THÈSE DE L'UNIVERSITÉ PIERRE ET MARIE CURIE
PARIS VI

en vue de l'obtention du titre de

Docteur de l'Université Paris VI

spécialité:
INFORMATIQUE

présentée par
Dan REGEP

Sujet de la thèse

L*f*P : un langage de spécification pour supporter une démarche de développement par prototypage pour les systèmes répartis

Soutenue le 17.12.2003 devant le jury composé de :

Claude Girault, Professeur à l'Université Pierre et Marie Curie..... **Président**
Laurence Duchien, Professeur à l'Université des Sciences et Technologies de Lille . **Rapporteur**
Didier Buchs, Professeur à l'Université de Genève..... **Rapporteur**
Charles-François Ducateau, Professeur à l'Université Paris 5..... **Examinateur**
Jean-Michel Couvreur, Maître de Conférences à l'Université de Bordeaux ..... **Examinateur**
Jean-Pierre Velu, Ingénieur de recherches à Sagem S.A. ..... **Examinateur**
Fabrice Kordon, Professeur à l'Université Pierre et Marie Curie ..... **Directeur**

Je tiens à remercier vivement :

A ma femme *Laura*, elle sait pourquoi ! **Aux membres du jury :** A Monsieur *Fabrice Kordon*, pour m'avoir proposé un tel sujet de thèse et pour avoir dirigé mes recherches d'une façon exemplaire. A Monsieur *Claude Girault*, pour m'avoir accepté au sein de son équipe et surtout pour m'avoir conseillé et aidé pendant les moments difficiles de ma vie de doctorant. Je tiens également à le remercier pour les nombreux séminaires, de qualité exceptionnelle, qu'il a organisé au LIP6. A Madame *Laurence Duchien*, Professeur à l'Université des Sciences et Technologies de Lille, pour l'intérêt qu'elle a porté à mes travaux. A Monsieur *Didier Buchs*, Professeur à l'Université de Genève, pour avoir accepté de rapporter mon mémoire. A Monsieur *Charles-Francois Ducateau*, Professeur à l'Université Paris 5, pour m'avoir honoré de sa présence comme membre du jury et pour m'avoir accepté au sein de l'équipe ACSI et guidé dans mon activité d'enseignant. A Monsieur *Jean-Michel Couvreur* et à Monsieur *Jean-Pierre Velu*, pour s'être intéressés à mes recherches. **Aux membres du groupe LIP6-SRC et surtout à :** *Jean-Michel Ilié*, pour sa collaboration d'enseignement, son caractère agréable et pour ses qualités humaines exceptionnelles. *Denis Poitrenaud*, pour son esprit pédagogique et son ouverture d'esprit. *Philippe d'Darche*, pour sa collaboration et pour son organisation exemplaires et pour sa présence d'esprit. *Luciana Arantes*, pour son sourire chaleureux ! *Xavier Blanc*, pour ses conseils et ses e-mails ! *Xavier Bonnaire*, pour son ouverture d'esprit, pour l'accueil chaleureux et pour le Quake. *Claude Dutheillet*, pour les bons conseils et le chocolat. *Bertil Folliot*, pour ses remarques. *Marie-Pierre Gervais*, pour sa convivialité. *Emmanuel Paviot-Adet*, pour les remarques sur LfP et son esprit collégial. *Lionel Seinturier*, pour ses conseils. *Pierre Sens*, pour ses conseils. *Isabelle Vernier-Mounier* & *Jean-Luc Mounier*, pour l'aide et leur présence agréables. *Lanfroy Thierry*, pour tous ces petits (mais très nombreux) coups de pouce. *Yann Thierry-Mieg*, pour sa collégialité, ses remarques utiles et pointues et pour ses idées sur la vérification et la sémantique de LfP. *Frederic Gilliers*, pour sa collégialité, son esprit d'équipe et pour ses remarques utiles. *Damien Collard*, l'amitié et pur le Quake 3. *Khristian Kury*, pour le support et les parties de Quake. *Florin Muscutariu*, pour être mon ami. **Aux membres de l'équipe ACSI et Programmation Internet de l'IUT de Paris 5** et surtout à : *Véronique Heiwy* et *Catherine Daubise*, pour leurs collaborations exemplaires. **A ceux qui m'ont aidée :** A Madame *Doina Cioranescu*, Professeur à l'Université Pierre et Marie Curie, pour son aide constante. A Monsieur et Madame le Professeur *Ioan* et *Miranda Nafornita*, pour leurs bons conseils et pour m'avoir offert la possibilité de suivre cette thèse. A monsieur *Yasid Sabeg*, le Président général du Groupe CS pour avoir financé une partie importante de cette thèse. **A ma famille & à mes amis ...**

Table des Matières

CHAPITRE 1

Introduction Générale

Non seulement la conception et la réalisation d'applications réparties deviennent sans cesse plus complexes mais de plus le marché exige qu'elles soient développées rapidement et qu'elles puissent évoluer constamment. Le problème des applications réparties est rendu plus difficile encore par :

- la constante évolution des standards et des systèmes d'exploitation,
- l'arrivée de nouvelles contraintes comme la mobilité, l'embarquement (au moins partiel) et le passage à l'échelle.

Un exemple typique réside dans le marché des services de télécommunications : le téléphone mobile cesse d'être un moyen de communication élémentaire pour devenir le point d'entrée de toutes sortes d'offres nouvelles (accès internet, outil de repérage par GPS, etc.). Dans ce contexte, les opérateurs fondent leur compétitivité sur les services complémentaires qu'ils offrent. Ces services doivent donc évoluer rapidement et rester compatibles avec les précédents, ce qui ajoute à la complexité des programmes embarqués à bord d'un terminal téléphonique.

Pour faire face à ces nouveaux défis, la communauté doit pouvoir considérer des techniques de développement nouvelles, basées sur la construction aussi rapide que possible de programmes exécutables et évaluables dans divers environnements d'exécution. Ces approches de développement nouvelles sont regroupées sous le terme «*prototypage*» et s'appuient sur des Langages de Description d'Architectures (ADL).

Nos objectifs ont donc été de définir un langage de spécification adapté aux applications réparties afin de capturer leurs éléments caractéristiques, de les développer dans le contexte du développement par prototypage et de les déployer de manière transparente sur des architectures hétérogènes. Nous avons eu constamment à l'esprit les contraintes suivantes :

- permettre la génération automatique des éléments qui assurent le contrôle réparti puisqu'il s'agit du point le plus délicat de ces systèmes,
- rester compatible avec les démarches de développement de l'industrie et en particulier avec des standards comme UML (Unified Modelling Language)**[90]**, afin d'apporter une aide aux ingénieurs sans perturber leur travail,
- supporter l'évaluation des prototypes par simulation mais aussi la vérification de leurs propriétés par utilisation des méthodes formelles.

1.1 Les différentes formes de prototypes

Un prototype est une aide à la mise en œuvre en vue de l'évaluation d'un système. Cela concerne aussi bien le matériel que le logiciel . Pour cela, les prototypes doivent être exécutables. Ils sont alors utilisés pour mieux formuler le cahier des charges et les spécifications d'un système et pour étudier des choix d'implémentation. La notion de «*prototypage rapide*» **[115]** ajoute un facteur de vitesse et de faible coût dans le développement de prototypes.

Le prototypage défini par l'IEEE **[19]** comme une approche «privilégiant le développement de programmes dès les premières étapes du cycle de vie du logiciel afin d'obtenir des réactions et des analyses utiles pour la suite du processus de développement»(1) apporte une solution en reliant plus étroitement spécification et produit. Cette définition du prototypage est diversement interprétée tout au long du cycle de vie du logiciel.

Maquettage

Une maquette est un prototype oublié lorsque l'on considère qu'il a apporté suffisamment d'informations. Ce type de prototypes est principalement utilisé dans la phase de conception mais parfois aussi pendant l'implémentation pour évaluer des choix de réalisation. Le principal avantage du maquettage est de n'avoir que peu de contraintes. On peut utiliser n'importe quelle technique de développement, des langages dédiés à un domaine (par exemple, pour la description d'ateliers flexibles), des simulateurs ou des langages de programmation. Une maquette tolère des restrictions qui seraient inacceptables dans l'environnement d'exécution final, ce qui est raisonnable dans la mesure où l'on ne veut investir que peu de ressources à un stade où le projet est encore en phase de négociation.

En fait, l'effort de réalisation ne contribue pas directement au produit final mais permet de mieux aprréhender les éléments qui permettront de le produire. Ainsi cet effort est perdu et, selon la manière dont il est analysé, ne détecte parfois pas des problèmes éventuels. Le plus grave est que, bien qu'elle contribue à l'élaboration du cahier des charges, la maquette ne diminue pas le fossé entre la spécification et la réalisation d'un système.

Prototypage incrémental

On peut voir le prototypage incrémental comme une «*approche éclairée de développement*». Le prototype est alors construit sur un «*prototype d'architecture*», lequel est fixé très tôt dans le projet (on recommande de le faire avant que 20% des ressources soient consommées). Cette architecture est alors vue comme une plate-forme d'accueil qui sera enrichie au fur et à mesure de composants nouveaux. A un stade donné, certains composants sont dans un état final, d'autres sont encore en phase de développement. Lorsque tous les composants sont dans un état de développement satisfaisant par rapport à leur spécification initiale, on considère le prototype comme achevé. Il est alors plus proche du produit final que dans le cas précédent.

