}}^{*}u \\ &\triangleq \exists (u_i)_{1..n} \in \mathcal{C} \text{ tel que nonrentrant}(u_1) \ \wedge \ \forall k < n : u_i \mathcal{S}_{\text{nr}} u_{i+1} \wedge u_n \mathcal{S}_{\text{nr}} u_1\end{aligned}$$

Un assemblage sain, avec ports non réentrants, vérifie la propriété d'absence d'interblocage externe :

**Théorème 3 (Absence d'interblocage externe, ports non réentrants)**
*Si* $\models_{\text{nr}} \mathcal{A}$, *alors* $\mathcal{A} \vDash P_{\text{edf}} \ \wedge \ P_{\text{edf}_{\text{nr}}}$
*avec*

$$\begin{aligned}P_{\text{edf}} &\triangleq \forall \mathcal{C}, \mathcal{A} \rightarrow^{*} \mathcal{C} \Rightarrow \neg \text{Ext_deadlock}(\mathcal{C}) \\ P_{\text{edf}_{\text{nr}}} &\triangleq \forall \mathcal{C}, \mathcal{A} \rightarrow^{*} \mathcal{C} \Rightarrow \neg \text{Ext_deadlock}_{\text{nr}}(\mathcal{C})\end{aligned}$$

La propriété $P_{\text{edf}}$ est l'absence d'interblocage externe selon la section 5.3.3 page 86. La propriété $P_{\text{edf}_{\text{nr}}}$ est l'absence d'un interblocage externe mettant en jeu un port serveur non réentrant.

### 6.3.1 Démonstrations intermédiaires

Nous avons besoin de démontrer le lemme suivant, pour alléger la démonstration. Il montre qu'une relation de dépendance est déclarée indirectement lors de l'assemblage.

**Lemme 5**
*Si* $u \rightarrowtail v$ *ou* $u \rightarrowtail_{\text{nr}} v$ *alors, à l'assemblage :* $\textit{parent}(u) \overset{\diamond}{\rightarrowtail} \textit{parent}(v)$
*Si* $u \dashrightarrow v$ *ou* $u \dashrightarrow_{\text{nr}} v$ *alors, à l'assemblage,* $\exists (u_i)_{1..n}$ *tel que :*
$u_1 = \textit{parent}(u) \ \wedge \ u_n = \textit{parent}(v)$ *et* $\forall i : u_i \overset{\clubsuit}{\leftrightarrow} u_{i+1} \ \vee \ (u_i \multimap u_{i+1}) \ \vee \ (u_{i+1} \multimap u_i)$

**Preuve.** La première ligne du lemme est évidente, puisque vérifiée par la règle SUSPEND.

Pour démontrer la deuxième ligne, il suffit de montrer par induction structurelle que, pour toute référence $u$ vérifiant l'une des conditions :

1. $v : M(\ldots, u, \ldots) \in Com.w$ ;
2. $\exists B(\mathsf{P}, \mathsf{R}, \mathsf{T})$ tel que $u \in \mathsf{R}$.

Alors, à l'assemblage : $\exists (u_i)_{1..n}$ avec $u_1 = \text{parent}(u)$ et

1. Pour la condition **1** : $u_n = \text{parent}(v)$ ;
2. Pour la condition **2** : $u_n \in \mathsf{P}$ et $u_n$ peut soit recevoir la référence $u$, soit être partenaire de $u$.

tel que : $\forall i < n : u_i \overset{\clubsuit}{\leftrightarrow} u_{i+1} \ \vee \ (u_i \multimap u_{i+1}) \ \vee \ (u_{i+1} \multimap u_i)$

Les seules règles qui posent problèmes sont celles qui concernent les envois et les réceptions de message.

**1.** SEND

Soit $u$ une référence que le port $v$ envoie. Nous appelons $B(\mathsf{P},\mathsf{R},\mathsf{T})$ le composant contenant le port $v$. Nous avions, avant l'application de la règle, et pour le port $u$, le cas numéro **2** de notre propriété :
$\exists(u_i)_{1..n}$ avec $u_1 = u$ et $u_n \in \mathsf{P}$ tel que : $\forall i : u_i \overset{\clubsuit}{\leftrightarrow} u_{i+1} \vee (u_i \multimap u_{i+1}) \vee (u_{i+1} \multimap u_i)$

$u_n$ est soit le partenaire de $u$, soit peut recevoir $u$. Nous avions donc, à l'assemblage, la relation $u_n \overset{\clubsuit}{\leftrightarrow} v$ (il s'agit des cas **2** – $v$ envoie le partenaire de $u_n$ – et **4** – $v$ envoie une référence reçue par $u_n$ – de la définition de $\overset{\clubsuit}{\leftrightarrow}$ ). Après l'envoi du message, le cas numéro **1** de la propriété recherchée est donc vrai : nous prenons la suite $(u_i)_{1..n+1}$ avec $u_{n+1} = v$.

**2.** RECV, RECVx, RECV-UN, RECV*

Soit $v$ la référence effectuant la réception du message, et $B(\mathsf{P},\mathsf{R},\mathsf{T})$ le composant contenant le port $v$. Deux cas se présentent :

**$v$ reçoit la référence $u$**

Soit $w$ la référence ayant envoyé le message en question ($w : M(..,u,..) \in Com.v$), et $B_w(\mathsf{P}_w,\mathsf{R}_w,\mathsf{T}_w)$ tel que $w \in \mathsf{P}_w$.

Nous avons donc, pour la référence $u$, le cas numéro **1** de la propriété : soit $(u_i)_{1..n}$ la suite de référence vérifiant cette propriété. Notamment, $u_1 = u$ et $u_n = w$.

Le cas numéro **2** de la propriété est valable aussi pour la référence $v$ ($v \in \mathsf{R}_w$), nous avons alors $(u'_i)_{1..n}$ avec $u'_1 = v$ et $u'_n \in \mathsf{P}_w$. Nous pouvons trouver une suite telle que $u'_n = w$. En effet, si $u'_n \neq w$ alors nous pouvons choisir $u'_{n+1} = w$ car nous avons par induction :

- soit $u'_n$ est la référence qui peut recevoir $v$. Dans ce cas, nous avons $u'_n \overset{\clubsuit}{\leftrightarrow} w$ (cas **3** de la définition de $\overset{\clubsuit}{\leftrightarrow}$ : $u'_n$ reçoit le partenaire de $w$) ;
- soit $u'_n$ peut être partenaire de $v$. Dans ce cas, nous avons aussi la relation $u'_n \overset{\clubsuit}{\leftrightarrow} w$ (cas **5** où $v$ change de partenaire, mais pas de composant).

En résumé, nous avions, avant la réception de la référence $u$, l'existence de la suite : $u = u_1,\ldots,u_n = w = u'_n,u'_{n-1},\ldots,u'_1 = v$. Dès que $v$ a reçu la référence $u$, nous avons $u \in \mathsf{R}$, et le cas **2** de la propriété est vérifié avec cette suite.

**$v$ reçoit un message de la part de $u$**

Soit $B_u(\mathsf{P}_u,\mathsf{R}_u,\mathsf{T}_u)$ tel que $u \in \mathsf{P}_u$.

La propriété est vérifiée si $u \in \mathsf{R}$ avant la réception (pas de changements).

Dans le cas contraire, alors nous avions, pour la référence $v \in \mathsf{R}_u$, le cas **2** de la propriété. Soit $(u_i)_{1..n}$ la suite vérifiant la propriété, et telle que $u_1 = v$. Nous pouvons trouver la suite $(u_i)_{1..n}$ telle que $u_n = u$ (cas similaire au précédent : soit $u_n$ a reçu la référence $v$, soit $u_n$ était partenaire de $v$. Dans les deux cas $u_n \overset{\clubsuit}{\leftrightarrow} u$ est vérifiée).

Après la réception, nous avons $u \in \mathsf{R}$, et une suite $u = u_n,\ldots,u_1 = v$ (soit $(u_i)_{1..n}$ renversée) qui vérifie la propriété.

**3.** Conclusion

Soit $u \dashrightarrow v$, ou $u \dashrightarrow_{\mathrm{nr}} v$. Nécessairement, l'un des deux ports ($u$ ou $v$) est attaché à l'autre port.

Choisissons le cas ($u \multimap v$) (l'autre cas est semblable). Soit $B_u(\mathsf{P}_u, \mathsf{R}_u, \mathsf{T}_u)$ tel que $u \in \mathsf{P}_u$. Nous avons $v \in \mathsf{R}_u$, de part l'attachement des deux ports. La propriété donnée en début de démonstration nous permet d'exhiber la suite $(u_i)_{1..n}$ telle que :

$$\exists (u_i)_{1..n} \text{ avec } u_1 = \mathrm{parent}(v) \text{ et } u_n \in \mathsf{P}_u \text{ telle que :}$$
$$\forall i < n : u_i \overset{\spadesuit}{\leftrightarrow} u_{i+1} \ \vee \ (u_i \multimap u_{i+1}) \ \vee \ (u_{i+1} \multimap u_i)$$

Dans le cas où $u_n \neq u$, nous avons deux cas possibles :
- soit $u_n$ peut recevoir $v$. Dans ce cas nous avons $u_n \overset{\spadesuit}{\leftrightarrow} u$ ($u_n$ reçoit le partenaire de $u$) ;
- soit $u_n$ est partenaire de $v$, nous avons toujours $u_n \overset{\spadesuit}{\leftrightarrow} u$ (cas **5** de la définition de $\overset{\spadesuit}{\leftrightarrow}$ ).

Nous avons donc bien l'existence d'une suite $(u_i)_{1..n}$ respectant les conclusions. □

### 6.3.2 Démonstration d'absence d'interblocage externe

Nous devons montrer qu'il n'y a pas d'interblocage avec des ports non réentrants, dont nous rappelons la définition :

$$\mathrm{Ext_deadlock}_{\mathrm{nr}}(\mathcal{C}) \triangleq \exists (u_i)_{1..n} \in \mathcal{C} \text{ tel que } \mathrm{nonrentrant}(u_1) \ \wedge \ \forall k < n : u_i \mathcal{S}_{\mathrm{nr}} u_{i+1} \wedge u_n \mathcal{S}_{\mathrm{nr}} u_1$$

**Preuve (Absence d'interblocage externe avec ports non réentrants).**

La démonstration se fait par induction structurelle sur les règles étendues du respect de contrat. Nous ne démontrerons pas la propriété d'absence d'interblocage externe que du point de vue des ports pairs ou réentrants ($P_{\mathrm{edf}}$). En effet, comme $\models \mathcal{A}$, la propriété $P_{\mathrm{edf}}$, qui n'est pas concernée par l'extension de ce chapitre, est vérifiée par le théorème 2 page 87.

