Habilitation à diriger des recherches de l'Université Toulouse 1

Spécialité : INFORMATIQUE

par

Rémi Bastide

Spécification comportementale par réseaux de Petri : Application aux systèmes distribués à objets et aux systèmes interactifs

---

Soutenue le 18 janvier 2000 devant la commission d'examen composée de :

G.	**AGHA**	*Professeur, University of Illinois at Urbana-Champaign, USA*
M.F.	**BARTHET**	*Professeur, Université Toulouse I*
M.	**BEAUDOUIN-LAFON**	*Professeur, Université d'Aarhus, Danemark*
J.P.	**BRIOT**	*Directeur de Recherches, CNRS et Université Paris 6*
M.	**DIAZ**	*Directeur de Recherches, LAAS/CNRS, Toulouse*
R.	**GUERRAOUI**	*Professeur, EPFL, Suisse*
J.B.	**STEFANI**	*Directeur de Recherches, FranceTelecom R&D*

Table des matières

1 INTRODUCTION 1

1.1 LES DOMAINES D'APPLICATION 2
1.2 ORGANISATION DU DOCUMENT 3

2 OBJETS ET RESEAUX DE PETRI 5

2.1 OBJETS DANS LES RESEAUX DE PETRI 6
2.2 RESEAUX DE PETRI DANS LES OBJETS 7
2.3 NOTRE FORMALISME DE REFERENCE : LES OBJETS COOPERATIFS 9
2.3.1 Interfaces IDL et classes d'OC 11
2.4 SEMANTIQUE DENOTATIONNELLE DES OBJETS COOPERATIFS 14

3 SPECIFICATIONS COMPORTEMENTALES DE SYSTEMES DISTRIBUES A OBJETS 17

3.1 PRESENTATION DU DOMAINE 19
3.2 LE PROJET SERPICO 19
3.2.1 Problématique scientifique 20
3.2.2 Objectifs 21
3.3 MES CONTRIBUTIONS DANS CE DOMAINE 22
3.3.1 Publications choisies 22
3.3.2 L'environnement PetShop 22
3.3.3 Une expérience de « rétro-spécification » d'un service CORBA 26
3.4 PERSPECTIVES 39
3.4.1 Développement des techniques d'analyse 39
3.4.2 Prise en compte du temps 40
3.4.3 Amélioration de la prise en compte des exceptions 40
3.4.4 Génération de test fonctionnel 40

4 SPECIFICATIONS FORMELLES POUR L'INTERACTION HOMME-MACHINE 43

4.1 PRESENTATION DU DOMAINE 44
4.1.1 Génie du logiciel interactif 44
4.1.2 Spécification du composant "Contrôleur du Dialogue" 45
4.1.3 Conception, implémentation et validation du composant Dialogue 46
4.1.4 Les approches formelles dans les interfaces homme-machine 46
4.1.5 Modélisation de la tâche 47
4.2 LE PROJET ESPRIT MEFISTO 48
4.2.1 Problématique scientifique 48
4.2.2 Résultats attendus 49
4.3 MES CONTRIBUTIONS DANS CE DOMAINE 49
4.3.1 Publications choisies 54
4.4 PERSPECTIVES 60

5 CONCLUSION 63

6 BIBLIOGRAPHIE 65

1 Introduction

Mes travaux de recherche se situent dans le domaine de l'*Application* des réseaux de Petri (RdP), plutôt que dans celui de la théorie des réseaux de Petri à proprement parler. J'ai toujours cherché à développer l'utilisation des RdP dans de nouveaux domaines d'application, en particulier le domaine de l'Interaction Homme-Machine (IHM) et celui des systèmes distribués à objets.

En tant que formalisme de spécification, les réseaux de Petri présentent un certain nombre de qualités très importantes :

- Ils disposent d'une définition formelle : ce caractère formel permet de produire des spécifications exemptes d'ambiguïté ; chaque construction des modèles possède une sémantique parfaitement définie.
- Ils présentent un grand pouvoir d'expression : les RdP sont notamment très bien adaptés à décrire des comportements complexes, réactifs ou concurrents.
- Ils sont exécutables : les modèles peuvent être interprétés par un programme construit à partir de la définition formelle de la notation, ce qui permet de simuler le fonctionnement du système en cours de spécification. Le modélisateur profite ainsi d'une vision dynamique du système qu'il spécifie pour approfondir la compréhension qu'il a de son comportement.
- Ils disposent de nombreuses techniques de vérification automatique des propriétés des modèles. Il est possible de rechercher des propriétés génériques telles que le caractère borné, vivant ou réinitialisable, ou des propriétés spécifiques telles que l'existence d'invariants.
- Ils disposent d'une représentation graphique attrayante, qui accroît la lisibilité et facilite la compréhension des modèles. Cette représentation graphique est également très utile lors de l'exécution interactive des modèles, servant alors de « débogueur » graphique.

Toutefois, malgré ces qualités, il semble que les réseaux de Petri souffrent d'un déficit d'image dans certains cercles de la communauté informatique. Les reproches que l'on entend le plus souvent adresser aux RdP sont :

- Leur manque de structuration : l'utilisation des RdP produirait des modèles dont la taille croît rapidement avec la complexité du système, et qui deviennent rapidement impossibles à comprendre et à gérer.
- Leur difficulté à prendre en compte l'aspect « structure de données » : A l'origine, les RdP étaient essentiellement destinés à décrire la *structure de contrôle* d'un système, son évolution dynamique. Les réseaux Place/Transition échouent en effet à décrire la structure des données manipulées par le système, et la manière dont ces données peuvent influencer le comportement dynamique.

Ces reproches ont certainement été légitimes à une époque, mais ils ont été entendus par la communauté RdP, qui a développé des techniques de structuration des réseaux, et qui a traité

le problème de la prise en compte des données en développant la théorie des réseaux de Petri de haut niveau [Jensen, 96].

Pour que les RdP soient effectivement utilisés dans les domaines d'application que je vise (ingénierie de l'interaction homme-machine et systèmes distribués à objets), il faut :

- Que le pouvoir d'expression de la notation soit suffisant pour permettre de modéliser simplement les problèmes typiques du domaine. Par exemple, dans mes deux domaines d'application, je me place résolument au sein du paradigme « orienté objet ». Il apparaît donc nécessaire de faire en sorte que les constructions de base des systèmes à objet (instanciation, invocation, polymorphisme) soient des primitives de base de la notation, plutôt que de nécessiter la combinaison de constructions plus simple. Chaque domaine d'application apporte de plus ses propres paradigmes, que la notation doit bien évidemment intégrer. Je détaille au paragraphe 3.1 du présent mémoire les spécificités du domaine des systèmes distribués à objet, et en § 4.1 celles du domaine des IHM.
- Que la notation possède de bonnes propriétés de « passage à l'échelle » (scalability). Un bon critère heuristique de scalabilité est que la taille des modèles croisse linéairement avec la complexité des systèmes que l'on souhaite spécifier.

Malheureusement, le pouvoir d'expression d'une notation s'accroît en général au détriment des possibilités d'analyse formelle des modèles. Il est bien connu, par exemple, que certaines propriétés deviennent indécidables dès qu'un formalisme atteint le pouvoir d'expression d'une machine de Turing. Ainsi, l'ajout aux réseaux de Petri d'arcs inhibiteurs (qui permettent de tester *l'absence de jeton* dans une place) accroît sensiblement le confort de modélisation, mais réduit les possibilités d'analyse des réseaux[1] [Peterson, 81]. Ce compromis entre pouvoir d'expression et pouvoir d'analyse est bien connu des praticiens des RdP, et se retrouve d'ailleurs dans toute autre notation à caractère formel ou semi-formel.

Une des préoccupations de mes travaux a toujours été de développer l'utilisation des réseaux de Petri, que ce soit dans le domaine des IHM ou des objets distribués (certains diront que cette préoccupation tourne parfois au prosélytisme). C'est pourquoi, dans ce compromis, j'ai toujours privilégié le pouvoir d'expression par rapport au pouvoir d'analyse. Cette recherche d'une notation puissante et expressive ne s'est jamais faite au détriment de son caractère formel, ni à celui de son caractère exécutable. Par contre, en s'autorisant des domaines de haut niveau pour les jetons, ou des constructions telles que les arcs inhibiteurs, on se prive sciemment de certains résultats d'analyse développés pour des modèles de RdP plus simples. On verra en § 3.4.1 que certains résultats restent exploitables, mais qu'ils donnent parfois une indication sur la présence ou l'absence d'une propriété dans un modèle plutôt qu'une assurance sur cette propriété.

1.1 Les domaines d'application

Mes travaux de recherche portent sur l'application des réseaux de Petri à deux domaines :

[1] Si la place reliée à l'arc inhibiteur est bornée, on peut trouver une construction équivalente qui se dispense de l'arc inhibiteur. Dans ce cas l'arc inhibiteur est simplement du « sucre syntaxique », et n'accroît pas le pouvoir d'expression. Le problème se ramène alors à prouver que la place est bornée.

- L'Approche Objet, et plus particulièrement la conception des systèmes concurrents et distribués à objets. Mes travaux les plus récents portent sur la spécification formelle de systèmes basés sur le standard CORBA.
- L'Interaction Homme-Machine et plus particulièrement la conception de systèmes interactifs critiques.

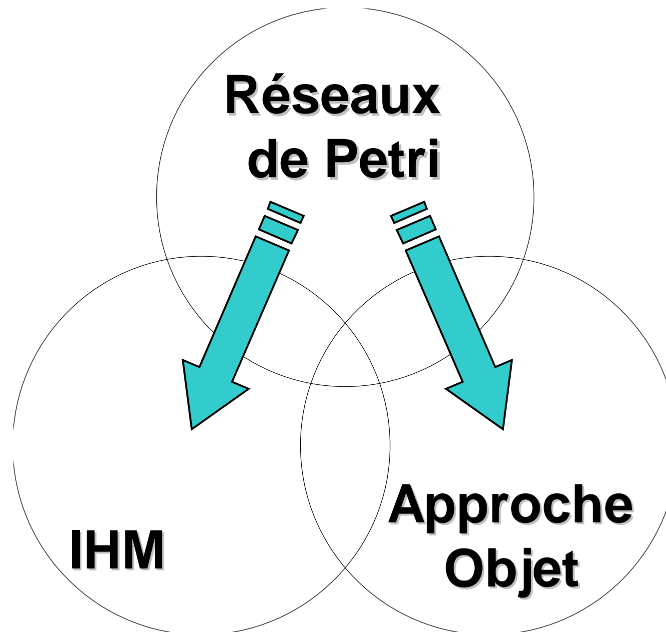


Figure 1 : Mes domaines d'application

Depuis ma thèse, une préoccupation constante de mes activités de recherche a été de tisser des liens que j'espère féconds entre les réseaux de Petri et ces deux domaines d'application.

Ce mémoire s'attache à montrer que la mise en relation de ces trois domaines (RdP, IHM et approche objet) est porteuse de fécondité scientifique, et que ce champ de recherche offre encore de nombreuses pistes prometteuses à explorer.

1.2 Organisation du document

Le mémoire d'habilitation présente tout d'abord (§ 2) les approches sur lesquelles se fonde mon travail : l'approche objet et les réseaux de Petri. Ce chapitre présente également notre formalisme de référence, les Objets Coopératifs (§ 2.3).

Suivent ensuite deux chapitres, dédiés respectivement à mes travaux dans le domaine des systèmes distribués à objets (§ 3) et de l'Interaction Homme-Machine (§ 4). Ces deux chapitres ont la même structure : Ils débutent par une présentation du domaine, présentent un projet caractéristique que nous menons dans ce domaine, présente mes contributions et détaille les perspectives de recherche.

2 Objets et Réseaux de Petri

A partir de la fin des années 80, de nombreux travaux visant à intégrer les concepts orientés-objet avec les réseaux de Petri sont apparus. Il apparaît en effet que ces deux approches présentent des éléments importants de complémentarité :

- L'approche objet est particulièrement bien adaptée à décrire les aspects *statiques* ou *structurels* d'un système. Les concepts fondamentaux que sont l'encapsulation, l'héritage et le polymorphisme permettent de définir des composants logiciels qui sont à la fois fortement cohérents et faiblement couplés avec leur environnement, ce qui en accroît les possibilités de réutilisation. La structuration par classes produit des composants dont la taille reste réduite, et qui n'offrent à leur environnement qu'un accès parfaitement contrôlé et spécifié par l'intermédiaire d'une interface en général fortement typée. Ce n'est donc pas un hasard si l'approche objet est aujourd'hui le paradigme dominant du génie logiciel.
- Les réseaux de Petri, quant à eux, sont particulièrement bien adaptés à décrire les aspects *dynamiques* ou *comportementaux* d'un système. Des concepts tels que la concurrence ou la synchronisation entre activités s'expriment aisément dans le cadre des réseaux de Petri. Les réseaux de Petri, à l'inverse d'autres formalismes, traitent les états et les actions sur un pied d'égalité, et permettent au concepteur de se concentrer sur les états ou sur les actions à différentes phases de son activité. De plus, les RdP présentent des aspects qui les rendent particulièrement bien adaptés à la modélisation des systèmes distribués : Un réseau de Petri modélise naturellement un système dont l'état est distribué, et pour lequel les changements d'états sont locaux. L'état distribué est modélisé par une distribution de jetons dans les places du réseau, et le franchissement d'une transition est complètement caractérisé par le marquage des places qui lui sont directement adjacentes. C'est la localité du franchissement des transitions qui permet aux réseaux de Petri d'exprimer la concurrence vraie, caractéristique des systèmes distribués.

La complémentarité entre objets et réseaux de Petri apparaît alors clairement : une approche par objets nécessite un formalisme puissant et expressif qui lui permette de modéliser le comportement d'un système d'objets qui interagissent. Pour illustrer ce propos, on peut citer UML [Rational Software Corporation, 97], dernier avatar d'une lignée de méthodes d'analyse et de conception par objets. UML s'est dotée d'un formalisme de description du comportement, en l'occurrence une variante des StateCharts [Harel & Gery, 97] pour décrire la dynamique des objets. Il m'arrive de regretter que le choix des pères de la méthode UML ne se soit pas porté plutôt sur les réseaux de Petri, qui bénéficient d'un corpus théorique et méthodologique plus étendu que celui des StateCharts. Ceci est peut être une manifestation du déficit d'image dont souffrent les réseaux de Petri dans certaines communautés.

Les réseaux de Petri, comme tout formalisme qui ambitionne de modéliser des systèmes complexes, se sont très rapidement vus confrontés à la nécessité de se doter de méthodes de structuration. Les premiers travaux dans ce domaine ont tout naturellement cherché à appliquer les principes de décomposition hiérarchique qui étaient alors dominants dans le domaine des langages de programmation. Les chercheurs ont donc développé des notions

telles que les macro-places ou les macro-transitions, ou des notions plus sophistiquées telles que les « pages » que l'on trouve dans les réseaux de Petri colorés de Jensen [Hubert et al., 89]. Ces primitives permettaient en fait d'émuler les notions de « macros » ou de « modules » présents dans de nombreux langages de programmation. Il était donc assez naturel que les RdP suivent l'évolution des langages de programmation, et se tournent vers les techniques de structuration plus élaborées apportées par les langages à objets.

Dans les travaux visant à intégrer objets et réseaux de Petri, deux tendances se sont dégagées, que l'on pourrait résumer par les slogans « Objets dans les réseaux de Petri » et « Réseaux de Petri dans les objets ». Nous verrons que l'approche que j'ai développée dans ma thèse visait à intégrer ces deux tendances

2.1 Objets dans les réseaux de Petri

Cette approche est la directe héritière des travaux menés sur les réseaux de Petri de haut niveau (RdpHN) [Jensen, 96]. Lorsqu'on modélise un système complexe, il est fréquent de constater que le système comporte plusieurs répliques d'éléments fonctionnellement identiques. Dans un atelier de production, par exemple, on rencontrera plusieurs exemplaires de la même machine ou de la même cellule de transport. Quand on modélise un tel système par réseaux Place/Transition classiques, on est pratiquement conduit à faire du « couper-coller » des sous-réseaux qui décrivent le fonctionnement individuel de tels éléments pour les mettre bout à bout et obtenir ainsi un modèle du système complet. Bien entendu, cette pratique est intellectuellement peu satisfaisante et conduit à des modèles difficiles à comprendre et à analyser.
Pour résoudre ce problème, les réseaux de Petri de haut niveau enrichissent la notion de jeton des réseaux Place/Transition : au lieu d'être une entité sans dimension, le jeton peut prendre des valeurs distinctes (des « couleurs ») dans un domaine de valeurs bien défini. Muni de tels jetons, un seul réseau peut alors décrire le fonctionnement d'une « famille » de composants fonctionnellement identiques. Dans le cas de l'atelier flexible cité ci-dessus, ont peut alors « replier » les réseaux des *n* différentes machines en un seul. Pour différentier les différentes machines au sein de ce réseau, les jetons portent une valeur issue d'un domaine énuméré à *n* éléments. Puisque les jetons ne sont plus indifférenciés, les arcs d'un réseau de Petri de haut niveau doivent porter des inscriptions qui permettent de spécifier quels jetons seront pris en compte lors de l'occurrence de la transition.
Les recherches sur les réseaux de Petri de haut niveau ont porté leurs investigations sur des domaines de valeurs plus sophistiqués que de simples énumérations, permettant ainsi au jetons de prendre leur valeur dans de véritables types de donnée. Ainsi, [Vautherin, 87] et [Krämer, 87] utilisent des notations algébriques ou des types abstraits pour décrire les domaines de valeurs des jetons.

L'étape suivante était tout naturellement de considérer des réseaux ayant des « objets comme domaine » comme l'exprime [Battiston & Chizzoni, 96]. Les jetons qui circulent dans le réseau sont alors des instances de classes décrites dans un langage à objets. Ceci conduit parfois à reconsidérer la sémantique de l'occurrence d'une transition : on ne souhaite plus considérer qu'une transition crée ou détruit des objets à chaque occurrence, mais plutôt qu'une transition déplace des objets d'une place à une autre, en appelant éventuellement des méthodes sur les objets déplacés par le franchissement. Bien entendu, une transition peut avoir pour effet d'instancier dynamiquement un objet, ou de le détruire.

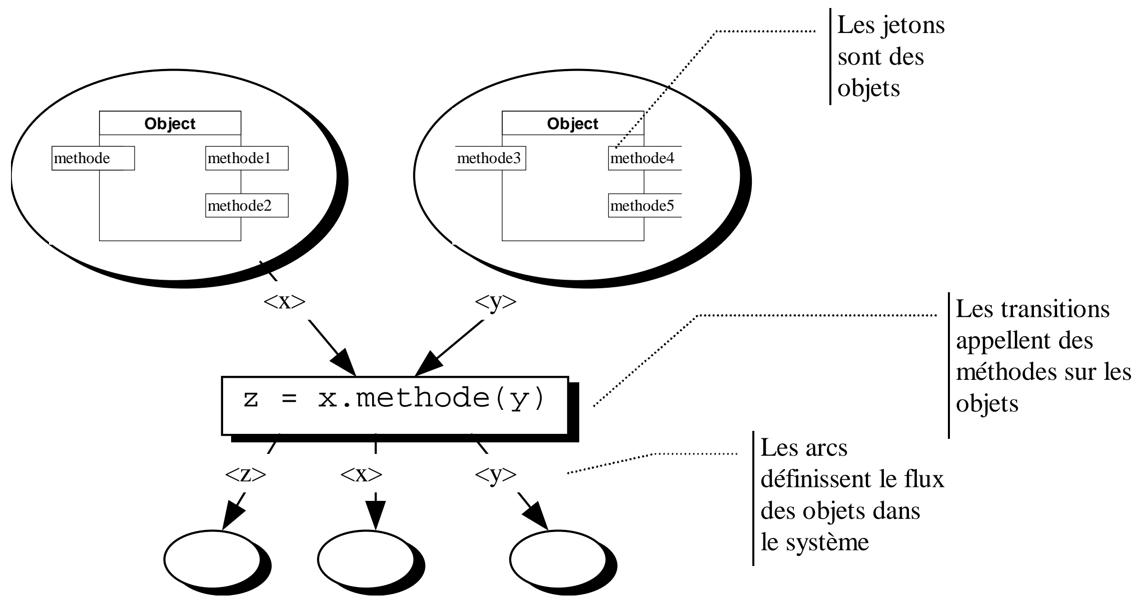


Figure 2 : Le paradigme « Objet dans les réseaux de Petri »

La philosophie sous-jacente de l'approche « objets dans les réseaux » est de modéliser un système par un réseau de Petri unique, qui manipule dans ses jetons des entités structurées porteuses d'information. Ce réseau unique n'est pas nécessairement monolithique, mais peut être structuré en utilisant une décomposition hiérarchique du type macro-transitions ou macro-places. L'important est que le réseau modélise la *structure de contrôle* du système, alors que les jetons et leurs domaines de valeur modélisent la *structure de données* du système. Les domaines des jetons sont décrits dans un formalisme extérieur aux réseaux de Petri, par exemple une notation algébrique ou un langage de programmation [Lakos, 91], [Sibertin-Blanc, 85].

2.2 Réseaux de Petri dans les objets

L'autre tendance majeure d'intégration entre objets et réseaux de Petri consiste à utiliser les réseaux pour décrire le comportement interne des objets. Dans cette approche, le marquage du réseau modélise l'état interne de l'objet et les transitions du réseau modélisent l'exécution d'une méthode par l'objet. Ainsi, la structure du réseau spécifie la disponibilité d'une méthode en fonction de l'état interne de l'objet, et indique les séquences légales d'exécution de méthodes par l'objet. L'intérêt des réseaux de Petri est alors qu'il est très naturel de décrire des objets intrinsèquement concurrents, qui sont capables d'exécuter plusieurs méthodes à la fois. De plus, certaines transitions du réseau peuvent rester « cachées » ou protégées à l'intérieur d'un objet, et modélisent ainsi le comportement interne, spontané d'un objet par opposition aux services qu'il offre à son environnement.

L'intérêt fondamental de ce type d'approche est de permettre l'utilisation des concepts issus de l'approche à objets (classification, encapsulation) pour décrire la structure du système, au lieu d'utiliser une structuration purement hiérarchique.

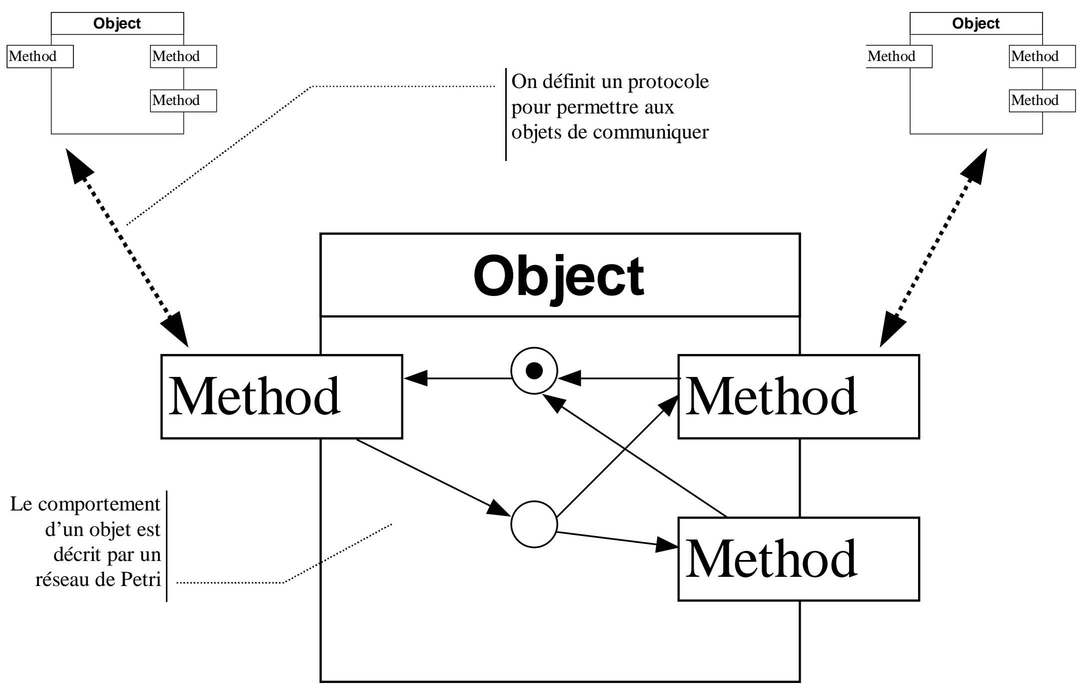


Figure 3 : le paradigme « Réseaux de Petri dans les objets »

Comme le système se présente sous la forme d'un ensemble d'objets (chacun étant décrit par un RdP), il faut bien évidemment permettre à ces différents objets de communiquer. Toutes les approches de cette catégorie définissent donc un protocole de communication entre objets, définissant comment les réseaux de Petri internes aux objets synchronisent leurs activités. Dans des approches « basées objet » qui n'incluent pas des primitives telles que l'instanciation et le polymorphisme, ces protocoles peuvent être très simples, définis par exemple en termes de fusion de places ou de transitions [Paludetto, 91]. Dans des approches orientées objet, ces protocoles de communication doivent gérer le polymorphisme, la liaison dynamique et l'instanciation dynamique d'objets. Ces protocoles peuvent être définis de manière idiosyncrasique pour une approche particulière. Ainsi CO-OPN2 [Buchs & Biberstein, 95] définit des opérateurs de simultanéité (//), de séquence (…) et d'alternative (+) pour la communication entre objets. Les protocoles peuvent eux-mêmes être définis directement en termes de réseaux de Petri. Par exemple des protocoles de type « envoi de message asynchrone » ou « question-réponse » à la mode client-serveur se spécifient aisément en termes de RdP ([Ramamoorthy & Ho, 80], [Sibertin-Blanc, 93], [Jensen, 96] Volume 1 p 38-39).

Dans le paradigme « Réseaux de Petri dans les objets », un système est décrit comme un ensemble d'objets qui communiquent, le comportement de chaque objet étant décrit en termes de réseaux de Petri. Le plus souvent, ces approches sont à base de classe, ce qui permet d'associer un RdP à une classe d'objets plutôt qu'à un objet individuel. Le problème de ces approches est toujours de se doter d'une sémantique bien définie, qui permette de définir le comportement dynamique de cet ensemble d'objets en combinant les comportements individuels décrits dans les classes et les protocoles qui permettent aux objets de coopérer. Il faut notamment veiller à ce que cette sémantique respecte bien un principe fondamental des réseaux de Petri, à savoir la localité de l'état et du franchissement des transitions.

2.3 Notre formalisme de référence : les Objets Coopératifs

J'ai défini dans ma thèse [Bastide, 92] le formalisme des Objets Coopératifs, que l'on peut présenter comme une unification des paradigmes « Objets dans les RdP » et « RdP dans les Objets ». L'approche objet nous fournit les concepts suffisants pour définir la structure des objets et leurs interrelations, afin de structurer le système suivant les principes de forte cohésion et de faible couplage ; la théorie des réseaux de Petri, quant à elle nous permet de modéliser le comportement des objets et les communications qu'ils entretiennent, de telle sorte que l'on puisse exprimer la concurrence aussi bien entre les différents objets qu'au sein même d'un objet.