L'avantage principal du prototypage incrémental est de ne requérir qu'une démarche méthodologique précise s'appuyant sur un environnement de développement et une approche rigoureuse du travail d'implémentation. En revanche, la robustesse du produit final dépend complètement de la pertinence du prototype d'architecture.

Prototypage par raffinements

Ces deux formes de prototypage s'appuient bien sur une notion de modèle, mais dans le maquettage, le modèle n'est qu'un instrument pour raffiner des spécifications fonctionnelles et dans le prototypage

(1) La citation originale est: «A type of prototyping in which emphasis is placed on developping programs early in the development process to permit early feedback and analysis in support of the development process».

incrémental, il est décrit directement avec le langage de programmation qui sera utilisé pour le développement. Au contraire, le prototypage par raffinements distingue le modèle du prototype. Le modèle est un ensemble de spécifications exécutables servant de base à l'évaluation ou si la notation le permet, à l'analyse. A partir de ce modèle, on obtient par génération de programmes une implémentation conforme qui peut s'exécuter dans l'environnement cible.

L'avantage de cette démarche est de permettre non seulement cette conformité mais aussi une plus grande flexibilité, au prix d'un lourd environnement de développement. Elle exige en effet des outils de modélisation, de simulation et/ou d'analyse, des générateurs de code, etc. En revanche, non seulement le modèle est plus maléable, mais le choix d'un générateur de code plutôt qu'un autre permet de jouer sur les choix d'architectures et les stratégies d'implémentation. Surtout la distinction entre modèle et produit ouvre un fort potentiel vers l'utilisation des méthodes formelles pour évaluer des propriétés du système. C'est pour cette raison essentielle que nous avons adopté cette démarche et cherché à l'étayer en concevant **L*f*P**, notre *Langage pour le Prototypage* et en montrant comme l'intégrer dans une méthode de développement.

1.2 Prototypage et méthodes de développement

Le prototypage par raffinements est déjà une réalité dans de nombreux domaines d'application. Citons en particulier la conception de systèmes sur puces (le modèle est alors exprimé en VHDL), la conception de systèmes de commande en avionique (approche basée sur les langages synchrones) ou les systèmes d'information (UML est alors utilisé comme langage de modélisation).

La tendance actuelle est de considérer le prototypage comme une aproche de développement dans laquelle on peut se référer à tout moment à une implémentation fonctionnant dans un environnement d'exécution réaliste. Cette référence permet, outre de répondre à des objectifs fonctionnels, d'analyser finement le comportement de l'application dans des conditions réalistes.

Mais ériger le prototypage en démarche de développement implique que les modèles soient définis de manière appropriée. Cela est particulièrement indispensable dans le domaine des systèmes répartis qui sont difficiles à capturer. Si la conception orientée objet de ces systèmes fournit une aide méthodologique aux concepteurs, le problème de la pertinence du langage de spécification subsiste de sorte que les approches actuelles ont du mal à capturer les aspects dynamiques de ces systèmes.

Cependant, on observe une tendance lourde dans le développement d'applications : le recours non pas à des langages de programmation mais à des modèles de spécification. A ce titre, la démarche MDA (Model Driven Architecture **[9]**) ou le développement basé sur des modèles **[13]** sont des événements marquants de cette évolution.

1.3 Contexte de nos recherches

Nos recherches ont débuté sur la base de travaux précédents, en particulier des expérimentations réalisées dans le thème Systèmes répartis et Coopératifs du LIP6, afin de faciliter l'usage des méthodes formelles et de la génération de code. Nous nous sommes en particulier inspiré des travaux d'A. Diagne sur les OF-Class **[29]** et de W. El Kaim sur H-COSTAM **[64]**.

Chacun dans leur domaine, OF-Class et H-COSTAM proposaient un langage de spécification de haut niveau. Le premier avait pour objectif la synthèse de spécifications formelles (au moyen de réseaux de Petri), le second, la description d'informations pertinentes pour la génération et le déploiement de pro-

grammes répartis. Malgré des résultats encourageants, il s'est avéré, lors d'expérimentations dans le cadre des projets CARISMA et FORMA, que ces propositions souffraient de quelques lacunes :

- Si OF-Class était efficace pour faciliter le travail de modélisation formelle, il se prêtait peu à la génération d'une application répartie.
- Si H-COSTAM proposait un modèle de composant et une notion élémentaire de liens de communication (à la manière de certains ADL aujourd'hui), il n'était pas pertinent comme support pour la vérification formelle.
- Pour etre utilisable dans une démarche de prototypage, il fallait utiliser OF-Class puis H-COSTAM, ce qui multipliait les transformations entre spécifications et les problèmes qu'elles engendrent.

Entre temps, les différentes notations pour la conception orientée objet avaient abouti à UML 1.4 **[90]** qui est devenu le standard, de fait incontournable, pour la conception et la réalisation d'applications. Dans le domaine des systèmes répartis, la démarche méthodologique promue impliquait clairement l'utilisation d'intergiciels dédiés dont plusieurs étaient pleinement opérationnels (par exemple, CORBA **[87]**, Ada95/DSA **[60]**, MPI **[119]**). Si l'utilisation conjointe d'UML et d'intergiciels facilitait la réalisation et le déploiement d'applications réparties, certains aspects de la modélisation restaient peu abordés, en particulier :

- la notion de contrat d'exécution proposée dans OF-Class et H-COSTAM,
- la notion de mécanisme de communication et la description des liaisons entre composants telles qu'on les trouve dans RM-ODP **[62]**.

Nous nous sommes donc fixé comme premier objectif de proposer une notation unifiée permettant la liaison avec les méthodes de vérification formelle ainsi que la génération de programmes. Nous avons également pris en compte les recommandations de standards comme RM-ODP (la notion de vues sur le système, la notion de point de liaison, la structuration d'un nœud dans un système réparti) et UML (modèle d'exécution compatible et d'architecture de l'application via le diagramme de classes).

Nous avons choisi de promouvoir la notion de média, permettant de séparer les aspects communication des aspects traitement dans la description du système et défini avec soin les liaisons entre les composants et le système de communication. Enfin, nous avons structuré la description des entités d'un système réparti (aspects traitements et communications) autour de la notion de contrat comportemental, qui permet de décrire de manière non ambiguë les protocoles d'interaction de chaque élément du système.

1.4 Plan de la thèse

Ce document est organisé sur plusieurs chapitres :

Le **Chapitre 2** sert comme échafaudage pour placer le contexte de notre travail, préciser nos objectifs de recherche et identifier les directions d'investigation à suivre. Après avoir illustré notre vision du style architectural d'une application répartie, nous nous intéressons au développement d'applications réparties conformes à leur spécification. La notion de modèle joue un rôle central; elle dénote à la fois une spécification et un prototype exécutable (on parle de prototypage évolutif par raffinements **[70]**). Notre analyse débouche sur une liste de critères de sélection pour un langage de spécification adapté à nos objectifs. L'adéquation du langage pour décrire l'architecture logicielle d'implémentation et la formalisation de la définition du langage sont les deux principaux critères.

Le **Chapitre 3** concerne la description d'applications réparties du point de vue de leur implémentation. Il présente une analyse systématique (qui couvre touts les aspects de modélisation, c'est dire la descrip-

tion des composants, des connecteurs et des ports ainsi que celle des relations de composition et de liaison) des principaux langages de description d'architectures. Nous présentons de plus les normes UML et RM-ODP. Cette analyse justifie le besoin d'une nouvelle notation formelle complémentaire et compatible avec UML, afin de synthétiser ces deux normes et d'apporter à UML les capacités d'un vrai ADL compatible avec l'architecture de RM-ODP.

Le **Chapitre 4** étudie la relation entre la conception et l'implémentation. Notre objectif est de réaliser l'indépendance de la conception face aux choix d'implémentation sans pour autant restreindre les capacités d'expression de notre langage. Notre idée novatrice est de concevoir un modèle d'exécutif canonique, neutre, basé sur un micro noyau de composants de communication et de services basiques. Pour cela nous analysons les principaux mécanismes de communication en local (e.g. les IPC) ou à distance (e.g. intergiciel, langages de programmation réparties etc.) afin de synthétiser :

- *un ensemble de modèles de communication simples*, normalisés.
- *une structure d'invocation abstraite* qui se prête aussi bien à la communication par messages ou par flots de caractères et à l'invocation de méthodes locales ou à distance,
- *un ensemble de services de base* suffisants pour la mise en œuvre des principaux types d'applications réparties.