**1.** CREAT, REMVPORT, REMVREF, DEACT, OTHER

Ces règles n'ajoutent pas de relation de dépendance.

**2.** BIND

Il y a éventuellement création d'une relation de dépendance. Deux cas se présentent suivant la nature de $v$ (le port en réception, qui est attaché en tant que partenaire) :

**$v$ est un port serveur non réentrant**

Si $v$ est actif, alors il n'y a pas de création de dépendance. (il y aura dépendance lors de l'application de la règle RECVx – voir plus loin dans la démonstration).

Si $v$ est oisif, il y a création de la relation de dépendance $u \dashrightarrow_{\mathrm{nr}} v$. Raisonnons par l'absurde et supposons :

$$\exists (u_i)_{1..n} \text{ tel que } \forall k \leqslant n : u_k \mathcal{S}_{\mathrm{nr}} u_{k+1} \text{ et } \exists l : \mathit{nonreentrant}(u_l)$$

En particulier, $u$ et $v$ appartiennent à ce cycle. Nous choisissons $u_1 = v$ et $u_n = u$. Nous avons bien $u\mathcal{S}_{\text{nr}}v$.
Avant l'attachement, nous avions donc la même suite $(u_i)_{1..n}$, mais telle que $\neg(u_n\mathcal{S}_{\text{nr}}u_1)$.
Nous allons démontrer que (voir figure 6.5 page 112 pour avoir une représentation des suites) :

1. à l'assemblage, il existait une suite $(v_i)_{1..m}$ indiquant que $u$ et $v$ seraient partenaire ;
2. à l'assemblage, il existait une suite $(w_i)_{1..l}$ indiquant que la suite $(u_i)_{1..n}$ se produirait ;
3. les deux suites sont telles que $P_{\text{risque}}(v)$ est vraie à l'assemblage.

Ces trois points se démontrent de la manière suivante :

1. (suite $(v_i)_{1..m}$)
   Il s'agit d'une application du lemme 5 page 113. Nous avons $v_1 = \text{parent}(u)$ et $v_m = \text{parent}(v) = v$. Cette suite a de plus les caractéristiques suivantes :

   $$\forall i, v_i \overset{\clubsuit}{\leftrightarrow} v_{i+1} \ \vee \ \text{partenaire}(v_i, v_{i+1})$$

2. (suite $(w_i)_{1..l}$)
   Cette suite est issue de la suite $(u_i)_{1..n}$, où les dépendances externes $\dashrightarrow$ et $\dashrightarrow_{\text{nr}}$ sont remplacées celles du lemme 5. Elle se construit donc de la manière suivante :
   - $w_1 = u_1 = v$ ;
   - si $w_i = \text{parent}(u_k)$ et $u_k \rightarrowtail u_{k+1}$ alors $w_{i+1} = u_{k+1}$. Nous avons bien sûr $w_i \overset{\diamondsuit}{\rightarrowtail} w_{i+1}$ déclaré à l'assemblage. Si $u_k \rightarrowtail_{\text{nr}} u_{k+1}$ alors $u_k = w_i = \text{parent}(u_{k+1})$ ;
   - si $w_i = \text{parent}(u_k)$ et $u_k \dashrightarrow u_{k+1}$ alors : d'après le lemme 5, $\exists (u'_j)_{1..p}$ tel que $u'_1 = \text{parent}(u_k)$, $u'_p = \text{parent}(u_{k+1})$ et $u'_j \overset{\clubsuit}{\leftrightarrow} u'_{j+1} \ \vee \ (u'_j \multimap u'_{j+1}) \ \vee \ (u'_{j+1} \multimap u'_j)$. Nous prenons alors $w_{i+q}$ tel que $\forall q, 1 \leqslant q < p$ : $w_{i+q} = u'_{q+1}$. On remarquera que $w_{i+p-1} = u'_p = u_{k+1}$.

   La suite ainsi construite se termine par $\text{parent}(u_n)$ : $w_l = \text{parent}(u_n) = \text{parent}(u)$.
   De plus, la dernière dépendance de la suite $(w_i)$ est $\overset{\diamondsuit}{\rightarrowtail}$. Ceci se démontre comme suit :
   Nous nous intéressons à la suite $(u_i)$, notamment à $u_n = u$, le futur client du serveur $v$ ; on remarquera que la première action de $u$ est un envoi : $u!^{\sigma}$. Nous avons $u_{n-1}\mathcal{S}_{\text{nr}}u_n$, ce qui se traduit par l'une des dépendances :

   $$\begin{array}{rl} u_{n-1} \rightarrowtail u_n & \\ u_{n-1} \dashrightarrow u_n & \text{qui implique, sachant } u_n!^{\sigma} : \quad (u_n \multimap u_{n-1}) \\ u_{n-1} \rightarrowtail_{\text{nr}} u_n & \text{qui implique : } \quad \text{instance}(u_{n-1}, u_n) \text{ donc } \neg(u_n \multimap \bot) \\ u_{n-1} \dashrightarrow_{\text{nr}} u_n & \text{qui implique, entre autres : } \quad \text{nonreentrant}(u_n) \end{array}$$

   La seule relation de dépendance possible avant l'application de la règle BIND est donc $u_{n-1} \rightarrowtail u_n$, donc $w_{l-1} \overset{\diamondsuit}{\rightarrowtail} w_l$.

3. les deux suites $(v_i)$ et $(w_i)$ forment un cycle qui est le suivant :

$$v = w_1 \xrightarrow{\mathcal{A}} {}^*w_{l-1} \overset{\diamond}{\rightsquigarrow} w_l = \text{parent}(u) = \underbrace{v_1, \ldots, v_m}_{(v_i \overset{\blacklozenge}{\leftrightarrow} v_{i+1} \ \vee \ \text{partenaire}(v_i, v_{i+1}))} = v$$

Soit $P_{\text{risque}}(v)$ vraie, ce qui est contradictoire avec la définition de l'assemblage sain.

**$v$ est un port pair**

Il y a création de la relation de dépendance $v\mathsf{?}^{\mathsf{a}} \dashrightarrow u!^{\sigma}$. Tout comme précédemment, nous raisonnons par l'absurde. Soit

$$(u_i)_{1..n} \text{ tel que } \forall i \leqslant n : u_i \mathcal{S}_{\text{nr}} u_{i+1} \text{ et } \exists k : \textit{nonreentrant}(u_k)$$

Nous choisissons la suite de telle sorte que $u_1 = v$ et $u_n = u$. Nous prenons aussi $k$ tel que $\textit{nonrentrant}(u_k)$. $k > 1$ car $v$ est un port pair.

Nous avons $u_{k-1}$ client de $u_k$, car la seule relation de dépendance possible est $u_{k-1} \dashrightarrow_{\text{nr}} u_k$. La définition de cette relation donne $u_{k-1}!^{\sigma}$. Par conséquent :
- $u_{k-1} \neq v$ et $k > 2$ ;
- $u_{k-2} \rightarrowtail u_{k-1}$ par un raisonnement similaire au cas précédent (c'est la seule dépendance possible).

Comme précédemment, nous construisons deux suites $(v_i)_{1..m}$ et $(w_i)_{1..l}$ telles que :

1. (suite $(v_i)_{1..m}$)
   $v_1 = u_{k-1}$ client de $u_k$, et $v_m = u_k$ serveur non réentrant. Le lemme 5 page 113 nous assure l'existence de cette suite, avec :

$$\forall i, v_i \overset{\blacklozenge}{\leftrightarrow} v_{i+1} \ \vee \ \text{partenaire}(v_i, v_{i+1})$$

2. (suite $(w_i)_{1..l}$)
   Nous prenons $w_1 = u_k$ et $w_l = u_{k-1}$. La suite se construit à partir de $(u_i)$ selon le même principe que précédemment, mais auquel nous ajoutons :
   Si $w_i = \text{parent}(u_n) = u$ : le port $u$ s'attache la référence $v$, donc par le lemme 5 nous avons la suite $(w_p)_{i..i+q}$ telle que :

$$w_{i+q} = v = u_1 \text{ et } w_{i+j} \overset{\blacklozenge}{\leftrightarrow} w_{i+j+1} \ \vee \ \text{partenaire}(w_{i+j}, w_{i+j+1})$$

La fin du raisonnement est similaire au cas précédent. Nous avons l'existence, à l'assemblage, de la suite (avec $u_k$ port serveur non réentrant) :

$$u_k = \underbrace{w_1 \xrightarrow{\mathcal{A}} {}^*w_{l-1} \overset{\diamond}{\rightsquigarrow} w_l}_{u, v \in (w_i)_{1..l}} = \text{parent}(u_{k-1}) = \underbrace{v_1, \ldots, v_m}_{(v_i \overset{\blacklozenge}{\leftrightarrow} v_{i+1} \ \vee \ \text{partenaire}(v_i, v_{i+1}))} = u_k$$

Ce qui contredit $\neg P_{\text{cycle}}(u_k)$ (avec $u_k$ serveur non réentrant).

**$v$ est un port serveur**

Il n'y a pas d'interblocage, par définition du comportement d'un port serveur.

**3.** SEND

Soit $u$ le port effectuant l'envoi d'un message vers le port $v$. Nous avons alors création d'une dépendance entre $u$ et $v$ ($u \dashrightarrow v$ ou $v \dashrightarrow u$). Nous raisonnons comme

pour la règle BIND, en distinguant si $v$ est non réentrant ou pas. La construction des suites $(v_i)$ et $(w_i)$ est similaire.