L'idée de base de notre approche est de permettre aux jetons de contenir des *références* vers d'autres objets du système. Comme le comportement des objets est décrit par réseaux de Petri, on obtient donc des réseaux qui se référencent mutuellement[2].

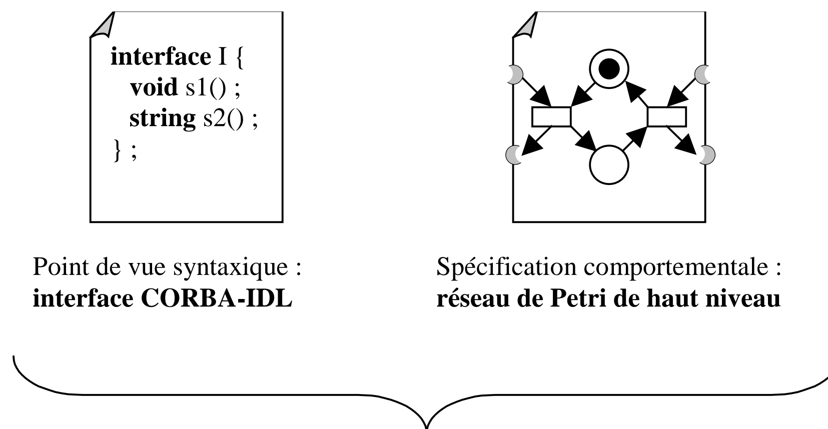


Figure 4 : Structure d'une classe d'Objets Coopératifs

Une classe d'OC spécifie une classe d'objets en décrivant leur **interface** et leur **comportement** :

- L'interface décrit les aspects syntaxiques de l'utilisation d'un objet de cette classe. Elle spécifie quels sont les services offerts par cet objet, et quelle est la signature de ces services. L'interface d'une classe d'OC est décrite en termes du langage de définition d'interface (Interface Definition Language ou IDL) de CORBA[3] (Common Object Request Broker Architecture).

[2] Une autre voie consiste à définir des jetons qui sont eux-mêmes des réseaux de Petri, et non pas des références vers d'autres réseaux. Cette voie a été suivie par d'autres chercheurs [Valk, 98]. Nous considérons que, même si cette approche peut donner des résultats théoriques intéressants, elle s'éloigne trop des concepts de base de l'approche objet. Dans une très large classe de problèmes, en effet, il est naturel de modéliser des objets référencés par plusieurs autres objets du système. C'est le cas de la quasi-totalité des systèmes client-serveur, où un serveur est en général utilisé simultanément par plusieurs clients.

[3] Dans les premières publications présentant les Objets Coopératifs (y compris certaines des publications incluses dans ce mémoire), l'interface d'une classe d'OC était décrite par un IDL idiosyncrasique, inspiré du langage Eiffel. Dans les publications plus récentes, nous avons adopté CORBA-IDL, d'une part à cause de son caractère standard, et d'autre part parce que nos travaux se sont orientés vers la spécification de systèmes distribués à objets, et qu'il était donc souhaitable de se rapprocher du standard CORBA.

- Le comportement d'une classe d'objet est appelé son ObCS (pour Object Control Structure, terme emprunté à la méthode HOOD). L'ObCS décrit les aspects sémantiques et comportementaux d'un objet de cette classe. L'OBCS décrit le cycle de vie des objets de la classe. Il spécifie dans quelle mesure l'invocation de services sur ces objets dépend de leur état interne, et réciproquement comment cet état est modifié par les invocations de services. Les objets que nous décrivons sont intrinsèquement concurrents : ils sont capables de répondre simultanément à plusieurs invocations de service. L'ObCS décrit donc les contraintes de synchronisation propres à cet objet, induites par son caractère concurrent.

La définition formelle des Objets Coopératifs, énoncée pour la première fois dans ma thèse, est reprise dans plusieurs articles joints à ce mémoire. L'article [Bastide et al., 99a], notamment, contient une version de cette définition qui prend en compte les développements récents du formalisme relatifs à CORBA, notamment la description des exceptions. Je rappelle simplement ici les caractéristiques principales de ce formalisme.

L'ObCS d'une classe d'OC est défini par un réseau de Petri de haut niveau. Plus précisément :

- Le système de types[4] (noté *TypeSet*) de ce réseau est défini en termes de CORBA-IDL : CORBA-IDL offre un ensemble de types de base prédéfinis (long, double, string …), permet de définir des types construits (sequence, array, struct) et surtout permet de décrire des types d'objets en donnant leur interface. Les variables dont le type est une interface IDL seront appelées des *références*.
- Les jetons (Tokens) sont des n-uplets de valeurs typées dans le système de types défini ci-dessus. L'arité d'un jeton est le nombre de valeurs qui le composent, et les jetons d'arité zéro correspondent aux jetons conventionnels des réseaux de Petri simples. On appelle Token-type un n-uplet de types, décrivant les types individuels des valeurs qui composent un jeton. Les Token-types sont notés $<Type_1, \ldots\ Type_n>$ ($Type_1, \ldots\ Type_n \in$ TypeSet) ou simplement <> pour dénoter le type des jetons d'arité zéro.
- Un Token-type est associé à chaque place du réseau, et la place ne peut recevoir que des jetons de ce type. Une place stocke un multi-ensemble de jetons, un jeton donné peut donc être présent plusieurs fois dans la même place.
- A chaque arc est associé un n-uplet de variables, avec une multiplicité donnée. L'arité d'un arc est celle de son n-uplet de variables, nécessairement identique à celle de la place à laquelle l'arc est connecté. Le type de chaque variable est déduit du Token-type de la place. La multiplicité d'un arc définit le nombre de jetons identiques qui seront traités lors du franchissement de la transition à laquelle l'arc est connecté. La forme générale de l'inscription attachée à un arc est multiplicité*<v1,… vn>. Une multiplicité égale à 1 peut être omise (ainsi 1*<v1,… vn> s'abrège en <v1,… vn>) une liste vide de variables peut également être omise (ainsi 2*<> s'abrège en 2)

[4] Le système de types de l'ObCS est équivalent à l'ensemble des *color sets* pour les réseaux colorés de [Jensen, 96].

- Les transitions ont une précondition (expression booléenne de leurs variables d'entrée[5]) et une action, qui peut faire appel à tout service défini par les variables d'entrée. La portée de chaque variable est locale à la transition.

Une transition est franchissable si :
- On peut construire une substitution des variables d'entrée par des valeurs présentes dans les places d'entrée de la transition.
- La multiplicité de chaque jeton substitué dans les places d'entrée est supérieure ou égale à la multiplicité de l'arc d'entrée,
- La précondition de la transition s'évalue à vrai pour la substitution considérée.

Le franchissement d'une transition exécute l'action de la transition, calcule de nouveaux jetons et les dépose dans les places de sortie de la transition. Les OC offrent deux extensions d'arcs [Lakos, 94]: les arcs de test et les arcs inhibiteurs.

2.3.1 Interfaces IDL et classes d'OC

La Figure 5 illustre une interface CORBA-IDL très simple[6]. Il s'agit d'un tampon qui stocke des messages (en l'occurrence des chaînes de caractères). Les mots en caractère gras sont des mots réservés de CORBA-IDL. Cette interface propose deux services :
- put, qui prend comme paramètre d'entrée un message ;
- get, qui renvoie un message.

```
interface Tampon {
     void put(in string message);
     string get();
}
```

Figure 5 : Tampon - une interface CORBA-IDL

Même une interface aussi simple ne peut pas se dispenser d'une description fonctionnelle ou comportementale. Il convient de répondre à nombre de questions, parmi lesquelles :
- Que se passe-t-il lorsqu'on invoque get() sur un tampon vide ?
- Les messages sont-ils délivrés par get() dans un ordre particulier (par exemple, dans leur ordre d'arrivée) ?
- Le tampon a-t-il une taille maximum ?

On pourrait tenter de répondre à ces questions en associant à cette interface une spécification fonctionnelle en langage naturel (c'est le choix que fait l'OMG pour les services qu'il

[5] Les préconditions portent uniquement sur les variables de types primitifs, de manière à pouvoir être évaluées localement, et conserver ainsi le principe de localité des réseaux de Petri. Il est notamment interdit d'invoquer un service sur une variable de type référence.

[6] Je m'excuse auprès des lecteurs du caractère trivial de cet exemple : je suis bien conscient qu'un reproche souvent fait aux approches formelles est d'être limitées à traiter des exemples de taille réduite. Ainsi on attribue à Bertrand Meyer cette citation cruelle : « *Les Types Abstraits sont un formalisme excellent, destiné à modéliser la pile* ». L'exemple donné ici a uniquement pour but d'illustrer les aspects syntaxiques des Objets Coopératifs. Le formalisme a été utilisé pour traiter des problèmes plus ambitieux (voir par exemple l'article [Bastide et al., 99b] joint à ce mémoire, § **Erreur ! Source du renvoi introuvable.**). La capacité des OC à traiter des problèmes de taille industrielle est discutée en § 3.3.3.

spécifie, comme nous en discutons de manière approfondie en § 3.3). On pourrait également envisager d'utiliser une spécification plus abstraite, comme par exemple une spécification algébrique [Guttag et al., 93]. Le paragraphe 3.1 discute des limitations de ce type d'approches dans le cadre des systèmes distribués. Dans le formalisme des Objets Coopératifs, la spécification comportementale associée à cette interface sera donnée en termes de réseaux de Petri de haut niveau.

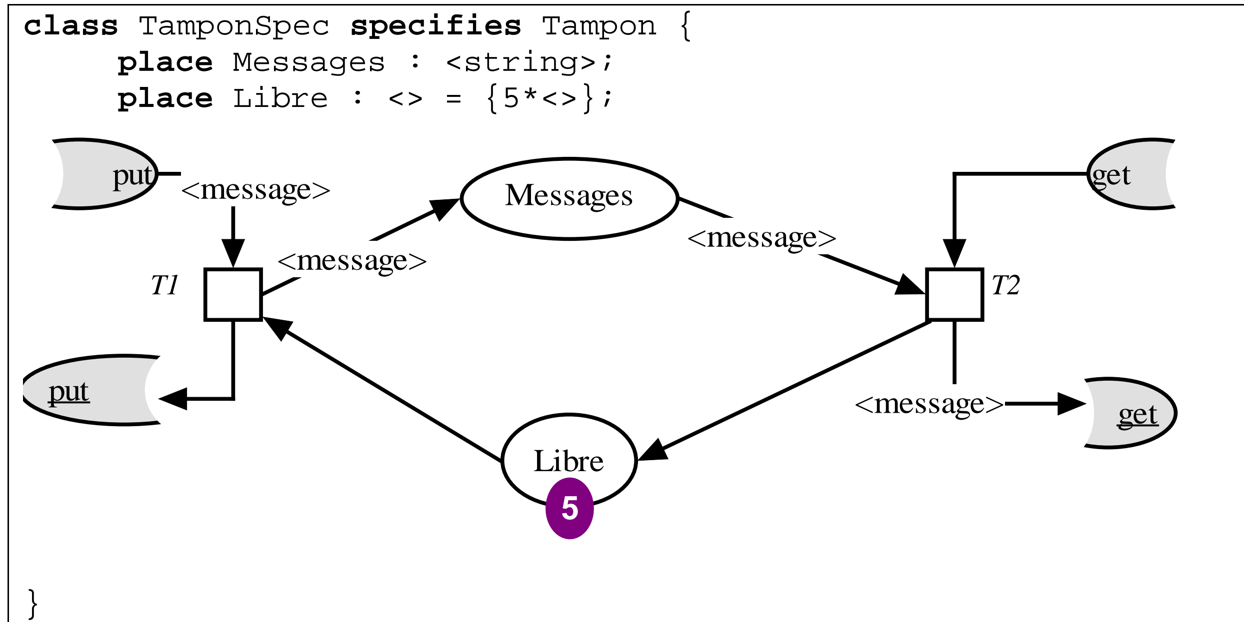


Figure 6 : TamponSpec - Une classe d'OC qui spécifie l'interface Tampon

Une classe d'OC spécifie le comportement d'une ou plusieurs interfaces IDL, de même qu'une classe Java, par exemple, peut implémenter une ou plusieurs interfaces. La classe d'OC décrite en Figure 6 spécifie une seule interface, en l'occurrence *Tampon*. Le mot clé *specifies* est suivi de la liste d'interfaces IDL spécifiées par la classe d'OC.

Description des services. Chaque service *op* défini dans une interface IDL est décrit par deux ou trois places dans l'ObCS : On trouve systématiquement un port d'entrée (*SIP*, pour *Service Input Port*) noté op, et un port de sortie (*SOP*, pour *Service Output Port*) noté op. Dans le cas ou le service peut lever une exception, on trouve également un port d'exception (*SEP* pour *Service Exception Port*) noté ~~op~~. Ces trois places sont directement déduites de l'IDL, comme suit :

- Le Token-type du SIP est la concaténation des types IDL des paramètres *in* ou *inout* du service.
- Le Token-type du SOP est la concaténation :
  - du type IDL du résultat du service (dans le cas ou le service renvoie un résultat)
  - de la liste des paramètres *out* et *inout* du service.
- Le Token-type du SEP est <*Exception*>, où *Exception* est le super-type de toutes les exceptions IDL.

Ainsi, conformément à l'interface décrite en Figure 5, le service *get* se trouve traduit par deux places : *get* pour le SIP et get pour le SOP.

L'invocation d'un service dépose un jeton contenant tous les paramètres *in* et *inout* dans le SIP. Le rôle de l'ObCS est d'appliquer un traitement aux valeurs contenues dans ce jeton,

pour finalement déposer un jeton contenant le résultat du service (plus les éventuels paramètres *out* et *inout*) dans le SOP. Si une exception est levée durant le traitement, aucun jeton ne sera déposé dans le SOP, mais un jeton décrivant l'exception sera déposé dans le SEP.

La Figure 6 illustre à la fois la syntaxe graphique d'une classe d'OC, et les annotations textuelles nécessaires à compléter sa description. Ces annotations textuelles sont :

- La liste des interfaces IDL spécifiées par la classe d'OC (mot clé *specifies*). Dans ce cas, la classe *TamponSpec* spécifie l'interface *Tampon*.
- La description du Token-type des places. Par exemple, la place *Messages* contient des n-uplets de la forme <string> qui représentent un message précédemment déposé dans le tampon. La place *Libre* contient des jetons sans dimension, notés <>. Le type des SIP, SOP et SEP n'a pas à être décrit, puisqu'il est directement déduit de la signature IDL du service associé. Ainsi, le type du SIP *put* et celui du SOP *get* sont-ils *<string>*.
- Le marquage initial des places : Ceci permet de décrire l'état initial des instances au moment de leur instanciation. Dans l'exemple la place *Libre* contient initialement 5 jetons.
- La description des préconditions et des actions pour les transitions. Par défaut, une action est vide et une précondition s'évalue identiquement à vrai. Dans l'exemple de la Figure 6, toutes les actions et toutes les préconditions sont vides.

La place *Libre* modélise les emplacements libres dans le tampon. Une occurrence de la transition *T1* consomme un jeton dans cette place, une occurrence de la transition *T2* y redépose un jeton. Dans le cas ou un client invoque "*put*" quand la place Libre est vide, il est bloqué jusqu'à ce qu'un emplacement se libère lorsqu'un autre client invoquera "*get*".

La place *Message* modélise les messages reçus par "*put*", et non encore transmis par "*get*". Le réseau présente un invariant : | *Messages* | + | *Libre* | : le nombre total de jetons dans ces deux places est constant (en l'occurrence, 5 jetons, le marquage initial de Libre).

La Figure 6 est une spécification fonctionnelle complète de l'interface décrite en Figure 5. Elle apporte notamment une réponse à toutes les questions mentionnées plus haut :

- Que se passe-t-il lorsqu'on invoque get() sur un tampon vide ? Cette invocation est matérialisée par l'arrivée d'un jeton dans la place *get*. Dans ce cas, la transition T2 n'est pas franchissable, et ne le deviendra que quand un jeton aura été déposé dans la place Message après l'invocation du service put(). Le client qui a initié l'invocation de get est alors bloqué jusqu'à ce qu'un message soit entré dans le tampon.
- Les messages sont-ils délivrés par get() dans un ordre particulier (par exemple, dans leur ordre d'arrivée) ? La spécification donnée en Figure 6 est indéterminisme : si la place Message contient plusieurs jetons, le jeton choisi pour le franchissement de T2 n'est pas spécifié. Aucun ordre des messages n'est donc préservé. Il serait aisé de spécifier que les messages sont fournis dans leur ordre d'arrivée, ou par ordre alphabétique…
- Le tampon a-t-il une taille maximum ? Compte tenu du marquage initial de la place *Libre*, le nombre de messages dans le tampon est limité à 5.

La communication entre objets est décrite par des transitions d'invocation, dont l'action et l'invocation d'un service sur un des objets capturés par l'occurrence de la transition. L'invocation entre objets est synchrone, et la sémantique des invocations repose sur un protocole client/serveur décrit en termes de réseaux de Petri. Les détails techniques de ce protocole (et notamment la manière dont il peut être étendu pour prendre en compte le

déclenchement d'exceptions lors de l'invocation) sont donnés dans [Bastide et al., 99a], article joint à ce mémoire (§ **Erreur ! Source du renvoi introuvable.**).

Certaines contraintes structurelles doivent être respectées par les ObCS pour que les invocations soient traitées correctement. Informellement, on veut exprimer le fait que pour chaque invocation, l'objet fournira soit un résultat, soit une exception, et qu'il ne fournira de résultats que s'il a été préalablement invoqué. Ceci se traduit dans l'ObCS par le fait que l'arrivée d'un jeton dans un SIP produira nécessairement un jeton et un seul soit dans le SOP soit dans le SEP. Ces contraintes structurelles s'expriment aisément en termes de réseaux de Petri, et peuvent être vérifiées automatiquement. La description détaillée de ces contraintes se trouve également dans [Bastide et al., 99a], où on définit également la notion d'OpCS (Operation Control Structure). Informellement, l'OpCS d'un service est un sous-réseau de l'ObCS de la classe, comprenant les places SIP, SOP et SEP de ce service.

Depuis sa création, notre formalisme à subi quelques modifications, le plus souvent dans le sens de la simplification. Ainsi quelques constructions peu orthogonales ou à l'utilité méthodologique douteuse ont-elles été abandonnées (objets composites, accès aux attributs…). D'autre part, le formalisme s'est rapproché autant que possible de la pratique industrielle, notamment en adoptant de standard CORBA-IDL pour décrire l'interface des classes d'OC au lieu de définir son propre IDL. D'autres modifications purement syntaxiques ont été apportées, comme par exemple la représentation graphique de certains éléments du formalisme.

2.4 Sémantique dénotationnelle des Objets Coopératifs

Un des enjeux principaux de ma thèse était de donner aux constructions typiques des langages à objets (instanciation dynamique, héritage, polymorphisme) une sémantique fondée sur les réseaux de Petri. Le caractère dynamique des systèmes à objet prend plusieurs dimensions :

- Instanciation dynamique de nouveaux objets : Tous les objets ne sont pas prédéfinis, mais au contraire des objets peuvent être créés dynamiquement ou détruits pendant l'activité du système. On veut pouvoir décrire cette création dynamique d'objets en termes de réseaux de Petri sans pour autant avoir des réseaux dont la structure évolue dynamiquement.
- Dynamicité de la relation d'utilisation entre objets : On a une relation d'utilisation entre un couple d'objets A et B lorsque A possède une référence vers B. A est alors en position d'invoquer un des services offerts par B par l'intermédiaire de la référence qu'il possède. Dans un système orienté objet, cette relation d'utilisation n'est pas statique, mais évolue constamment lorsque des objets se transmettent des références, en tant que paramètres ou résultats des invocations de services. En termes de réseaux de Petri, cela se caractérise par le fait que différentes occurrences d'une même transition peuvent concerner des références différentes, qui dénotent donc des réseaux différents.
- Liaison dynamique : le fait de prendre en compte les notions d'héritage et de polymorphisme conduit au fait que la classe exacte d'une référence n'est, en général, connue qu'au moment de l'exécution. Une référence déclarée comme étant d'une classe A peut prendre ses valeurs parmi les instances de n'importe quelle classe dérivée de A.

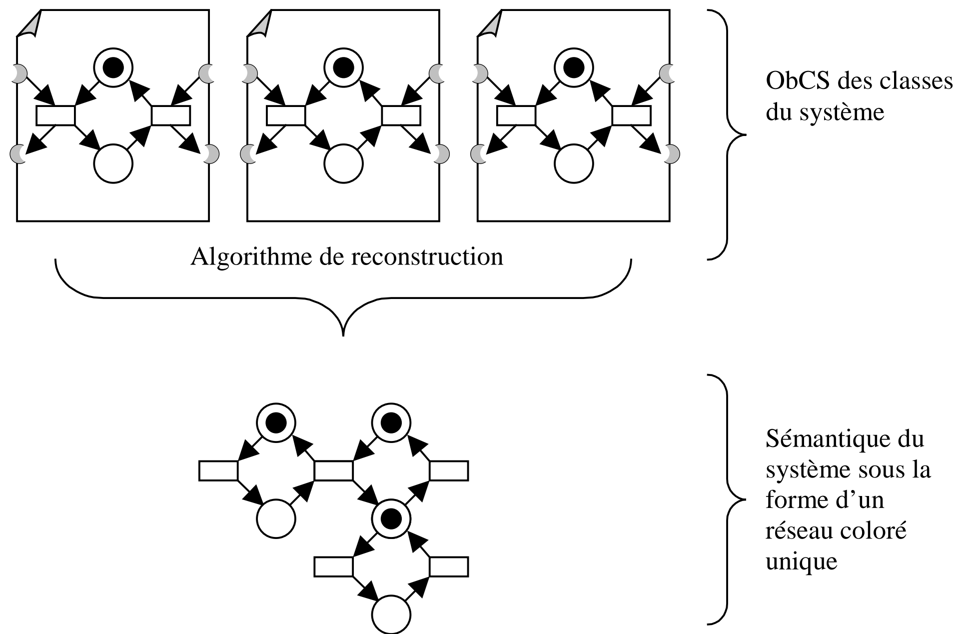


Figure 7 : Principe d'une sémantique dénotationnelle pour les Objets Coopératifs

Un des résultats principaux de ma thèse a été de démontrer que les concepts principaux de l'approche objet (encapsulation, héritage, instanciation dynamique, polymorphisme) peuvent se définir en restant strictement dans le cadre de la théorie des réseaux de Petri de haut niveau. Cette démonstration se fait dans le style de la sémantique traductionnelle ou dénotationnelle [Meyer, 92] : D'après B. Meyer, « *l'idée de la sémantique traductionnelle est d'exprimer la signification d'un langage par un schéma de traduction qui, pour tout programme dans ce langage, produit un programme dans un langage plus simple* ». Bien entendu, cette approche pose le problème de la sémantique du langage cible lui-même. Dans notre cas, le langage cible est celui des réseaux de Petri colorés [Jensen, 96] augmenté de quelques extensions d'arcs [Lakos, 94] (arcs inhibiteurs et arcs de test). La sémantique de ces derniers étant bien définie, notre schéma de traduction se rapproche davantage d'une sémantique dénotationnelle qui « *exprime la sémantique d'un langage dans un schéma de traduction qui associe une signification (dénotation) à chaque programme du langage* » (Meyer, op. cit.), la dénotation étant alors un objet mathématique. J'ai décrit dans ma thèse un algorithme qui, à partir des ObCS des différentes classes d'OC présentes dans un système, construit un réseau de haut niveau unique, non structuré, qui modélise le comportement du système dans son ensemble.

Dans la pratique, cette génération du réseau global n'est faite qu'à des fins d'analyse, afin d'explorer les propriétés du système modélisé dans son ensemble. La définition formelle de cet algorithme fait l'objet du chapitre V de ma thèse [Bastide, 92].

Cette possibilité d'obtenir un réseau de Petri global qui représente le fonctionnement du système dans son ensemble est de peu d'intérêt pratique : la structure de ce réseau est nécessairement beaucoup moins lisible que celle des classes initiales, et le concepteur aura donc du mal a s'y référer. En ce qui concerne l'exécution dynamique des modèles, nous disposons déjà dans l'environnement PetShop (§ 3.3.2) d'un interprète réparti qui exécute directement les ObCS des classes en les faisant communiquer dans un environnement distribué via CORBA. Il est donc plus intéressant d'effectuer cette simulation distribuée que d'effectuer une simulation centralisée en interprétant le réseau global.

Si l'intérêt pratique de la reconstruction du réseau global est limitée, son intérêt théorique, par contre, est très grand : il nous permet d'utiliser les techniques d'analyse développées dans le cadre des réseaux colorés (non orientés-objet) pour vérifier des propriétés sur le système modélisé. L'outil Design/CPN par exemple ([Jensen, 96] vol 2) offre un module de génération de graphes d'occurrence sur des modèles de RdP colorés. A partir de ces graphes d'occurrence, de nombreuses propriétés peuvent être vérifiées. Un problème demeure, toutefois : comme les propriétés sont évaluées sur le réseau global du système, leur interprétation en termes du comportement des classes individuelles peut être difficile. Cet effort sur l'applicabilité des techniques d'analyse fait partie des perspectives de notre travail (§3.4.1).

3 Spécifications comportementales de systèmes distribués à objets

Les systèmes distribués à objet ne sont plus depuis quelques années des rats de laboratoire, mais bel et bien des outils couramment utilisés pour le développement d'applicatifs ambitieux et opérationnels.

CORBA (Common Object Request Broker Architecture) est un standard proposé par l'Object Management Group (OMG) pour promouvoir l'interopérabilité entre les systèmes à objets distribués. Le développement d'un tel standard témoigne que le domaine des technologies distribuées à objets a maintenant acquis une maturité industrielle, et motive de nouvelles recherches sur l'ingénierie de tels systèmes.
Un des composants principaux de CORBA est l'Interface Definition Language (IDL), qui permet de spécifier l'interface des objets du système. CORBA-IDL est indépendant de tout langage de programmation (bien qu'il soit clairement dérivé de C++), et orienté objet, permettant la spécialisation d'interfaces par héritage. Une interface IDL définit au niveau syntaxique les services offerts par une classe d'objets, en spécifiant leur signature : liste des paramètres reçus, mode de passage de ces paramètres, valeur de retour, exceptions pouvant être levées durant l'exécution du service.

Une limitation reconnue de CORBA est que les classes d'objets sont définies seulement en termes de leur interface. CORBA-IDL couvre uniquement les aspects syntaxiques de l'utilisation d'un objet distant, et ne fournit aucune description sémantique ou comportementale de ces objets, alors qu'une telle description est bien évidemment indispensable aux utilisateurs potentiels de l'objet : un composant CORBA, lors de son existence, suit un cycle de vie spécifique, qui caractérise dans quel ordre et sous quelles contraintes il est susceptible de répondre aux séquences de services qui vont lui être soumises par ses clients. Pour pouvoir utiliser un composant CORBA, il faut connaître non seulement son interface (les services qu'il offre et leur signature) mais également son comportement.