Le **Chapitre 5** décrit notre langage de modélisation. Celui-ci peut être présenté comme un diagramme hiérarchique qui décrit l'architecture logicielle d'une application à l'aide des classes (patrons d'implantation) et des média (protocoles de communication) qui sont reliées au niveau de leurs ports de communication par des binders (contrats de liaison simples). Contrairement à UML qui utilise plusieurs diagrammes pour exprimer soit la structure soit le comportement, **L*f*P** propose une démarche de description hiérarchique qui intègre dans la description des classes et des média à la fois des aspects de spécification structuraux (déclarations de variables, méthodes et types locaux) et des spécifications comportementales (contrats comportemental d'une classe, protocole d'un média ou flot de contrôle d'une méthode). Notons que, **L*f*P** se distingue de UML sur plusieurs aspects. Au contraire de ce dernier une spécification **L*f*P** est :

- *centrée sur la description de l'architecture d'implémentation* d'une application répartie (et non celle de l'architecture logique). La description d'un système **L*f*P** se fait en termes de : 1) *classes d'implémentation instanciables* (non abstraites et élaborées), 2) *ports d'interaction physiques* (et non d'interfaces logiques UML), 3) *média de communication protocolaires* (et non des associations UML), 4) *binders* **L*f*P** *paramétrables* (qui offrent un puissance d'expression supérieure à celle des files de messages prioritaires UML et ROOM).
- *complète* car elle inclut *la structure* et *le comportement* détaillés, y compris ceux des méthodes.
- *unificatrice* car nous utilisons un seul type de diagramme (les **L*f*P**-BD) pour modéliser à la fois le contrats d'utilisation des classe, les protocoles de communication exhibées par les média ou le flot de contrôle des méthodes. Cela permet entre autres d'assurer la cohérence des modèles.
- *par raffinements* par raffinements vu que le comportement d'une application est réalisé par composition hiérarchique (et modulaire) de diagrammes de comportement. Chacune d'elles peut être raffinée pour décrire des prototypes de plus en plus perfectionnés.

Enfin, le **Chapitre 6** présente nos conclusions et les perspectives de recherche.

Une annexe complète cet ouvrage. Elle présente une étude de cas réalisée avec **L*f*P**. Il s'agit d'un travail coopératif mené avec Yann Thierry-Mieg et Frédéric Gilliers.

CHAPITRE 2

De la conception au développement et à l'exécution d'applications réparties vérifiables

2.1 Introduction

Notre thèse vise le développement par modélisation d'*applications réparties à composants logiciels interopérables* dans le sens de la **Définition 2-1**. Notre définition de l'interopérabilité raffine celle de Wagner **[139]** dans le sens que hétérogénéité n'est plus traitée d'une manière unitaire mais elle est multiplexée par rapport à l'étape de développement où elle intervient (conception, implémentation, etc.).

Définition 2-1: L'*interopérabilité* des composants logiciels se traduit par leur capacité de coopérer en dépit de leur hétérogénéité de conception (e.g. modèle de programmation, interfaces logiques ou physiques) et d'implémentation (e.g. langage de programmation et environnements d'exécution cibles).

Le spectre d'applications que nous visons est très variable. Il inclut des applications :

- de taille fixe ou variable,
- déployées statiquement ou dynamiquement,
- intègrant des composants logiciels préexistants,
- bénéficiant d'une architecture ouverte, évolutive.

Ce chapitre a donc trois objectifs :

- *décrire le cadre général de notre travail :* les systèmes et les applications réparties, et proposer une terminologie homogène,
- *préciser l'objectif de nos travaux* : le développement rapide, évolutif et formel d'applications réparties à géométrie variable centrée sur la modélisation,
- *identifier les directions de recherche* : la modélisation formelle d'architectures logicielles réparties et l'influence des choix d'implémentation sur la modélisation.

La **Section 2.2** précise le cadre général de notre recherche qui cible les systèmes répartis. Nous examinons le spectre des systèmes visés par notre thèse, c'est-à-dire ceux pour lesquels l'environnement d'exécution (vu comme une couche d'adaptation entre l'application répartie et l'infrastructure matérielle) peut avoir une structure et une épaisseur variables nécessitant une grande souplesse d'adaptation.

La **Section 2.3** restreint l'horizon de notre recherche au domaine des applications réparties. Elle échafaude un style architectural et une terminologie associée adéquats à la description d'applications réparties.

La **Section 2.4** présente le développement d'applications réparties dans la perspective d'une démarche de conception par prototypage rapide évolutif **[70]**. Elle identifie des critères et des bases pour un langage de modélisation adéquat à la description d'architectures réparties, à la vérification formelle et à l'implémentation conforme des modèles.

2.2 Systèmes répartis

Dans la littérature, le terme *système réparti* est souvent utilisé pour exprimer des concepts différents. Plusieurs définitions émergent, un système réparti est identifié à :

- une collection d'ordinateurs indépendants qui, du point de vue d'un utilisateur, ont l'apparence d'être un seul ordinateur **[130]**.
- un ensemble d'ordinateurs autonomes interconnectés via un réseau de communication et équipés d'un système logiciel réparti leur permettant de coordonner leurs activités et de partager leurs ressources matérielles, logicielles ou données.
  L'utilisateur d'un tel système doit le percevoir comme une ressource de calcul unique, même si son implémentation est basée sur un ensemble d'ordinateurs physiquement répartis **[26]**.
- une collection d'équipements informatiques indépendants interconnectés **[118]**.

Les deux premières définitions font référence aux *systèmes d'exploitation répartis natifs* (comme CHORUS ou AMOEBA **[131]**, QNX **[100]**, etc.) ou aux *méta-systèmes d'exploitation* (comme Legion **[49]**) qui ont l'avantage d'assurer au niveau applicatif une transparence totale par rapport à la répartition physique des ressources matérielles, logicielles ou des données. Ces deux définitions nous apparaissent trop restrictives pour inclure les *systèmes d'exploitation réseaux* c'est à dire un système d'exploitation intégrant des services de communication réseau **[134]**. De tels systèmes sont par exemple ceux de la famille UNIX, qui malgré une transparence plus faible face à la répartition, sont également adaptés et utilisés pour la réalisation d'*applications réparties coopératives*. La troisième définition, basée sur une vision totalement différente, considère qu'un système reparti est uniquement constitué par l'infrastructure d'exécution matérielle et exclut le système d'exploitation.

Nous souhaitons raffiner cette vision afin de mieux situer le cadre de notre travail.

Définition 2-2: Un *système réparti* est constitué des éléments suivants (**Figure 2-1**) :
- l'**application répartie elle-même**, c'est-à-dire les programmes spécifiques écrits par une équipe de développement,
- l'**environnement d'exécution** composé par l'ensemble systèmes-intergiciels,
- l'**infrastructure matérielle** qui est le réseau d'ordinateurs autonomes.

Notre définition sépare mieux l'application de son environnement d'exécution et celui-ci de l'infrastructure d'exécution. Elle nous semble aussi plus souple vu qu'elle rassemble les systèmes d'exploitation et les intergiciels dans une couche logicielle intermédiaire (l'environnement d'exécution). Notre vision est moins contraignante vu que la «transparence totale» par rapport à la répartition n'est plus un objectif à assurer par l'environnement d'exécution.

La portabilité, l'évolutivité et l'interoperabilité des applications réparties modernes sont désormais des caractéristiques importantes pour les *applications réparties*. L'utilisation d'intergiciels est un moyen d'atteindre ces objectifs.

Définition 2-3: Un *intergiciel* est une couche logicielle d'adaptation qui définit un *protocole de communication* et propose un ensemble de services facilitant la construction d'applications réparties au dessus d'un réseau hétérogène d'ordinateurs équipés de systèmes d'exploitation réseau (NOS pour Network Operating System).

Si le rôle d'un système d'exploitation est d'assurer un certain niveau de transparence par rapport à l'infrastructure d'exécution (équipements matériels et réseaux de communication), celui d'un intergiciel est complémentaire : *il assure la transparence de l'application face à l'hétérogénéité des systèmes d'exploitation, des langages de programmation et des modèles de répartition et protocoles*. C'est pourquoi nous considérons comme évidente leur intégration dans la composition d'un système réparti.

Bien qu'attractive d'un point de vue Génie Logiciel, l'utilisation d'environnements d'exécution généralistes, d'une complexité importante, a deux inconvénients :

- *elle introduit un surcoût* (mémoire, vitesse d'exécution, etc.) qui peut être prohibitif dans le cas des systèmes,
- *elle complique la vérification et la validation du comportement* d'une application répartie en introduisant des mécanismes dont la sémantique précise peut varier selon les implémentations ; l'étude menée dans **[127]** illustre parfaitement ce problème sur les COS-services de CORBA.

C'est pourquoi dans le contexte de systèmes plus contraints (embarqués, temps-réel, etc.), on utilise des environements d'exécution dédiés (comme CORBA-RT **[88]**, Ravnscar **[30]**), ou plus «légers» qui n'offrent que les primitives nécessaires aux besoins considérés **[110, 135]**. Lorsque les contraintes sont plus fortes, l'environnement d'exécution est alors réduit au seul noyau supportant l'exécution de l'application et son implémentation est souvent intégrée directement dans l'application lors de l'édition de liens.

Notre objectif est de capturer une large gamme d'applications réparties : des *applications embarquées* aux applications basées sur des intergiciels sophistiqués. Cela semble possible si on paramètre la spécification de manière à y intégrer de manière fine les éléments de description des différents niveaux du système (applicatif, intergiciel, matériel).