En fait, la création d'une dépendance entre les ports $u$ et $v$ nous donne la même démonstration que pour BIND. Si nous avons un changement de dépendance ($u \dashrightarrow v$ modifiée en $v \dashrightarrow u$), la démonstration peut tirer parti du fait que ces deux dépendances sont, à l'assemblage, modélisées de la même manière (par des dépendances bidirectionnelles $\overset{\clubsuit}{\leftrightarrow}$ et $\mathit{partenaire}(\cdot,\cdot)$)

**4.** RECV,RECV-UN,RECV*,RECVx

Les démonstrations pour les règles RECV, RECV-UN et RECV* sont soit évidentes, soit similaires aux précédentes.

Pour la règle RECVx, nous avons un port serveur non réentrant, $u$, qui reçoit le message provenant d'un client $v$. Le port $u$ crée un port $w$ qui traitera avec le client, et se suspend pendant toute la durée du traitement (soit : tant que $w$ n'est pas supprimé). Nous avons création des dépendances :

$$u\mathsf{?}^{\mathsf{i}} \rightarrowtail_{\mathrm{nr}} w\mathsf{!}^{\mathsf{a}} \text{ et } u = \mathrm{parent}(w) \qquad v \dashrightarrow_{\mathrm{nr}} w$$

où il est évident qu'il n'y a pas de cycles. La dépendance $u \rightarrowtail_{\mathrm{nr}} w$ sera supprimée avec la règle REINIT.

**5.** REINIT

Le port $u$ est supprimé. Le port serveur non réentrant $s$ devient actif en réception, et la dépendance $s \rightarrowtail_{\mathrm{nr}} u$ est supprimée.

**6.** SUSPEND

Démonstration similaires aux précédentes. On remarquera que si $u \rightarrowtail v$ est créée suite à cette dépendance, alors à l'assemblage nous avons $\mathrm{parent}(u) \overset{\diamond}{\rightarrowtail} \mathrm{parent}(v)$. □

## 6.4 Exemple : le dîner de philosophes

Nous appliquons l'intégration des serveurs non réentrants à l'étude de cas classique des philosophes.

Le problème est le suivant : des philosophes (en général cinq, mais nous ne considérerons que deux philosophes pour simplifier) sont installés autour d'une table circulaire, et passent leur temps à penser et à manger. La table est mise avec cinq fourchettes qui sont disposées entre chacun des philosophes. De temps en temps, un philosophe a faim et essaye de prendre les fourchettes qui sont immédiatement à coté de lui. Un philosophe a besoin de deux fourchettes pour manger, et ne peut évidemment pas prendre une fourchette qui est dans la main d'un voisin. Quand un philosophe a ses deux fourchettes dans les mains en même temps, il mange sans libérer ses fourchettes. Dans le cas contraire, il doit attendre que celles-ci deviennent libres. Enfin, quand il a finit de manger, il repose ses deux fourchettes et commence à penser à nouveau.

Nous avons alors deux composants : `Philosophe` et `Fourchette`. Le composant `Fourchette` a un seul port, $f$, qui est un serveur non réentrant et typé de la manière suivante :

fourchette = **may ? x[** prendre ; **must ! [** donner ; **must ? [** rendre ; **0 ] ] ]**

Il faut demander la ressource *fourchette* par le message *prendre*. La réception du message *donner* fournit l'accès à cette ressource, que l'on doit impérativement rendre par le message *rendre*. Si la fourchette est déjà utilisée par un autre philosophe, il faut attendre que ce dernier repose la fourchette.

Le composant `Philosophe` a deux port, $fd$ et $fg$ correspondant aux fourchettes à sa droite et à sa gauche. Le type de ces ports est le suivant :

main = **may ! [** prendre ; **must ? [** donner ; **must ! [** rendre ; **0 ] ] ]**

Le composant `Philosophe` est tel qu'il demande d'abord l'accès à la fourchette droite, puis à la fourchette gauche. Son diagramme d'état est donné figure 6.6. On remarque que le port $fd$ se suspend au port $fg$ une fois qu'il a reçu le message *donner*. A l'assemblage, la relation $fd \overset{\diamond}{\rightarrowtail} fg$ est connue.

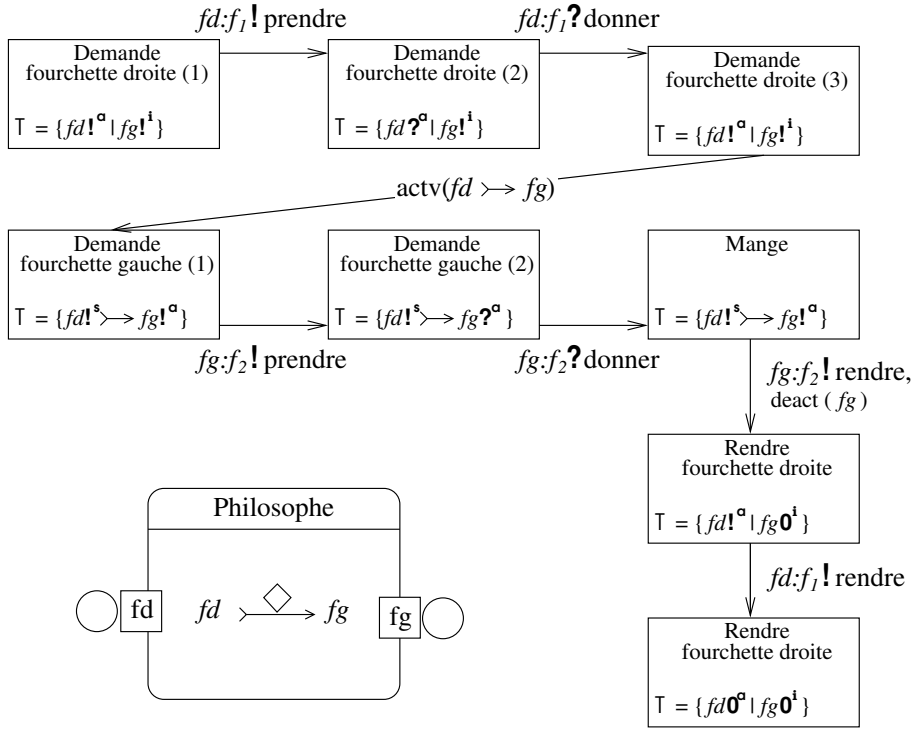



**Fig. 6.6:** *Diagramme d'état (partiel) pour le composant* `Philosophe`

Plusieurs assemblages sont possibles. La figure 6.7 page suivante donne un assemblage où un interblocage est possible : la propriété $P_{\text{risque}}(f_1)$ est vraie. Une solution, classique, est d'intervertir, pour un seul philosophe, les fourchettes droites et gauches.

Dans la figure 6.8 page 121, le philosophe du bas ne connais pas, à l'assemblage, les références vers les ports serveurs non réentrants $f_1$ et $f_2$. Il demandera la référence à un port serveur $s^*_{1,2}$.

Pour le composant `Fourchette`, le serveur $s_{1,2}$ a le type :

serveur_fourchette = **may ? *[** demande_ref ; **must ! [** ref (fourchette) ; **0 ] ]**

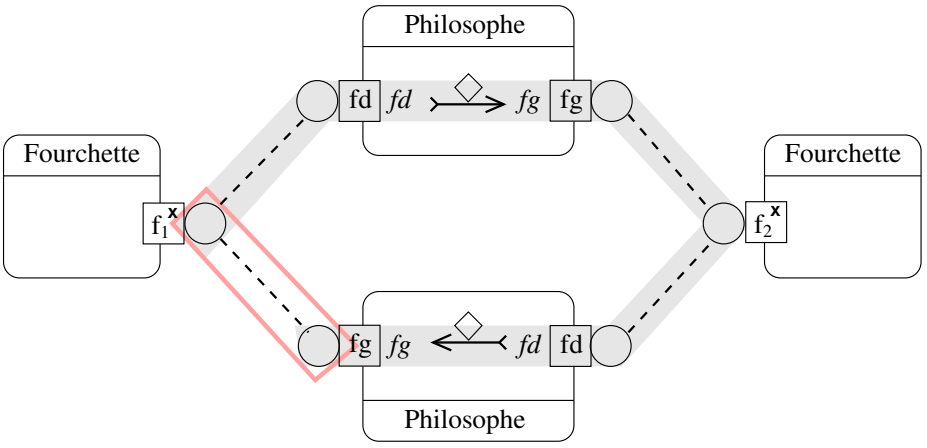



**FIG. 6.7:** *Dîner de philosophes : liens statiques*

Pour le composant `Fourchette`, nous avons, pour les ensembles de la section 6.1.3 page 104 :

$$\mathcal{E}(\text{serveur_fourchette}) = \{\text{fourchette}\}$$

Les autres ensembles $(\mathcal{R}(\cdot), \mathcal{S}(\cdot))$ sont vides. Nous avons donc la relation $f_{1,2} \overset{\spadesuit}{\leftrightarrow} s_{1,2}$.

De même, pour le composant `Philosophe2` nous avons :

client_fourchette = **may ! [** demande_ref **; must ? [** ref (fourchette) **; 0 ] ]**

$$\mathcal{R}(\text{client_fourchette}) = \{\text{fourchette}\}$$

Les autres ensembles sont vides, et nous avons $fd, fg \overset{\spadesuit}{\leftrightarrow} c_{1,2}$, car $Comp(\text{fourchette}, \text{main})$ est vraie. Sur la figure, seules les relations $fg \overset{\spadesuit}{\leftrightarrow} c_1$ et $fd \overset{\spadesuit}{\leftrightarrow} c_2$ sont représentées.

Un cycle tel que nous l'avons définit est présent à l'assemblage, donc il y a un risque d'interblocage.

On remarquera, par contre, qu'intervertir les fourchettes droites et gauches ne résout pas le problème dans ce cas précis : cela est dû au fait que nous avons les relations $fg \overset{\spadesuit}{\leftrightarrow} c_2$ et $fd \overset{\spadesuit}{\leftrightarrow} c_1$, alors que la référence reçue par $c_2$ ne sera pas utilisée par $fg$. Une solution serait d'ajouter un ensemble de règles vérifiant le mouvement des références à l'intérieur du composant : si $fg \overset{\spadesuit}{\leftrightarrow} c_2$ n'est pas déclarée, alors la référence reçue par $c_2$ ne peut être attachée au port $fg$.