Une spécification comportementale doit en particulier décrire :

- Les préconditions et postconditions de l'invocation d'un service, et plus généralement les contraintes sur les séquences d'invocation des services offerts.
- Une relation abstraite entre les paramètres reçus et les résultats fournis par une invocation.
- Les contraintes de concurrence propres à l'objet distant : l'objet supporte-t-il des invocations concurrentes ou requiert-il une sérialisation des invocations ?
- Les conditions qui peuvent amener à la levée d'une exception durant l'exécution d'un service.

CORBA bénéficierait d'une spécification de la sémantique associée à une interface IDL, fournie à un niveau d'abstraction suffisant pour ne pas apporter de contraintes sur l'implémentation de l'objet. On pourrait ainsi fournir aux objets concurrents et distribués un

support conceptuel comparable a celui que les types abstraits de données (TAD) apportent à la spécification d'un type de données séquentiel.

Notre but est de définir une notation formelle adaptée à la spécification de systèmes à objets distribués conformes au standard CORBA. Nous voulons pouvoir décrire le comportement d'un ensemble d'objets qui interagissent, et pas seulement le comportement d'un objet isolé. Tout formalisme poursuivant cet objectif se doit de remplir certaines contraintes :

- Traiter les flots de données sur un pied d'égalité avec les flots de contrôle : le formalisme doit pouvoir décrire des échanges de valeurs typées, et pas seulement de pures relations de causalité. En effet, le comportement d'un objet CORBA dépend fréquemment non seulement de l'historique des invocations précédentes, mais aussi des valeurs échangées durant ces invocations. Pour un objet dans un état donné, une invocation peut réussir ou échouer selon la valeur des paramètres de l'invocation. Les formalismes qui ne prennent pas en compte les valeurs typées (tel que les réseaux de Petri simples, par exemple) ne sont donc pas adaptés à notre objectif.
- Prendre en compte la dynamicité des références : toute spécification formelle d'un système CORBA doit respecter son modèle d'objets de base, et plus précisément doit permettre de désigner des objets distants par l'intermédiaire de références, qui peuvent être échangées lors des invocations. La topologie de la relation de référence entre objets est donc dynamique, d'autant plus que de nouveaux objets peuvent être instanciés pendant l'activité du système. Les approches algébriques CCS ou CSP présupposent une topologie statique des canaux entre processus, et sont donc mal adaptées à notre objectif. De même, certaines approches qui combinent réseaux de Petri et objets ([Valk, 98], [Lakos & Keen, 94]) évitent de traiter le problème des références, et considèrent les jetons comme des objets, et non pas des références à des objets. Des approches algébriques plus récentes [Gaspari & Zavattaro, 99] s'affranchissent de cette limitation et visent explicitement le domaine d'application de CORBA.
- Permettre de spécifier des objets dont le comportement interne est concurrent. Ce point est particulièrement important, car les objets CORBA sont souvent des entités partagées par un grand nombre de clients, qui offrent des services dont l'exécution peut réclamer un délai non négligeable. Il est donc irréaliste d'exiger que chaque service s'exécute de manière atomique, conduisant à ce qu'au plus un service soit actif à tout instant pour un objet donné. De fait, tous les environnements CORBA (ORB) disponibles actuellement permettent d'implémenter des objets « multi-fils », capables d'exécuter plusieurs services simultanément. Plusieurs propositions ont été faites pour étendre la théorie des types abstraits de données au domaine des objets concurrents et distribués. La plupart de ces approches considèrent un objet comme un moniteur, permettant un seul service actif à la fois, et donc ne sont pas adaptées à la spécification d'objets CORBA multi-fils.
- Etre adapté à la fois aux implémenteurs de la classe serveur dérivée de l'IDL, et aux concepteurs de systèmes qui utiliseront la classe ainsi spécifiée. Pour les premiers, la spécification comportementale doit être suffisamment complète et précise pour leur indiquer de manière non ambiguë quel comportement ils doivent implémenter, dans le langage de programmation de leur choix. La spécification doit d'autre part être suffisamment abstraite pour ne pas contraindre les choix d'implémentation du programmeur. Les utilisateurs du serveur, d'autre part, doivent pouvoir utiliser la spécification comportementale pour comprendre sans ambiguïté la sémantique de chacun des services offerts.

3.1 Présentation du domaine

La spécification comportementale des systèmes à objets est un champ de recherches à part entière [Kilov & Harvey, 96]. En ce qui concerne plus spécifiquement les systèmes distribués à objets, de nombreux chercheurs ont reconnu l'importance de fournir des spécifications comportementales adaptées au modèle objet de l'OMG. Ainsi, Sriram Sankar [Sankar, 96] mentionne, non sans une certaine malice, que l'usage de méthodes formelles peut au moins aider à maintenir « le niveau actuel de qualité du logiciel », alors que la complexité des systèmes à construire s'accroît du fait de la présence de composants distribués. Il note également que le surcoût induit par l'usage des approches formelles est souvent mieux accepté dans le domaine des systèmes distribués, où il apparaît proportionnellement moindre.

Plusieurs auteurs ont tenté d'adapter aux systèmes à objets distribués le principe de la « programmation par contrat », popularisé par le langage Eiffel. Toutefois, par leurs fondements basés sur la théorie des Types Abstraits de Données, ces approches font souvent l'hypothèse implicite que l'exécution d'un service par un objet est toujours atomique. Ainsi, pour [Meyer, 93] « *Tout client accédant à un objet par l'intermédiaire d'un service doit avoir la garantie d'un accès exclusif à l'objet pendant toute la durée de l'appel. Le niveau le plus fin de granularité pour l'accès exclusif à un objet est l'exécution d'un service* ». Bien qu'une telle limitation puisse être légitime dans le domaine des systèmes concurrents, il nous apparaît qu'elle présente de sérieux inconvénients dans le domaine des systèmes distribués tels que CORBA. La plupart des méthodes fondées sur l'utilisation des pré et post-conditions considèrent implicitement tout objet comme un moniteur [Hoare, 85] permettant au plus à un service d'être actif à tout instant, et donc se prêtent mal à la description de serveurs CORBA présentant une concurrence interne.

Les travaux de [Bryan, 96] se placent dans cette catégorie. Bryan propose un langage appelé ASL (Architecture Specification Language) qui inclut la description comportementale, ainsi que d'autres constructions destinées à la description d'architectures logicielles. La notation de spécification comportementale, fondée sur les pré et post-conditions, ne permet pas de décrire de la concurrence interne aux objets. Les travaux de [Sankar, 96], qui utilise le langage textuel Borneo, présentent les mêmes limitations. Le langage de spécification LARCH [Guttag et al., 93] a été également utilisé pour la spécification comportementale d'interfaces CORBA-IDL. Ainsi, les travaux de [Leavens & Cheon, 95] ne traitent pas de la concurrence intra-objet, alors que ceux de [Sivaprasad, 95] la prennent en compte par l'introduction d'opérations atomiques et non-atomiques. Certains auteurs proposent la notation Z [Spivey, 94] comme formalisme de spécification comportementale pour les objets distribués [Bryant & Evans, 96], mais les possibilités de Z pour décrire des comportements concurrents apparaissent limitées. D'autres travaux, tels ceux de [Puntigam, 97] sont de nature très théorique, assez éloignés des contraintes opérationnelles propres à CORBA.
Parmi les divers formalismes proposés, les StateCharts [Harel & Gery, 97] répondent bien aux critères détaillés ci avant, et ont été choisis comme formalisme de description comportementale dans la populaire méthode UML.

3.2 Le projet SERPICO

Le projet SERPICO (**S**pécification **E**t **R**eprésentation Comportementale, **P**rototypage et **I**ntégration de Composants **CO**RBA) est financé par le CNET (Centre National des Etudes de

France Telecom), dans le cadre de la CTI (Consultation Thématique Informelle) N° 98 1B 059.

Ce projet, démarré en mars 1998 pour une durée de trois ans, bénéficie d'un budget de plus de 1MF. Le projet finance en totalité la thèse de Ousmane Sy (cf. §**Erreur ! Source du renvoi introuvable.**).

3.2.1 Problématique scientifique

La nécessité de proposer une notation formelle adaptée à la spécification comportementale des systèmes CORBA, justifiée plus haut, est au cœur de la problématique scientifique du projet SERPICO.

CORBA n'inclut aucune notation susceptible de décrire de manière abstraite les comportements, mais il apparaît qu'une description complète et non ambiguë du comportement des classes de composants CORBA est nécessaire, à tous les stades du développement et de l'intégration de ces composants :

- Conception des composants : Une fois que l'interface du composant CORBA est définie en termes d'IDL, et avant de passer à la phase d'implémentation, il est essentiel de disposer d'une spécification à la fois abstraite et précise du comportement du composant. C'est cette spécification qui servira de référence et de guide lors de l'implémentation du composant dans un langage de programmation supporté par CORBA.
- Extension de composants existants : CORBA supporte les notions orientées-objet d'héritage et de polymorphisme. Toutefois, si on souhaite étendre un composant CORBA par héritage, il faut s'assurer non seulement que l'interface de la sous-classe est compatible avec celle de la classe parente, mais aussi que les comportements de ces deux classes sont compatibles. Les problèmes d'héritage entre objets concurrents donnent lieu à d'épineux problèmes théoriques, dont celui bien connu de l' « inheritance anomaly ». Il faudrait donc disposer d'un moyen de s'assurer qu'une sous-classe CORBA définit bien un comportement compatible avec son ancêtre.
- Bibliothèques de composants et de services : Suivant l'exemple des Common Object Services (COS) définis par l'OMG, il faut s'attendre à voir se développer une offre de composants logiciels offrant des services réutilisables de haut niveau. A l'heure actuelle, ces modules sont définis par des modules d'interfaces IDL complexes, accompagnés de « modes d'emplois » informels, souvent écrits en langage naturel. Il apparaît absolument nécessaire de se doter d'une notation permettant à l'utilisateur d'un de ces composants de comprendre son fonctionnement et ses contraintes d'utilisation.
- Intégration de composants : Lors de l'intégration au sein d'un même applicatif de plusieurs composants CORBA, on se trouve toujours confronté à la possibilité d'avoir entre composants une utilisation incorrecte, c'est à dire une utilisation qui ne respecte pas les contraintes imposées par le cycle de vie d'un des composants qui interagissent. Faute d'une description abstraite du comportement, de tels dysfonctionnements ne peuvent être détectés que par une phase de tests intensifs, qui sont difficiles à mettre en œuvre et d'une fiabilité aléatoire. Si l'on dispose d'une description formelle du comportement des objets qui interagissent, il est possible de vérifier automatiquement que les objets coopèrent au sein de l'application en respectant leurs contraintes mutuelles de comportement, et donc de vérifier automatiquement des propriétés de bon fonctionnement (non interblocage, par exemple).

3.2.2 Objectifs

L'objectif scientifique du projet est la construction d'un environnement logiciel permettant la spécification comportementale, la validation, le prototypage et l'intégration de composants CORBA. L'idée sous-jacente est de donner une nouvelle dimension à l'ingénierie des composants CORBA en l'appuyant sur des techniques formelles de spécification et de vérification.

Cet objectif général se décompose en plusieurs sous-objectifs distincts :

Définition d'un formalisme adapté

La première étape, bien entendu indispensable à la suite du projet, consiste à proposer un formalisme adapté à la spécification comportementale des composants CORBA. Les caractéristiques nécessaires à un tel formalisme sont les suivantes :

- Caractère formel : la notation proposée doit avoir un niveau de formalité suffisant pour permettre la vérification automatique des comportements décrits et la génération de code.
- Expressivité : le formalisme doit permettre de décrire de manière simple et lisible des comportements qui peuvent être complexes, incluant des notions de parallélisme, de synchronisation, etc. Il doit également permettre une spécification des exceptions qui peuvent être levées lors des invocations de méthodes.
- Exécutabilité : le formalisme proposé doit être exécutable : il est dès lors possible de prototyper et de tester rapidement un composant CORBA dès que son comportement a été spécifié.

Dans le cadre du projet SERPICO, nous avons proposé l'utilisation du formalisme des Objets Coopératifs, étendu pour prendre en compte les spécificités de CORBA.

Etude des techniques de validation associées

Le formalisme choisi doit se prêter à des possibilités de vérification automatique des comportements décrits :

- Validation unitaire : lors de la définition du comportement d'un composant CORBA, ce comportement doit pouvoir être validé « en isolation » : on veut pouvoir prouver que ce composant est exempt de défauts internes (blocages, opérations inaccessibles, défauts de synchronisation).
- Validation par héritage : lorsqu'on étend un composant par héritage, il faut prouver que le comportement du nouveau composant est compatible avec celui spécifié par sa super-classe : notamment, toute séquence d'invocation de services acceptée par une instance de la super-classe doit l'être également par la sous-classe, faute de quoi le nouveau composant ne pourra pas être utilisé de manière polymorphe.
- Validation d'intégration : lorsqu'on intègre plusieurs composants CORBA dans une application, on doit pouvoir prouver que ces différents composants coopèrent en respectant leurs comportements mutuels.

Construction de l'environnement

Pour être utilisable, le formalisme choisi doit disposer d'un environnement logiciel permettant de construire les modèles, de les simuler et de les analyser. Le but de cette étape est à terme de disposer d'un environnement de programmation complet, permettant l'exécution distribuée et l'intégration de composants CORBA.

Définition d'une démarche méthodologique

Le formalisme et l'outil doivent être accompagnés d'une démarche méthodologique détaillée, décrivant comment ils doivent s'intégrer dans un processus de développement logiciel fiable, allant de la définition de l'architecture initiale du système jusqu'au test des implémentations.

3.3 Mes contributions dans ce domaine

C'est dans le cadre du projet SERPICO que mon équipe a réalisé ses contributions au domaine des systèmes distribués à objets. Bien que le projet n'en soit pas même à mi-chemin de ses échéances, les travaux effectués ont déjà donné lieu à plusieurs publications scientifiques, notamment lors des conférences ECOOP'99 [Bastide et al., 99b], ICATPN'99 [Bastide et al., 99a] et DOA'99 [Bastide et al., 99c].

Certains des objectifs scientifiques du projet nous paraissent proches d'être atteints : Le formalisme des Objets Coopératifs a été étendu pour prendre en compte les spécificités de CORBA, et nous avons traité des exemples suffisamment complexes pour que nous soyons confiants quant à l'applicabilité réelle de la notation (cf. § 3.3.3). L'expérience que nous avons acquise en traitant ces études de cas nous permet également d'envisager des améliorations d'ordre syntaxique que nous pourrions apporter à la notation afin d'améliorer son utilisabilité (cf. § 3.4.3). L'environnement PetShop est opérationnel (au moins au sein de notre laboratoire, un gros travail de finition et de documentation reste à faire pour qu'il puisse être distribué à d'autres utilisateurs dans de bonnes conditions de robustesse et de portabilité).
Les travaux continuent au sein du projet SERPICO pour atteindre les autres objectifs du projet, notamment le développement des techniques d'analyse § 3.4.1 et de génération de tests fonctionnels § 3.4.4.

3.3.1 Publications choisies

"Formal Specification and Prototyping of CORBA Systems." *13th European Conference on Object-Oriented Programming, ECOOP'99*, Lisbon, Portugal, June 14-18, 1999. Rachid Guerraoui, Volume editor. Lecture Notes in Computer Science, no. 1628. Springer (1999) 474-94.
Cet article illustre les avantages que l'on peut retirer de l'utilisation d'une notation formelle adaptée à CORBA. L'article donne également l'architecture de l'environnement PetShop.

"Petri-Net Based Behavioural Specification of CORBA Systems." *20th International Conference on Applications and Theory of Petri Nets, ICATPN'99*, Williamsburg, VA, USA, June 21-25, 1999. Susanna Donatelli, and Jetty Kleijn, Volume editors. Lecture Notes in Computer Science, no. 1639. Springer (1999) 66-85.
Cet article détaille les fondements théoriques de nos travaux. La sémantique du protocole client-serveur est donnée en termes de réseau de Petri, et les contraintes structurelles sur les ObCS sont détaillées.

3.3.2 L'environnement PetShop

Un avantage bien connu des réseaux de Petri est qu'ils constituent une notation à la fois formelle et exécutable. Notre approche met à profit cette exécutabilité en permettant d'interpréter dynamiquement une spécification comportementale dès qu'elle a été définie en termes d'Objets Coopératifs, ce qui permet d'obtenir une compréhension plus approfondie de la dynamique du système.

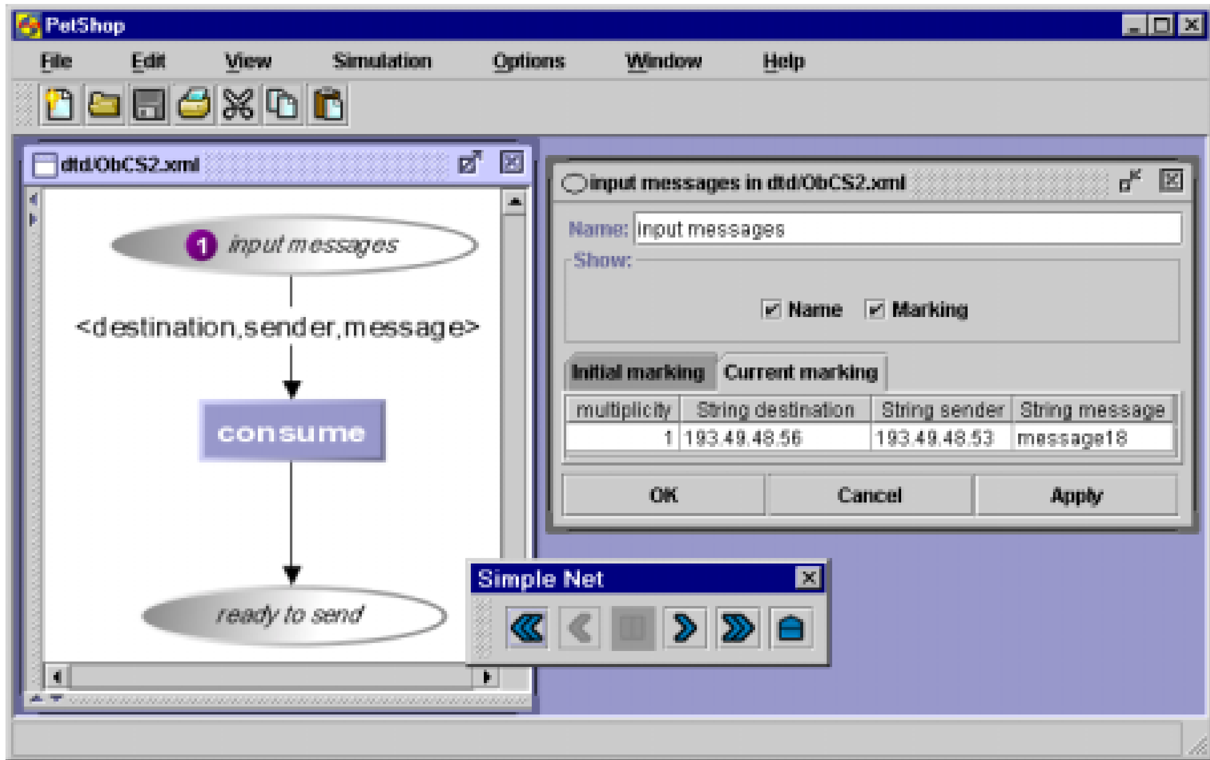


Figure 8 : L'environnement PetShop

L'approche s'appuie sur un outil appelé PetShop (Figure 9), qui inclut une implémentation distribuée de l'interprète de réseaux de Petri de haut niveau décrit en [Bastide & Palanque, 95], réécrit en Java. L'implémentation de l'outil est réalisée de telle sorte que :

- Un ObCS interprété peut invoquer (par une transition d'invocation) un service offert par un serveur CORBA tiers, qui s'exécute hors de l'environnement ;
- Réciproquement, un client CORBA externe peut invoquer un service offert par un Objet Coopératif interprété, sans se soucier du fait que ce service sera en fait exécuté par interprétation de l'ObCS.

Ceci permet d'obtenir une interopérabilité complète entre le monde CORBA et notre formalisme, et permet à l'environnement de fonctionner dans un contexte réaliste, où l'on possède la spécification formelle de quelques composants seulement, mais où l'on peut néanmoins accéder à des objets quelconques, dont on connaît seulement l'interface IDL.

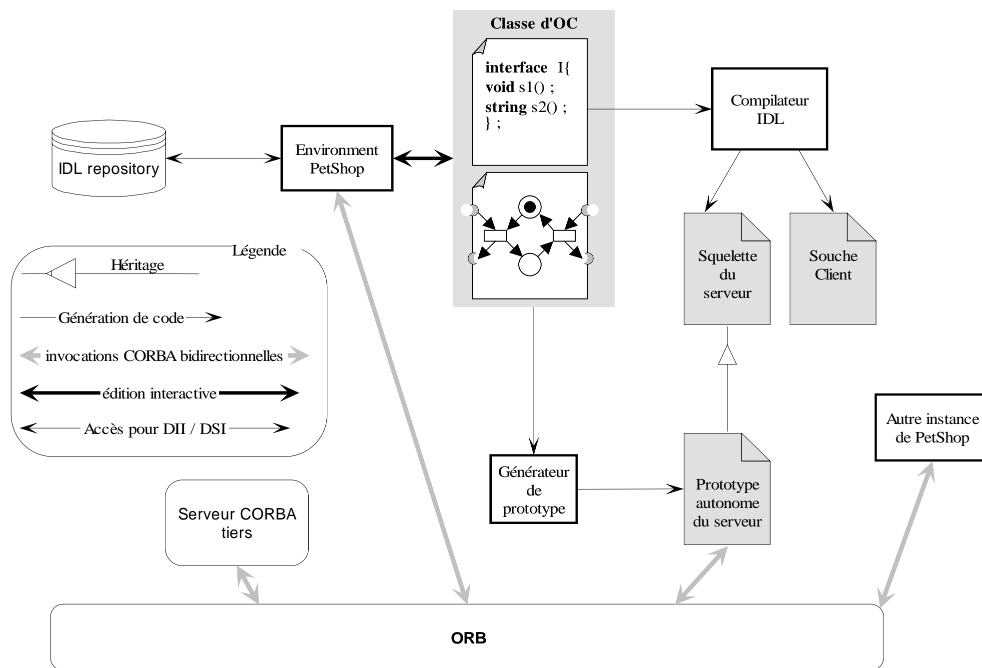


Figure 9 : Architecture de l'environnement PetShop

L’outil PetShop est conçu pour s’intégrer au mieux dans le cycle de développement d’un système CORBA, et offre un environnement intégré permettant de prendre en compte les phases suivantes :

- Edition de la spécification comportementale en tant que classe d’OC.
- Génération de l’IDL.
- Analyse mathématique des modèles comportementaux.
- Interprétation et déverminage des modèles.
- Génération de prototypes exécutables

Edition de la spécification comportementale : l’environnement inclut un éditeur syntaxique de réseaux de Petri, permettant la saisie des parties graphiques et textuelles de la spécification.

Génération d’IDL : le but principal de notre approche est d’ajouter une spécification comportementale aux interfaces IDL. Une activité fréquente dans ce contexte est de partir d’un ensemble d’interfaces défini par ailleurs (telles que les interfaces que l’on trouve dans les nombreux Common Object Services, COS, définis par l’OMG) et décrire une spécification comportementale pour les complémenter. Le concepteur peut également partir de la spécification comportementale, et générer l’IDL correspondant.

Analyse mathématique du comportement : L’outil inclut les algorithmes classiques d’analyse, qui permettent d’explorer des propriétés intéressantes sur les ObCS, telles que les invariants ou les conflits (propriétés structurelles) ou le caractère vivant, borné et réinitialisable du réseau (propriétés comportementales).

Interprétation et déboguage des modèles : dans notre environnement, chaque instance d’Objet Coopératif en cours d’édition est exécutée dynamiquement par l’interprète décrit en [Bastide & Palanque, 95]. Cette instance peut interagir librement avec des objets CORBA tiers, en utilisant les mécanismes d’invocation dynamique (Dynamic Invocation Interface et Dynamic Skeleton Interface) mis à disposition par CORBA. L’interprétation est étroitement couplée avec l’édition graphique du modèle, ce qui permet de tester interactivement des

nouveaux comportements sans nécessiter de recompilation. La représentation graphique du réseau, qui donne accès à tout instant au marquage des différentes places (et donc à l'état interne de l'instance) constitue par ailleurs un environnement de déboguage convivial et productif. La Figure 8 illustre l'interface utilisateur de l'environnement PetShop. On voit une instance d'OC en cours d'exécution, avec, dans une fenêtre séparée, le marquage d'une des places du réseau. Ce marquage évolue en temps réel lors de l'interprétation du modèle. L'interprétation peut être faite en continue (l'interprète sélectionne alors automatiquement les transitions à franchir) ou en mode pas à pas (l'utilisateur choisit alors interactivement la transition qu'il souhaite franchir, et même la substitution avec laquelle la franchir).

Simulation distribuée : L'environnement PetShop est lui-même un objet CORBA, et plusieurs exemplaires de PetShop peuvent communiquer pendant l'interprétation des modèles. Il est par exemple possible de demander, à partir de l'environnement, l'instanciation d'une nouvelle classe sur un exemplaire distant de PetShop, et de communiquer librement avec cette nouvelle instance.

Génération de prototypes exécutables : Lorsque la spécification comportementale est complète et que les modèles ont été validés par analyse et par interprétation, le concepteur peut générer un prototype autonome correspondant à la spécification. Ces prototypes, qui peuvent être exécutés sans le support de l'environnement graphique, sont des implémentations « quasi-fonctionnelles », dans le sens que seuls les aspects comportementaux sont pris en compte. D'autres aspects liés à la qualité de service, tels que la performance, la réplication, la persistance ou la tolérance aux pannes doivent également être traités dans une implémentation complètement fonctionnelle. Le prototype que l'on génère est toujours une instance de RdP interprétée, mais le fait de fonctionner en dehors de l'éditeur graphique lui permet de fonctionner plus efficacement, et d'être exécuté sur des machines serveur, qui n'ont pas de possibilités graphiques.