La **Figure 2-1** synthétise notre vision des éléments d'un système réparti. L'«épaisseur» de la couche logicielle d'adaptation (système d'exploitation et intergiciel) étant variable, nous considérons cette cou-

che comme étant optionnelle. Dans certains cas, elle sera présente; en revanche, lorsque les contraintes le rendent nécessaire, elle sera intégrée à l'application sous la forme d'un exécutif léger.

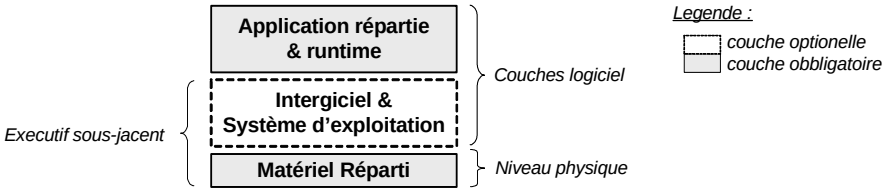


Figure 2-1 : Structure d'un système réparti.

Remarquons la similitude de l'architecture d'un système réparti avec celle d'un programme compilé. L'approche de conception d'une application répartie prônée dans le langage Ada-95 est ainsi pleinement justifiée (son implémentabilité a été démontrée). La notion d'exécutif se généralise au niveau de l'infrastructure d'exécution d'un système réparti pour offrir un support d'exécution réparti.

2.3 Conception componentielle des applications réparties

La structure d'une application répartie dépend du paradigme choisi pour guider son élaboration. Plusieurs paradigmes sont actuellement définis : les objets répartis, les composants logiciel répartis (e.g. les processus communiquants), les résources réparties partagées (objets ou segments de mémoire), etc. Le choix d'un paradigme force un style architectural.

Définition 2-4: Un *style architectural* introduit un vocabulaire (i.e. ontologie) pour la description d'architectures logicielles ainsi qu'un ensemble de règles et contraintes sur son utilisation **[38]**. Cela se formalise en définissant un méta modèle de l'architecture logicielle.

Nous nous sommes consacré à la réalisation d'applications réparties.

Définition 2-5: Une *application répartie* est un assemblage cohérent de *composants logiciels* communiquants déployés au dessus d'un exécutif réparti.

Une large variété d'environnements d'exécution répartis doit supporter l'exécution de ces applications. *Notre objectif est d'assurer une grande indépendance de la conception vis-à-vis des choix de réalisation* (e.g. choix de l'intergiciel, langages de programmation, systèmes d'exploitations répartis, etc.) *et vis-à-vis des styles architecturaux*.

Nous considérons que l'architecture logicielle d'une application répartie est constituée autour de trois notions : *composants logiciels*, *communications à distance*, *configurations* (assemblage). Cette décomposition est directement inspirée de la notion d'ADL (*Langage de Description d'Architectures ou Architectural Description Language)* **[80]** qui considère que les composants, les connecteurs les reliant et les configurations sont les briques de base constituant une architecture logicielle.

2.3.1 Composants logiciels

La notion de composant logiciel varie selon la littérature et même parfois, selon les publications d'un même groupe. Nous avons retenu plusieurs interprétations :

- Pfister et Szyperski **[98]** expliquent en quoi un composant logiciel diffère d'un objet. Un composant logiciel est considéré comme *un ensemble d'objets coopératifs fortement couplés* ayant une

interface bien définie. Bien qu'historiquement l'une des premières, cette définition est restrictive pour deux raisons : 1) elle est n'adaptée qu'au paradigme de programmation orientée objets ; 2) le terme «couplage fort» manque d'interprétation.

- Une révision de la définition précédente est proposée par Szyperski dans **[129]**. Un composant logiciel y est assimilé à *une unité binaire de déploiement et de composition sans état interne*. L'interprétation est ici plus générale car elle ne se rattache plus uniquement à l'approche orientée objet. Elle donne en outre une interprétation au terme «couplage fort» dans le sens de «à déployer ensemble». Cette définition introduit une nouvelle idée de «composition» sans expliquer comment celle-ci doit être réalisée.
  Cependant, ne considérer que les composants logiciels sans état nous semble restrictif. Bien qu'il existe des plates-formes dont les composants vérifient cette définition (e.g. DCOM **[57]**), elle ne peut être généralisée. Par exemple, les composants EJB **[28]** ou CORBA-CCM **[89]** supportent les composants logiciel avec ou sans état.
- Heineman **[53]** généralise la notion de composant logiciel. Un composant logiciel est toujours consideré comme *une unité indépendante de déploiement et de composition* mais l'auteur introduit la nécessité de définir et respecter un «*modèle de composants*» et un «*standard de composition*».
  Le *modèle de composants* spécifie la structure et la sémantique des interfaces fournies et requises par un composant tandis que le *standard de composition* spécifie comment les composants seront répartis, configurés, instanciés et connectés selon une topologie de déploiement donnée.
- Pour la définition de ces règles, les auteurs font appel au modèle de référence RM-ODP **[62]**, un framework défini par l'ITU-T pour la description d'applications réparties ouvertes.

La dernière interprétation nous semble plus précise et complète c'est pourquoi nous nous en inspirons pour adopter la définition suivante.

Définition 2-6: Un *composant logiciel* est une entité logicielle conforme à un *modèle de composants* pouvant être déployés et composés indépendamment, sans modification, à l'aide d'un *standard de composition*.

2.3.2 Modèle de répartition

Le «modèle de répartition» est une notion centrale dans les systèmes répartis. Elle permet de classer la manière dont l'infrastructure d'exécution offre les services nécessaires à la répartition (communication, gestion du contexte réparti, etc.). A l'heure actuelle, plusieurs modèles sont couramment admis :

- l'envoi de messages qui constitue une réponse efficace aux problèmes de calcul scientifique **[119]**,
- l'appel de sous-programmes distants **[123, 132]** ou sa variante plus moderne, l'appel de méthodes sur objets répartis **[87]** qui sont des solutions plus lourdes pour l'infrastructure d'exécution mais aussi plus structurantes du point de vue de la conception des applications réparties,
- la mémoire partagée répartie ou tout autre mécanisme de stockage partagé qui se répandent actuellement **[4]**.

A chacun de ces modèles de répartition correspond une philosophie d'exécution répartie. Cette philosophie doit être mise en œuvre au niveau de l'infrastructure d'exécution. Elle l'est actuellement au moyen de couches logicielles complémentaires du système d'exploitation : les intergiciels qui sont une «colle» entre les services réseaux d'un système d'exploitation classique et le code de l'application. Dans certains cas, l'intergiciel est composé à la fois de bibliothèques de fonctions de base pour la répartition et de code spécifique à l'application. Cette génération peut être explicite, réalisée à partir d'un IDL comme dans CORBA, ou s'appuyer sur des directives de compilations et de configuration comme dans Ada95/DSA.

Mais une application répartie peut ne pas utiliser exclusivement un seul modèle de répartition. L'interaction entre les composants devient alors complexe à maîtriser et un travail au niveau d'un modèle peut s'avérer structurant. A ce titre, la notion de PIM (Platform Independant Model) introduites dans l'approche MDA apporte un point de vue intéressant. Les interactions entre composants sont gérées à partir de leur sémantique opérationnelle au niveau du PIM et sont ensuite implémentées de manière conforme. Cela nécessite une réflection sur les interactions entre composants d'une application répartie d'une part, et les services de communication offerts par les systèmes d'exploitation d'autre part.

2.3.3 Interactions entre composants d'une application répartie

Elaborer une application répartie à l'aide de composants nécessite de la «colle» (on parle fréquemment de «glueware»), c'est à dire de passerelles assurant les liaisons sur une topologie de déploiement. Nous parlons de connecteurs ADL.

Définition 2-7: Un *connecteur* est une entité logicielle de liaison pour les composants. Il précise un contrat d'interaction exprimant un protocole ainsi que des contraintes de liaison. Au contraire des composants, les connecteurs sont souvent implémentés dans l'environnement d'exécution.