## 6.5 Conclusion

Nous venons de voir une extension de notre langage aux ports serveurs non réentrants. Ces ports ne peuvent servir qu'un seul client à la fois, et se suspendent sur le port effectuant le traitement en cours.

Pour garantir la propriété d'absence d'interblocage, nous raffinons la spécification du composant de la manière suivante :

- les dépendances entre les ports $(x \rightarrowtail y)$ sont déclarées ;
- les mouvements de référence à l'intérieur du composant (par exemple réception de la référence $u$ sur un port, puis attachement de $u$ à un autre port) sont extrapolées à partir de la structure des types de tous les ports du composant.

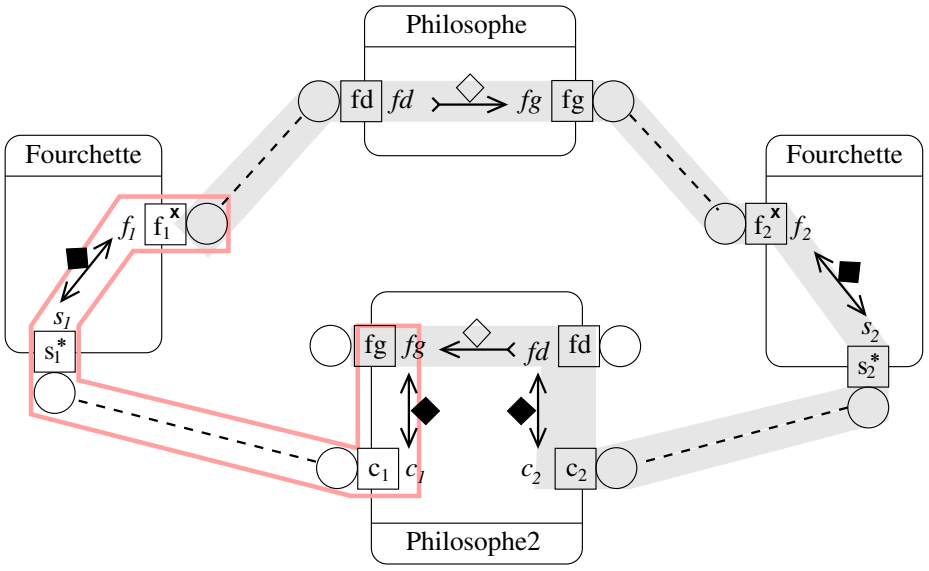



**FIG. 6.8:** *Dîner de philosophes : liens non connus à l'assemblage*

A l'assemblage, nous pouvons construire un graphe de dépendances construit à partir des dépendances entre les ports, des mouvements de référence dans le composant, et des liens d'assemblage entre les ports. Il y a absence d'interblocage si un cycle contenant un port serveur non réentrant apparaît dans ce graphe.

Notre inférence sur les mouvements de références est cependant trop contraignante : elle indique *tous* les mouvements possibles, notamment ceux qui n'auront pas lieu. Pour résoudre ce problème, il faut ajouter des règles vérifiant que ces mouvements sont bien déclarés à l'assemblage.

Enfin, nous pensons que l'extension qui est proposée dans ce chapitre ne restreint pas les autres modèles. En effet, la condition d'absence d'interblocage fait intervenir les ports serveurs non réentrants, qui étaient absents dans les chapitres précédents. Par exemple, un cycle peut apparaître à l'assemblage dans le graphe dépendances, sans qu'il ne contienne de ports serveurs non réentrants.

Chapitre 7

# Conclusion

## 7.1 Bilan des travaux

Nous avons défini un langage de type d'interface comportementale, qui s'inspire des types réguliers de Nierstrasz [Nie95], est basé sur une sémantique de messages, et offre des notions d'actions permises et obligatoires. Ces obligations et permissions sont des contraintes imposées aux deux parties d'une communication point-à-point (par exemple, une obligation de recevoir contraint l'interlocuteur). Une relation de sous-typage permet une architecture plus souple, autorise la spécialisation des composants et la vérification de liens de délégation. Une relation de compatibilité entre interfaces nous permet de vérifier la validité des liens d'assemblage et de s'affranchir des erreurs de communication de type «message non compris». Il est possible de spécifier les interfaces de type flux ou événements avec un type récursif. Dès lors, toutes les interfaces des composants EJB ou CCM peuvent être typées avec notre langage.

Notre langage définit alors un contrat comportemental pour le composant. Les règles de vérification de respect de contrat reposent sur une abstraction du comportement du composant. Notre sémantique de composant se base sur une observation de l'état des ports (essentiellement émission/réception, activé/suspendu). Les composants sont multi-tâches : notre modèle de tâches se concentre uniquement sur l'état des ports et les dépendances entre les ports (à savoir un port $u$ attend qu'un port $v$ termine son action).

Enfin, notre approche autorise deux types de liens entre les ports : client/serveur et point-à-point. Ces liaisons sont dynamiques : création mais aussi modification (changement d'interlocuteur grâce à l'envoi de référence). Les serveurs peuvent être réentrants ou non.

Notre travail permet de spécifier des applications fiables, développées sur la base des paradigmes de composants. Deux vérifications ont lieu pour assurer cette fiabilité :

- le composant doit respecter le contrat qui lui est associé. Cette vérification est statique, et peut se faire à la compilation du composant ;
- l'assemblage impose que les interfaces communicantes soient compatibles. Le comportement du composant n'entre pas du tout en ligne de compte lors de cette vérification, ce qui est très utile lors de l'utilisation de composants sur étagère (COTS). Cependant, le cas des serveurs non réentrants demande un

raffinement supplémentaire : le composant doit spécifier les dépendances potentielles entre ses ports. La vérification de la compatibilité des interfaces est suffisamment rapide pour pouvoir se faire au déploiement du composant.

Nous montrons des propriétés de sûreté (absence d'interblocage externe), et de vivacité (les messages sont compris et, moyennant l'équité entre les tâches du composant, consommés).

Notre approche présente quelques limites, en plus de problèmes d'équité, qui portent sur la sémantique du composant. La principale est qu'un port ne peut se suspendre directement sur un port *en réception*. Par exemple, si le port $u$ se suspend sur le port $v$, alors la première action de $v$ doit être une émission avant de pouvoir passer en réception. La raison sous-jacente est que le port $u$ demande un service au port $v$ : donc ce port doit envoyer un premier message avant de recevoir le résultat de la demande. Cette contrainte ne permet pas d'effectuer des appels asynchrones de fonctions avec un retour des résultats : en effet, cela voudrait dire qu'un port ayant besoin de ces résultats devra se suspendre sur le port qui attend de recevoir les données, ce qui est interdit. Cependant, vu la complexité mise en jeu lors d'appels de telles fonctions, nous pensons que les application s'orientent plutôt vers un appel synchrone de fonctions.

### 7.1.1 Travaux similaires

Les travaux les plus proches sont dans la mouvance des types réguliers.

Colaço, Colin, Dagnat, Pantel et Sallé travaillent sur l'inférence de type pour le langage d'acteur CAP [CTP03, Col02, DPCS00]. Ils s'intéressent aussi au problème de la consommation des messages : assurer qu'un message envoyé sera finalement consommé. Par contre, les auteurs ont des files d'attente qui ne sont pas ordonnées. La détection d'un message non consommé, ou «orphelin», est alors plus complexe à formaliser : il faut vérifier si dans le futur, l'acteur aura un comportement pour consommer le message. Leur approche s'appuie sur une description en logique temporelle. Ils définissent tout d'abord le potentiel de traitement et d'émission d'un acteur, c'est-à-dire l'ensemble des messages qu'un acteur peut recevoir (traiter) ou émettre. Le potentiel d'un acteur est représenté par un multi-ensemble de messages, chacun étiqueté d'une multiplicité. Cela leur permet de compter le nombre de fois qu'un message peut être consommé (ou envoyé), et de déduire quels sont les messages «orphelins». Ces orphelins sont en fait les messages qui ne seront pas consommés dans le futur. L'aspect comportemental, dont la syntaxe s'inspire de TyCO [VT93], est défini selon une abstraction intuitive du comportement [Col97, CPDS99], qui est l'union des multi-ensembles provenant de toutes les branches d'exécutions possibles à un moment donné. La différence essentielle avec notre travail est que les auteurs infèrent le type de l'acteur, et que les files d'attente sont des ensembles non ordonnés. Nous proposons une approche plus restrictive où nous vérifions qu'un composant est conforme à un type donné ; comme nous avons des files FIFO, nous vérifions seulement si le premier message de la file est consommé dans les prochaines actions du composant. Par contre, notre approche apporte le problème complexe d'interblocage, mais que nous avons résolu par des contraintes sur le comportement du composant.

Il est à noter que notre travail est la poursuite des travaux de Najm et Nimour [NNS99b, NNS99a, NN97], qui proposent un calcul d'objet pour interfaces non

uniformes. Nous avons repris la notion d'interfaces publiques (accessibles par plusieurs clients), et privées (accessible par un seul client) ; par contre, nous avons laissé de coté la communication qui était synchrone, et qui ne convenait pas dans le cadre composants distribués. Un type, chez Najm et Nimour, est décrit sous la forme d'une somme $\alpha ::= \Sigma_i l_i(\tilde{\alpha_i}).\alpha_i$. Leur système de type assure que tout message envoyé sera compris (et consommé) par le récepteur. Les types sont applicables sur des comportements réguliers ou à états infinis (le cas des états infinis est traité par un ensemble de prédicats ajouté au type). Par contre, les communications sont unidirectionnelles : les types ne décrivent que les enchaînements de réception ou d'émission de messages ; un mélange des deux n'est pas autorisé. Ce système de type s'applique donc difficilement à un port qui alterne émissions et réceptions. Une autre différence est la définition des relations entre les types (sous-typage et compatibilité). Ces relations sont définies à l'aide de bisimulations fortes, donc plutôt une notion sémantique du sous-typage et de la compatibilité. Nous avons défini nos relations selon une notion syntaxique. Arnaud Bailly [Bai02] s'est lui aussi inspiré du travail de Najm et Nimour : il propose un système de type pour le $\pi$-calcul temporisé. Il a la notion de permission (laisser le temps s'écouler jusqu'à un certain point) et la notion d'obligation (l'action doit avoir lieu avant le délai indiqué). Le temps-réel n'était pas notre sujet d'étude, mais il serait intéressant de regrouper les deux travaux, notamment pour imposer des contraintes plus fortes sur les obligations '**must**'.