3.3.2.1.1 Un "langage de scripts formel"

L'architecture que nous avons donnée à l'environnement PetShop est inspirée de CorbaScript [Merle et al.,], qui définit un langage de scripts interprétés dédié à CORBA. L'utilité principale des langages de script est de promouvoir la rapidité et la flexibilité du développement logiciel, afin de permettre un prototypage rapide des applications, et de permettre d'explorer à moindre coût des alternatives de conception. Un des objectifs de conception de l'environnement PetShop est d'être un « langage de script formel » :

- Nous souhaitons fournir un outil qui offre la même flexibilité et la même simplicité de développement qu'un langage de script. Il faut notamment, dans le contexte de CORBA, que la mise en œuvre d'un serveur CORBA opérationnel soit sensiblement plus rapide en utilisant PetShop qu'en utilisant un langage algorithmique conventionnel tel que C++. Ceci implique que PetShop prenne à sa charge tous les aspects fastidieux et répétitifs de la mise en œuvre (connexion à l'ORB, exploration des interfaces disponibles, nommage et localisation des objets…), afin de permettre à l'utilisateur de se concentrer sur la modélisation comportementale en oubliant les aspects accessoires.
- Nous voulons également que l'outil préserve les avantages que l'on peut retirer de l'utilisation d'une notation formelle, en termes de complétude et de non-ambiguïté des spécifications produites. Les travaux en cours dans le projet SERPICO explorent deux directions : la génération de test fonctionnel à partir de la spécification, et l'utilisation des techniques d'analyse propres aux RdP de haut niveau pour vérifier les propriétés du système modélisé.

Nous avons donc consacré beaucoup d'effort à l'interface homme-machine de l'outil, et à son intégration la plus transparente possible dans un environnement CORBA :

- L'interface générale de l'outil reprend dans une large mesure celle que l'on rencontre dans les éditeurs de RdP conventionnels (Figure 8) : on dispose d'un éditeur graphique, interactif et convivial, qui utilise les techniques classiques de la manipulation directe. L'outil n'intègre pas (pour le moment) des techniques d'interaction plus modernes et plus ambitieuses, telles que celles qui sont développées au sein du projet CPN2000 à Aarhus (manipulation à deux mains, fenêtres semi-transparentes). Par contre, notre travail a essentiellement porté sur l'interactivité de l'environnement, qui offre une mode de fonctionnement beaucoup plus souple que ce que l'on rencontre généralement. On s'affranchit par exemple du caractère « modal » de la plupart des environnements RdP de la génération précédente. De tels environnements imposent une distinction complète entre la phase d'édition de la structure du réseau, la phase d'exécution ou d'interprétation dynamique, et les phases éventuelles d'analyse : chacune de ces phases est réalisée dans un « mode » particulier du logiciel, et ces phases doivent s'exécuter séquentiellement. Dans PetShop, au contraire, ces différentes activités peuvent être réalisées simultanément, de manière transparente : Il n'y a pas de différence entre la phase d'édition d'un modèle et la phase d'exécution : Pendant qu'une instance d'OC est en cours d'exécution, la structure de son ObCS peut être modifiée dynamiquement par l'utilisateur[7], sans interrompre son fonctionnement. Ceci s'avère particulièrement utile, par exemple, dans le cas où la classe d'OC décrit le fonctionnement d'une interface-utilisateur : on peut alors conduire des sessions de prototypage, où l'on présente à l'utilisateur une interface opérationnelle, et où on peut directement évaluer avec lui différentes alternatives de comportement en changeant le modèle, sans recompilation. Cette technique de « prototypage formel » a été développée et utilisée au sein du projet MEFISTO (§ 4.2). De même, l'analyse de l'ObCS est réalisée concurremment à son édition, par des « agents d'analyse » qui fonctionnent en arrière plan, et qui fournissent leur résultat sans interrompre l'activité du modélisateur.
- En ce qui concerne l'intégration dans un environnement distribué, il est par exemple possible de faire communiquer plusieurs exemplaires de l'environnement sur un réseau local. Il est ainsi possible de créer des instances d'OC à distance, et donc de réaliser très simplement une simulation distribuée.

3.3.3 Une expérience de « rétro-spécification » d'un service CORBA

Les résultats décrits dans le présent chapitre sont une contribution originale de ce mémoire, et n'ont pas encore été publiés.

L'OMG a entrepris de standardiser un ensemble de service (COS, Common Object Services [Object Management Group, 98a]) destinés à devenir les briques de base de la construction des systèmes distribués. Certains de ces services sont dits « horizontaux » c'est à dire qu'ils correspondent à une fonctionnalité typique que l'on s'attend à retrouver dans une large classe de systèmes distribués. Parmi ceux-ci on peut citer le service de nommage (Naming Service) qui permet d'associer un nom symbolique à un objet, le service de contrôle d'accès concurrents (Concurrency Control Service) qui permet à plusieurs objets de se coordonner

[7] Dans ces cas de modification interactive du comportement, on se place sous la responsabilité de l'utilisateur. Comme dans un debugger, il peut faire des modifications incohérentes : il peut par exemple interrompre une invocation en cours, ce qui se traduirait plus tard par une exception "timeout" chez le client de l'invocation. L'utilité principale de ce mode est de corriger des erreurs dans les valeurs des différents jetons, de manière à pouvoir continuer l'exécution avec des valeurs correctes sans interrompre le fonctionnement.

pour accéder à une ressource partagée, ou le service d'événements (Event Service) qui permet la communication asynchrone par événements. Les services peuvent être également « verticaux », c'est à dire correspondre aux besoins d'un domaine d'application particulier (par exemple la banque, ou les télécommunications).

L'OMG a publié un document [Object Management Group, 98a] contenant la spécification de 14 de ces services. Ce document, qui ne compte pas moins de 1074 pages, spécifie chacun des 14 services en fournissant leur IDL, et en décrivant leur fonctionnement en langage naturel. La spécification s'agrémente parfois d'un automate fini qui décrit de manière plus précise certains états accessibles du composant spécifié. Cette description sous forme d'automate est d'ailleurs assez rare, et n'est jamais exhaustive, l'automate servant surtout à clarifier le texte en langage naturel.

Un des motivations de notre travail est la conviction que ce mode de spécification n'est pas approprié au domaine des systèmes distribués à objets. Nous pensons que des spécifications d'une telle complexité, si elles ne s'appuient pas sur une notation formelle, contiendront presque obligatoirement des ambiguïtés, des imprécisions voire des contradictions. Ceci nous apparaît comme spécialement problématique dans le contexte des COS, puisque les textes de l'OMG servent de références aux entreprises qui développent et vendent des implémentations de ces différents services. Le problème est rendu encore plus aigu par les objectifs d'interopérabilité sur lesquels se fonde CORBA. Il se peut sans doute que les implémenteurs d'un COS résolvent les ambiguïtés ou les sous-spécifications des documents de l'OMG en faisant appel à leur bon sens et à leur expérience des systèmes distribués. Cependant, il y a peu de chance que deux implémenteurs différents résolvent ces sous-spécifications de la même façon. C'est d'autant plus regrettable que les diverses implémentations des services CORBA sont conçues pour interopérer : deux « Event Services » hétérogènes, par exemple, doivent pouvoir s'échanger des événements. Linda Northrop a justement mentionné, lors d'un débat de la conférence ECOOP'99, que ce style de spécifications était particulièrement propice à produire des « *components that plug but just don't play* » : Les composants peuvent se « brancher » (plug), interagir au niveau syntaxique, mais ils ne « jouent » pas (play) ensemble, c'est à dire qu'ils ne peuvent pas interagir dynamiquement, car ils font des hypothèses erronées sur leur comportement mutuel.

Nous avons donc mené une expérience destinée à prouver le bien-fondé de notre travail, et à motiver le développement de l'usage des méthodes formelles dans le contexte de CORBA.

Notre expérimentation visait à confirmer l'hypothèse suivante :

Hypothèse : une spécification en langage naturel d'un service CORBA est nécessairement incomplète ou ambiguë. Des implémentations distinctes construites à partir d'une telle spécification présenteront des incompatibilités fonctionnelles importantes.

Nous souhaitions également établir un corollaire qui justifie le choix de notre formalisme, et notamment la capacité de notre formalisme à gérer la spécification de systèmes complexes.

Corollaire : Le formalisme des Objets Coopératifs est adapté à la spécification de systèmes présentant un niveau de complexité équivalent à celui d'un service CORBA typique. On ne constate pas de problème de « facteur d'échelle » (explosion de la taille des modèles, par exemple) lorsque l'on passe à ce niveau de complexité.

Pour tester ces hypothèses, nous avons procédé de la manière suivante :

- Nous avons sélectionné un service CORBA caractéristique (l'Event Service),
- nous avons procédé à la spécification formelle de ce service par les Objets Coopératifs,
- à partir de cette spécification formelle, nous avons dérivé un jeu de tests fonctionnels,
- nous avons soumis un certain nombre d'implémentations du service CORBA au jeu de test, et analysé les résultats obtenus.

Cette procédure nous a effectivement permis de constater que les implémentations testées présentaient des différences fonctionnelles importantes, au point d'empêcher leur interopérabilité.

L'objet du présent chapitre n'est pas de présenter in extenso la spécification de l'Event Service (ce qui dépasserait le cadre de ce mémoire) mais plutôt de détailler la démarche que nous avons suivie et les leçons que nous avons pu en retirer.

3.3.3.1 Sélection du service CORBA

Parmi les quatorze services spécifiés en [Object Management Group, 98a], nous en avons sélectionné un, en l'occurrence le service d'événements (Event Service). Notre choix s'est porté sur ce service, pour plusieurs raisons :

- Ce service est « auto-contenu » : il ne fait référence à aucun autre service CORBA, à la différence de nombreux COS qui dépendent de l'existence d'autres services pour fonctionner. Le service de cycle de vie (Life-Cycle) par exemple dépend de l'existence du service de nommage. Le service d'événements ne dépend pas non plus de fonctionnalités propres à CORBA, telles que l'existence d'un Interface Repository. Il est donc possible de fournir une spécification complète de ce service, sans faire d'hypothèse sur des services de plus bas niveau.
- Le service présente un niveau de complexité intéressant, dû à sa grande généralité[8]. En particulier, la connexion à un canal d'événements nécessite un protocole assez complexe entre producteurs et consommateurs d'événements.
- Ce service fait partie de ceux qui sont le plus souvent implémentés par les fournisseurs d'ORB. Une implémentation existe dans la quasi-totalité des ORB disponibles. Ce n'est pas le cas, loin s'en faut, de tous les autres COS : dans sa rage spécificatrice, l'OMG a en effet standardisé certains services pour lesquels la demande du marché semble faible, et que les développeurs d'ORB sont donc réticents à implémenter. Pour tester nos hypothèses, il était nécessaire de disposer de plusieurs implémentations réelles du service choisi.

3.3.3.1.1 Présentation de l'Event Service

Le modèle objet de CORBA propose uniquement un modèle de communication synchrone entre objets : l'invocation d'un service ne se termine pour l'objet client qu'après la fin de l'exécution du service par l'objet serveur. D'autre part, la communication supportée par CORBA est de type « 1 vers 1 » : l'objet client doit toujours posséder une référence vers l'objet qu'il souhaite invoquer. Il n'existe pas de primitive de type « diffusion » (multicast).

[8] Cette généralité est d'ailleurs jugée excessive par certains auteurs [Schmidt & Vinoski, 97], qui considèrent que ce service ne sera jamais utile dans toute sa flexibilité, et que les utilisateurs seront donc tentés de réimplémenter une partie seulement de ses fonctionnalités avec des interfaces plus spécifiques.

Cette contrainte a pour conséquence qu'un client qui cherche à diffuser une information vers un ensemble de serveurs doit impérativement connaître les références de chacun d'eux, et communiquer de façon séquentielle avec chacun.
L'Event Service a été proposé pour s'affranchir de ces limitations. Ce COS met en scène trois rôles :

1. le producteur d'événements,
2. le consommateur d'événements,
3. le canal événementiel.

Le consommateur implémente une interface définie par l'Event Service (`PushConsumer` ou `PullConsumer`) pour signifier son désir de communiquer et pour être informé de l'état de la communication avec un producteur. De même, le producteur utilise une interface (`PushSupplier` ou `PullSupplier`) pour indiquer son intention de communiquer et se tenir au courant de l'état de la liaison avec le consommateur. Une connexion est réalisée par la mise en relation d'une interface de consommation avec une interface de production. Ces interfaces permettent la déconnexion et, selon celui qui prend l'initiative de la communication, la consommation ou la production d'événements.
Les rôles de consommateur et de producteur ne suffisent pas pour s'affranchir du modèle d'invocation synchrone de CORBA. Le troisième et dernier rôle est celui d'un canal événementiel (event channel) qui s'interpose entre un nombre quelconque de consommateurs et de producteurs pour diffuser des événements. Les consommateurs et producteurs vus précédemment sont alors des clients du canal. Ce canal rend inutile la connaissance réciproque des consommateurs et des producteurs. Il a deux fonctions principales : l'administration et la propagation des événements. L'administration du canal est confiée à une interface de canal (`EventChannel`) qui d'une part délègue la gestion des clients du canal à des interfaces d'administration (`ConsumerAdmin` et `SupplierAdmin`) et d'autre part peut décider de l'état du canal. Ces interfaces d'administration ont pour rôle de créer des consommateurs et des producteurs internes au canal, appelés proxies (`ProxyPushConsumer`, `ProxyPushSupplier`, `ProxyPullConsumer`, `ProxyPullSupplier`), qui connectent les clients au canal et par lesquels transitent les événement transportés (Figure 10).

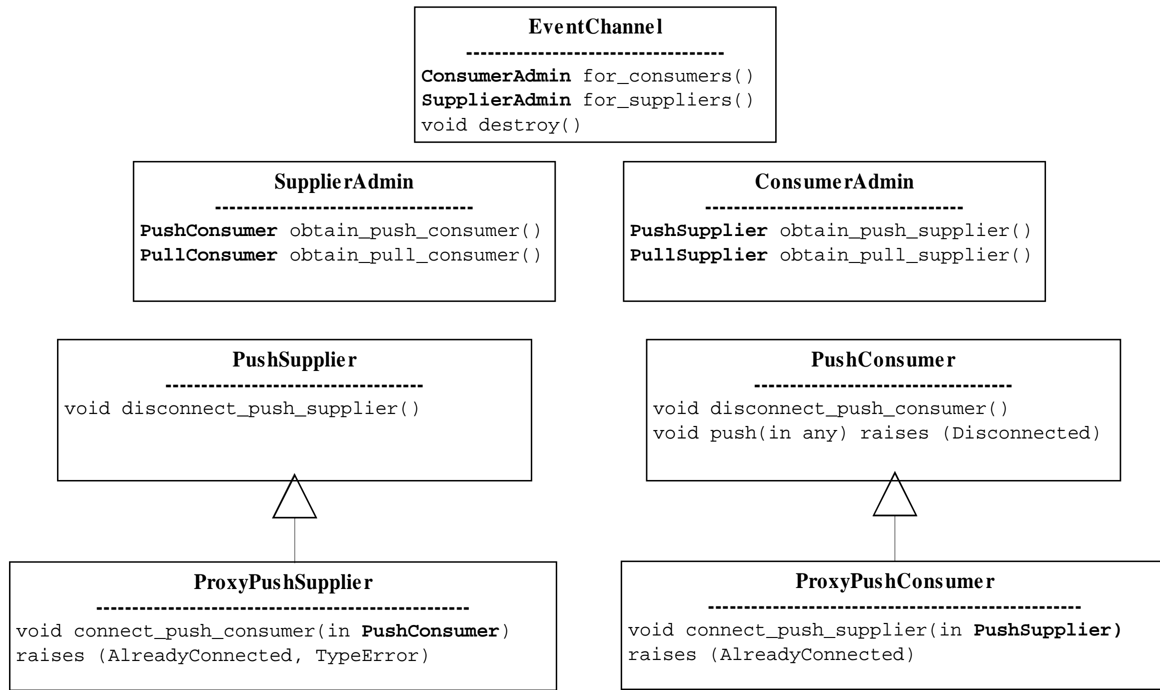


Figure 10 : interfaces de l'Event Service (mode « push »)

Deux modèles de communications sont proposés pour marquer l'initiative de la communication :

- Le modèle « push » dans lequel le consommateur subit les invocations du producteur qui peut générer des événements à sa guise. Le consommateur peut néanmoins décider de mettre fin à la connexion en se déconnectant ou, s'il dispose de la référence du producteur, l'informer de la terminaison de la connexion en le déconnectant.
- Le modèle « pull » dans lequel le consommateur est libre d'invoquer le producteur pour réclamer un nouvel événement. Le producteur peut interrompre la communication s'il dispose de la référence du consommateur, en le déconnectant. Le producteur permet deux types de requêtes : le premier est bloquant tant qu'un événement n'est pas disponible (*pull*), le second n'est pas bloquant et renvoie une indication sur la validité du résultat de la requête (*try_pull*).

3.3.3.1.2 Scénario d'utilisation de l'Event Service

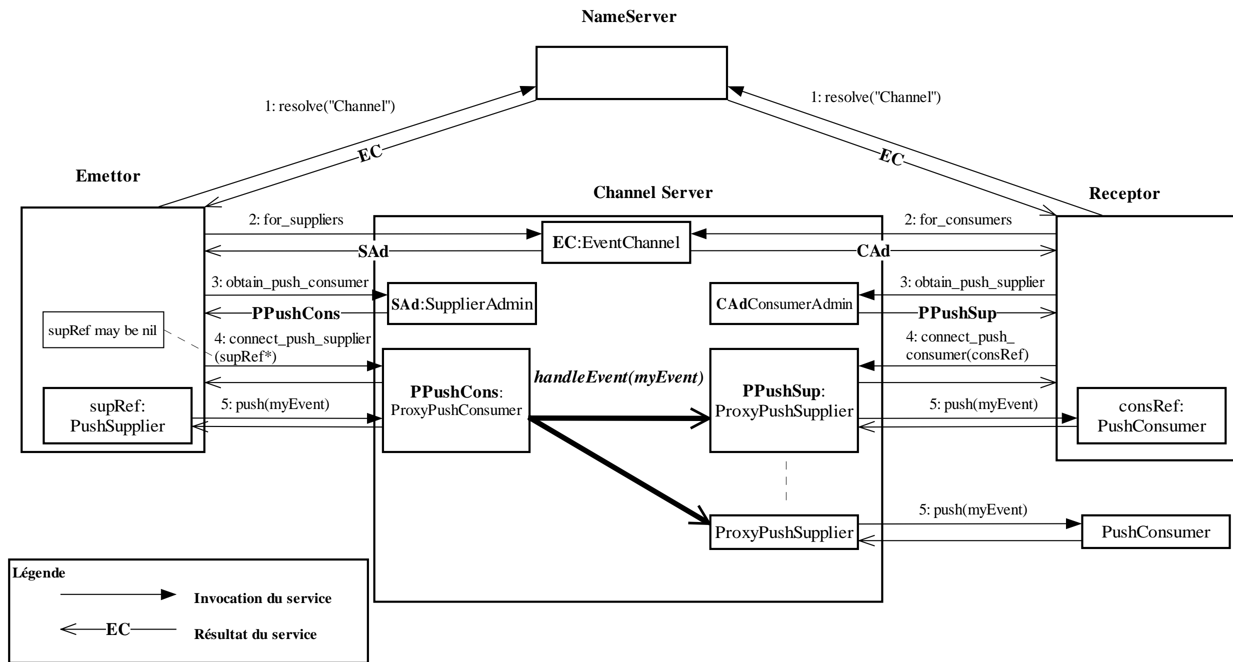


Figure 11 : Scénario d'utilisation de l'Event Channel

La Figure 11 illustre l'utilisation de l'Event Channel en mode « push ». On considère le scénario typique suivant :

Un objet (*Receptor*) souhaite s'abonner à un canal événementiel pour jouer le rôle de consommateur d'événements. Un autre objet (*Emettor*) souhaite assumer le rôle de producteur d'événements pour ce canal. Les objets *Emettor* et *Receptor* se connectent indépendamment sans se concerter au canal événementiel. Le canal événementiel remplit son rôle de tampon en connectant chacun des clients à un proxy, en suivant un protocole de connexion en quatre étapes :

1. *Emettor* et *Receptor* obtiennent une référence vers le canal événementiel (par exemple, en faisant appel à un serveur de noms pour obtenir la référence d'un objet nommé « *Channel* ».Cette référence est notée EC ;
2. *Emettor* et *Receptor* demandent à EC les références d'un administrateur approprié et obtiennent respectivement un administrateur de fournisseurs (*SAd*, sur invocation de *for_supplier*) et un administrateur de consommateurs (*CAd* sur invocation de *for_consumer*) ;
3. *SAd* renvoie un proxy consommateur (*PPushCons*) à *Emettor* sur invocation de *obtain_push_consumer* et *CAd* renvoie un proxy producteur (*PPushSup*) à *Receptor* sur invocation de *obtain_push_supplier* ;
4. La liaison est établie entre le canal et ses utilisateurs après invocation des opérations *connect_push_supplier* sur *PPushCons* et *connect_push_consumer* sur *PPushSup*. Cette deuxième opération exige le passage comme paramètre de la référence d'un consommateur (*consRef*), délégué de *Receptor*, pour que le fournisseur *PPushSup* puisse invoquer l'opération *push* quand des événements sont disponibles.

La communication s'effectue alors par invocation des méthodes *push* sur les consommateurs d'abord à l'entrée du canal du fait du producteur *supRef*, délégué d'*Emettor*, et à la sortie du canal du fait de *PPushSup*. Elle peut être interrompue en invoquant les opérations de

déconnexion définies par les interfaces de consommation et de production de *supRef*, *consRef*, *PPushCons* et *PPushSup*, ou en détruisant le canal EC.

Le transfert d'information entre les proxies n'est pas couvert par la spécification de l'Event Service et reste un choix d'implémentation. Nous l'avons modélisé par une méthode *handleEvent* dans le cas du modèle Push. Quand *supRef* transmet l'événement *myEvent* comme paramètre de l'opération *push* à *PPushCons*, celui-ci invoque *handleEvent* sur *Chanel Server* pour router le message vers les proxies producteurs, comme *PPushSup*. A son tour *PPushSup* invoque l'opération *push* sur *consRef* avec le paramètre *myEvent*.

3.3.3.2 Rétro-spécification formelle

On voit que l'Event Service a été conçu pour fournir une flexibilité maximale, mais qu'en contrepartie sa structure et son fonctionnement sont loin d'être simples et intuitifs. Nous avons donc entrepris de construire une spécification formelle de ce service en utilisant les Objets Coopératifs.

Cette tentative appelle quelques commentaires :

3.3.3.2.1 Sous-spécification volontaire ou involontaire

La spécification OMG de l'Event Service est volontairement incomplète : en particulier elle ne précise pas la *qualité de service* qu'une implémentation conforme devrait fournir. Par exemple, le document OMG n'impose pas la transmission fiable des événements par le canal : les événements peuvent être perdus ou dupliqués. D'une manière un peu caricaturale, le document OMG précise que « Clairement, une implémentation du canal qui perd tous les événements n'est pas une implémentation utile » (*Clearly, an implementation of an event channel that discards all events is not a useful implementation*). On considère donc qu'un implémenteur de l'Event Service devra choisir une qualité de service, et préciser ce choix dans la documentation qu'il fournira à ses clients. De même, dans le mode « pull », la spécification ne précise pas la politique à suivre par le canal pour se procurer des événements auprès de ses fournisseurs. Une implémentation devrait donc choisir une politique particulière (round-robin, en parallèle, …) et documenter ce choix pour ses utilisateurs.

Le but de notre spécification formelle n'est pas de lever ces sous-spécifications volontaires des documents de l'OMG. Nous pensons au contraire qu'une spécification formelle volontairement indéterministe peut être fournie afin de laisser de la flexibilité aux implémenteurs, mais que la spécification doit pouvoir être raffinée à volonté, afin de spécifier une qualité de service ou une politique particulière, documentant alors de manière complète le comportement à implémenter.

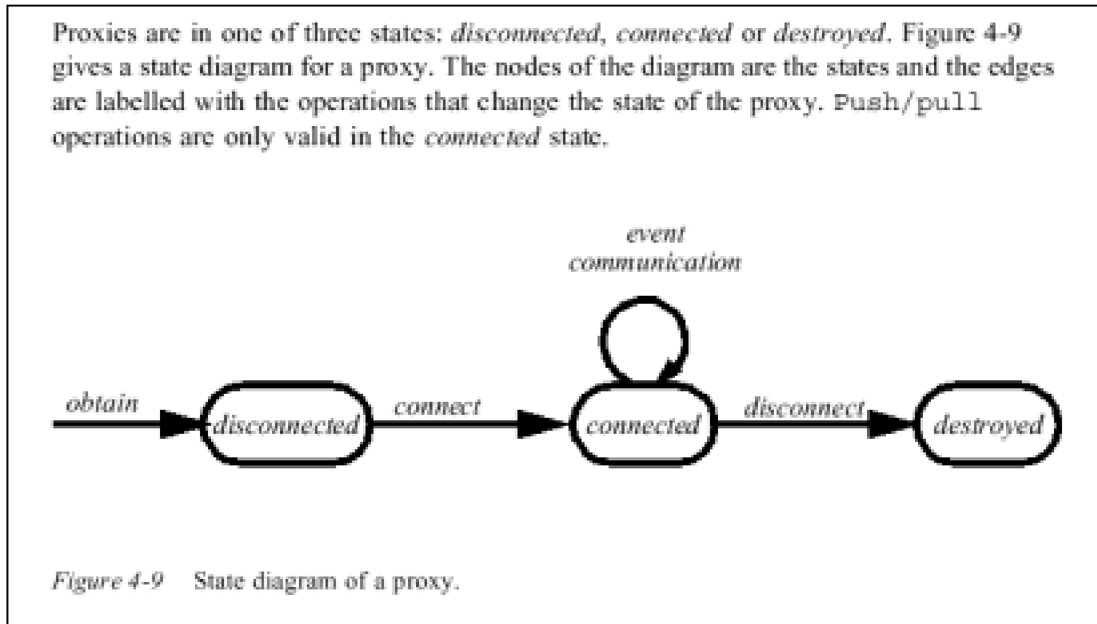
3.3.3.2.2 Gestion des ambiguïtés dans la spécification initiale

Comme le point de départ de notre spécification est un document en langage naturel, il faut s'attendre à y rencontrer des sous-spécifications (involontaires, cette fois), des ambiguïtés, voire des contradictions. L'objectif de la spécification formelle est précisément de détecter de tels défauts dans la spécification initiale, et d'y remédier. Dans l'expérience relatée ici, nous avons effectivement détecté des ambiguïtés et des sous-spécifications, mais pas de contradiction.
Lorsque nous avons détecté de tels problèmes, notre attitude a été la suivante :

- Documenter précisément le problème : si nous détectons le problème lors de la spécification formelle, il est vraisemblable que les implémenteurs du service le détecteront

également lors de l'implémentation, et qu'ils y apporteront une solution. Le fait d'avoir identifié ce problème potentiel nous guidera lors de la phase de test des implémentations réelles.

- Compléter et préciser la spécification, en faisant le choix qui nous apparaît le plus logique, le plus simple et le plus conforme à la philosophie du service. Bien entendu, il s'agit là d'un choix arbitraire, et d'autres spécificateurs pourraient ne pas partager notre avis.



Proxies are in one of three states: *disconnected*, *connected* or *destroyed*. Figure 4-9 gives a state diagram for a proxy. The nodes of the diagram are the states and the edges are labelled with the operations that change the state of the proxy. `Push/pull` operations are only valid in the *connected* state.