Certains environnements d'exécution définissent des services d'interaction à distance. La façon de réaliser cette interaction diffère d'une solution à l'autre mais plusieurs paradigmes d'interaction émergent :

- *Communication par flots de caractères.* Ce modèle est de bas niveau. L'unité binaire d'échange est composée d'un seul caractère (octet). La communication par flots de caractères peut être assimilée à un cas particulier de communication par messages si l'on considère que chaque caractère est un message de taille fixe sans entête.
- *Communication par messages.* Ce modèle de communication fait appel aux communications par échange «direct» (e.g. point-à-point) ou par «intermédiaires» (e.g. lecteurs-écrivains, producteurs-consomateurs) de paquets d'information à travers un réseau. Le rôle de l'exécutif est d'implémenter un protocole de communication permettant de fixer à la fois la structure et la représentation binaire des messages, mais également la sémantique des communications, c'est à dire l'automate implémentant un protocole d'échange.
  La structure des messages peut être hiérarchique, par couches, l'exécutif ayant le droit d'utiliser et modifier certaines de ces informations (i.e. pour router). C'est pourquoi les messages sont souvent structurés en deux parties : plusieurs entêtes (connues par l'exécutif) et le corps du message (opaque pour son regard).
- *Communication par invocations à distance.* Basé sur le modèle de communication par messages, ce type d'interaction offre un meilleur niveau abstraction des communications. Il permet l'invocation transparente de services à distance entre deux ou plusieurs composants répartis. Selon le paradigme de programmation utilisé (procédural ou orienté objet) ce modèle de communication est connu sous divers noms : *Appels de Procédures à distance, Invocation de Méthodes distantes* etc. Son principe consiste dans l'échange réciproque de messages représentant des requêtes, des réponses associées ou éventuellement des messages d'exception et de synchronisation.
- *Communication par ressources réparties partagées.* Elles sont réalisées à l'aide des objets (i.e. variables ou instances de classes) **[11]** ou segments mémoires partagés **[4]**. La gestion des ressources partagées revient à l'exécutif qui fournit un ensemble de primitives d'accès protégé (synchronisé) sur les données. Bien que les communications soient toujours basées sur l'échange de messages, elles sont cachées par l'environnement d'exécution qui rend transparent l'accès aux ressources partagées.

D'autres paradigmes d'interaction entre les composants sont également possibles. Mehta **[82]** réalise une taxonomie détaillée des connecteurs ADL. Un connecteur est une entité logicielle dédiée pour la communication.

2.3.4 Noyau canonique de modèles d'interaction

L'analyse précédente constitue notre point de départ pour la synthèse d'un «*noyau canonique de modèles d'interaction*». Nous souhaitons intégrer cette idée (de noyau canonique de modèles de communication) et celle de «*composition de connecteurs*» **[121]** comme moyen de synthèse pour les protocoles de communication de plus haut niveau.

Le *paradigme d'interaction* choisi a une influence directe sur la structure, les performances et la programmation d'une application. C'est pourquoi nous souhaitons concevoir des applications de manière la plus transparente possible par rapport au modèle de communication employé pour son déploiement. Ce problème est analysé en détail dans le **Chapitre 4**: *Abstraction de l'environnement d'exécution pour la modélisation* qui étudie l'impact des choix d'implémentation sur la conception d'applications réparties.

Assemblage des composants

La notion de composant logiciel permet de moduler la conception d'une application répartie en vue de son déploiement. Ce dernier est vu comme une étape ultérieure à la conception qui revient à une procédure de configuration et d'assemblage de composants. Comme précisé dans la **Définition 2-6**, l'assemblage de composants doit respecter un standard de composition, ce qui oblige à considérer un style architectural de déploiement.

Nous nous inspirons à la fois du modèle de référence RM-ODP **[62]**, des concepts introduits dans les ADL **[38]** ainsi que des modèles architecturaux proposés dans les plates-formes à composants logiciels EJB **[28]** et CORBA-CCM **[89]**. L'objectif est de reprendre les concepts communs à ces approches comme la base d'un consensus permettant de garantir la faisabilité de réalisation d'une application répartie. Ces concepts communs sont : le *nœud*, la *capsule*, le *port*, le *binder* et le *média* dont nous donnons la définition ci-après.

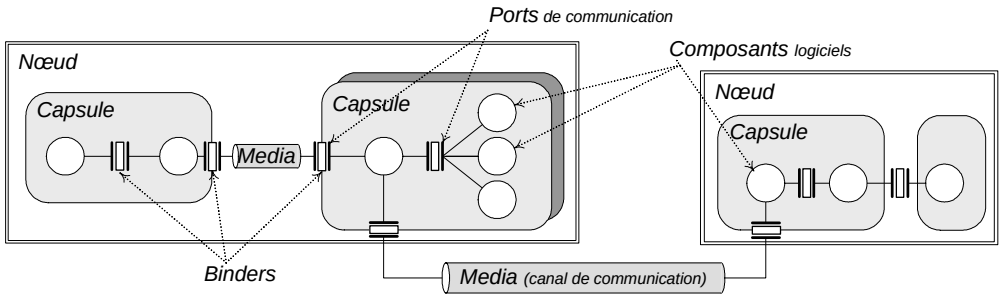


Figure 2-2 : Architecture logicielle de déploiement pour les applications réparties à composants logiciels.

La **Figure 2-2** illustre notre vision de l'architecture de déploiement d'une application répartie. Pour en formalise la composition nous introduisons plusieurs notions dont nous proposons les définitions ci-après :

Définition 2-8: Un *nœud* est une ressource de traitement physiquement autonome (e.g. ordinateur) permettant l'exécution des capsules. Il est sous le contrôle d'un noyau d'exécution (le système d'exploitation ou l'exécutif réduit qui remplit ces fonctions) offrant les services élémentaires pour la gestion des ressources et celle des communications.

Définition 2-9: Une *capsule* est une unité logicielle de protection pour l'exécution **[62]**. Elle définit son propre espace d'adressage et possède des capacités de stockage et une partie des ressources de traitement du noeud. Chaque capsule dispose d'un gestionnaire contrôlant les composants hébergées et les connexions avec l'extérieur.

Définition 2-10: Un *port* est un point de communication physique permettant à un composant de coordonner son activité avec d'autres entités logicielles.

Définition 2-11: Un *binder* est un connecteur dédié pour les interactions directes, en local. Il précise un contrat d'interaction simple (e.g. rendez-vous synchrone, tampon de messages ou des caractères, etc.) permettant de typer la connexion de ports.

Définition 2-12: Un *média* est connecteur complexe encapsulant un protocole de communication (e.g. un *canal de communication*). Comparé à un binder, son rôle est d'assurer la coopération entre les composants embarqués dans différentes capsules et ne partageant donc pas le même espace d'adressage.

Nous constatons que :

- La notion de capsule est équivalente à celle de *conteneur d'exécution* CORBA-CCM ou *maison* DCOM **[57]**. Celle-ci ne constitue rien d'autre qu'un composant logiciel spécialisé qui encapsule les fonctionnalités offertes par l'exécutif sous-jacent (e.g. gestion du cycle de vie, référentiel de noms, service de liaison etc).
- La notion de Port exprime un port ADL. Celle-ci correspond également à un point de liaison (interface) dans RM-ODP.
- Contrairement à d'autres approches qui utilisent les connecteurs ADL pour modéliser à la fois les interactions locales (souvent normalisées) et les communications à distance, nous préférons séparer ces notions par des entités logicielles spécialisées : les binders et les media. La notion de binder est équivalente à celle de d'objet de liaison simple RM-ODP et celle de Média à un canal de communication.

L'architecture logicielle d'une application répartie revient ainsi à un assemblage cohérent de composants logiciels au moyen de connecteurs (binders ou media) selon une *configuration architecturale* (topologie) donnée. Cette configuration architecturale doit être cohérente par rapport aux règles de composition du modèle de répartition (ou de son métamodèle).

Définition 2-13: Une *configuration architecturale* décrit la structure d'une application sous la forme d'un graphe de composants logiciels connectés. Le rôle d'une configuration est d'imposer des contraintes de composition (e.g. correspondance des signatures d'interfaces de connexion, protocoles de communication à utiliser, etc.) compatibles avec un style architectural de déploiement. Elle permet de déterminer la sémantique globale d'un système à partir des entités le constituant (composants logiciel et média de communication reliés par des binders au niveau de leurs ports).

2.4 Développement d'applications réparties conformes à leurs spécifications

L'implémentation et le déploiement d'une spécification à base de composants sur une architecture répartie est une tâche difficile car l'utilisation de techniques et méthodologies de développement «classiques» (i.e. celles des applications non réparties) ne sont pas adaptées. Par exemple, les modèles de communication utilisés dans ces spécifications n'ont pas de correspondance directe avec les mécanismes offerts par les environnements d'exécution. Il faut donc développer du code spécifique permettant d'assurer cette correspondance.

Le fil conducteur de notre démarche est de structurer l'analyse des problèmes de développement d'applications réparties par rapport au cycle de vie du logiciel. Nous devons donc considérer le développement comme un ensemble d'étapes : spécification puis modélisation, vérification, implémentation, test et déploiement. *Nos travaux se focalisent donc sur la modélisation, la vérification et l'implémentation ainsi que la relation entre ces trois étapes:*

- **Conception versus implémentation.** Pour assurer un *développement évolutif* permettant la construction (la spécification et la modélisation) d'applications portables et interopérables nous considérons que :
  - la conception d'applications doit s'abstraire des choix d'implémentation **[9]**;
  - la conception d'applications doit se baser sur l'emploi d'abstractions. Les gabarits de conception **[39]**, les gabarits d'interaction **[82]** et la séparation entre l'interface et l'implémentation des entités logicielles **[120]** sont des exemples d'abstractions (d'autres existent dans la littérature).