Un autre travail très proche de celui de Najm et Nimour, est celui de Ravara et Vasconcelos [RRV02, RV00]. Le type a une syntaxe similaire, mais les auteurs y ajoutent un opérateur de composition parallèle et les transitions silencieuses $\tau$. Les relations entre les types comme le sous-typage sont là aussi définies avec une bisimulation. Ils s'intéressent aussi aux erreurs de type «message non compris», mais leur notion d'erreur est trop lâche. Une erreur est caractérisée par le fait que le message n'est pas accepté par un objet *persistant* ; c'est-à-dire que, en regard du couple objet-message, il n'y a pas d'erreur si le message est consommé ou si l'objet est éphémère.

Enfin, récemment, Gay et Vasconcelos, dans [GH99, VVR02, GH03] ont repris l'idée des types de session (*session types*) de Honda et al. [THK94, HVK98]. Les types de session sont associés aux canaux de communication. Un type de session spécifie le protocole d'interaction qui a lieu sur ce canal. De plus, un constructeur de branchement et un constructeur de choix permettent de typer respectivement les serveurs et les clients : le branchement symbolise l'union des services proposés, tandis que le choix symbolise la possibilité pour le client de choisir tel ou tel message à envoyer. Leur système de type assure que les canaux de communication sont utilisés correctement (ce qui revient à la détection d'erreurs «message non compris»). Par contre, les auteurs n'ont pas associé de modalités sur les actions.

Concernant les travaux sur les systèmes de type pour le $\pi$-calcul, le travail le plus proche est celui de Kobayashi [Kob02]. Il a étendu le travail qu'il a fait avec Pierce, Turner et Igarashi [KPT99, IK01], et où le système de type dénotait les séquences d'interaction d'émission et de réception. Kobayashi apporte la propriété d'absence de blocage d'un processus (à la fois interblocage et verrou vivant – *deadlock* et *livelock*). Il préfixe les actions qui sont requises dans le processus ; les actions décrites par le type d'un canal de communication de ce processus sont alors estampillés de deux

compteurs : l'un donne les capacités, l'autre les obligations, en nombre de transitions nécessaires pour effectuer l'action. Deux processus sont compatibles si les capacités de l'un correspondent aux obligations de l'autre. Par exemple, le type $I_3^\infty.O_\infty^3$ indique que le processus est d'abord en réception ($I$), puis en émission ($O$) ; la réception doit avoir lieu dans un délai de 3 transitions, tandis que l'émission ne se fera qu'après un délai de 3 transitions. Ce type n'est pas compatible avec $O_\infty^1.I_1^\infty$, car la réception ($I_1^\infty$) doit avoir lieu en une seule transition. Le principal problème de ce système de type est de donner la valeur de ces compteurs. Notre approche oblige le composant à avoir une transition possible pour les actions obligatoires ; notre relation de compatibilité est alors plus souple (nous n'avons pas de notion de nombre de transitions, qui se rapproche assez des notions temps réels), il nous reste un problème d'équité entre ces transitions, problème qui est absent chez Kobayashi.

Plusieurs différences ou similitudes sont à noter entre notre approche et les travaux que nous venons de présenter :

- peu de systèmes de type arborent des modalités, avec actions permises et obligatoires. Cette notion nous semble importante, et a été illustrée au chapitre 3 par un exemple de compte en banque et une étude de cas ;
- les types comportementaux commencent seulement à mélanger émissions et réceptions sur le même port ;
- les travaux ne se positionnent pas vis-à-vis d'un assemblage de composant. Notre approche se focalise entre autres sur la vérification des liens à l'assemblage. De plus, nous apportons la notion de lien dynamique (création et modification de lien), qui est absente dans les modèles à base de composant (comme les ADL) ;
- la distinction entre un port pair et un port client nous semble importante dans le cadre du typage comportemental. La raison principale est d'assurer, dans un dialogue complexe, une progression des types cohérente entre les interlocuteurs. Une première solution est de restreindre les communications à des liaisons point-à-point ; c'est d'ailleurs le choix qui semble avoir été fait indirectement dans les travaux précédents. Le cas du port serveur nous autorise les liaisons 1-à-n. Nous pensons que ces deux types de communication nous permettront de spécifier les autres modèles de communication (en utilisant un connecteur servant alors de médium de communication) ;
- nous ne proposons pas d'inférer le type des ports. Nous travaillons plutôt à vérifier qu'un composant respecte le contrat (ou les types) explicites.

## 7.2 Travaux futurs

Plusieurs points sont encore à explorer, que l'on peut regrouper en deux axes. D'une part, poursuivre l'étude théorique concernant la sémantique du composant et le langage d'interface. Il s'agit d'améliorer la faisabilité de la vérification de contrat, et d'étudier le cas des composants composites. D'autre part, l'implantation de nos travaux dans un outil de conception logicielle basé sur UML, et aux plates-formes de composants. Il s'agit de produire un profil UML, et d'étudier l'intégration d'un contrat dans les descripteurs de déploiement.

### 7.2.1 Vérification de contrats

La vérification du respect de contrat est difficilement utilisable en l'état, ou du moins se heurte au problème classique d'explosion du nombre d'états des techniques de vérification de modèles. Nous avons illustré notre sémantique de composants par un exemple simple de la manière suivante : chaque tâche est spécifiée par un diagramme d'état, dans lequel chaque état inclut un comportement interne ; la création de tâches d'exécution permet de spécifier une partie des automates à états infinis.

Dès lors, la vérification pourrait se faire en deux parties. La première serait de vérifier qu'une tâche respecte le contrat. La principale difficulté concerne alors les appels de procédure ou de fonction internes. Une première approche est de modéliser l'appel de procédure en s'inspirant de [ÁMdBdRS03][1]. Nos tâches peuvent en fait être formalisées de manière plus précise par des piles d'exécution de la forme $(p_i, s_{ij}) \circ (p_k, s_{kl}) \circ \cdots$, où $p_{i,k}$ sont les noms de procédure (avec $p_i$ appelée par $p_k$), et les $s_{ij,kl}$ les états courants de ces procédures. Ce procédé présente le double avantage de tirer parti des techniques de vérification compositionnelles, et celui de faciliter la vérification des propriétés de vivacité : chaque $p_i$ doit terminer pour que $p_k$ puisse reprendre son exécution, et les appels récursifs terminent (soit : $p_i$ apparaît un nombre fini de fois dans la pile d'exécution).

La deuxième partie de la vérification serait d'étudier les dépendances entre tâches d'exécution[2]. Cette vérification peut être modélisée avec un réseau de Petri. L'idée est d'utiliser deux formalismes : l'un pour les séquences d'exécutions, l'autre pour le parallélisme des tâches. Il faut alors définir la projection de la sémantique du composant sur ces deux formalismes.

Enfin, nous n'avons abordé que les composants, et laissé de coté les connecteurs. Notre relation de dualité peut être utilisée pour spécialiser la prise du connecteur auquel un port est attaché (la prise a le type dual du port correspondant). Nous voyons ensuite deux types de connecteurs :

- les connecteurs qui font office de médiateur : ils ne comprennent pas les messages, et ne font que les transférer. Il est nécessaire de s'assurer que la prise où les messages sont reçus est compatible avec la prise d'où les messages sont émis ;
- les connecteurs qui comprennent la sémantique des messages. La vérification que le connecteur respecte le type de ses prises a lieu de la même manière qu'avec les composants.

L'inconvénient de ces méthodes est que la vérification que le connecteur respecte le type de ses interfaces a lieu à l'assemblage (les prises doivent être instanciées).

### 7.2.2 Composant composite

Nous n'avons pas abordé, ou presque, la notion de composant composite. Notre relation de sous-typage permet de spécifier des connecteurs de délégation utilisés

[1] Les auteurs ont défini une logique de Hoare pour la vérification des tâches Java ; la vérification se fait en ajoutant des annotations aux programmes, et en les vérifiant avec l'outil de vérification de modèles PVS.

[2] il s'agit de vérifier la précondition de la règle C-THSUSP du tableau 4.1 page 67 (un port ne se suspend pas indirectement à un port actif en réception), et de vérifier que le prédicat Must($\mathsf{T}'$) est respecté (un port typé **must**! n'est pas suspendu par un port typé **may?**).

dans les composants composites : un port d'un composant interne est sous-type d'un port du composant global (composant à gauche de la figure 7.1).

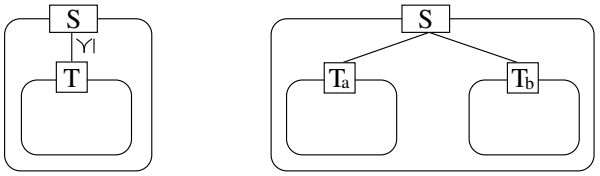



**Fig. 7.1:** *Composant composite : délégation simple ($T \preceq S$) et multiple ($(T_a, T_b) \preceq S$)*

Cependant, ces relations ne sont que 1-à-1. Les relations de délégations du composant à droite de la figure 7.1 posent deux problèmes :

- les messages de $T_a$ peuvent aussi être reçus par $T_b$ : vers quel port orienter le message reçu ? Une restriction possible est d'avoir des messages pour $T_a$ et $T_b$ qui soient distincts ;
- des problèmes quant à la synchronisation des comportements des deux composants internes : le type $T_a$ fera évoluer le type de $S$. Cette évolution doit se faire de telle sorte qu'elle permette au type $T_b$ d'être à nouveau, dans le futur, sous-type de $S$.
  Cela doit se faire en imposant des contraintes sur la structure des types. Une contrainte forte est que la relation de sous-typage est vraie à tout moment, donc chaque évolution du type de $S$ doit être super-type de $T_b$. Cela impose notamment des contraintes sur le type de $T_a$. Une contrainte plus souple demanderait la définition d'opérations de composition d'interfaces disjointes, selon une composition d'automates disjoints.
  La définition de la composition d'interfaces devra étudier d'une part l'adéquation des modalités (composer une modalité **may** avec une modalité **must** ; nous avons vu par les exemples (ex. 3.7 page 55) que les interfaces typées **may** doivent revenir à leur état initial), et d'autre part les émissions et les réceptions. Ce dernier point a été soulevé dans la conclusion du chapitre 3 page 43 : $I = (\textbf{must?}\ M) + (\textbf{may!}\ N)$. Dans ce cas, une solution serait d'avoir un port typé **must?** $M$, et un autre typé **may!** $N$. On remarquera aussi que, dans cet exemple précis, $J = \textbf{must?}\ M$ est sous-type de $I$, donc se comporte correctement vis-à-vis du partenaire.