Figure 4-9 State diagram of a proxy.

Figure 12 : Extrait de la spécification originale de l'Event Service

La Figure 12 illustre le type de sous-spécification que nous avons souvent rencontré dans le document de l'OMG. Cette figure donne un diagramme d'état « informel » d'un proxy, agrémenté de commentaires en langage naturel. Cette spécification est largement incomplète : le diagramme d'état n'est pas complet, puisque plusieurs transitions d'état potentielles sont passées sous silence (par exemple, que se passe-t-il quand l'opération *disconnect* survient dans l'état *disconnected* ?). D'autre part, le commentaire en langage naturel est ambigu (« *operations are only valid in the connected state* » : que signifie précisément le fait d'être valide, pour une opération ?).

3.3.3.2.3 Spécification formelle par Objets Coopératifs

La spécification formelle complète du COS Event Service est décrite dans [Sy & Bastide, 99a], document fourni dans le cadre du projet SERPICO. Nous espérons pouvoir diffuser ces résultats dans un cadre plus large, après avoir obtenu l'autorisation de publier auprès du CNET, qui finance ces travaux.

Pour illustrer le type de spécification formelle que nous avons produit à partir du document OMG, je donne ici la spécification de l'interface `ProxyPushConsumer`, qui représente l'objet intermédiaire instancié par le canal pour gérer la communication avec un producteur unique (en l'occurrence un PushSupplier). C'est une des catégories d'objet proxy décrite informellement en Figure 12.

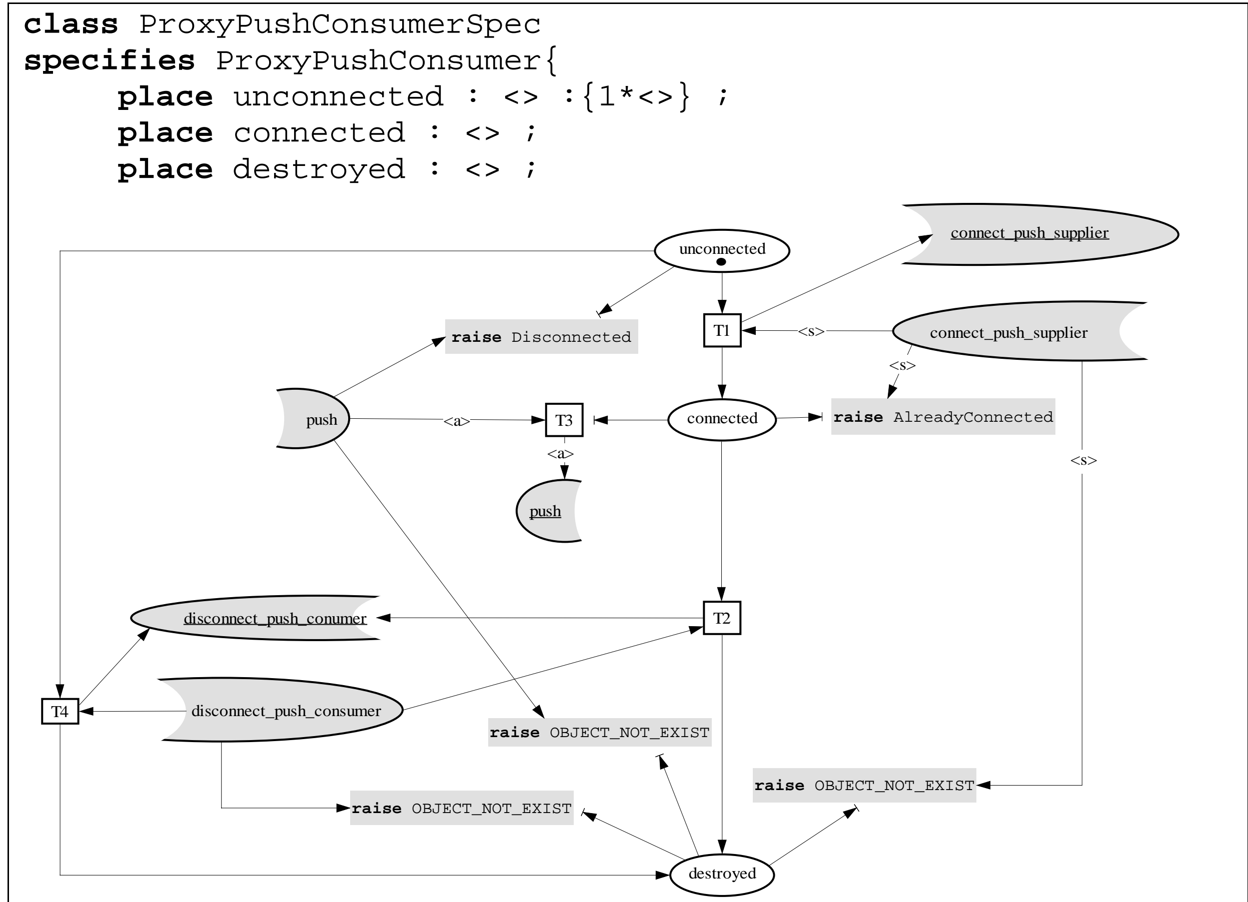


Figure 13 : Spécification de l'interface ProxyPushConsumer

La Figure 13 décrit la classe d'OC *ProxyPushConsumerSpec* qui spécifie l'interface *ProxyPushConsumer*. Cette interface hérite de *PushConsumer*. La figure illustre également la syntaxe graphique des transitions d'exception (qui apparaissent en grisé, et contiennent le mot clé *raise*, suivi du type de l'exception qui est levée). La sémantique de ces transitions d'exception (et notamment la manière dont elles enrichissent le protocole client-serveur qui régit l'invocation entre objets) est donnée en [Bastide et al., 99a]. Intuitivement, une transition d'invocation interromps l'exécution d'un service en retournant une exception au client. Les autres résultats éventuels du service sont indéfinis.

L'ObCS décrit clairement le cycle de vie de l'objet, qui va passer de l'état *unconnected* dans lequel il se trouve au moment ou il est instancié (marquage initial de la place *unconnected*) vers l'état *connected* (après invocation du service *connect_push_supplier*, par le franchissement de la transition T1) puis à l'état *destroyed* (après invocation du service *disconnect_push_consumer*, par le franchissement de la transition T2). Il spécifie également quelles exceptions sont susceptibles d'être levée lors de l'invocation des services, en fonction de l'état interne de l'objet, par les transitions d'exception : on voit ainsi que l'invocation du service *push* lorsque l'instance est dans l'état *unconnected* lève l'exception *Disconnected*.

On voit également qu'une requête sur *connect_push_supplier* peut se terminer de trois façons :

- Un succès, transition *T1* ;
- Un échec si l'instance est déjà connectée signalée par la levée de l'exception *AlreadyConnected* ;

- Un échec si le proxy a été détruit par invocation de *disconnect_push_supplier,* signalé par la levée de l'exception *OBJECT_NOT_EXIST*.

Par comparaison avec le document de l'OMG (Figure 12), cette spécification est certes plus complexe, mais elle est exempte d'ambiguïté, et ne nécessite aucun commentaire en langage naturel supplémentaire. La représentation sous forme de réseau de Petri peut, dans ce cas particulier, paraître exagérément compliquée par rapport à une représentation sous forme d'automate. Il faut toutefois remarquer qu'il s'agit d'un objet très simple, qui n'exhibe aucune concurrence interne. La présence de concurrence interne aura en général pour effet de rendre une représentation par automate plus complexe, nécessitant des notations plus élaborée telles que les StateCharts [Harel & Gery, 97], et les réseaux de Petri reprendraient dans ce cas l'avantage. D'autre part, le comportement de cet objet n'est pas influencé par les données qu'il transporte. Ce n'est pas le cas en général, et dans le cas contraire un formalisme tel que les réseaux de Petri de haut niveau montre également tout son intérêt, puisqu'il permet de décrire la structure des données à l'intérieur de la structure de contrôle.

3.3.3.3 Dérivation d'un jeu de tests fonctionnels

Après avoir complété la spécification de l'Event Service, nous avons souhaité comparer plusieurs implémentations réelles de ce service. L'objectif de ces tests doit être clarifié : il ne s'agit pas de prouver que les implémentations sont conformes à notre propre spécification. En effet, lors de notre spécification, nous avons été amenés à lever des ambiguïtés et des sous-spécifications dans les documents OMG. Il n'y a pas de raison pour que les implémenteurs aient levé ces ambiguïtés de la même manière que nous. Cependant, le fait de réaliser la spécification formelle nous a conduit à identifier de manière exhaustive toutes ces sous-spécifications. Nous avons donc pu produire des jeux de tests qui visent spécifiquement à déterminer comment les implémenteurs ont résolu les sous-spécifications, et à voir si les solutions qu'ils ont retenues sont compatibles entre elles.

Les imprécisions que nous avons pu détecter dans les spécifications de l'OMG ont le plus souvent trait aux comportements « limites » des différents objets, et plus particulièrement à la relation entre les invocations de services et le « cycle de vie » des objets décrits. Dans le présent document, je me limite à quelques problèmes que nous avons mis à jour en ce qui concerne le comportement des objets « proxy » (ProxyPushConsumer, ProxyPushSupplier…), et sur lesquels le document de l'OMG n'apporte pas de réponse. Le compte rendu complet des tests effectués est disponible sous la forme d'un rapport interne du LIHS [Sy & Bastide, 99b].

- Les proxies ont une opération de connexion (par exemple connect_push_consumer pour un ProxyPushSupplier), et une opération de déconnexion (disconnect_push_supplier pour le ProxyPushSupplier). Quel est le comportement d'un proxy sur lequel on invoque d'abord l'opération disconnect, puis l'opération connect ? Pour notre part, nous avons décidé dans notre spécification formelle que l'opération disconnect aurait systématiquement pour effet de mettre l'instance dans l'état destroyed, quel que soit sont état antérieur. Le test réalisé pour évaluer les comportements des implémentations sera appelé dans la suite **Test 1**.
- Est-il possible d'appeler plusieurs fois de suite l'opération disconnect sur un proxy ? Cette séquence d'invocation va-t-elle lever des exceptions ou réussir ? Notre choix de spécification était que toute invocation de méthode suivant l'opération disconnect lève l'exception OBJECT_NOT_EXIST. Le test réalisé pour évaluer les comportements des implémentations sera appelé dans la suite **Test 2**.

- Quel est l'effet d'effectuer la séquence d'invocations (connect, disconnect, push*), sur un ProxyPushConsumer ? Notre choix était de lever systématiquement l'exception OBJECT_NOT_EXIST sur tous les appels de push qui suivent un appel de disconnect. Le test réalisé pour évaluer les comportements des implémentations sera appelé dans la suite **Test 3**.

Nous avons également décidé de tester certains points qui sont couverts par la spécification de l'OMG, mais dont l'interprétation nous avait semblé difficile lors de la spécification formelle : par exemple, quand un PushSupplier se connecte à un ProxyPushConsumer (en invoquant le service connect_push_supplier), le document OMG spécifie qu'il peut transmettre sa propre identité en tant que paramètre du service, mais qu'il peut également transmettre la valeur nil. En effet, cette valeur n'est théoriquement jamais nécessaire au proxy pendant la suite de son fonctionnement. Nous voulions savoir si toutes les implémentations réagissaient bien en recevant la valeur nil. Le test réalisé pour évaluer les comportements des implémentations sera appelé dans la suite **Test 4**.

3.3.3.4 Test de 4 implémentations

Nous avons manuellement construit un jeu de tests nous permettant de déterminer comment des implémenteurs de l'Event Service avaient eux-mêmes résolu les ambiguïtés mentionnées plus haut.

Nous avons retenu quatre implémentations du COS Event provenant de quatre fournisseurs différents, ayant pour point commun de pouvoir fonctionner avec l'ORB Visibroker d'Inprise, qui est un des ORB de référence du projet SERPICO :

1. celle d'Inprise (http://www.inprise.com), Visibroker version 3.4. Elle est distribuée avec l'environnement de développement JBuilder d'Inprise;
2. celle d'Exemplar Development (http://www.exemplardev.com) utilisable gratuitement avec une restriction dans la durée de fonctionnement à 30 minutes ;
3. celle de Prism technologies (http://www.prismtechnologies.com), Open Fusion version beta. Cette société propose plusieurs des services du COS et a récemment signé un accord de distribution avec Inprise ;
4. celle de DSTC (http://www.dstc.com), COS Notification version 1.0.2. DSTC est un organisme de recherche australien qui a développé une version de COS Notification [Object Management Group, 98b] qui fonctionne avec Visibroker. Le COS Notification est une version étendue de COS Event qui contient toutes les fonctionnalités de COS Event.

Les tests ont été effectués sur un ordinateur équipé d'une architecture Intel Pentium II à 350 MHz, fonctionnant sous système d'exploitation Windows NT 4.0. Le langage de programmation utilisé pour le pilote de test est Java et nous utilisons le JDK 1.1.8. Chaque implémentation a été testée en isolation des autres. Les programmes de tests sont rigoureusement identiques pour toutes les implémentations testées.

3.3.3.5 Résultats expérimentaux et conclusions

La conception de l'Event Service étant très symétrique entre le mode « push » et le mode « pull », les problèmes que nous avons détectés se posent également dans les deux modes. Les tests que nous avons effectués se sont donc limités au mode « push », et ont consisté à effectuer des séquences d'appel sur des objets répondant à l'interface « ProxyPushConsumer ».

- Le test **Test1** invoque la séquence « disconnect_push_consumer() – connect_push_supplier() ».
- Le test **Test2** appelle la séquence « connect_push_supplier() - disconnect_push_consumer() – disconnect_push_consumer() ».
- Le test **Test3** appelle la séquence « connect_push_supplier() - disconnect_push_consumer() – push() ».
- Le test **Test4** appelle la séquence « connect_push_supplier(nil) - push() ».

	Visibroker	**Exemplar**	**DSTC**	**OpenFusion**
Test1	Aucune erreur n'est détectée, aucune exception n'est levée.	L'invocation de connect lève l'exception OBJECT_NOT_EXIST	Aucune erreur n'est détectée, aucune exception n'est levée.	L'invocation de connect lève l'exception OBJECT_NOT_EXIST
Test2	La deuxième invocation de disconnect lève l'exception OBJECT_NOT_EXIST	La deuxième invocation de disconnect lève l'exception OBJECT_NOT_EXIST	Aucune erreur n'est détectée, aucune exception n'est levée.	La deuxième invocation de disconnect lève l'exception BAD_PARAMETER
Test3	L'invocation de push lève l'exception OBJECT_NOT_EXIST	L'invocation de push lève l'exception OBJECT_NOT_EXIST	L'invocation de push lève l'exception DISCONNECTED	L'invocation de push lève l'exception BAD_PARAMETER
Test4	L'invocation de connect échoue silencieusement et push lève ensuite l'exception DISCONNECTED. **Non conforme à la spécification de l'OMG.**	L'invocation de connect lève l'exception BAD_PARAMETER. **Non conforme à la spécification de l'OMG.**	Aucune erreur n'est détectée, aucune exception n'est levée.	Aucune erreur n'est détectée, aucune exception n'est levée.

Figure 14 : Synthèse des résultats expérimentaux

Les tests que nous avons effectués sur les 4 implémentations de l'Event Service ne prétendent nullement à l'exhaustivité. Leurs résultats sont cependant éloquents : on remarque en particulier que les implémentations ne s'accordent sur le comportement à donner pour aucun des quatre tests détaillés ici.

Les différences entre les implémentations sont de natures diverses :

- Pour Test1 et Test2 certaines implémentations acceptent la séquence et fonctionnent ensuite correctement. D'autres lèvent une exception, ce qui était également notre choix de spécification.
- Pour Test3, toutes les implémentations lèvent une exception, mais sont en désaccord sur le type de l'exception à lever. Les implémenteurs de DSTC ont fait le choix de lever l'exception DISCONNECTED. Lever OBJECT_NOT_EXIST nous paraît être un

meilleur choix, car il permet à l'Event Channel de libérer les ressources qu'il a allouées lors de l'instanciation du proxy.
- Test4, enfin, met en lumière les dysfonctionnements les plus graves : deux implémentations sont clairement en contradiction avec les spécifications de l'OMG. Ces deux implémentations, de plus, n'échouent pas de la même façon.

Quelles conclusions faut-il tirer de cette expérience ?
- Les 4 implémentations que nous avons testées sont incompatibles (elles ne sont pas **substituables**, au sens de Liskov [Liskov & Winy, 94]) : un client qui fonctionne correctement avec une de ces implémentations pourra échouer si on substitue une implémentation différente. Par exemple, un client qui exécute plusieurs fois l'opération *disconnect* (lors d'une séquence de déconnexion mal maîtrisée, par exemple) peut fonctionner parfaitement avec DSTC, mais échouer avec VisiBroker.
- Aucune des implémentations dont nous avons pu disposer ne spécifie clairement les hypothèses additionnelles que les implémenteurs ont adoptées pour pallier les sous-spécifications du document de l'OMG. En pratique, cela veut dire qu'un utilisateur de ces implémentations devrait effectuer des tests similaires aux nôtres pour déterminer comment se comporte l'Event Service dans des conditions limites.
- Certaines implémentations (parmi les plus réputées) violent des spécifications explicites de l'OMG. Cela peut s'expliquer par la complexité des spécifications, et par le fait que l'OMG n'a pas adjoint à ses spécifications un jeu de tests de validation, qui permettrait d'établir expérimentalement la conformité d'une implémentation. Un des axes de recherche que nous poursuivons est justement de dériver automatiquement un tel jeu de test à partir de la spécification formelle du service (cf. § 3.4.4).

En ce qui concerne l'intérêt des approches formelles et l'applicabilité du formalisme des Objets Coopératifs, cette étude a également donné des résultats encourageants :
- La phase de spécification formelle à permis de détecter un certain nombre d'ambiguïtés et d'imprécisions dans les documents OMG. Il n'y a là rien de surprenant : la discipline imposée par l'usage d'une notation formelle oblige le spécificateur à se poser toutes les questions pertinentes, et à y répondre de manière complète et non ambiguë. Ce point n'est pas spécialement à mettre au crédit de notre notation, il est probable que ces problèmes auraient été détectés également avec une autre approche formelle. Ce qui est plus intéressant, c'est que les problèmes détectés ne sont pas de nature théorique : tous les implémenteurs y ont été confrontés, et ont malheureusement produit des implémentations incompatibles. Ce point à lui seul nous paraît justifier la nécessité impérieuse de disposer d'une notation formelle adaptée à CORBA.
- Le formalisme des Objets Coopératifs a « bien résisté » lors du traitement d'un problème d'une telle complexité. Les modèles produits sont restés de taille raisonnable, et les primitives du formalisme ont été suffisantes pour traiter le problème complètement. Toutefois, en traitant ce problème, nous avons pu constater que le formalisme pourrait bénéficier de quelques aménagements de nature syntaxique, qui permettraient de traiter avec plus d'élégance certaines constructions qui sont récurrentes dans le domaine de CORBA. Il s'agit en particulier de la gestion des exceptions, qui sont omniprésentes dans CORBA puisque chaque invocation peut lever des exceptions propres au fonctionnement distribué de l'ORB. Il nous apparaît nécessaire de définir des mécanismes permettant de gérer simplement les exceptions sans pour autant surcharger les modèles. Nous sommes actuellement au travail sur ce point, et j'en reparlerai en conclusion de ce chapitre (cf. § 3.4.3).

3.4 Perspectives

Les travaux en cours au sein de notre équipe dans le domaine des systèmes distribués à objets portent sur plusieurs axes :

- Axe théorique : il s'agit d'une part de prendre en compte autant que possible les résultats d'analyse issus de la théorie des réseaux de Petri pour vérifier les modèles d'Objets Coopératifs (§ 3.4.1), et d'autre part d'étendre la notation pour prendre en compte les aspects temporels dans la spécification (§ 3.4.2).
- Axe pratique : Nous voulons apporter des améliorations syntaxiques à la notation, afin de permettre une expression plus aisée de constructions qui sont très fréquentes dans le domaine (§ 3.4.3).
- Axe méthodologique : Nous voulons proposer une démarche de développement complète basée sur notre notation, depuis la spécification jusqu'au test (§ 3.4.4).

3.4.1 Développement des techniques d'analyse

J'ai mentionné au début de ce mémoire le dilemme qui se pose à nous entre le désir d'accroître le pouvoir d'expression de notre notation, et celui de lui conserver un certain pouvoir d'analyse mathématique.

A l'heure actuelle, l'outil PetShop (cd § 3.3.2) inclut plusieurs modules classiques d'analyse des réseaux de Petri (calcul d'invariants, évaluation du caractère borné, vivant ou réinitialisable du réseau). Ces différentes techniques d'analyse ont été développées depuis longtemps dans le cadre des réseaux Place/Transition. Dans notre outil, ces algorithmes d'analyse opèrent sur le *réseau sous-jacent* des ObCS (c'est à dire son ObCS dépouillé de tous les types de données des jetons). Comme l'ObCS est en général un réseau de haut niveau, l'interprétation des résultats fournis par ces modules d'analyse est délicate : en général, ces modules ne peuvent pas fournir d'assurance formelle sur la validité d'une propriété dans le modèle, mais peuvent seulement donner une indication sur la présence ou l'absence d'une propriété. Ces résultats conservent toutefois une valeur intéressante, à cause d'une caractéristique particulière au modèle de RdP de haut niveau que nous utilisons : le fait qu'une transition soit franchissable dans le RdP de haut niveau implique qu'elle est également franchissable dans le réseau sous-jacent. Les inscriptions du réseau de haut niveau ont toujours pour effet de réduire les possibilités de franchissement, et jamais de les augmenter.

Ceci permet dans certains cas de préserver les résultats d'analyse obtenus sur le réseau sous-jacent :

- Si le réseau sous-jacent est borné, le réseau de haut niveau est également borné. Si le réseau sous-jacent n'est pas borné, on ne peut rien affirmer sur le réseau de haut niveau : son caractère borné peut en effet être rétabli par les inscriptions du réseau.
- Si le réseau sous-jacent n'est pas vivant, le réseau de haut niveau ne l'est pas non plus. Par contre, le fait que le réseau sous-jacent soit vivant n'implique pas que le réseau de haut niveau le soit : les inscriptions de ce dernier peuvent détruire sa vivacité.

Malgré leur caractère partiel, ces résultats d'analyse apportent une aide très importante lors de l'élaboration des modèles, comme nous avons pu le constater nous-mêmes à maintes reprises. Avec un peu d'expérience des réseaux de Petri, le modélisateur s'attend en général à constater la présence de certaines propriétés dans ses modèles, et même des résultats partiels peuvent lui être utiles pour gagner une confiance accrue dans la qualité de sa modélisation. Prenons le

cas la vivacité : Le modélisateur s'attend, par exemple, à ce que son réseau soit vivant. Le module d'analyse peut lui prouver l'absence de cette propriété, et donc le conduire à chercher la faille dans son modèle. Le module peut au contraire lui indiquer que le réseau sous-jacent est bel et bien vivant, mais sans garantie sur le réseau de haut niveau. Le modélisateur peut alors s'appuyer sur ce résultat pour développer une preuve ad hoc de la propriété sur le réseau de haut niveau, s'il estime que c'est nécessaire.

Nos travaux actuels dans ce domaine portent sur l'utilisation des techniques d'analyse propres aux RdP non seulement pour prouver des propriétés sur un objet isolé, mais aussi pour analyser des aspects caractéristiques des systèmes à objets. Ainsi, quand deux interfaces IDL sont reliées par la relation d'héritage, on s'attend à ce qu'une forme d'héritage du comportement [van der Aalst & Basten, 97] soit respectée pour les classes qui spécifient ces interfaces. On souhaite aussi analyser la coopération entre plusieurs instances, pour vérifier des propriétés sur un système d'objets en interaction. Ceci est rendu possible par le fait que le protocole d'interaction entre classes d'OC [Bastide et al., 99a] est lui-même spécifié en termes de réseaux de Petri, ce qui permet de générer un seul réseau de Petri statique à partir des ObCS de chaque classe d'un système, comme nous l'avons vu au § 2.4.

3.4.2 Prise en compte du temps

Dans sa définition actuelle, le formalisme des Objets Coopératifs permet de décrire le comportement dynamique d'un système d'objets en termes de relation de causalité, mais sans référence quantitative au temps réel qui s'écoule lors des invocations.
Dans le contexte de CORBA, il nous apparaît souhaitable d'être capables de modéliser le temps réel, notamment pour spécifier des phénomènes de timeout : un objet peut par exemple accepter d'attendre le résultat d'une invocation pendant un certain délai, et à l'expiration de ce délai choisir d'effectuer une action de remplacement (par exemple déclencher une alarme). Nous envisageons d'intégrer dans notre notation des techniques issues des réseaux temporisés dans le but de pouvoir décrire ces phénomènes. Nous évaluons à l'heure actuelle plusieurs stratégies d'intégration du temps dans les OC, mais la technique la plus adaptée semble être celle des transitions temporisées, en adoptant la stratégie dite « enabling memory policy » [Ajmone Marsan et al., 95].

3.4.3 Amélioration de la prise en compte des exceptions

Nous avons pu constater en traitant des exemples de taille conséquente (§ 3.3.3) l'omniprésence des exceptions dans un système distribué basé sur CORBA. Même si la signature d'un service n'inclut pas la définition explicite d'exceptions, l'invocation de ce service peut tout de même lever un certain nombre d'exceptions prédéfinies (TIMEOUT, OBJECT_NOT_EXIST…). L'extension du protocole d'invocation entre objets que nous avons défini en [Bastide et al., 99a] permet au client de récupérer et de traiter ces exceptions, mais parfois au prix d'une complexification importante de son ObCS. Nous évaluons à l'heure actuelle plusieurs extensions syntaxiques du formalisme visant à une prise en compte plus implicite des exceptions, à la manière des langages tels que Java ou Eiffel.

3.4.4 Génération de test fonctionnel

Nous avons pu constater (§ 3.3.3.5) que, dans l'état actuel de la pratique industrielle, il est difficile de se contenter de spécifications informelles pour développer de manière fiable et interopérable des serveurs CORBA. Une direction prometteuse pour améliorer cet état de fait consiste à combiner les techniques de spécification formelles et les techniques de test, pour permettre le test fonctionnel des composants développés [Gaudel, 95] : On peut alors définir deux rôles distincts dans le développement d'un système basé sur CORBA :

- Le « donneur d'ordre », a qui incombe la spécification fonctionnelle du système à développer. Il doit fournir l'IDL de ce système, assorti d'une spécification décrite dans une notation formelle adaptée (nous avons décrit en § 3.1 les qualités que doit offrir une telle notation, et nous pensons bien entendu que les Objets Coopératifs sont bien adaptés). Il fournit, en plus de la spécification fonctionnelle, un jeu de test qui servira à valider une implémentation. Ce jeu de test a été généré de manière automatique ou semi-automatique à partir de la spécification formelle.
- L'implémenteur, a qui incombe le développement logiciel de serveurs compatibles avec la spécification. Il reçoit la spécification formelle, qui lui indique sans ambiguïté ce qu'il doit implémenter, et le jeu de test, qui lui permet de vérifier que son implémentation est conforme.