  Cependant, nous remarquons que ces hypothèses sont rarement respectées dans la pratique.
- **Modélisation**. La construction de modèles exécutables, ne serait-ce que pour une analyse par simulation, permet de détecter des erreurs ou inconsistances de conception dans le système très tôt dans le cycle de vie du logiciel **[120]**. Cependant, la pertinence des observations est directement liée au langage de modélisation choisi. Des critères comme la puissance d'expression ou la précision de la sémantique opérationnelle doivent être considérés dans le choix d'un tel langage **[3, 90]**.
- **Conformité du développement.** La modélisation et l'analyse d'une spécification exécutable n'a pas de sens si elle n'est pas accompagnée de techniques visant à assurer l'adéquation du code par rapport au modèle **[140]**. L'utilisation de générateurs de programmes dans des Ateliers de Génie Logiciels permet d'assurer une forte correspondance **[133, 35]**.

2.4.5 Conception versus implémentation

Nous cherchons à concevoir des applications réparties à composants logiciels coopératifs en nous focalisant sur leur portabilité et leur évolutivité. Pour satisfaire ces objectifs, plusieurs techniques de développement sont généralement considérées :

- l'utilisation d'*intergiciels*,
- la réalisation de *passerelles statiques* entre intergiciels,
- l'utilisation d'*intergiciels génériques* ou *paramétrables*,
- l'utilisation d'*intergiciels réflexifs*,
- l'utilisation d'*intergiciels* dynamiquement adaptables.

Les intergiciels constituent notre première piste. L'idée est de rendre le développement «neutre» par rapport à un langage de programmation, un environnement d'exécution et un protocole de communication. Cela revient à proposer un modèle de répartition qui normalise un schéma de conception. Par

exemple, CORBA **[87]** intègre ces trois aspects au moyen d'un langage de description d'interface (IDL: Interface Definition Language) et d'outils de génération automatique de talons («stub» et «skeleton») utilisant un protocole générique (GIOP) sur différents couples <*langage*, *système d'exploitation*>. Cette solution introduit cependant un nouveau type d'hétérogénéité au niveau de l'intergiciel lui-même. C'est pourquoi L.Pautet **[95]** parle du «paradoxe de l'intergiciel»: Bien que les intergiciels soient un moyen pour cacher l'hétérogénéité des choix d'implementation, ils sont souvent incompatibles entre eux et constituent donc une nouvelle source d'hétérogénéité de plus haut niveau.

La deuxième approche est la construction de passerelles d'adaptation statiques entre intergiciels comme dans CIAO **[103]** ou RMI-HPC++**[18]**. Ces passerelles assurent une communication transparente entre composants hétérogènes. Cette approche est efficace du point de vue des performances mais trop rigide du point de vue de l'évolutivité et elle interdit tout passage à l'échelle (pour chaque couple d'intergiciels dans le système, il faut concevoir une passerelle dédiée).

Les intergiciels paramétrables comme Quarterware **[117]**, Jonathan **[85]** ou ACT **[37]** constituent une étape supplémentaire vers la construction d'intergiciels réellement intéropérables. Ces travaux mettent en évidence la nécessité de définir un modèle réduit d'*intergiciel neutre, générique* permettant la construction rapide d'intergiciels personnalisées par un processus de paramétrage.

Des travaux parallèles, basés sur l'emploi des mécanismes de réflexion montrent que l'interopérabilité entre les intergiciels peut être obtenue si on la traite au niveau «Meta» **[14]**, du modèle d'intergiciel. *Cette observation est essentielle pour notre travail car elle ouvre la voie vers une démarche à base de méta-modélisation et non plus centrée sur une approche technique.*

Des travaux plus récents combinent les deux précédentes démarches : une conception centrée sur un modèle personnalisable et des techniques de réflexion. Cela permet de construire des intergiciels dynamiquement adaptables comme PolyORB **[102]**, UBICORE **[109]** ou DynamicTao **[69]**. Les intergiciels dits «schizophrènes»**[95]** changent de «personnalité» dynamiquement pour s'adapter aux variations d'interconnexions demandées par les applications.

Comme dans le cas de ces intergiciels dynamiques, nous souhaitons mettre l'accent sur l'indépendance de la conception par rapport aux choix de déploiement. Cependant, notre approche consiste à intégrer l'intéropérabilité au niveau d'un modèle plutôt que de s'appuyer sur des environnements de programmation. Ainsi, nous souhaitons définir un *modèle d'intergiciel pivot* offrant des mécanismes de base proposés à travers un langage de modélisation dédié aux systèmes répartis. Le déploiement d'un modèle s'effectuera sur un intergiciel concret implémentant les fonctions qui sont référencées en s'appuyant sur des techniques de personnalisation. Cela revient à définir une forme canonique d'environnement d'exécution réparti qui sera utilisée dans la conception de notre langage de modélisation orienté prototypage (**L*f*P**, Language for Prototyping). L'analyse des environnements d'exécution (principalement celle des intergiciels) et la synthèse d'un modèle neutre d'*environnement d'exécution canonique* sont abordées dans le **Chapitre 4**: *Abstraction de l'environnement d'exécution pour la modélisation*.

Le langage **L*f*P** est notre contribution majeure à la conception d'applications réparties. Son objectif est permettre la modélisation d'applications réparties selon une démarche de conception par prototypage incrémental. Nous souhaitons également avec **L*f*P** ouvrir des passerelles vers la vérification formelle d'une part (couplage avec les réseaux de Petri étudiés dans le cadre de la thèse de Yann Thierry Mieg) et de la génération automatique de programmées répartis (étudié dans le cadre de la thèse de Frédéric Gilliers). Même si nous n'avons pas mis en œuvre **L*f*P** comme un profil UML (ce qui sera fait dans le cadre du projet RNTL MORSE), nous avons souhaité le rendre compatible avec la démarche de modélisation d'UML.

2.4.6 Modélisation

La modélisation d'applications réparties à base de composants logiciels est une tâche délicate car la qualité de la conception et l'éventail des architecture potentielles d'exécution dépendent de la notation utilisée. UML ne peut être écarté vu l'étendue de son utilisation dans l'industrie. Cependant, s'il apporte un avantage indéniable en étant un standard reconnu, les problèmes que l'on rencontre en l'utilisant pour des systèmes répartis sont désormais identifiés **[68]** :

- UML n'est qu'imparfaitement adapté à la description d'architectures logicielles **[65, 80]**;
- UML n'est pas assez formel pour bien supporter un processus de vérification formel sur l'ensemble des diagrammes **[21]**;
- il est difficile de rassembler l'information éparpillée sur plusieurs diagrammes même s'ils sont du même type **[67, 97]**.

Les langages de description d'architectures logicielles **[38, 80]** constituent notre deuxième piste pour parachever celle des intergiciels. Plusieurs propositions existent, la difficulté est de trouver un langage flexible adapté pour supporter un choix varié de styles architecturaux tout en restant compatible avec UML.

La description d'architectures logicielles n'a pas seulement besoin de notations adaptées. Elle doit également respecter des gabarits de conception (les modèles de référence d'ODP **[62]**) définissant les règles architecturales à respecter. Le modèle de référence RM-ODP est actuellement un standard pour la construction d'applications réparties ouvertes. C'est pourquoi, notre démarche utilise les points de vue *Traitement*, *Ingénierie* et *Technologie*.

Notre approche de modélisation formelle ouvre la voie à la vérification formelle de modèles par model-checking, à la simulation et, à la génération de programmes conformes aux spécifications (i.e. certification). Le probléme de la modélisation formelle est un sujet d'intenses recherches. La formalisation ponctuelle **[51, 34, 94, 22]** ou dans son ensemble **[21, 97, 36]** du langage UML, la proposition des ADL formels aptes à la simulation (e.g. RAPIDE **[66]**) ou à la vérification formelle (e.g. Wright **[3]**) ainsi que d'autres langages de modélisation formels (e.g. AltaRica **[99]** et CO-OPN2 **[13]**) sont des démarches positives.

Cependant la modélisation formelle soulève plusieurs questions impliquant des choix concernant le formalisme à utiliser :

- Quel formalisme est le mieux adapté à la description d'applications à composants répartis ?
- Existe-t-il un besoin réel pour la conception complète d'un nouveau formalisme ou l'emploi des approches existantes suffit-il ?
- Dans quelle mesure la puissance d'expression limite-t-elle la vérification des propriétés sur les modèles et quelle est la maturité des techniques de vérification et de simulation propres à un langage ?
- Quelles sont les difficultés d'utilisation et comment les simplifier ?

Les aspects concernant la modélisation d'applications réparties sont examinés dans **Chapitre 3** : *Modélisation d'Applications Réparties*. Celui-ci présente une analyse détaillée des langages de modélisation sous l'angle de leur adéquation pour la description d'applications réparties et leur définition formelle.

2.4.7 Prototypage pour la conformité du développement

La conformité du développement est un problème à caractère technique. Notre méthode de développement doit assurer la correspondance fonctionnelle et structurelle entre les modèles décrivant une appli-

cation d'une part, les programmes qui l'implémentent d'autre part. Nous souhaitons garantir à la fois la rapidité du développement et les performances des programmes produits.