Le cadre formel pour les composants composites est complexe. Dans cette optique, il est serait intéressant d'étudier l'adéquation de notre langage avec Fractal[3][Ste03, BCS03]. Fractal est un modèle de composants composites. Formellement, un composant est une expression $\alpha[P](Q)$ où $\alpha$ est une localité (sorte d'emplacement mémoire où se trouve le composant), $P$ est la membrane du composant (interface entre le composant et son environnement), et $Q$ le processus le représentant (le comportement interne du composant). Notre langage d'interface devrait pouvoir s'intégrer dans la membrane $[P]$ du composant : le comportement du composant, $Q$, doit être conforme à la spécification donnée par la membrane.

[3] `http://fractal.objectweb.org/`

### 7.2.3 Profil UML

L'intégration de notre travail à un outil de conception logicielle est important. Il est préférable de tirer parti d'outils de modélisation UML comme *Objecteering* (de la société *Softeam*), et de créer un profil UML. Cela consiste à modifier le méta-modèle UML pour ses propres besoins.

Le projet ACCORD, dans [ACC03], a formalisé avec un profil UML le modèle abstrait de composant présenté succinctement dans la section 2.1 page 10. Les contrats y sont exprimés dans une méta-classe UML «contrainte», sous forme textuelle, ce qui convient entièrement à notre langage d'interface. Cependant, cette forme n'est pas très lisible.

Nous proposons plutôt de mettre en œuvre ce profil de la façon suivante. Un stéréotype sur les ports indique qu'il est typé selon notre approche. A ce port, nous associons un diagramme d'état (UML 2.0 définit un tel port comme état complexe ; cependant, l'outil *Objecteering*, par exemple, n'a pas encore implanté cette fonctionnalité. Il est cependant possible de typer le port par une interface, qui contiendra le comportement du type).

Le diagramme d'état associé au port est la représentation graphique de notre langage. Les états de ce diagramme doivent être séréotypés selon les modalités : «**may?**», «**may!**», «**must?**», «**must!**», «**0**», «alias». Un état stéréotypé «alias» est tel que le nom de cet état est le nom de son type ; le nom de ce type peut être le nom du diagramme d'état représentant le type, ou le nom de l'interface contenant le diagramme correspondant. Les arcs portent les différents messages autorisés, mais il n'est toutefois pas possible d'avoir une structure aussi détaillée qu'un nom et une liste d'arguments.

Une fois ce méta-modèle mis en place, le profil UML peut vérifier les liens d'assemblage et de délégation (pour les composants composites) en utilisant les relations de compatibilité et de sous-typage. Il est à noter qu'il suffit de coder l'une ou l'autre, car des relations d'équivalence existent entre les deux.

La vérification qu'un composant respecte son contrat n'est pas possible dans un tel environnement. Il faut avoir recours à un compilateur (ou un *model-checker*) extérieur.

### 7.2.4 Intégration aux plates-formes de composants

Notre spécification de contrat doit pouvoir être intégrée plus ou moins facilement aux plates-formes de composants.

La première étape est d'étendre le descripteur de déploiement. Cette extension du descripteur peut être utilisée pour vérifier automatiquement la compatibilité du composant déployé avec les autres composants du système. Par exemple, le port du composant `Rapporteur` de notre étude de cas peut s'écrire :

```
<interface name="i_rapporteur">
  <send mod="must">
    <message name="acces_rapporteur">
      <arg type="string"/>
      <arg type="string"/>
```

```
      <arg type="integer"/>
      <nextstate>
        <receive mod="must">
          <message name="accorde">
            <arg type="strings"/>
            <nextstate>
              <alias name="entrer_rapport"/>
            </nextstate>
          </message>
          <message name="refuse">
            <nextstate>
              <null/>
            </nextstate>
          </message>
        </receive>
      </nextstate>
    </message>
  </send>
</interface>

<interface name="entrer_rapport">
  <send mod="must">
    ...
  </send>
</interface>
```

De plus, il est possible d'utiliser le descripteur pour vérifier le contrat à l'exécution. La raison principale d'effectuer cette vérification est que notre approche est optimiste : nous avons supposé que l'environnement respectait le contrat du composant, mais c'était dans l'optique où tous les composants ont été vérifiés. Dans le cas d'un environnement pessimiste, le contrat permet de bloquer les messages non désirés ; la mise en œuvre pourra s'inspirer de Fontaine et al. [FNR02], qui ont associé des contrats aux EJB pour modifier les interactions. Par contre, les propriétés que nous démontrons, comme l'absence d'interblocage externe, ne sont plus garanties.

## 7.3 Applications

Outre l'application pour des assemblages de composants effectués durant les phases de conception d'un logiciel, notre approche est aussi applicable dans un contexte d'exécution.

Tout d'abord, les composants adaptables peuvent tirer un certain profit de notre relation de sous-typage, qui permet une spécialisation du composant, donc une adaptation (certes limitée) de son comportement.

Une utilisation beaucoup plus intéressante est le remplacement de composant. Notre modèle propose des liens dynamiques : l'ancien composant peut, petit à petit, envoyer les références de ses partenaires au nouveau composant, effectuant ainsi une migration lente vers ce dernier.

Le véritable remplacement est aussi possible : reste le problème des ports en réception de l'ancien composant (les références vers de tels ports n'ont plus de signification avec le nouveau composant), et de la sauvegarde de l'état du composant. Pourtant, en supposant que l'intergiciel résout le premier problème[4], alors il suffit, lors du remplacement, de transférer l'état de l'ancien composant vers le nouveau. Le remplacement ne peut avoir lieu que si le type des ports du nouveau composant sont sous-types de ceux du composant à remplacer. Le typage comportemental montre ici toute sa puissance comparé à un simple regroupement de méthodes : si les ports ne sont pas positionnés au bon endroit dans le dialogue avec les partenaires, le remplacement est impossible.

Enfin, et cela rejoint en quelque sorte le déploiement de composant dans une application en cours d'exécution, notre langage de type peut être utilisé avec la mobilité du composant : moyennant la plate-forme mobile, l'arrivée du composant sur un site est accepté si les types *courants* de ses ports sont compatibles avec la configuration actuelle du site. Un bémol quant à cette application est que nos propriétés sont démontrées à partir d'un assemblage respectant des règles particulières. Le site d'arrivée doit non seulement vérifier la compatibilité, mais aussi que les règles de l'assemblage sont respectées. Cela amène soit à de fortes restrictions (par exemple le composant mobile ne peut se connecter qu'à des ports serveur), soit à une architecture sous-jacente complexe, qui doit avoir une abstraction suffisante de l'état de l'ensemble de la configuration.

[4] il peut garder un point d'accès aux anciens ports, et transmettre toutes les requêtes vers le nouveau port. On notera que dès que ce nouveau port passe en émission, le partenaire sera informé de sa présence : l'ancien port peut donc être supprimé.

# Bibliographie


[ACC02] ACCORD : Formalisation uml du modèle abstrait d'assemblage de composants. Rapport technique Livrable 1.3, Projet RNTL Accord, novembre 2002.

[ACC03] ACCORD : Métamodèle d'assemblage de composants par contrats. Rapport technique Livrable 1.5, Projet RNTL Accord, septembre 2003.

[AF99] Luís Filipe Andrade et José Luiz Fiadeiro : Interconnecting Objects via Contracts. Dans Bernhard Rumpe, éditeur : *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 de *LNCS*, Kaiserslautern, Germany, octobre 1999. Springer-Verlag.

[AG96] Robert Allen et David Garlan : The wright architectural specification language. Rapport technique CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, Pillsburgh, PA, septembre 1996.

[AHKV98] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman et Moshe Y. Vardi : Alternating refinement relations. *Lecture Notes in Computer Science*, 1466:163–178, 1998.

[AL93] Martín Abadi et Leslie Lamport : Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73 – 132, janvier 1993.

[AL95] Martín Abadi et Leslie Lamport : Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.

[All97] Robert J. Allen : *A Formal Approach to Software Architecture.* Thèse de doctorat, School of Computer Science, Carnegie Mellon University, mai 1997.

[AM98] Samson Abramsky et Guy McCusker : Game semantics. Dans In H. Schwichtenberg et U. Berger, éditeurs : *Logic and Computation : Proceedings of the 1997 Marktoberdorf Summer School.* Springer-Verlag, 1998.

[ÁMdBdRS03] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever et Martin Steffen : A hoare logic for monitors in java. Rapport technique TR-ST-03-1, University of Kiel, avril 2003.

[AMST93] Gul Agha, Ian A. Mason, Scott F. Smith et Carolyn L. Talcott : A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1993.

[AP03] Jiri Adamek et Frantisek Plasil : Behavior protocols capturing errors and updates. Dans *Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE 2003), ETAPS*, Poland, avril 2003. University of Warsaw.

[Bai02] Arnaud Bailly : *Assume / Guarantee Contracts for Timed Mobile Objects*. Thèse de doctorat, ENST, Paris, France, décembre 2002.

[BCC+03] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino et Erik Poll : An overview of jml tools and applications. Dans Thomas Arts et Wan Fokkink, éditeurs : *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, volume 80 de *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, juin 2003.

[BCS03] E. Bruneton, T. Coupaye et J.B. Stefani : The fractal component model, septembre 2003.

[BHR84] S. D. Brookes, C. A. R. Hoare et A. W. Roscoe : A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599, 1984.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau et Damien Watkins : Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.

[BM02] Mark Brörkens et Michael Möller : Jassda trace assertions, runtime checking the dynamic of java programs. Dans Ina Schieferdecker, Hartmut König et Adam Wolisz, éditeurs : *Trends in Testing Communicating Systems, International Conference on Testing of Communicating Systems*, pages 39–48, Berlin, Germany, mars 2002.