A l'heure actuelle, l'OMG joue souvent le rôle de donneur d'ordre lorsqu'il publie ses spécifications de COS, et les vendeurs d'ORB jouent le rôle d'implémenteurs en fournissant les implémentations de services. Malheureusement l'aspect « test » est absent ce qui conduit à des dysfonctionnements sérieux comme nous l'avons montré en § 3.3.3.

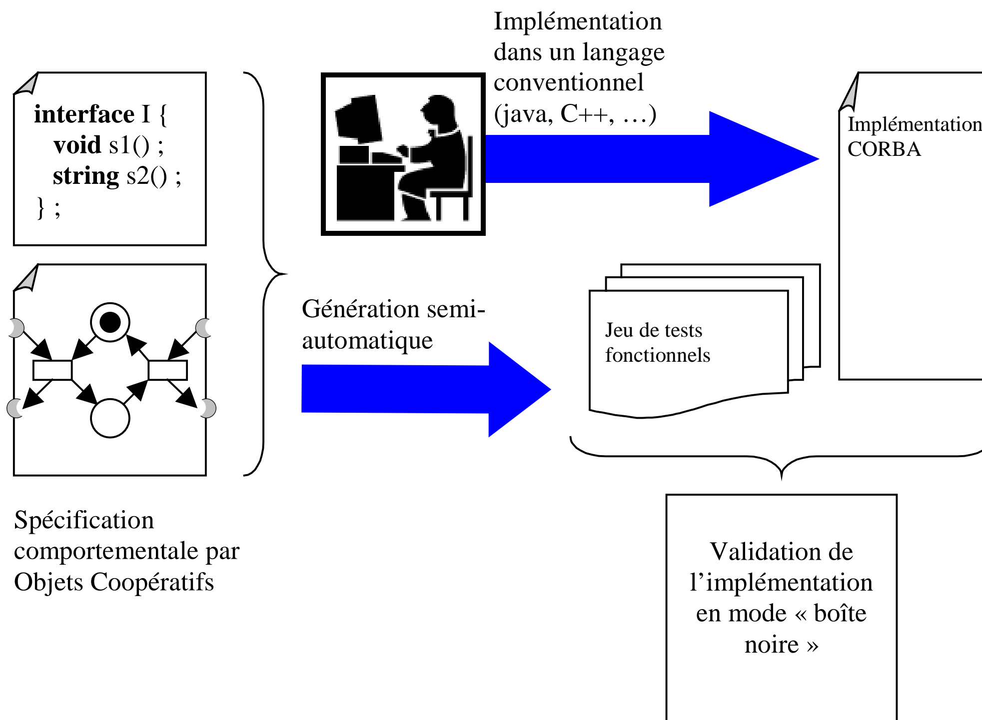


Figure 15 : Principes de la génération de tests fonctionnels

Cette approche basée sur la génération de jeux de tests nous paraît plus réaliste à moyen terme que des approches qui se fonderaient, par exemple, sur la génération automatique de code à partir de la spécification formelle. En ce qui concerne notre approche, une génération de code ne nous paraît pas possible, ni même souhaitable : la spécification par Objets Coopératifs est abstraite, dégagée de toute contingence d'implémentation. Le programmeur garde beaucoup de liberté (et doit faire preuve de créativité) pour développer l'implémentation de cette spécification. De plus la spécification n'inclut pas des exigences telles que la robustesse, la performance, la persistance, et ne prend pas en compte des contraintes technologiques telles que le choix d'utiliser un système de bases de données plutôt qu'un autre. Toutes ces

contingences doivent être prises en compte en plus de la spécification comportementale lors du développement.

La génération de test est actuellement à l'étude au sein du projet SERPICO. Nous cherchons notamment à intégrer les travaux déjà réalisés dans ce domaine au LRI et à l'EPFL [Barbey et al., 96b], [Barbey et al., 96a], [James & Gaudel, 99], et à voir comment ils peuvent s'appliquer à notre formalisme.
Les problèmes théoriques ne sont pas minces (prise en compte de l'indéterminisme dans les modèles, caractère concurrent des objets que l'on modélise, nécessité de définir un pilote de test qui soit lui-même concurrent pour émuler l'environnement normal de l'objet, problème de l'oracle…). Cependant, la démarche méthodologique a été très clairement défrichée par les travaux cités plus haut, et nous avons bon espoir d'arriver à des résultats concrets dans le cadre du projet SERPICO.

4 Spécifications formelles pour l'Interaction Homme-Machine

Les méthodes formelles sont encore peu utilisées dans le développement des systèmes informatiques interactifs. On leur reproche souvent d'être trop éloignées des principes et des techniques couramment utilisées par les informaticiens, de réclamer de ces derniers un bagage mathématique et théorique trop important, et surtout d'induire un surcoût en temps et en argent insupportable pour un projet classique. Ces arguments relèvent parfois davantage du mythe que de la réalité objective [Bowen & Hinchey, 94], mais force est de constater qu'on préfère souvent aux méthodes formelles des approches plus empiriques, basées par exemple sur le test intensif.

Il est cependant un domaine où les approches formelles sont bien acceptées et largement utilisées, c'est celui des systèmes critiques (safety-critical systems). Dans ce domaine, on considère que le surcoût du à l'utilisation des méthodes formelles est justifié par le fait que des vies humaines sont en jeu ou parce que le coût d'un dysfonctionnement du système serait largement supérieur au coût de sa conception.
Cependant, jusqu'à une époque assez récente, l'utilisation des méthodes formelles s'est cantonnée à la partie purement informatique (logicielle ou matérielle) du système critique. On rencontre cependant de plus en plus de systèmes critiques qui comprennent une composante « Interaction Homme-Système » importante : Un opérateur humain interagit avec le système critique, et les actions de cet opérateur peuvent avoir une influence déterminante sur la sécurité du couple homme-système dans son ensemble. A titre d'exemple de tels systèmes, on peut citer les poste de pilotage d'avion ou de train, les systèmes de contrôle aérien, etc.
Dans un tel système, il est fondamental que la partie interface homme-machine soit réalisée avec le même niveau de fiabilité que le reste du système, faute de quoi l'intégrité du système interactif dans son ensemble pourrait être compromise.

De manière assez surprenante, les travaux visant à l'application des méthodes formelles au domaine de l'interface homme-machine ne se sont développés qu'assez récemment. Certes, quelques travaux pionniers avaient débuté très tôt, par exemple ceux de David Parnas [Parnas, 69], Kieras et Polson [Booch, 94] ou van Biljon [van Biljon, 88]. Toutefois, ces travaux étaient restés dispersés et ne s'étaient pas fédérés en un champ de recherches bien défini. Au début des années 1990, on a vu apparaître des ouvrages dédiés à ce domaine [Harrison & Thimbleby, 90], [Dix, 91]. Vers le milieu des années 90, ce champ de recherches s'est cristallisé, notamment sur l'initiative du BCS FACS et du BCS HCI (British Computer Society, groupes Formal Aspects of Computer Science et Human-Computer Interaction) pour parvenir à l'émergence du domaine de recherche identifié comme « Formal Methods for Interactive Systems ». Des ateliers dédiés à ce sujet, puis des conférences, se sont développés. Ce thème a acquis droit de cité dans les programmes des conférences généralistes de l'interaction homme-machine. Notre équipe a d'ailleurs participé à ce courant en contribuant à

la fondation de la conférence DSV-IS (Design, Specification and Verification of Interactive Systems) qui est devenue un des forums importants pour les chercheurs de ce domaine [Palanque & Bastide, 95a].

4.1 Présentation du domaine

Mes travaux dans le domaine de l'interaction homme-machine ont touché deux sous-domaines :

- Le génie du logiciel interactif, où on montre l'intérêt d'utiliser des approches formelles dans le développement d'interfaces homme-machine.
- La modélisation de la tâche, où les approches formelles sont utilisées pour décrire le comportement ou l'activité des utilisateurs qui interagissent avec le système.

Un des postulats de ces travaux est que les méthodes formelles sont également utiles pour décrire l'aspect « machine » que l'aspect « homme » d'un système interactif, et que leur utilisation conjointe dans ces deux domaines apporte une synergie très riche.

4.1.1 Génie du logiciel interactif

La majorité des applications interactives développées à l'heure actuelle proposent à l'utilisateur un type d'interface graphique que l'on qualifie par l'acronyme anglo-saxon WIMP (Windows, Icons, Menus and Pointing). C'est pour ce type d'interfaces que les premiers standards de présentation et d'interaction ("look and feel") ont été définis et largement diffusés. Par leurs qualités ergonomiques (facilité d'utilisation, concision, cohérence, flexibilité, ...), ces interfaces ont immédiatement acquis la faveur des utilisateurs.

Une des raisons principales de ce succès provient du fait que l'utilisateur reste maître de l'interaction avec l'application tout au long de la session de travail. D'un point de vue informatique on dit que ces applications relèvent de la *programmation par événements* ; d'un point de vue ergonomique, on qualifie ces applications de *contrôlées par l'utilisateur*. Dans ce type d'interfaces, toute commande peut être initiée au moyen d'un élément graphique accessible par manipulation directe (icône, menu, bouton, ...). Ainsi, l'ensemble des commandes de l'application peut être visualisé à tout moment, réduisant d'autant la charge cognitive de l'utilisateur. L'interaction se caractérise à la fois par une grande liberté d'action (toute action licite à un instant donné est activable et présentée comme telle à l'utilisateur) et un niveau de guidage important (les actions illicites à un instant donné sont inactivables).

La mise en œuvre de ce type d'interfaces apporte toutefois une complexité nouvelle à la tâche des concepteurs. Pour le concepteur de telles interfaces la principale difficulté réside dans la conception du dialogue entre l'utilisateur et l'application. Le support de la communication entre ces deux interlocuteurs peut être considéré comme un langage et à ce titre peut être décrit en termes de lexique, syntaxe et sémantique. Ainsi, la conception du dialogue nécessite à la fois l'explicitation du langage d'interaction et de la répartition de la prise de parole dans cette interaction (c'est-à-dire la gestion de l'alternance du rôle d'émetteur et de récepteur entre l'homme et l'application).

Des problèmes spécifiques se posent tout au long du cycle de développement d'une application possédant une interface homme-machine contrôlée par l'utilisateur.
De nombreux modèles d'architecture ont été proposés pour les applications interactives. Le modèle ayant eu l'impact le plus important auprès des concepteurs d'application est sans nul

doute le modèle de Seeheim [Green, 86]. Ce modèle structure la communication homme-machine en trois composants conceptuels pour l'IHM : la présentation, le contrôleur du dialogue, et l'interface du noyau fonctionnel (voir Figure 16).

Une analogie a souvent été faite entre cette structure à trois composants et les trois niveaux classiques de description des langages : niveau lexical, niveau syntaxique et niveau sémantique. De nombreuses mises en gardes ont déjà été formulées à l'encontre d'une prise en compte trop littérale de cette structuration. Cette interprétation a servi de motivation pour l'usage de techniques de description de langages dans le cadre de la conception des IHM.

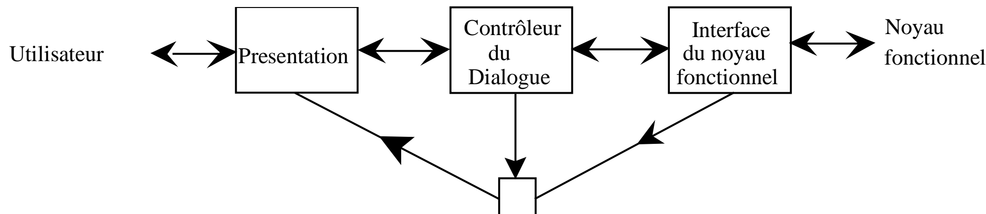


Figure 16 : le modèle de Seeheim

- Le composant "Présentation" prend en charge les aspects lexicaux de la communication homme-machine, à la fois en entrée et en sortie. Ce composant gère les interactions physiques et doit transformer les actions de l'utilisateur en unités de dialogues plus abstraites.
- Le composant "Contrôleur du Dialogue" supporte les aspects syntaxiques de l'interaction. Il structure l'interaction homme-machine à un haut niveau, et assure la répartition du contrôle entre l'utilisateur et le système. Il joue le rôle de médiateur entre le composant Présentation et le composant Interface du Noyau Fonctionnel.
- Le composant "Interface du Noyau Fonctionnel" donne une interprétation sémantique aux informations reçues du composant dialogue. L'exécution de l'action sémantique ainsi construite est à la charge du noyau fonctionnel lui-même.

Le composant "Contrôleur du Dialogue" pose des problèmes particuliers, du fait des caractéristiques spécifiques des applications contrôlées par l'utilisateur. Les travaux que nous avons réalisés dans le domaine de la spécification formelle des interfaces portent principalement sur l'ingénierie de ce composant et proposent une méthode formelle couvrant les différentes étapes de son cycle de développement : spécification, conception, validation.

4.1.2 Spécification du composant "Contrôleur du Dialogue"

La spécification du composant "Contrôleur du Dialogue" concerne plusieurs aspects : la description du **langage de commande** et la description des **interactions** entre le système (l'application que l'on spécifie) et son environnement (l'utilisateur).

- Le langage de commande de l'interface correspond à l'ensemble des séquences valides de commandes. Ces séquences peuvent être décrites en spécifiant l'ensemble des états possibles pour le dialogue et leurs relations [Coutaz, 90]. De nombreux formalismes tels que les grammaires hors-contexte [Olsen, 83], les diagrammes d'états et leurs dérivés [Wasserman, 85], [Booch, 94], [Jacob, 86] ont été utilisés pour modéliser le dialogue dans l'interaction homme-machine. Ces formalismes, adéquats pour la modélisation des applications directives, atteignent leurs limites dans le domaine des interfaces contrôlées par l'utilisateur. Ils sont en effet mal adaptés à la modélisation d'activités concurrentes (particulièrement nécessaire pour les dialogues multi-fils), manquent souvent de mécanismes permettant de structurer les modèles et de prendre en compte l'aspect

structure de données. Un certain nombre d'extensions, visant à remédier aux problèmes de ces formalismes ont été proposés, malheureusement le plus souvent au détriment de leur définition formelle. Pour illustrer ce propos, on peut citer [van Biljon, 88] qui remarque que « *de nombreuses extensions apportées aux automates à états ont simplement pour but de leur donner un pouvoir d'expression approchant celui des réseaux de Petri* ».

- Toute spécification se doit de décrire l'interaction entre le système spécifié et son environnement. Dans le cadre des IHM contrôlées par l'utilisateur cette spécification a des caractéristiques particulières : l'environnement est un être humain dont le comportement n'est pas modélisable. De plus, on souhaite donner à l'utilisateur le plus de liberté d'action possible dans ses interactions avec l'application, ce qui a pour effet d'accroître la difficulté de cette étape qui est particulièrement critique dans la mesure où la qualité de sa réalisation détermine l'utilisabilité de l'application.

4.1.3 Conception, implémentation et validation du composant Dialogue

La conception et l'implémentation du composant Dialogue posent des problèmes particuliers dans le cadre d'applications contrôlées par l'utilisateur. En effet, seules les techniques relevant de la programmation par événements permettent une implémentation efficace de telles applications. La mise en œuvre de ces techniques pose des problèmes que l'on ne rencontre habituellement que dans le cadre de la conception d'applications réactives.

Les méthodes formelles ont pour but de donner une description en intention de l'espace d'état et les techniques de validation raisonnent sur cette description sans énumérer explicitement l'ensemble des états possibles, ni les séquences de commandes conduisant à ces états. Un autre intérêt est de permettre la preuve d'un certain nombre de propriétés sur les modèles avant leur implémentation.

4.1.4 Les approches formelles dans les interfaces homme-machine

Les problèmes liés à l'ingénierie des systèmes interactifs sont aujourd'hui largement reconnus et bien identifiés. Les travaux dans ce domaine sont nombreux, et bien que partant des mêmes constatations empiriques, proposent des points de vue et des solutions remarquablement variées.

Malgré cette grande diversité, on peut dégager des constantes dans les approches proposées, ce qui nous paraît témoigner du fait que l'on s'approche d'une bonne maîtrise du domaine et de ses caractéristiques. Dans les approches les plus récentes, on constate toujours les caractéristiques suivantes :

- **une exigence de formalisation :** les approches simplement fondées sur l'utilisation d'outils, d'aussi haut niveau soient-ils, ont montré leurs limites. La plupart des propositions récentes s'appuient explicitement sur des approches formelles, en utilisant le plus souvent des formalismes généraux plutôt que de proposer des notations ad hoc. La conception du dialogue bénéficie ainsi des avancées réalisées dans d'autres domaines du génie logiciel ;
- **la prise en compte des aspects "structures de données" et "structures des traitements" :** les paradigmes de l'approche à objets ont clairement fait apparaître qu'on ne peut maîtriser la conception d'un système sans prendre en compte ses aspects statiques (ou structurels) en même temps que ses aspects dynamiques (ou comportementaux). Les méthodes formelles de conception d'IHM intègrent donc toutes ces deux composantes, souvent en tentant de marier le plus harmonieusement possible deux formalismes dédiés chacun à l'un de ces deux aspects ;
- **l'introduction de mécanismes de structuration :** comme n'importe quel autre composant logiciel, l'interface doit être intelligible, réutilisable et maintenable. Tout formalisme se

doit donc d'introduire des mécanismes de structuration et de composition, afin de produire des composants de taille suffisamment réduite pour que leur validation soit simple et qu'ils restent compréhensibles. Cette structuration doit autoriser et favoriser la construction de nouvelles interfaces à l'aide de composants prédéfinis et parfaitement spécifiés, et permettre la modification ou l'extension aisées d'interfaces déjà construites ;

- **la prise en compte explicite du parallélisme :** ce point peut paraître surprenant au premier examen. Tout d'abord, les interfaces sont souvent mises en œuvre au moyen de langages de programmation conventionnels et séquentiels. De plus, l'être humain lui-même, qui sera en définitive le moteur de l'interaction homme-machine, a des capacités d'action concurrente très limitées. Toutefois la prise en compte du parallélisme au niveau de la spécification est de première importance. En effet, même si l'utilisateur n'interagit avec le système que par l'intermédiaire d'un dispositif purement séquentiel (par exemple un clavier), les systèmes multifenêtres lui permettent de mener de front des tâches parallèles. Ces tâches sont même le plus souvent concurrentes, en ce sens qu'elles peuvent évoluer de manière indépendante, mais doivent se synchroniser et coopérer lorsqu'elles accèdent à des données qu'elles partagent. Il est donc nécessaire, lors de la spécification, de disposer d'un formalisme permettant de représenter cette concurrence conceptuelle. D'autre part, au point de vue purement technique, de nombreux systèmes d'exploitation modernes proposent des primitives de parallélisme interne (« threads ») qui sont particulièrement utiles pour augmenter le niveau d'interactivité et de réactivité du dialogue, et qui sont exploitées dans de nombreux SGIU. Ces primitives sont toutefois d'assez bas niveau, et il est utile de disposer d'un formalisme permettant d'utiliser des primitives concurrentes tout en faisant abstraction des détails d'implémentation.

4.1.5 Modélisation de la tâche

Un être humain utilise un artefact quelconque (en ce qui nous concerne, cet artefact est un logiciel interactif) comme un outil pour parvenir à un *but* qu'il s'est fixé. Nous considérons qu'un but peut être décrit comme un état accessible du couple humain/système interactif. Une *tâche* est un ensemble d'actions (physiquement ou mentales) que doit effectuer l'utilisateur pour atteindre son but. Une tâche peut être structurée, c'est-à-dire composée d'un ensemble de sous-tâches elles-mêmes décomposables. L'élément terminal et non décomposable de cette structure est *l'action*. Un modèle de tâches n'est pas nécessairement déterministe dans la mesure où il peut décrire des alternatives offertes à l'utilisateur. A un haut niveau d'abstraction, il est fréquemment concurrent.

Dans beaucoup de travaux les modèles de tâches sont exclusivement des descriptions de *structures de contrôle* des actions des utilisateurs (voir par exemple la notation pour décrire les tâches dans UAN [Hartson & Hix, 89]), se privant ainsi de la possibilité de décrire l'influence des données manipulées par l'utilisateur lorsqu'il désire atteindre un but. Les approches plus récentes cherchent à intégrer l'aspect structure de données et l'aspect structure de contrôle. Ainsi la méthode DIANE a été étendue dans [Tarby, 93] pour intégrer les objets nécessaires à la réalisation des tâches. Dans le même esprit, TLIM [Paternò & Mezzanotte, 95] associe à chaque modèle de tâches un objet.

Une fois que l'on a décidé des informations qui doivent être décrites dans le modèle de tâches, il reste à résoudre le problème du choix (ou de la définition) d'une technique de représentation pour ces informations. Par exemple, même si la notation proposée par UAN utilise des opérateurs proches de ceux de LOTOS, les auteurs mettent en avant l'extensibilité de leur notation en proposant à chaque utilisateur de définir, si nécessaire, d'autres opérateurs au gré de leurs besoins de modélisation. Les arguments développés par les auteurs des notations

présentées jusqu'ici se fondent sur le postulat selon lequel le processus d'analyse des tâches est informel et que, par conséquent, les tâches doivent être représentées au moyen de notations informelles. Un des principaux problèmes qui découle de l'utilisation de ces notations est le fait qu'il existe plusieurs interprétations possibles des modèles de tâches représentés. La notation peut donc réintroduire des ambiguïtés dans les modèles qui avaient pu être levées lors de l'analyse de l'activité des utilisateurs.

C'est pour éviter ce genre d'inconvénients que nous avons décidé d'utiliser une technique de spécification formelle permettant de représenter les données et les tâches de façon intégrée, pour décrire les activités, les tâches et les buts des utilisateurs.

4.2 Le projet ESPRIT Mefisto

Le projet MEFISTO est un projet Reactive Long Term Research du programme Esprit. Il a démarré le 18 septembre 1997 et a une durée de 3 ans. Il comprend 6 partenaires et 2 partenaires associés.

Les 6 partenaires sont:

- Alenia (Entreprise de développement logiciel, Rome, Italie)
- CENA (Centre d'Etudes de la Navigation Aérienne, Toulouse, France)
- CNUCE-C.N.R. (Centre National de Recherche Scientifique, Pise; Italie)
- D.R.A. (Defense Research Agency, Malvern, UK)
- Université Toulouse I (par l'intermédiaire du laboratoire d'informatique LIHS, France)
- University of York (York, UK)

Les 2 partenaires associés sont :

- E.N.A.V. (association de contrôleurs aériens, Rome, Italie)
- University of Siena (Sienne, Italie)

Coût total du projet : 2,048 KECU.
Financement de la communauté Européenne : 1,024 KECU
Financement attribué au laboratoire LIHS (100%): 151 KECU soit 1 200 000 FTTC.
Responsable scientifique du projet (pour l'UT1) : Philippe Palanque
Responsable administratif du projet (pour l'UT1) : Rémi Bastide

4.2.1 Problématique scientifique

Le projet Esprit Mefisto a pour but de contribuer à la conception des interfaces utilisateurs pour les systèmes interactifs critiques, en portant une attention toute particulière au domaine du contrôle aérien (ATC pour Air Trafic Control). Un système est considéré comme critique si le coût d'une défaillance dans le fonctionnement du système est sensiblement plus important que le coût de développement du système. Parmi ces systèmes on trouve bien entendu le contrôle aérien, mais aussi le pilotage de centrales nucléaires ou de satellites, etc.

On prévoit un fort accroissement en volume du trafic aérien au cours des prochaines décades, et seule la mise en œuvre de solutions logicielles semble à même de permettre de gérer les problèmes liés à cet accroissement du trafic. A l'heure actuelle, le nombre et la durée des délais, plus particulièrement dans les heures de pointe (dont le coût est estimé à 2 milliards de dollars par an) montre que les systèmes actuels de contrôle ne sont pas toujours capables de répondre à la demande des compagnies aériennes. De plus, même si à ce jour en France aucun

accident d'avion ne peut être attribué au contrôle aérien, l'accroissement de trafic augmente les risques d'accident et requiert des techniques de gestion du trafic plus efficaces et plus sures. Ce type d'application met en évidence le besoin de méthodes informatiques permettant de soutenir la construction d'interfaces utilisateur à la fois fiables et utilisables.

Mefisto a pour but de développer des méthodes (et des outils informatiques associés) pour l'ingénierie de la conception, du développement et de l'évaluation d'interfaces utilisateurs. Même si le domaine du contrôle aérien est privilégié, les résultats devront être applicables à d'autres systèmes interactifs critiques.
L'originalité du projet se trouve dans l'application de nouvelles technologies matérielles (interface multimodale, liaison de données entre sol et bord…) et logicielles pour des systèmes interactifs et critiques. Le travail qui sera réalisé par les partenaires se situera dans l'optique de la définition et de l'utilisation de techniques de spécifications formelles.

4.2.2 Résultats attendus

Les résultats attendus de ce projet sont :

- Le développement d'une méthode (et des outils informatiques associés) qui permette aux concepteurs d'appliquer des techniques de spécification formelles sur des applications informatiques de grande taille. Les outils doivent permettre d'évaluer la fiabilité et l'utilisabilité des applications produites.
- La définition d'une approche qui permette de passer d'une spécification formelle d'une application interactive à un système interactif tout en conservant les propriétés établies en ce qui concerne la spécification.
- Le développement de techniques d'évaluation d'interfaces utilisateur pour le contrôle aérien dans le but d'en accroître l'utilisabilité. Ces techniques devront être soutenues par des outils logiciels dans le but, en particulier, de gérer la quantité d'information recueillie lors des évaluations.

4.3 Mes contributions dans ce domaine

Le point de départ de notre travail dans le domaine de l'interaction Homme-Machine portait sur la spécification de la partie « interface » de l'application interactive, plus précisément la partie « dialogue » du modèle de Seeheim. Les préoccupations de ces premiers travaux relèvent essentiellement du Génie Logiciel :

- On souhaite favoriser la concision, la complétude et la non-ambiguïté des spécifications en fournissant un formalisme spécialement adapté à ce domaine, mais qui conserve tout son caractère formel.
- On veut explicitement prendre en compte le paradigme dominant pour les interfaces modernes, qui est la programmation par événements
- On veut s'assurer de l'exécutabilité de nos spécifications, de manière à rendre possible un prototypage rapide à partir des spécifications formelles, et ainsi permettre d'impliquer l'utilisateur dans le développement dès la phase de spécification.
- On veut clairement définir comment notre formalisme s'intègre dans des architectures logicielles largement utilisées, telles que MVC par exemple.

Ces objectifs se sont concrétisés par le développement du formalisme des ICO [Palanque & Bastide, 95b] (Interactive Cooperative Objects), qui a fait l'objet de la thèse de Philippe Palanque, et qui est une spécialisation du formalisme des Objets Coopératifs au domaine de l'interaction homme-machine.