Le prototypage rapide, par raffinements, est une méthodologie de développement adaptée pour la synthèse et la validation de spécifications **[120]**. Cela permet, entre autres avantages, d'assurer une cohérence étendue entre la spécification et l'implémentation d'un produit logiciel en même temps qu'un développement rapide **[47]**. Les méthodologies de prototypage dites «par raffinements» (evolutionary prototyping) **[70]**, visent à la production d'un produit par raffinements successifs de prototypes générés automatiquement. Ce type de prototypage s'oppose à la notion de «maquettage» dans lequel le protype n'est pas conservé: seules les informations qu'il fournit sont exploitées.

Les techniques de prototypage par raffinement sont principalement appliquées dans la phase de développement et s'appuient sur des langages de programmation classiques. Le passage à de gros projets entraine alors des problèmes liés aux difficultés de gestion. Par exemple, les modifications constantes du prototype interdisent souvent une bonne structuration des projets. De plus, toute la robustesse du système repose sur des choix d'architecture initiaux qui peuvent ne plus être adaptés après de nombreuses itérations. Pour rendre ce type d'approche efficace, il faut les associer à d'autres techniques de développement permettant d'assurer le passage à l'échelle avec une qualité de développement garantie.

Des AGL (Ateliers de Génie Logiciel) comme AUTOFOCUS **[58]** ou OBJECT-GEODE **[8]** ont démontré qu'il est avantageux d'utiliser une méthodologie de développement basée sur un modèle formel. Le modèle diffère du langage de programmation en ce sens qu'il est une abstraction de haut niveau du système à réaliser. Ces modèles pourront soit être exécutés par simulation, soit servir de base à la génération automatique de programmes.

La génération automatique de programmes est une technique employée depuis longtemps par les outils de développement rapide (ou RAD pour Rapid Application Developpment). Le modèle qui sert de base à la génération automatique de programmes (ou autres modèles) doit être formellement défini. Cette technique bénéficie de plusieurs avantages :

- La traduction des modèles vers un environnement d'exécution cible <*langage de programmation*, *environnement d'exécution*> est automatisée.
- Le moteur de transformations (générateur de programmes) est basé sur un ensemble de règles formelles qui spécifient un morphisme depuis la notation utilisée pour la modélisation vers le langage ciblé. Il est donc possible de tracer l'information entre modèles et programmes.
- Un générateur de programmes qui préserve la sémantique opérationnelle du modèle dans les programmes produits est conforme **[140]**. La conformité va plus loin que la traçabilité entre programmes. Cela impose cependant un processus de certification du générateur de programmes.

Le modèle MDA (*Model Driven Architecture* **[9]**) prôné par l'OMG soutient qu'il est avantageux d'associer une méthodologie de développement par modélisation. Nous pensons que cela doit se combiner à une démarche de prototypage par raffinements. Ainsi, la conception se structure à un niveau plus abstrait. La réutilisation et l'échange de modèles deviennent possibles **[92]**.

Cependant la génération automatique de programmes et la simulation de modèles nécessitent des langages de modélisation formels. Dans le cas contraire, il est impossible de définir de manière unifiée une correspondance entre le modèle et le code généré. C'est typiquement le cas des outils UML qui proposent chacun une interprétation des relations entre les différents diagrammes utilisés pour la génération de code. Cela nuit à la portabilité des modèles entre AGL.

C'est pourquoi nous proposons une démarche de «prototypage par raffinements» basée sur un modèle **[70]**. Au terme de MDA, nous préférons celui de *Model Based Development* (MBD) **[44]**. La génération automatique de programmes répartis assure la conformité entre le programme et le modèle.

2.4.8 Conclusion : critères et bases pour un langage de modélisation

Nous réalisé une première analyse sur la problématique de modélisation d'applications réparties dans la perspective des relations existant entre cette étape de conception d'un côté et les étapes de vérification formelle et d'implémentation de l'autre. Cette analyse nous permet d'identifier une liste de critères de sélection à utiliser pour le choix d'un langage de modélisation adapté à notre problématique, c'est à dire le développement rapide, évolutif d'applications réparties conformes à leurs modèles.

Les critères que nous préconisons pour le choix d'un langage de modélisation sont :

- l'adéquation pour la description d'architectures logicielles réparties,
- le niveau de formalisation dans la définition.

D'autres critères moins importants par rapport aux autres objectifs présentés sont également considérés:

- la compatibilité du langage avec les normes existantes,
- le niveau d'abstraction du langage,
- le passage à l'échelle des modèles,
- la réutilisation de composants.

Adéquation pour la description d'architectures logicielles réparties

Nous cherchons des langages de modélisation adaptés pour la description d'architectures logicielles réparties. Compte tenu de la multitude de propositions existantes **[80]** le problème est de trouver un ADL capable d'exprimer les architectures logicielles les plus usuelles. Pour rendre la conception d'une application indépendante de son implémentation et de son déploiement nous proposons un langage de description d'applications réparties **L*f*P** permettant leur prototypage conformément à leur spécification. Nous nous appuyons sur les propositions de RM-ODP pour définir les constructions de ce langage afin de rester dans le contexte d'applications réparties ouvertes.

Niveau de formalisation dans la définition

Nous ciblons les applications qui, de par leur conception et leur implémentation, doivent respecter des propriétés de fonctionnement. Pour garantir ces propriétés, nous souhaitons analyser le comportement du modèle de l'application à l'aide de méthodes formelles. La génération de programmes assure la conformité de l'implémentation du modèle. Par exemple, des approches formelles comme celle du langage Z **[122]** ou les réseaux de Petri **[45]** sont adaptées à la description d'applications certifiables : ils permettent la vérification et la génération de programmes. Ainsi, l'emploi d'une démarche de modélisation formelle ou, à défaut, la compatibilité du langage de conception avec une méthode formelle, nous paraît une condition *sine qua non* pour qu'un langage de modélisation soit pris en compte.

Compatibilité du langage avec les normes existantes

La compatibilité de la démarche de modélisation avec la notation UML ainsi que la compatibilité du style architectural supporté par le langage de modélisation choisi avec le modèle RM-ODP sont des objectifs que nous souhaitons satisfaire.

Niveau d'abstraction du langage

Les langages abstraits, de haut niveau, permettent une modélisation rapide. Les utilisateurs de tels langages peuvent se concentrer sur la description des aspects fonctionnels d'une application tout en faisant abstraction des détails d'implémentation qui seront à spécifier ultérieurement. Cependant, la puissance d'expression d'une notation ne doit pas nuire à sa lisibilité **[75]**. Nous pensons qu'une notation semi-graphique concise est plus lisible à un ingénieur que des formules mathématiques. C'est pourquoi nous

nous inspirons des notations comme les ADL et les Automates à Etats Finis (FSA-*Finite State Automata*).

Passage à l'échelle

Le passage à l'échelle des applications peut être assuré par plusieurs moyens. Une décomposition modulaire hiérarchique permet d'assurer de bien structurer les modèles et de mieux appréhender des systèmes de grande taille. Une autre forme de modularité est la prise en compte d'*aspects de conception* d'un système. RM-ODP les voit comme différentes vues qu'il faut étudier.

La complexité d'une application répartie rend difficile la conception. Il faut donc offrir des facilités de modélisation par composants («diviser pour régner») et promouvoir par conséquent la réutilisation de composants et modèles déjà étudiés et considérés comme fiables.

Réutilisation

La séparation conceptuelle entre l'interface et l'implémentation et le couplage faible dans la conception des composants comme énoncé par la le principe de Dementer **[84]** sont deux moyens pour assurer la réutilisation de modèles. Selon ce principe une classe doit dépendre dans sa conception au plus de celle de ses voisines directs. L'objectif de cette loi de conception est de limiter (à un seul niveau) la dépendance fonctionnelle entre les classes et par conséquence de stimuler à leur réutilisation modulaire. Notre proposition se force de respecter ces deux principes.

Evolutivité

UML est un langage à la fois extensible au moyen de profils dédiés et évolutif par modification de son méta-modèle (décrit en MOF **[91]**). Notre objectif est moins ambitieux car il se concentre principalement sur l'évolutivité du point de vue de l'application. L'évolutivité du langage, pour intégrer de nouveaux styles architecturaux, se réduit à la recherche d'un style architectural neutre, canonique qui peut être étendu par un processus de paramétrage et composition. Le travail difficile est donc la conception de ce modèle de style architectural.

Démarche adoptée

L'état de l'art que nous avons présenté nous a permis de dégager non seulement les critères précédents mais aussi une démarche de développement par raffinements basée sur la génération automatique des programmes. Les chapitres suivants proposent des solutions pour satisfaire les obligations inférentes à cette démarche :

1) pour traiter la géométrie variable des exécutifs supports, notre langage doit abstraire l'environnement d'exécution
2) pour permettre aux concepteurs/réalisateurs de réutiliser des foncionalités parmi un ensemble, la construction du système se fait à partir d'un «catalogue de concepts» utiles et disponibles (comme ce qui se fait sur les «frameworks»).
3) pour étayer les outils garantissant la conformité nous basons notre approche sur la modélisation- pour permettre la simulation, la vérification formelle et la génération de code automatique.