[Bou97a] G. Boudol : Typing the use of resources in a concurrent calculus. Dans *ASIAN'97, the Asian Computing Science Conference*, volume 1345 de *Lecture Notes in Computer Science*, pages 239–253, Kathmandu, Nepal, 1997.

[Bou97b] Gérard Boudol : The pi-calculus in direct style. Dans *Conference Record of POPL '97*, pages 228–241, 1997.

[Bry95] R. E. Bryant : Binary decision diagrams and beyond : Enabling technologies for formal verification. Dans *International Conference on Computer Aided Design*, pages 236–245, Los Alamitos, Ca., USA, novembre 1995. IEEE Computer Society Press.

[Car03] Eric Cariou : *Contribution à un processus de réification d'abstractions de communications*. Thèse de doctorat, Université de Rennes 1, juin 2003.

[CB02] Eric Cariou et Antoine Beugnard : The specification of uml collaborations as interactions components. Dans *Fifth International Conference on the Unified Modeling Language (UML 2002)*, volume 2460 de *LNCS*. Springer Verlag, Dresden, Germany, octobre 2002.

[CBJ02] Eric Cariou, Antoine Beugnard et Jean-Marc Jézéquel : An architecture and a process for implementing distributed collaborations. Dans *6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002)*, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland, septembre 2002. IEEE Computer Society.

[CdAH+02] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, Marcin Jurdziński et Freddy Y.C. Mang : Interface compatibility checking for software modules. Dans E. Brinksma et K.G. Larsen, éditeurs : *CAV 02 : Computer-Aided Verification*, volume 2404 de *Lecture Notes in Computer Science*, pages 428–441. Springer-Verlag, 2002.

[CdAHM02] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger et Freddy Y.C. Mang : Synchronous and bidirectional component interfaces. Dans E. Brinksma et K.G. Larsen, éditeurs : *CAV 02 : Computer-Aided Verification*, volume 2404 de *Lecture Notes in Computer Science*, pages 414–427. Springer-Verlag, 2002.

[CFN03a] Cyril Carrez, Alessandro Fantechi et Elie Najm : Behavioural contracts for a sound composition of components. Dans Hartmut König, Monika Heiner et Adam Wolisz, éditeurs : *23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003, IFIP TC 6/WG 6.1)*, volume 2767 de *LNCS*, pages 111–126. Springer-Verlag, Berlin, Germany, septembre 2003.

[CFN03b] Cyril Carrez, Alessandro Fantechi et Elie Najm : Contrats comportementaux pour un assemblage sain de composants. Dans *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 2003)*, Paris, France, octobre 2003. Traduction française de [CFN03a].

[CGT03] Cyril Carrez, Arnaud Georgin et Alexandre Tauveron : Spécification du profil uml d'assemblage cible ejb (version 2). Rapport technique Livrable 2.6, Projet RNTL Accord, mai 2003.

[CH74] R. H. Campbell et A. N. Habermann : *The specification of process synchronization by path expressions*, volume 16 de *Lecture Notes in Computer Science*, pages 89–102. Springer Verlag, 1974.

[CM03] Olivier Caron et Alexis Muller : Spécification du profil uml d'assemblage cible ccm (version 2). Rapport technique Livrable 2.7, Projet RNTL Accord, mai 2003.

[Col97] Jean-Louis Colaço : *Analyses Statiques d'un calcul d'Acteurs par typage*. Thèse de doctorat, Institut National Polytechnique de Toulouse, octobre 1997.

[Col02] Matthias Colin : *Analyse Statique de la communication dans un langage d'acteur fonctionnel*. Thèse de doctorat, Instituqt National Polytechnique de Toulouse, décembre 2002.

[CPDS99] Jean-Louis Colaço, Marc Pantel, Fabien Dagnat et Patrick Sallé : Safety analysis for non-uniform service availability in ac-

tors. Dans *Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, février 1999.

[CTP03] Matthias Colin, Xavier Thirioux et Marc Pantel : Temporal logic based static analysis for non-uniform behaviours. Dans Elie Najm, Uwe Nestmann et Perdita Stevens, éditeurs : *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 de *LNCS*, pages 94–108. Springer-Verlag, novembre 2003.

[dAH01a] Luca de Alfaro et Thomas A. Henzinger : Interface automata. Dans Volker Gruhn, éditeur : *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 de *SOFTWARE ENGINEERING NOTES*, pages 109–120, New York, septembre 10–14 2001. ACM Press.

[dAH01b] Luca de Alfaro et Thomas A. Henzinger : Interface theories for component-based design. Dans T.A. Henzinger et C.M. Kirsch, éditeurs : *EMSOFT 01 : Embedded Software*, volume 2211 de *Lecture Notes in Computer Science*, pages 148–165. Springer-Verlag, 2001.

[DPCS00] Fabien Dagnat, Marc Pantel, Matthias Colin et Patrick Sallé : Typing concurrent objects and actors. *L'OBJET*, 6(1), 2000.

[DW98] Desmond D'Souza et Alan Wills : *Objets, Components and Frameworks with UML – The Catalysis approach*. Addison Wesley, 1998.

[ECM02a] Common Language Infrastructure (CLI). ECMA TC39/TG3, document ECMA-335, décembre 2002.

[ECM02b] C♯ language specification. ECMA TC39/TG2, document ECMA-334, décembre 2002.

[FLA03] Gérard Florin Fabrice Legond-Aubry, Daniel Enselme : Modèle abstrait d'assemblage de composants par contrats. Rapport technique Livrable 1.4, Projet RNTL Accord, juin 2003.

[FLF02] Robert Bruce Findler, Mario Latendresse et Matthias Felleisen : Behavioral software contracts. Rapport technique TR02-402, Rice University Computer Science, Houston, Texas, USA, juin 2002.

[Flo67] R. W. Floyd : Assigning meaning to programs. Dans J. T. Schwartz, éditeur : *Mathematical aspects of computer science : Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.

[FNR02] Arnaud Fontaine, Elie Najm et Michel Ruffin : Une approche de la composition de services par contrats de collaboration. Dans *CFIP'02 – Colloque Francophone sur l'Ingénierie des Protocoles*, Montreal, Canada, mai 2002. Hermés-Sciences.

[FNS97] Arnaud Février, Elie Najm et Jean-Bernard Stefani : Contracts for odp. Dans *Transformation-Based Reactive Systems Development, 4th Internatio nal AMAST Workshop on Real-Time Systems and Concurrent and Distribute d Software, ARTS'97*, volume 1231 de *Lecture Notes in Computer Science*, pages 216–232. Springer, 1997.

[GH99] Simon J. Gay et Malcolm Hole : Types and subtypes for client-server interactions. Dans *European Symposium on Programming*, pages 74–90, 1999.

[GH03] S. J. Gay et M. Hole. : Types and subtypes for correct communication in client-server systems. Rapport technique TR-2003-131, Department of Computing Science, University of Glasgow, février 2003.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides : *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[HHG90] Richard Helm, Ian M. Holland et Dipayan Gangopadhyay : Contracts : Specifing behaviorial compositions in object-oriented systems. Dans *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'90), Proceedings.*, pages 169–180, 1990. pdf pas imprimable ?...

[Hoa69] C. A. R. Hoare : An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HQR98] Thomas A. Henzinger, Shaz Qadeer et Sriram K. Rajamani : You assume, we guarantee : Methodology and case studies. Dans A.J. Hu et M.Y. Vardi, éditeurs : *CAV 98 : Computer-Aided Verification*, volume 1427 de *Lecture Notes in Computer Science*, pages 440–451. Springer-Verlag, 1998.

[HQR00] Thomas A. Henzinger, Shaz Qadeer et Sriram K. Rajamani : Decomposing refinement proofs using Assume-Guarantee reasoning. Dans *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD-2000)*, pages 245–253, Los Alamitos, California, novembre 5–9 2000. IEEE Computer Society Press.

[HVK98] Kohei Honda, Vasco T. Vasconcelos et Makoto Kubo : Language primitives and type disciplines for structured communication-based programming. Dans Chris Hankin, éditeur : *Programming Languages and Systems : Proceedings of the 7th European Symposium on Programming (ESOP'98)*, volume 1381 de *LNCS*, pages 22–138. Springer, 1998.

[IK01] Atsushi Igarashi et Naoki Kobayashi : A generic type system for the Pi-calculus. *ACM SIGPLAN Notices*, 36(3):128–141, mars 2001. A full version will be published as a Technical Report from Department of Information Science, University of Tokyo.

[Jac00] Michael Jackson : *Guide de l'amateur de Malt Whisky*. Solar, 4è édition, 2000.

[Kob02] Naoki Kobayashi : A type system for lock-free processes. *INFC-TRL : Information and Computation (formerly Information and Control)*, 177:122 – 159, 2002.

[KPRS01] Y Kesten, A Pnueli, L Raviv et E Shahar : Model checking with strong fairness. Rapport technique MCS01-07, Mathematics &

Computer Science, Weizmann Institute Of Science, Rehovot, Israel, 2001.

[KPT99] Naoki KOBAYASHI, Benjamin C. PIERCE et David N. TURNER : Linearity and the Pi-Calculus. *ACM Transactions on Programming Languages and Systems*, 21(5):914–947, 1999.

[LAEF03a] Fabrice LEGOND-AUBRY, Daniel ENSELME et Gérard FLORIN : Assembling contracts for components. Dans *Phd Student Workshop FMOODS-DAIS 2003*, Paris, France, novembre 2003.

[LAEF03b] Fabrice LEGOND-AUBRY, Daniel ENSELME et Gérard FLORIN : Contrat d'assemblage de composants. Dans *RENPAR-ASF-CFSE 2003, Proceedings*, Paris, France, novembre 2003.

[LPS81] Daniel J. LEHMANN, Amir PNUELI et Jonathan STAVI : Impartiality, justice and fairness : The ethics of concurrent termination. Dans Shimon EVEN et Oded KARIV, éditeurs : *Automata, Languages and Programming, 8th Colloquium*, volume 115 de *Lecture Notes in Computer Science*, pages 264–277, Acre (Akko), Israel, 13–17 juillet 1981. Springer-Verlag.