Je suis issu d'une formation et d'une culture essentiellement informatique, et il n'est donc pas surprenant que mes premiers travaux aient essentiellement porté sur la partie « machine » ou « logiciel » du couple homme-machine. Toutefois, au contact de la communauté multidisciplinaire des IHM, j'ai rapidement pris conscience du fait que la technique informatique ne peut à elle seule résoudre tous les problèmes. L'enfer de l'informatique est hélas pavé de projets réalisés dans les règles de l'art du Génie Logiciel, et qui ont purement et simplement été refusés par les utilisateurs.

Cette prise de conscience de l'importance des facteurs humains m'a incité à orienter mes recherches vers les techniques d'analyse et de modélisation de la tâche. Il ne s'agit plus ici de décrire le fonctionnement du système interactif informatique, mais bien d'analyser et de modéliser les activités typiques d'un utilisateur qui interagit avec le système. Le but premier d'un système interactif est d'assister un être humain dans la réalisation de ses tâches, en le déchargeant autant que possible des traitements qui sont le mieux réalisés par l'ordinateur. Dans un monde idéal, la conception d'un système interactif devrait donc commencer par une analyse détaillée des tâches des utilisateurs, suivie d'une modélisation de ces tâches dans un formalisme adapté. Le système interactif serait alors conçu d'après ce modèle des tâches, en s'assurant que la structure du système est parfaitement adaptée aux tâches des utilisateurs.

Cette vision idyllique est malheureusement loin de la réalité, pour des raisons d'ordre culturel :

> Dans le domaine de l'analyse des tâches, les notations ont essentiellement été développées par et pour des scientifiques issus des Sciences Cognitives. Le but de ces notations est de fournir des modèles clairs et concis, qui se rapproche de leur théorie cognitive sous-jacente. Les notions de complétude, de non-ambiguïté ou de sémantique formelle, chères aux informaticiens, sont très rarement prises en compte : ces notations sont au mieux semi-formelles. De nombreux auteurs et praticiens ont ressenti de grandes difficultés à intégrer les techniques orientées tâches avec les techniques propres au génie logiciel. La raison en est simple : on ne dispose pas, en général, d'une articulation claire entre les notations utilisées par les adeptes des sciences cognitives et celles utilisées par les informaticiens. Il n'y a pas non plus de vision claire d'un processus de développement d'une application interactive qui intègrerait les apports des sciences humaines et cognitives avec le savoir-faire des informaticiens.

Pour contribuer modestement à cette tentative de rapprochement entre sciences cognitives et informatique, nous avons fait porter nos travaux explicitement sur l'articulation entre la modélisation de la tâche (issus des Sciences Cognitives) et la modélisation des applications interactives, domaine que nous avions couvert dans nos premiers travaux. Ceci nous a conduits à étendre la portée de notre notation formelle au côté humain de l'interaction homme-machine.
Nous avons tout d'abord constaté que les réseaux de Petri sont bien adaptés à la modélisation de la tâche, du moins quand il s'agit d'en fournir une description détaillée[9]. Dès qu'on veut parvenir à un niveau fin de la description de l'activité d'un utilisateur, on se trouve confronté à

[9] Si on souhaite se cantonner à un niveau d'abstraction plus élevé, des techniques plus légères sont mieux adaptées, par exemple une décomposition hiérarchique en buts et sous-buts. Nous avons dans plusieurs articles montré qu'il était possible d'aborder la spécification des tâches de cette manière, puis de la raffiner en utilisant les réseaux de Petri.

des problèmes de modélisation qui sont précisément ceux que les réseaux de Petri de haut niveau traitent le mieux : activités concurrentes, synchronisations, indéterminisme, prise en compte des données. On pourrait dire que certains formalismes de description de la tâche « réinventent la roue », en proposant un jeu d'opérateurs idiosyncrasiques qui émulent (parfois sans le savoir) des constructions connues depuis longtemps dans d'autres domaines, tels quel l'algèbre des processus par exemple.

On peut retirer de multiples avantages du fait de disposer d'une notation unique permettant de décrire aussi bien l'activité de l'utilisateur que la dynamique interne du logiciel, avec une interaction parfaitement spécifiée entre ces deux modèles. Il devient notamment possible de vérifier la cohérence des deux modèles.

Notre approche a principalement consisté à fournir une aide au recueil des informations devant être représentées dans les modèles de tâches et une technique de description formelle permettant de représenter à la fois les tâches et les caractéristiques des utilisateurs effectuant ces tâches. Toutefois, ces contributions ont été réalisées de façon intégrée avec nos travaux dans le domaine du génie du logiciel interactif. Nous avons proposé une approche intégrée du processus de développement d'un système interactif basé sur une technique de spécification formelle. Ce processus est présenté dans la Figure 17. Les éléments grisés correspondent aux points de départ du processus de conception d'un système interactif. On voit ainsi que la conception peut être initiée :

- soit à partir d'un modèle de tâches qui correspond alors aux tâches réalisées par les utilisateurs sur le système existant (qui peut être ou non un système informatisé) ;
- soit à partir d'un modèle du système qui peut être, soit créé de toutes pièces quand on désire construire un nouveau système indépendamment de tout système existant, soit une représentation d'un système existant que l'on désire améliorer.

Une fois le premier modèle construit, le processus de vérification de propriétés est mis en œuvre pour déterminer si la construction est correcte. Par exemple, dans le cas d'un modèle du système on peut être intéressé par le fait que le système soit réinitialisable ou qu'il ne présente pas de blocages potentiels. Dans le cas d'un modèle de tâches, le concepteur est en général plus intéressé par prouver des propriétés telles que l'accessibilité d'un état (ce qui correspondra par exemple au but que cherche à atteindre l'utilisateur lorsqu'il effectue les tâches décrites dans ce modèle de tâches).

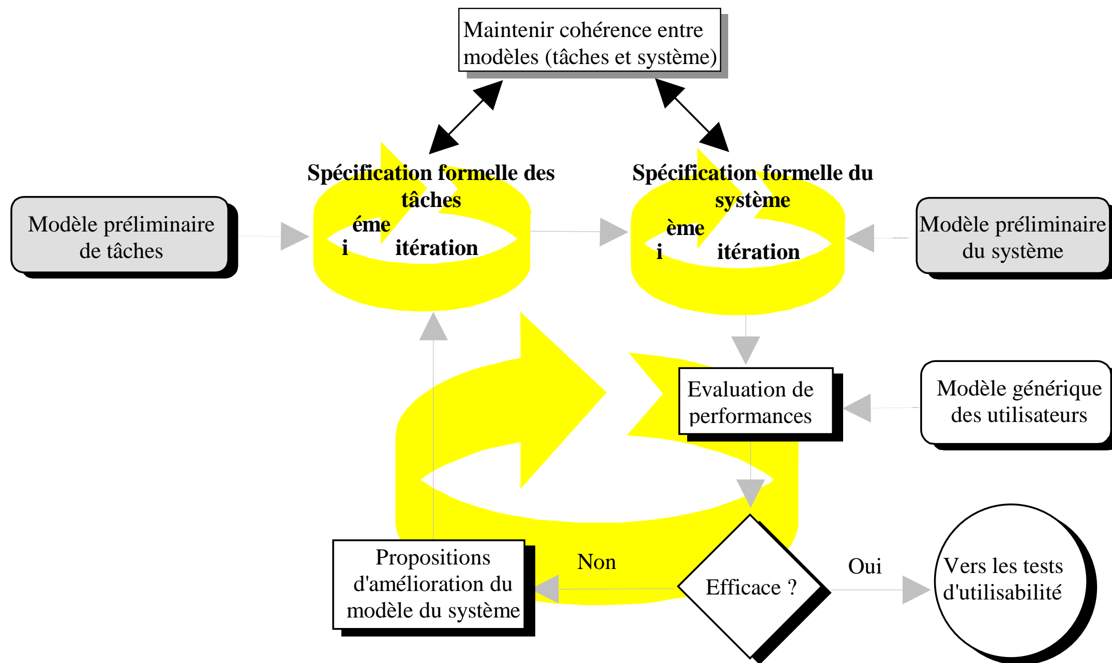


Figure 17 : Un cycle de développement de logiciels interactifs fondé sur l'utilisation des spécifications formelles

La Figure 17 représente le processus que nous proposons. Les composants « Spécification formelle du système » et "Spécification formelle des tâches » suivent eux-mêmes un processus itératif d'élaboration d'un modèle à l'aide d'une technique de description formelle qui est décrit dans la Figure 19.

L'action « Maintenir la cohérence entre modèles (tâches et système) » a pour objectif de vérifier que les actions qui doivent être réalisées par l'utilisateur dans le modèle de tâches sont effectivement offertes dans le modèle du système. Ce processus de vérification de compatibilité entre les modèles est similaire à celui utilisé pour vérifier la compatibilité de deux classes dans le modèle du système.

Une fois vérifiée la cohérence entre les modèles de tâches et du système, les modèles de tâches sont décorés par des informations temporelles sur les performances génériques des utilisateurs en matière d'interaction homme-machine. Il est important de noter que l'intérêt de cette étape est d'utiliser des valeurs de performance des utilisateurs pour comparer l'efficacité relative de plusieurs spécifications d'un système interactif. L'objectif n'est en aucun cas de donner une valeur prédictive du temps d'accomplissement d'une tâche avec le système spécifié. En effet, le modèle du processeur humain [Card et al., 83] que nous utilisons est trop simple ; il est reconnu que les valeurs en termes de performance des utilisateurs dépendent de l'environnement du travail (stress, bruit, etc.) et qu'aucune mesure actuelle ne concerne les interactions homme-machine. Toutefois, les calculs d'évaluation de performance sur les modèles de tâches fournissent certaines informations qui permettent d'évaluer si le système qui a été spécifié prend suffisamment en charge les actions de l'utilisateur. Par exemple, la présence de cycles dans les modèles de tâches signifie que l'utilisateur doit effectuer plusieurs fois les mêmes actions pour atteindre son but. Ce genre d'information peut servir pour voir s'il ne serait pas possible de faire réaliser cette répétition d'actions par le système ce qui revient à modifier le modèle du système.

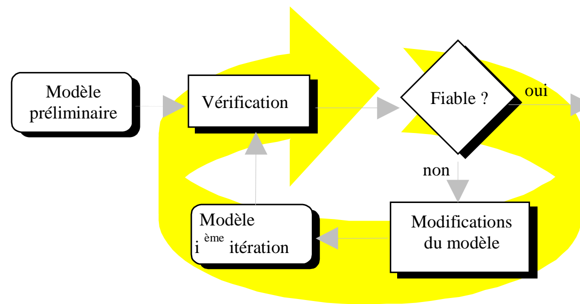


Figure 18 : processus d'élaboration d'un modèle à l'aide d'une notation formelle

Les tout premiers travaux que nous avons effectués dans le domaine de la prise en compte des modèles de tâches sont introduits dans l'article [Bastide et al., 95] qui est dédié à la description de l'utilisation des techniques de preuve sur les réseaux de Petri pour la vérification d'un système interactif formellement spécifié par ICO. Cet article montre également deux modèles de tâches différents applicables sur un même système et la façon dont on peut prouver que la séquence d'actions que doit réaliser l'utilisateur correspond bien à une séquence d'actions autorisée dans le modèle du système. Une autre partie de cet article établit un lien formel entre modèle de tâches et buts. Nous avons ainsi décrit qu'un but correspond à un état précis du modèle de tâches et que les techniques de preuve permettent de vérifier que l'état décrivant le but peut bien être atteint par une séquence d'actions effectuées sur le modèle de tâches.

Dans [Palanque et al., 95] nous avons proposé l'utilisation d'une passerelle entre une notation informelle pour décrire les modèles de tâches (la notation UAN) et le formalisme des ICO. Dans cet article nous appliquons le formalisme des ICO sur l'étude de cas "banc de test" du distributeur de billets. De façon parallèle, nous utilisons la notation UAN pour représenter la structure des tâches d'un utilisateur du distributeur de billets. Un algorithme de transformation des modèles UAN en réseaux de Petri de haut niveau est décrit et utilisé pour générer des modèles de tâches en réseaux de Petri à partir de ceux décrits en UAN. La dernière étape consiste à appliquer les techniques de vérification présentées dans le paragraphe précédent pour vérifier, au moyen des modèles de tâches, que les buts des utilisateurs peuvent bien être atteints en utilisant le distributeur de billets spécifié en ICO.

L'article [Bastide & Palanque, 96] pousse plus avant cette volonté d'intégration des modèles de tâches et de systèmes en proposant une première esquisse du cycle de développement de la Figure 17. Sur une étude cas très simple, plusieurs itérations sont ainsi réalisées sur ce cycle de développement. Ces itérations mettent en avant la possibilité de commencer la conception d'un système, soit à partir de modèles de tâches, soit à partir d'une spécification d'un système existant. Il introduit aussi le principe d'évaluation de la performance des modèles de tâches en vue de comparer l'efficacité respective de deux spécifications d'un système. Cette évaluation de performances est réalisée au moyen des techniques d'analyses disponibles pour les dialectes de réseaux de Petri permettant de représenter les aspects temporels quantitatifs. L'article [Palanque & Bastide, 96] présente précisément le modèle temporel quantitatif retenu. Il est basé sur les réseaux de Petri stochastiques [Ajmone Marsan et al., 95]. Les valeurs quantitatives ajoutées aux modèles proviennent du modèle générique de l'utilisateur.

L'article [Moher et al., 96] introduit une autre approche pour la prise en compte de l'utilisateur. Dans cet article, nous proposons la description dans un même modèle du système, de l'interface d'utilisation de ce système et du comportement (cognitif et physique) de

l'utilisateur. Cet article montre comment les connaissances qu'a l'utilisateur du système peuvent influer sur l'utilisation qui va en découler. Ce que le système montrera à l'utilisateur, à la fois au sujet de son comportement (les fonctions disponibles) et de son état courant, va ainsi non seulement influer sur l'utilisation qui va être faite du système, mais aussi sur la compréhension que l'utilisateur va avoir de ce système. Nous utilisons là encore un modèle de réseaux de Petri de haut niveau pour représenter toutes ces informations. Ces modèles qui décrivent le comportement de l'utilisateur en situation d'utilisation d'un système permettent aussi de représenter précisément les différentes stratégies d'appréhension d'un système interactif. L'application des même techniques d'analyse que précédemment permet ici de déterminer les stratégies les meilleures, non seulement en termes de nombre d'actions ou de durée d'utilisation mais aussi en termes d'informations apprises et/ou oubliées par l'utilisateur.

4.3.1 Publications choisies

Pour illustrer mes travaux dans ce domaine, j'ai choisi cinq publications représentatives. Les trois premières traitent du génie du logiciel interactif, et les deux suivantes traitent de la spécification formelle des tâches.

4.3.1.1 Thème « Spécification formelle des applications interactives »

Il s'agit ici d'appliquer le formalisme des Objets Coopératifs à l'ingénierie du logiciel interactif

"Spécifications formelles pour l'ingénierie des interfaces homme-machine." *Technique et Science Informatique* 14, no. 4 (1995) 473-500.
Cet article est une synthèse des travaux réalisés par Philippe Palanque et moi-même dans le domaine de l'ingénierie du logiciel interactif. Les motivations d'un formalisme spécifique à ce domaine (les ICO, développés dans la thèse de Philippe Palanque) sont établies.

"A Petri-Net Based Environment for the Design of Event-Driven Interfaces." *16th International Conference on Applications and Theory of Petri Nets, ICATPN'95*, Torino, Italy, June 1995. Giorgio De Michelis, and Michel Diaz, Volume editors. Lecture Notes in Computer Science, no. 935. Springer (1995) 66-83.
Cet article montre comment intégrer un interprète de réseau de Petri pour piloter le dialogue d'une application interactive. Les principes de fonctionnement de l'environnement PetShop sont présentés.

"Integrating Rendering Specifications into a Formalism for the Design of Interactive Systems." in *5th Eurographics Workshop on Design, Specification and Verification of Interactive Systems, DSV-IS'98*, Abingdon, U. K. June 3-5,1998. Springer-Verlag (1998)
Cet article traite plus spécialement de la communication dans le sens ordinateur ➔ humain, ce qu'on exprime en anglais par le terme "rendering", ou "retour d'information" en français. On décrit une taxonomie des retours d'information, et on montre comment les différents éléments de cette taxonomie peuvent être décrits au sein de notre notation formelle.

4.3.1.2 Thème « Modélisation de la tâche »

Dans ces articles, le formalisme est appliqué non plus à la spécification de l'application interactive, mais à la modélisation de la tâche et de l'activité de l'utilisateur qui interagit avec le système informatique. On montre l'intérêt d'utiliser le même formalisme pour décrire l'activité de l'utilisateur et le comportement du système utilisé.

"Synergistic Modelling of Tasks, Users and Systems Using Formal Specification Techniques." *Interacting With Computers* 9, no. 2 (1997) 129-53.

"Formal Specification As a Tool for the Objective Assessment of Safety Critical Interactive Systems." In *Interact'97, 6th IFIP TC13 Conference on Human-Computer Interaction*, Sydney, Autralia, July 14-18, 1997. Chapman et Hall (1997) 323-30.

4.3.1.3 Une application au domaine du contrôle aérien

Pour illustrer les travaux que nous avons menés à la fois dans le domaine du génie des logiciels interactifs que dans celui de la modélisation des tâches, je résume ici une étude que nous avons menée dans le domaine du contrôle aérien, et qui a été prolongée dans le cadre du projet Mefisto. L'étude complète se trouve dans l'article [Palanque et al., 97].

L'espace aérien est divisé en secteurs, et chaque secteur est géré par deux contrôleurs aériens qui travaillent de manière coopérative. Les informations sur les avions (identification, vecteur de vitesse) sont affichées sur un écran radar non interactif. Le contrôleur organise son travail en utilisant des rubans de papiers appelés strips. Les strips sont émis par un système informatique, qui les renseigne avec le plan de vol de l'avion. Le contrôleur annote ensuite manuellement le strip pour garder la trace de ses communications avec le pilote. Les communications entre pilote et contrôleur se font en utilisant un équipement VHF. La bande de fréquence est partagée par tous les avions d'un secteur, et tous les pilotes entendent donc l'intégralité des communications sur le secteur.

Cette organisation induit un certain nombre de problèmes :

- Limite de la bande passante : la fréquence de communication est partagée par tous les avions d'un secteur. Le trafic est parfois si dense que les contrôleurs doivent accélérer leur élocution pour trouver le temps de communiquer avec tous les pilotes, d'autant plus que, pour éviter toute ambiguïté dans les ordres, chaque communication doit impérativement commencer et se terminer par l'identification du contrôleur et de l'avion concerné.
- Mauvaise compréhension : l'utilisation de la voix comme médium de communication induit fréquemment des mauvaises interprétations des ordres ou des paramètres de ces ordres (altitude, orientation…)

Pour résoudre ces problèmes, une organisation différente appelée Data-Link est à l'étude dans de nombreux pays : l'idée générale est d'utiliser un médium de communication numérique entre sol et bord pour éviter les problèmes de bande passante et de compréhension. L'étude de ces systèmes s'accompagne en général d'une réflexion approfondie sur les nouveaux types d'interface homme-machine qu'il faut leur fournir. Une des stratégies que nous avons évaluées est de rendre l'écran radar interactif, et de l'utiliser comme une interface de communication avec les pilotes.

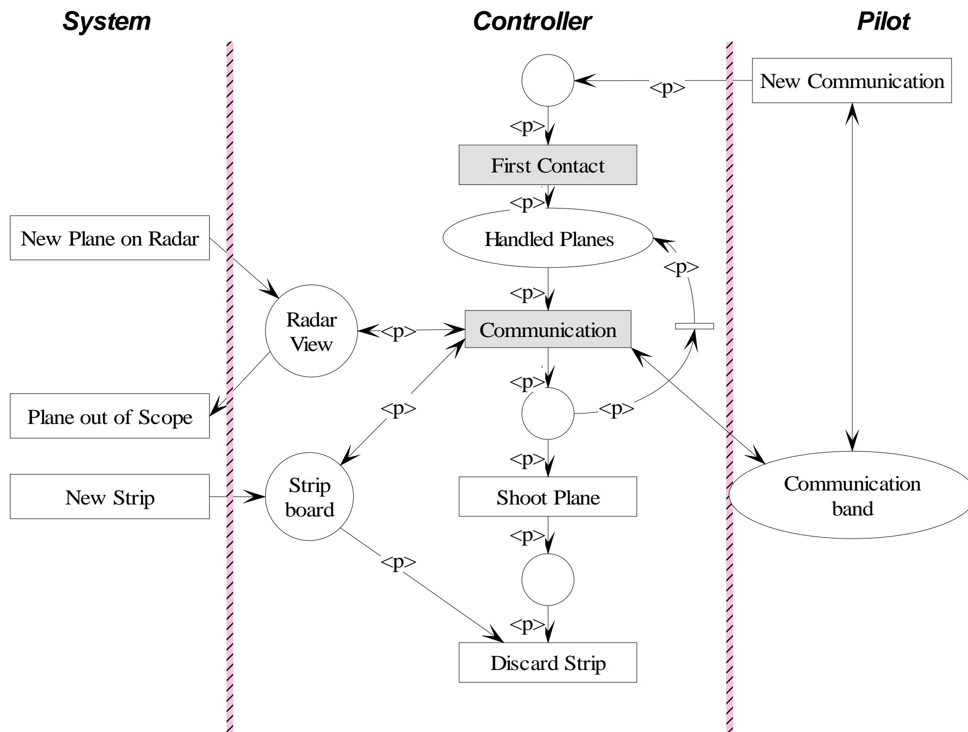


Figure 19 : Le modèle de tâches initial

La Figure 19 détaille le modèle de tâches d'un contrôleur dans le système actuel. Ce modèle montre comment les informations pertinentes pour le contrôleur (avions sur le radar, communications radio) sont fournies par le système ou par d'autres intervenants du système (en l'occurrence les pilotes). Une analyse de ce modèle de tâches fait apparaître un point particulièrement critique de l'activité du contrôleur : avant de communiquer avec un avion, le contrôleur doit faire une association mentale entre la représentation de l'avion sur le radar et le strip papier qui décrit le plan de vol de l'avion. Ceci est visible sur le modèle, au niveau de la transition « Communication ». Celle-ci, pour être franchissable, nécessite une unification entre des jetons contenus dans la place « radar view » (qui modélisent les avions visibles sur le radar) et des jetons contenus dans la place « strip board » (qui modélisent des strips papier). Cette capacité à décrire simplement l'influence des données sur la réalisation de la tâche est d'ailleurs très caractéristique de notre approche.

Sur un critère purement formel (la complexité du modèle des tâches) ont peut établir un problème cognitif potentiel que les utilisateurs pourraient connaître dans l'accomplissement de leur travail.

Une interface de type data-link permet de résoudre en partie ce problème, en faisant migrer la complexité depuis le modèle de la tâche vers celui du système. Le contrôleur n'a plus a faire l'identification entre l'avion sur le radar et le strip papier : il interagit directement sur le radar, et le système se charge de faire parvenir le message à l'avion qui a été désigné par manipulation directe.

Figure 20 : Le modèle de tâches avec data-link

Dans un système avec data-link, le modèle de la tâche du contrôleur se trouve simplifié, comme l'illustre la Figure 20 : la nécessité d'identifier l'avion par son strip papier avant de pouvoir communiquer avec lui a disparu. La simplicité du nouveau modèle de tâche donne une indication que ce choix de conception est positif : il a effectivement fait décroître la charge cognitive de l'utilisateur.

Notre travail dans le domaine de la modélisation de la tâche se base sur le postulat suivant :
Postulat : il est possible d'identifier des problèmes cognitifs potentiels dans un système homme-machine par l'analyse d'un modèle formel qui combine la tâche de l'utilisateur et le comportement du système.

Ce postulat a reçu une confirmation expérimentale que nous considérons comme très importante, d'autant plus qu'elle a été fournie par une équipe complètement indépendante de la nôtre :
Wendy Mackay et son équipe [Mackay et al., 98], dans le cadre d'une étude sur l'utilisation de la réalité augmentée dans le domaine du contrôle aérien, a conduit une étude ethnographique approfondie sur l'activité des contrôleurs aériens, et notamment sur la manière dont ils organisent leur travail autour des strips papier. Parmi les résultats de cette étude, elle a pu constater de manière empirique le même phénomène que nous avions prévu avec notre modèle théorique de la tâche des contrôleurs : l'activité qui consiste à relier l'avion présent sur l'écran avec sa représentation sous forme de strip papier est cognitivement complexe, et les contrôleurs eux-mêmes ont réclamé un lien direct entre le strip et le radar. Dans son expérimentation, Wendy Mackay a testé une interface de type « réalité augmentée » dans laquelle l'avion est automatiquement sélectionné sur le radar lorsque le contrôleur tape sur le strip avec son crayon. Cette nouvelle interface a effectivement pour effet de réduire la charge cognitive de l'utilisateur, en faisant en sorte qu'une activité qui relevait de sa propre activité (identifier un strip avec un avion sur le radar) passe dès lors sous la responsabilité du système, en déchargeant alors l'utilisateur.

Nous avons poursuivi nos travaux sur le contrôle aérien dans le cadre du projet Mefisto. Nous avons notamment développé des techniques de prototypage formel. La Figure 21 montre une spécification formelle (partielle) de l'application data-link, interprétée dans l'environnement PetShop. La figure montre également l'interface utilisateur de l'application data-link, qui est entièrement pilotée par le réseau en cours d'interprétation. L'environnement rend très simple la modification interactive de la spécification de l'interface, permettant ainsi de tester des alternatives de conception. Le développement de ce type de techniques est au cœur de la problématique scientifique du projet Mefisto.

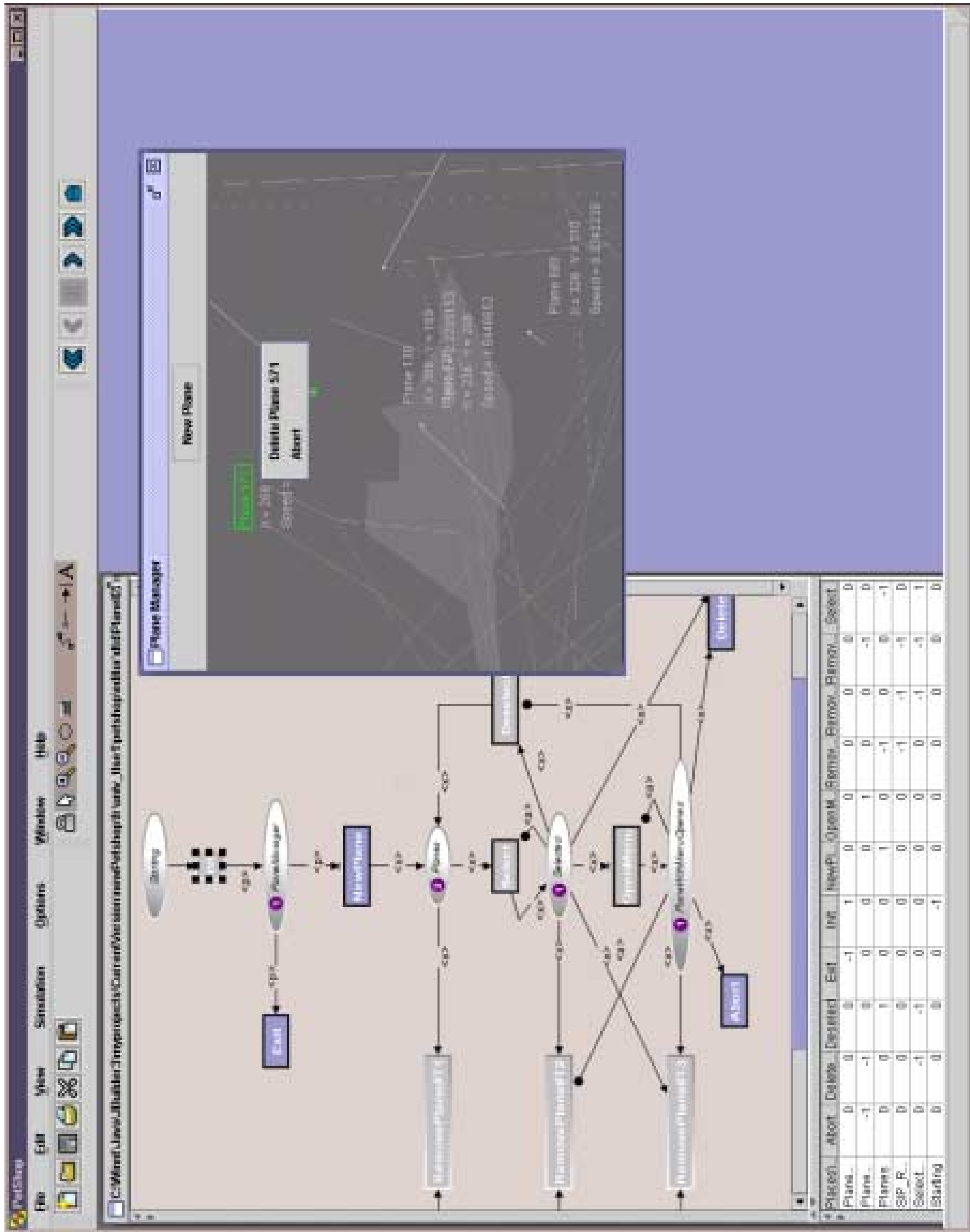


Figure 21 : L'application data-link en phase de prototypage dans l'environnement PetShop

4.4 Perspectives

A l'heure actuelle, la plupart des problèmes pratiques concernant l'utilisation de notre notation formelle dans le cadre des interfaces homme-machine ont été résolus de manière satisfaisante. L'intégration est maintenant effective au niveau des outils : on peut complètement piloter le comportement d'une interface homme-machine à partir d'un réseau interprété dans l'environnement PetShop. Les détails techniques de cette intégration sont donnés notamment dans l'article [Bastide et al., 98].

Nos travaux cherchent maintenant à améliorer l'intégration de notre outil avec les environnements de développement les plus modernes, et notamment les environnements à base de composants.
La conception d'une interface homme-machine se fait le plus souvent de manière visuelle, en assemblant graphiquement les différents éléments de l'interface. Les environnements à base de composant (et notamment ceux qui s'appuient sur le modèle JavaBeans, développé pour le langage Java) vont plus loin encore dans ce sens, en permettant de définir la dynamique de l'application de manière visuelle et graphique. On peut, par exemple, relier graphiquement un bouton de l'interface et une méthode offerte par un composant, pour signifier qu'un clic sur le bouton doit déclencher l'exécution de la méthode.

L'idée que nous poursuivons est d'encapsuler un Objet Coopératif sous la forme d'un composant JavaBean, de manière à pouvoir l'intégrer dans un environnement supportant ce standard, et permettre ainsi le développement visuel d'une interface utilisateur qui accède aux services offerts par l'OC.

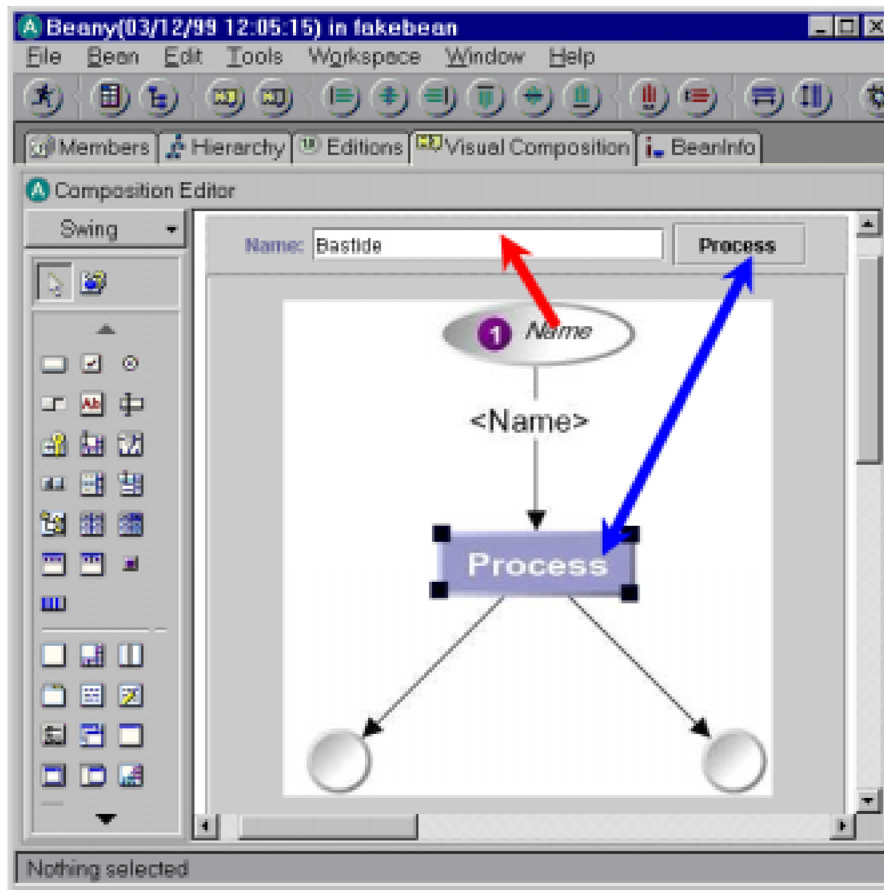


Figure 22 : Un scénario d'intégration des Objets Coopératifs dans un environnement à base de composants.

La Figure 22 illustre le type d'intégration que nous souhaitons atteindre : cette copie d'écran montre l'environnement VisualAge pour java (un des environnements de programmation à base de composants les plus aboutis). L'environnement offre une palette d'outils permettant la création visuelle de l'interface. La figure montre également les relations entre éléments d'interface et éléments du réseau de Petri : la zone de texte « Name » est une « vue » (dans le sens du paradigme Model-View-Controller) de la place correspondante du réseau. Chaque fois qu'un jeton est déposé dans la place, la valeur de ce jeton doit être visualisée dans la zone de texte. Le bouton « process » est à la fois une « vue » et un « contrôleur » de la transition du même nom : c'est un contrôleur, car un clic sur le bouton va déclencher le franchissement de la transition ; c'est une vue, car l'état du bouton reflète la franchissabilité de la transition : quand la transition est franchissable le bouton est actif, quand la transition n'est pas franchissable le bouton est grisé.

Nous sommes en phase d'évaluation de plusieurs scénarios possibles d'intégration dans de tels environnements, et nous évaluons également les possibilités et les limites de la technologie JavaBeans, mais c'est à ce type d'intégration que nous travaillons.

5 Conclusion

Mes premiers travaux ont porté sur l'intégration des concepts de l'approche Objet dans le domaine des Réseaux de Petri (RdP). Ces travaux participent d'un courant de recherche visant à augmenter l'expressivité des RdP pour pouvoir produire des modèles plus concis et plus proches du domaine du problème. Le résultat de ces travaux a été la définition d'un modèle de réseaux de Petri de haut niveau structuré suivant les concepts de l'approche à objets, les Objets Coopératifs (cf.§ 2.3) dans ma thèse de doctorat [Bastide, 92].

Plus récemment, mes travaux à l'intersection des RdP et de l'approche Objet se sont tournés vers le domaine des systèmes distribués à objets, notamment en relation avec le standard CORBA. L'objectif scientifique est ici de démonter l'utilité et l'utilisabilité d'une approche formelle fondée sur les RdP pour la spécification de tels systèmes. Cette démarche nécessite à la fois la rigueur théorique issue des approches formelles, la définition d'une démarche méthodologique apte à intégrer efficacement ces concepts dans un développement logiciel opérationnel, et la construction d'outils logiciels supportant à la fois à la notation formelle et la démarche méthodologique. Ces travaux sont au cœur de la problématique scientifique du projet SERPICO (cf. § 3.2).

A l'intersection des réseaux de Petri et de l'Interaction Homme-Machine, mes travaux portent sur l'utilisation des méthodes formelles dans le domaine des systèmes critiques qui comportent une composante interaction homme-machine importante. Ces travaux ont été menés depuis leur origine en collaboration étroite avec Philippe Palanque. A l'origine, ces travaux nous ont conduit à démontrer que les RdP sont un outil puissant et efficace pour la description du dialogue Homme-Machine, et se sont concrétisés par la définition du formalisme des ICO (Interactive Cooperative Objects) dans la thèse de Philippe Palanque [Palanque, 92] (cf. § 4.2). Nos travaux dans ce domaine se sont ensuite développés, nous conduisant notamment à développer l'utilisation de cette notation formelle au-delà du domaine de la machine elle-même, pour l'appliquer à la modélisation du comportement de l'opérateur humain, et plus précisément à la description de son « modèle de tâche » [Palanque & Bastide, 97]. Ces travaux nous ont notamment permis de démonter les avantages d'utiliser une notation commune pour la description de l'activité de l'opérateur et la description du comportement du système avec lequel l'opérateur interagit [Palanque et al., 97].

Mes travaux dans ce domaine ont également porté sur des aspects fondamentaux, participant à un courant de réflexion sur la nature profonde des systèmes interactifs, sur ce qui les différentie d'autres catégories de systèmes réactifs, et sur ce qui conduit au développement d'un corpus méthodologique et théorique propre à ce domaine. Un des résultats de ces travaux a été la définition d'une taxonomie des retours d'information dans un système interactif [Bastide et al., 98], et la prise en compte de cette taxonomie dans le formalisme des ICO.

A l'intersection des domaines « approche Objet » et « Interaction Homme-Machine », mes travaux portent actuellement sur la « programmation par composants ». Ces travaux, menés à la fois au sein du projet MEFISTO (§ 4.2) et du projet SERPICO (§ 3.2) ont notamment pour but d'intégrer les Objets Coopératifs Interactifs dans une architecture de composants du type Java Beans.

Enfin, au point central où se rejoignent réseaux de Petri, Interaction Homme Machine et systèmes distribués à objets, je situe mes travaux dans le domaine du collecticiel. Ces travaux ont été menés en partie au sein du groupe de travail SCOOP (Systèmes Coopératifs) du GDR-PRC CHM, où ils nous ont conduit notamment à produire un travail de synthèse sur les notations formelles dans le domaine du Workflow [Attali et al., 98]. Ces travaux ont également porté sur des aspects méthodologiques et organisationnels (thèses de Narjès Bellamine et de Renaut Zorolla).

6 Bibliographie

Ajmone Marsan, M., G. Balbo, C. Conte, Susanna Donatelli, and G. Franceschinis. *Modelling With Generalized Stochastic Petri Nets*. Wiley (1995).

Attali, Isabelle, Bastide, Rémi, Blay, Mireille, Dery, Anne Marie, and Palanque, Philippe. "Spécification formelle et approche objet pour les applications Workflow." *Technique et Science Informatique* 17, no. 2 (1998) 181-209.

Barbey, Stéphane, Buchs, Didier, and Péraire, Cécile. "Overview and Theory for Unit Testing of Object-Oriented Software." *Tagungsband "Qualitätsmanagement Der Objektorientierten Software-Entwicklung"*, Basel, Switzerland, October 24. (1996a) 73-112.

Barbey, Stéphane, Buchs, Didier, and Péraire, Cécile. "A Theory of Specification-Based Testing for Object-Oriented Software." *European Dependable Computing Conference (EDDC2)*, Taormina, Italy, October 1996. Lecture Notes in Computer Science, no. 1150. Springer-Verlag (1996b) 303-20.

Bastide, Rémi. "Objets Coopératifs : un formalisme pour la modélisation des systèmes concurrents." Ph.D. thesis, Université Toulouse III (1992).

Bastide, Rémi, and Palanque, Philippe. "A Petri-Net Based Environment for the Design of Event-Driven Interfaces ." *16th International Conference on Applications and Theory of Petri Nets, ICATPN'95*, Torino, Italy, June 1995. Giorgio De Michelis, and Michel Diaz, Volume editors. Lecture Notes in Computer Science, no. 935. Springer (1995) 66-83.

Bastide, Rémi, and Palanque, Philippe. "A Life-Cycle for the Formal Design of Interactive Systems." *BCS-FACS Workshop on the Formal Aspects of the Human-Computer Interface, FAHCI'96*, Sheffield, UK, September 1996. (1996)

Bastide, Rémi, Palanque, Philippe, Le, Duc-Hoa, and Muñoz, Jaime. "Integrating Rendering Specifications into a Formalism for the Design of Interactive Systems." in *5th Eurographics Workshop on Design, Specification and Verification of Interactive Systems, DSV-IS'98*, Abingdon, U. K. June 3-5,1998. Springer-Verlag (1998)

Bastide, Rémi, Palanque, Philippe, and Sengès, Valérie. "Task Model-System Model: Towards an Unifying Formalism." *6th International Conference on Human-Computer*

Interaction, HCI International'95, Yokohama, July 9-14, 1995. (1995) 489-94.

Bastide, Rémi, Palanque, Philippe, Sy, Ousmane, Le, Duc-Hoa, and Navarre, David. "Petri-Net Based Behavioural Specification of CORBA Systems." *20th International Conference on Applications and Theory of Petri Nets, ICATPN'99*, Williamsburg, VA, USA, June 21-25, 1999. Susanna Donatelli, and Jetty Kleijn, Volume editors. Lecture Notes in Computer Science, no. 1639. Springer (1999a) 66-85.

Bastide, Rémi, Sy, Ousmane, and Palanque, Philippe. "Formal Specification and Prototyping of CORBA Systems." *13th European Conference on Object-Oriented Programming, ECOOP'99*, Lisbon, Portugal, June 14-18, 1999. Rachid Guerraoui, Volume editor. Lecture Notes in Computer Science, no. 1628. Springer (1999b) 474-94.

Bastide, Rémi, Sy, Ousmane, and Palanque, Philippe. "Formal Support for the Engineering of CORBA-Based Distributed Object Systems." *IEEE International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, September 5-6, 1999. IEEE Computer Society (1999c) 264-72.

Battiston, E., and A. Chizzoni. *Modeling a Cooperative Development Environment With CLOWN*. Universita degli Studi di Milano, Department of Computer science, 1996.

Booch, Grady. *Object-Oriented Analysis and Design With Applications* Benjamin/Cummings (1994).

Bowen, Johnathan P., and Hinchey, M. G. "Seven More Myths of Formal Methods." *Formal Methods Europe, FME'94*, 1994. Springer-Verlag (1994)

Bryan, Doug. "Exactness and Clarity in a Component-Based Specification Language." *Object-Oriented Behavioral Specifications.* Haim Kilov, and William Harvey, editors. New-York: Kluwer Academic Publishers (1996) 1-15.

Bryant, Antony, and Evans, Andy. "A Formal Basis for Specifying Object Behaviour." *Object-Oriented Behavioral Specifications.* Haim Kilov, and William Harvey, editors. New-York: Kluwer Academic Publishers (1996) 17-30.

Structured Algebraic Nets With Object-Orientation. University of Geneva, CUI, 1995.

Card, Stuard K., T. P. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates (1983).

Coutaz, Joëlle, Author. *Interfaces Homme-Ordinateur: Conception Et Réalisation* Dunod, Paris (1990).

Dix, Alan J. *Formal Methods for Interactive System* Academic Press (1991).

Gaspari, Mauro, and Zavattaro, Gianluigi. "A Process Algebraic Specification of the New Asynchronous CORBA Messaging Service." *13th European Conference on Object-Oriented Programming, ECOOP'99*, Lisbon, Portugal, June 14-18, 1999. (1999)

Gaudel, Marie-Claude. "Testing Can Be Formal, Too." *TAPSOFT '95: Theory and Practice of*

Software Development. 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995. Lecture Notes In Computer Science, eds. G. Goos, J. Hartmanis, and J. van Leeuwen, no. 915. Springer Verlag, Heidelberg (1995) 82-126.

Green, Mark. "A Survey of Three Dialogue Models." *ACM Transaction on Graphics* . (1986)

Guttag, John V., James J. Horning, S. J. Garland, A. Jones, and J. M. Wing. *Larch: Languages and Tools for Formal Specification* Springer-Verlag, New-York (1993).

Harel, David, and Gery, Eran. "Executable Object Modeling With Statecharts." *IEEE Computer* 30, no. 7 (1997) 31-42.

Harrison, Michael, and Harold Thimbleby, Editors. *Formal Methods in HCI* Cambridge University Press (1990).

Hartson, Rex, and Hix, Deborah. "Human-Computer Interface Development: Concepts and Systems for Its Management." *ACM Computing Surveys* 21, no. 1 (1989) 5-92.

Hoare, C. A. R. *Communicating Sequential Processes* Prentice Hall (1985).

Hubert, P., Jensen, Kurt, and Shapiro, R. "Hierarchies in Coloured Petri Nets." *10th International Conference on Application and Theory of Petri Nets, ICATPN'89*, Bonn, Germany, 1989. Springer Verlag (1989)

Jacob, J. K. Robert. "A Specification language for Direct Manipulation User Interfaces." *ACM Transactions on Graphics* 5, no. 4 (1986) 283-317.

James, P. R., and Gaudel, Marie-Claude. "Testing Algebraic Data Types and Processes: a Unifying Theory." *Formal Aspects of Computing* 10, no. Special issue. Best papers of FMICS98 (1999) 436-51.

Jensen, Kurt. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. 2nd edition ed., Vol. 2 Springer-Verlag (1996).

Kilov, Haïm, and William Harvey, editors. *Object-Oriented Behavioral Specifications* Kluwer Academic Publishers (1996).

Krämer, B. "SEGRAS - A Formal and Semigraphical Language Combining Petri Nets and Abstract Data Types for the Specification of Distributed Systems." *9th International Conference on Software Engineering* , Washington, 1987. IEEE Computer Society Press (1987) 116-25.

Lakos, Charles. *Language for Object-Oriented Petri Nets*. Report #91-1. Department of Computer Science, University of Tasmania, 1991.

Lakos, Charles. "A General Systematic Approach to Arc Extensions for Coloured Petri Nets." *15th International Conference on Application and Theory of Petri Nets, ICATPN'94*, June 1994. Lecture Notes in Computer Science, no. 815. Springer (1994) 338-57.

Lakos, Charles, and Keen, C. D. "LOOPN++: a New Language for Object-Oriented Petri Nets." *European Simulation Multiconference*, Barcelona, Spain, June 1994. (1994)

Leavens, Gary T., and Yoonsik Cheon. *Extending CORBA IDL to Specify Behavior With LARCH*. TR #93-20. Iowa State University, Department of Computer Science, 1995.

Liskov, Barbara, and Winy, Jeanette M. "A Behavioral Notion of Subtyping." *ACM TOPLAS* 16, no. 6 (1994) 1811-41.

Mackay, Wendy, Fayard, Anne-Laure, Frobert, Laurent, and Médini, Lionel. "Reinventing the Familiar: Exploring an Augmented Reality Design Space for Air Traffic Control." *ACM Conference on Computer Human Interaction (CHI'98)*, Los Angeles (USA), 18-23 April 1998. ACM Press (1998) 558-65.

Merle, Philippe, Gransart, Christophe, Geib, Jean-Marc, and Laukien, Marc. "The CorbaScript Language." *ORBOS OMG Meeting on Scripting Languages*, Slides, Helsinki, Finland, 27-31 July 1998. ftp://ftp.omg.org/pub/docs/orbos/98-07-02.pdf.

Meyer, Bertrand. *Introduction à la théorie des langages de programmation*. InterEditions (1992).

Meyer, Bertrand. "Systematic Concurrent Object-Oriented Programming." *Communications of the ACM* 36, no. 9 (1993) 56-80.

Moher, Tom, Dirda, Victor, Bastide, Rémi, and Palanque, Philippe. "Monolingual, Articulated Modeling of Users, Devices and Interfaces." *Design, Specification and Verification of Interactive Systems'96, Proceedings of the Eurographics Workshop*, Université Notre-Dame de la Paix, Namur (Belgium), June 5-7, 1996. François Bodart, and Jean Vanderdonckt, editors. Wien, Springer-Verlag (1996) 312-29.

Object Management Group. *Common Object Services Specification /98-07-05*, Framingham, MA (1998a).

———. *Notification Service. Orbos/*, Framingham, MA (1998b).

Olsen, D. R. "SYNGRAPH : a Graphical User Interface Generator." *ACM Computer Graphics* . (1983) 43-50.

Palanque, Philippe. "Modélisation par Objets Cooperatifs Interactifs d'interfaces homme-machine dirigées par l'utilisateur." Ph.D. thesis, Université Toulouse I (1992).

Palanque, Philippe, and Rémi Bastide, Editors. *Design, Specification and Verification of Interactive Systems'95*, Springer-Verlag, Wien (1995a).

Palanque, Philippe, and Bastide, Rémi. "Spécifications formelles pour l'ingénierie des interfaces homme-machine." *Technique et Science Informatique* 14, no. 4 (1995b) 473-500.

Palanque, Philippe, and Bastide, Rémi. "Performance Evaluation As a Tool for Evaluating the Formal Design of Interactive Systems." *IEEE-SMC CESA'96* , Université de Lille, France, July 9-12, 1996. IEEE Press (1996) 328-33.

Palanque, Philippe, and Bastide, Rémi. "Synergistic Modelling of Tasks, Users and Systems Using Formal Specification Techniques." *Interacting With Computers* 9, no. 2 (1997)

129-53.

Palanque, Philippe, Bastide, Rémi, and Paternò, Fabio. "Formal Specification As a Tool for the Objective Assessment of Safety Critical Interactive Systems." In *Interact'97, 6th IFIP TC13 Conference on Human-Computer Interaction*, Sydney, Autralia, July 14-18, 1997. Chapman et Hall (1997) 323-30.

Palanque, Philippe, Bastide, Rémi, and Sengès, Valérie. "Validating Interactive System Design Through the Verification of Formal Task and System Models." *6th IFIP Conference on Engineering for Human-Computer Interaction, EHCI'95*, Garn Targhee Resort, Wyoming, USA, August 14-18. Chapman et Hall (1995)

Paludetto, Mario. "Sur La Commande De Procédés Industriels: Une Méthodologie Basée Objets Et Réseaux De Petri." Ph.D. thesis, Université Paul Sabatier (1991).

Parnas, D. L. "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System." *24th ACM Conference*. (1969) 379-85.

Paternò, fabio, and Mezzanotte, Menica. "Formal Analysis of User and System Interactions in the CERD Case Study." *6th IFIP Conference on Engineering for Human-Computer Interaction, EHCI'95*, Grand Targhee Resort, U.S.A., August. Chapman et Hall (1995)

Peterson, James Lyle. *Petri Net Theory and the Modeling of Systems* Prentice-Hall (1981).

Puntigam, F. "Types for Active Objects Based on Trace Semantics." *Formal Methods for Open Object-Based Distributed Systems.* Elie Najm, and Jean Bernard Stefani, editors. Chapman & Hall (1997) 4-19.

Ramamoorthy, C. V., and Ho, G. S. "Performance Evaluation of Asynchronous Concurrent Systems." *IEEE Transactions of Software Enginnering* 6, no. 5 (1980) 440-449.

Rational Software Corporation. *UML Notation Guide*. 1.1 ed.1997.

Sankar, Sriram. "Introducing Formal Methods to Software Engineers Through OMG's CORBA Environment and Interface Definition Language." *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96*, Munich, Germany, July 1-5, 1996. Martin Wirsing, and Maurice Nivat, Editors. Lecture Notes In Computer Science, no. 1101. Springer (1996) 52-61.

Schmidt, Douglas C., and Vinoski, Steve. "Overcoming Drawbacks in the OMG Events Service." *SIGS C++ Report* 9, no. 6 (1997)

Sibertin-Blanc, Christophe. "High-Level Petri Nets With Data Structure." *6th European Workshop on Petri Net and Applications*, Finland. (1985)

Sibertin-Blanc, Christophe. "A Client-Server Protocol for the Composition of Petri Nets." *14th International Conference on Application and Theory of Petri Nets, ICATPN'93* , Chicago, USA, June 21-25 1993. Lecture Notes in Computer Science, no. 691. Springer (1993)

Sivaprasad, Gowri Sankar. *Larch/CORBA: Specifying the Behavior of CORBA-IDL*

Interfaces. TR #95-27a. Iowa State University, Department of Computer Science, 1995.

Spivey, J. M. *La Notation Z*. Paris: Masson (1994).

Sy, Ousmane, and Rémi Bastide. *Spécification Du COS Event Et Évaluation Du Formalisme*. SERPICO/Lot 1/FOR2. Laboratoire LIHS - FROGIS, Université Toulouse I, 1999a.

———. *Compte Rendu De Tests Sur Une Sélection D'Implémentations Du COS Event* Laboratoire LIHS - FROGIS, Université Toulouse I (1999b).

Tarby, Jean Claude. "Gestion Automatique Du Dialogue Homme-Machine à Partir De Spécifications Conceptuelles." PhD thesis, Université de Toulouse I (1993).

Valk, Rüdiger. "Petri Nets As Token Objects: an Introduction to Elementary Object Nets." *19th International Conference on Application and Theory of Petri Nets, ICATPN'98*, Lissabon, Portugal, June 1998. Springer (1998)

van Biljon, W. "Extending Petri Nets for Specifying Man-Machine Dialogues." *International Journal on Man-Machine Studies* , no. 28 (1988) 437-55.

van der Aalst, W. M. P., and Basten, T. "Life-Cycle Inheritance, a Petri-Net Based Approach." *18th International Conference on Application and Theory of Petri Nets, ICATPN'97*, Toulouse, France, June 1997. Pierre Azéma, and Gianfranco Balbo, editors. Lecture Notes in Computer Science, no. 1248. Springer (1997) 62-81.

Vautherin, J. "An Algebraic Approach to High Level Petri Nets." *Eighth European Workshop on Application and Theory of Petri Nets*, Zarogoza, Spain, 1987. Universidad de Zaragoza (1987)

Wasserman, A. I. "Extending State-Transition Diagrams for the Specification of Human-Computer Interaction." *IEEE Transactions on Software Engineering* 11, no. 8 (1985)