[LS84] Leslie LAMPORT et Fred B. SCHNEIDER : The "hoare logic" of CSP, and all that. *Programming Languages and Systems*, 6(2):281–296, 1984.

[LSW95] Kim G. LARSEN, Bernhard STEFFEN et Carsten WEISE : A constraint oriented proof methodology based on modal transition sytems. Dans Ed BRINKSMA, Rance CLEAVELAND, Kim Guldstrand LARSEN, Tiziana MARGARIA et Bernhard STEFFEN, éditeurs : *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS'95, Proceedings*, volume 1019 de *Lecture Notes in Computer Science*, pages 17–40. Springer, Aarhus, Denmark, mai 1995.

[LT87] Nancy A. LYNCH et Mark R. TUTTLE : Hierarchical correctness proofs for distributed algorithms. Dans Fred B. SCHNEIDER, éditeur : *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, BC, Canada, août 1987. ACM Press.

[LW94] B. H. LISKOV et J. M. WING : A behavioral notion of subtyping. *ACM Trans. Prog. Lang. and Sys.*, 16(1):1811–1841, novembre 1994.

[LX01] Edward A. LEE et Yuhong XIONG : System-level types for component-based design. Dans *First Workshop on Embedded Software, EMSOFT2001*, Lake Tahoe, CA, USA, octobre 2001.

[MC81] Jayadev MISRA et K. Mani CHANDY : Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, juillet 1981.

[McM99] Kenneth L. MCMILLAN : Circular compositional reasoning about liveness. Rapport technique, Cadence Berkeley Labs, Berkeley, CA, USA, février 1999.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach et J. Kramer : Specifying Distributed Software Architectures. Dans W. Schafer et P. Botella, éditeurs : *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[MDK94] Jeff Magee, Naranker Dulay et Jeff Kramer. : Regis : A constructive development environment for distributed programs. *IEE/IOP/BCS Distributed Systems Engineering*, 1(5):304–312, septembre 1994.

[Mey91] Bertrand Meyer : *Eiffel : The Language*. Prentice Hall, octobre 1991.

[Mey92] Bertrand Meyer : Applying "design by contract". *Computer*, 25(10):40–51, octobre 1992.

[Mey00] Bertrand Meyer : What to compose. *Software Development*, mars 2000.

[Mic] Microsoft : Shared source cli (codename : Rotor). `http://msdn.microsoft.com/net/sscli/`.

[Mil83] Robin Milner : *A calculus of communicating systems*, volume 158 de *Lecture Notes in Computer Science*. Springer-Verlag, New York-Berlin-Heidelberg, 1983. Also see S-V's LNCS Vol. 92 for paper of same title and author, 1980.

from Steve Stevenson (fpst@hubcap.clemson.edu).

[Mil93] R. Milner : The polyadic pi-calculus : a tutorial. Dans F. L. Bauer, W. Brauer et H. Schwichtenberg, éditeurs : *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993. Available also as Technical Repost ECS-LFCS-91-180, University of Edinburgh, UK, 1991.

[MKB98] Robert Meolic, Tatjana Kapus et Zmago Brezočnik : Testing equivalence with binary decision diagrams. Dans *Proceedings of the 7th Electrotechnical Conference ERK'98*, Portorož, Slovenia, septembre 1998.

[MMC02] Raphaël Marvie, Philippe Merle et Olivier Caron : Le modèle de composant corba. Rapport technique Livrable 1.1-4, Projet RNTL Accord, mars 2002.

[Möl02] Michael Möller : Specifying and checking java using CSP. Dans *Workshop on Formal Techniques for Java-like Programs - FTfJP'2002, Technical Report NIII-R0204*, Computing Science Department, University of Nijmegen, 2002.

[MT00] Nenad Medvidovic et Richard N. Taylor : A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[MW93] J.-J. Ch. Meyer et R. J. Wieringa, éditeurs. *Deontic Logic in Computer Science. Normative System Specification. 1st Int. Workshop on*

*Deontic Logic in Computer Science (DEON '91), Dec. 11-13, 1991, Amsterdam*, Chichester, 1993.

[Nie95] Oscar Nierstrasz : Regular types for active objects. Dans Oscar Nierstrasz et Dennis Tsichritzis, éditeurs : *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[NK95] Keng Ng et Jeff Kramer : Automated support for distributed software design. Dans *Proceedings of 7th International Workshop on Computer-aided Software Engineering (CASE'95),*, Toronto, Canada, juillet 1995.

[NN97] Elie Najm et Abdelkrim Nimour : A calculus of object bindings. Dans *Proceedings of Second IFIP conference on Formal Methods for Open Object-based Distributed Systems – FMOODS'97*, Canterbery, UK, July 1997. Chapman & Hall.

[NNS99a] Elie Najm, Abdelkrim Nimour et Jean-Bernard Stefani : Guaranteeing liveness in an object calculus through behavioral typing. Dans Kluwer, éditeur : *Proceedings of FORTE/PSTV'99*, Beijing, China, octobre 1999.

[NNS99b] Elie Najm, Abdelkrim Nimour et Jean-Bernard Stefani : Infinite types for distributed objects interfaces. Dans *Proceedings of third IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'99*, Firenze, Italy, February 1999. Kluwer.

[OMG01] Unified Modeling Language (UML) 1.5. Object Management Group, document formal/03-03-01, septembre 2001.

[OMG02a] CORBA Component Model, version 3.0. Object Management Group, document formal/2002-06-65, juin 2002.

[OMG02b] Meta-Object Facility MOF, version 1.4. Object Management Group, document formal/2002-04-03, avril 2002.

[OMG03a] UML 2.0 OCL 2nd revised submission. Object Management Group, document ad/03-01-07, janvier 2003.

[OMG03b] UML 2.0 superstructure final adopted specification. Object Management Group, document ptc/03-08-02, août 2003.

[PP99] Franz Puntigam et Christof Peter : Changeable interfaces and promised messages for concurrent components. Dans *Proceedings of the ACM Symposium on Applied Computing (SAC' 99)*, San Antonio, Texas, USA, février 1999.

[PS93] Benjamin C. Pierce et Davide Sangiorgi : Typing and subtyping for mobile processes. Dans *Proceedings 8th IEEE Logics in Computer Science*, pages 376–385, Montreal, Canada, 1993.

[Pun97] Franz Puntigam : Coordination requirements expressed in types for active objects. Dans *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 de *Lecture Notes in Computer Science*, pages 367–387. Springer-Verlag, juin 1997.

[Pun99] Franz PUNTIGAM : Non-regular process types. Dans P. AMESTOY et al., éditeurs : *Proceedings of the 5th European Conference on Parallel Processing (Euro-Par'99)*, numéro 1685 dans LNCS ?, Toulouse, France, 1999. Springer-Verlag.

[PV02] Frantisek PLASIL et Stanislav VIŠNOVSKÝ : Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11), novembre 2002.

[RRV02] A. RAVARA, P. RESENDE et V. VASCONCELOS : An algebra of behavioural types. Preprint, Section of Computer Science, Department of Mathematics, Instituto Superior Técnico, 1049-001 Lisboa, Portugal, 2002. Submitted for publication.

[RV00] António RAVARA et Vasco Thudichum VASCONCELOS : Typing non-uniform concurrent objects. Dans C. PALAMIDESSI, éditeur : *11th International Conference on Concurrency Theory, CONCUR 2000, Proceedings.*, volume 1877 de *Lecture Notes in Computer Science*, pages 474–488, University Park, Pennsylvania, 2000. Springer.

[SD02] Jean-Marc SOUCÉ et Laurence DUCHIEN : État de l'art sur les langages de description d'architecture. Rapport technique Livrable 1.1-2, Projet RNTL ACCORD, juin 2002.

[SG96] M. SHAW et D. GARLAN : *Software Architecture : Perspective on an Emerging Discipline*. Prentice-Hall, 1996.

[Sma03] Patrick SMACCHIA : *Pratique de .NET et C♯*. O'Reilly, juin 2003.

[Ste03] Jean-Bernard STEFANI : A calculus of higher-order distributed components. Rapport technique 4692, INRIA, SARDES project, janvier 2003.

[SUN01] Enterprise JavaBeans Specification, version 2.0. Sun Microsystems, California, USA, août 2001.

[SUN03] Java$^{tm}$ 2 standard edition, specification. Sun Microsystems, document J2SE V1.4.2, California, USA, juin 2003.

[Szy02] Clemens SZYPERSKI : *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 2 édition, novembre 2002.

[Tea97] Rapide Design TEAM : Draft guide to the rapide 1.0 language reference manuals, juillet 1997.

[THK94] Kabu TAKEUCHI, Kohei HONDA et Makoto KUBO : An interaction-based language and its typing system. Dans *Proc. of PARLE'94*, numéro 817 dans Lecture Notes in Computer Science, pages 398–413. Springer-Verlag, 1994.

[TJ92] Jean-Pierre TALPIN et Pierre JOUVELOT : The type and effect discipline. Dans *Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.

[Ves93] Steve VESTAL : A cursory overview and comparison of four architecture description languages. Rapport technique, Honeywell Technology Center, février 1993.

[Viš02] Stanislav Višnovský : *Modeling Software Components Using Behavior Protocols*. Thèse de doctorat, Department of Sofbvare Engineering, Faculty of Mathematics and Physics, Charles University, Prague, 2002.

[von51] G. H. von Wright : Deontic logic. *Mind*, 60(237):1–15, 1951.

[VT93] Vasco T. Vasconcelos et Mario Tokoro : A typing system for a calculus of objects. Dans *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742, pages 460–474. Springer-Verlag, 1993.

[VVR02] Antonio Vallecillo, Vasco T. Vasconcelos et António Ravara : Typing the behavior of objects and components using session types. Dans *1st International Workshop on Foundations of Coordination Languages and Software Architectures (Foclasa 2002)*, Electronic Notes in Theoretical Computer Science. Elsevier, août 2002.

[Xim] Ximian : Projet mono. `http://www.go-mono.com/`.

[Yos02] Nobuko Yoshida : Type-based liveness guarantee in the presence of nonterminatio n and nondeterminism. MCS 2002-20, University of Leicester, April 2002.