L'avantage de notre démarche de développement est de faciliter, dans un contexte de développement par prototypage, la liaison avec les méthodes formelles d'analyse des systèmes répartis. De plus, la génération automatique de programmes répartis permet de garantir la conformité de l'implémentation par rapport à la spécification. Ces points ne sont actuellement pas couverts par les approches à base d'UML dans le domaine des systèmes répartis.

CHAPITRE 3

Modélisation d'Applications Réparties

3.1 Introduction

La rigueur de la démarche de modélisation, la validation des modèles et leur implémentation conforme au dessus d'un environnement d'exécution ciblé sont trois objectifs que nous souhaitons traiter. Nous sommes donc à la recherche d'un langage de modélisation adapté pour la description d'architectures logicielles réparties et d'une méthodologie de développement pour atteindre ces objectifs.

Puisque l'architecture logicielle d'une application est contrainte à la fois par le choix du langage de modélisation et par les contraintes architecturales qu'un exécutif ciblé impose, notre travail d'analyse s'étend sur deux directions complémentaires : l'aspect applicatif est traité dans ce chapitre tandis que les exécutifs répartis le seront dans le Chapitre 4 : *Abstraction de l'environnement d'exécution pour la modélisation*.

Nous avons comparé les principaux langages de modélisation existants afin d'identifier les caractéristiques d'un ADL (Architecture Description Language) unifié, supportant un style architectural neutre. Le langage de modélisation UML **[90]**, l'architecture logicielle proposé par le modèles de référence RM-

ODP **[62]** et des ADL tel que WRIGHT **[3]**, RAPIDE **[66]**, C2 **[79]** Meta-H **[136]**, ROOM **[111]** sont les travaux qui nous ont inspiré le plus. L'analyse de ces travaux est présenté dans la **Section 3.2** : *Description des applications réparties*.

La **Section 3.4** intitulée *Relation avec les approches formelles* est consacrée à l'étude des aspects formels dans la modélisation en vue de la vérification formelle et de la simulation de modèles. Notre étude est centrée sur l'utilisation des Réseaux de Petri (R*d*P) comme notation apte pour la formalisation et la vérification des notations de plus haut niveau. Ce choix a essentiellement été dicté par l'environnement de modélisation et de vérification disponible pour notre thèse.

3.2 Description des applications réparties

La **Section 2.4.8** a présenté nos critères pour le choix d'un langage de modélisation. L'adéquation d'une notation à la description d'architectures logicielles réparties constitue notre premier objectif, c'est pourquoi nous consacrons cette section à l'état de l'art des langages de modélisation et de description d'architectures logicielles.

Chaque langage de modélisation supporte un ensemble de styles architecturaux. Le choix du langage utilisé pour la modélisation restreint la gamme d'applications à un domaine précis. C'est pourquoi après avoir proposé des critères d'analyse (voir la **Section 3.2.1**), nous étudions et classons ces langages de modélisation afin d'identifier les caractéristiques propres à la description d'applications réparties et, le cas échéant, en retenir un.

3.2.1 Critères d'analyse

Pour raffiner le terme «description d'architectures logicielles réparties» nous nous inspirons de l'approche de Medevidovich **[80]** qui propose une analyse et une classification détaillée des principaux ADL. L'auteur compare leur capacité de modélisation selon trois axes :

- *modélisation de composants* logiciels,
- *description de connecteurs*,
- *facilités de configuration* et *assemblage d'architectures* (topologies).

Cette proposition est compatible avec le style architectural que nous avons proposé en **Section 2.3.4**. Cependant, en particulier pour étudier les problèmes de placement et de migration de composants et de données, nous souhaitons distinguer les connecteurs distants (pour les communications entre nœuds) des connecteurs locaux (les binders).

L'auteur raffine son étude sur chacun des trois axes d'analyse et propose une liste de critères hétérogènes mélangeant à la fois des aspects structurels, sémantiques ou de souplesse de modélisation. Bien que cette liste soit rigoureuse, sa construction nous paraît peu systématique. C'est ainsi que nous cherchons à élaborer une approche d'analyse systématique.

Pour justifier une liste de critères, nous proposons une vision «gros grain» sur la relation existant entre les concepts ADL. Ce méta-modèle est illustré par la **Figure 3-1** :

- les entités d'un ADL sont soit des composants soit des connecteurs;
- chaque entité peut être composée (hiérarchiquement ou par chaînage) d'autres entités;
- toute entité logicielle peut disposer des Ports de communication;

- la connexion d'entités se fait exclusivement par l'association des ports compatibles.

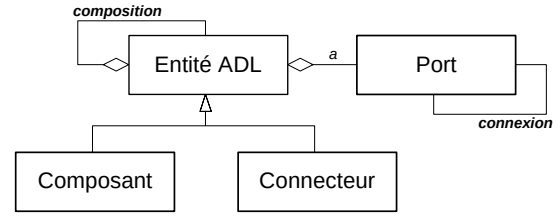


Figure 3-1 : Relations entre les concepts proposés par les ADL.

Notre démarche d'analyse est systématique; elle revient à analyser les capacités de modélisation des ADL étudiées par rapport à ce méta-modèle. Cela revient à identifier les contraintes structurelles et comportementales de modélisation de chaque ADL pour la description de composants, de connecteurs et de ports ainsi que les relations de composition entre ceux-ci. Ces derniers correspondent à des parcours dans le méta-modèle.

Considérons deux exemples afin d'illustrer notre propos :

- La relation «Composant-a-Port» de la **Figure 3-1** dépend de l'ADL. Certains d'entre eux ne permettent de définir qu'un port par composant (e.g. les StateCharts UML) alors que d'autres en autorisent plusieurs (SDL).
- L'association «connexion» de la **Figure 3-1** peut intégrer des contraintes. Par exemple, la connexion de deux ports n'est possible que si 1) leurs signatures sont compatibles ou 2) l'un appartient à un composant tandis que l'autre appartient à un connecteur.

Le **Tableau 3-1** presente nos critères d'analyse. Ceux-ci sont groupés en deux sections dont la première cible les concepts de base d'un style architectural (composants, connecteurs, ports) et la seconde se focalise sur leurs relations (des parcours dans le méta-modèle).

		Aspect de modélisation :	***Structure***	***Comportement***
Style architectural	***Concepts de base***	***Composants***	- structure interne des composants.	- comportement des composants.
		Connecteurs	- structure interne des connecteurs.	- protocole de communication - contraintes de comportement (e.g. la qualité de service).
		***Ports** d'un Composant ou d'un Connecteur*	- définition de services pour un port (polymorphisme). - contraintes sur le nombre et sur le regroupement de services dans un port.	- contrats d'interaction : types d'interactions supportées pour les services. - direction d'interactions pour les services.
	Relations entre les concepts de base	***Entité ADL – Ports*** ***(Composant ou Connecteur)***	- contraintes sur le nombre et le types de ports q'un entité supporte.	- rattachement dynamique de ports à l'exécution.
		Ports – Ports	- contraintes structurelles de connexion (binding) entre les ports.	- sémantique de connexion. - cycle de vie d'une connexion.
		Entité ADL – Entité ADL ***- composition de Composants, Connecteurs et de l'application (Composants et Connecteurs)***	- restrictions sur la composition hiérarchique et l'enchaînement de composants et des connecteurs. - restrictions sur la topologie d'une application.	- règles de coordination d'exécution entre les entités logicielles. - cycle de vie des entités logicielles.

Tableau 3-1 : Critères pour l'analyse des techniques de description d'architectures logicielles réparties.

Nous définissons chacun des éléments descriptifs correspondants aux lignes de ce tableau :

Composants : ils ont dédiés à la description des éléments fonctionnels d'une application.

- *Aspects structurels* : les ADL peuvent voir les composants soit comme une «boîte noire» (ils ne s'intéressent pas à leur structure), soit comme une «boîte blanche» (ils permettent d'en décrire la

structure).

- *Aspects comportementaux* : il faut identifier le type de comportement du composant (passif, actif, etc.) et décrire le flot d'exécution au moyen d'un automate.

Connecteurs : ils sont dédiés à l'expression des liaisons de communication entre composants.

- *Aspects structurels* : les services d'un connecteur sont en général implicites : aucune méthode ne permet de les paramétrer. Les connecteurs véhiculent des informations sans leur appliquer aucun traitement.
- *Aspects comportementaux* : il faut identifier chacun des éléments du contexte et, le cas échéant, un automate décrivant le protocole de communication. Au contraire d'un composant qui a un comportement plutôt séquentiel, les connecteurs peuvent représenter des comportements parallèles (par exemple, si l'on décrit un mécanisme de diffusion).

Ports : ils sont des points d'interaction liés à un connecteur ou à un composant.

- *Aspects structurels* : il faut décrire la structure des informations permettant d'identifier le traitement à appliquer, dans un composant ou dans un connecteur. Pour un composant cela revient à définir la signature des services offerts ou requis par le composant. Pour un connecteur il faut définir un en-tête décrivant un discriminant (e.g. l'initiateur ou le destinataire d'un service, la priorité de traitement, etc.).
- *Aspects comportementaux* : il s'agit des contraintes sur la sémantique d'interaction imposé par les ports (e.g. scenario distraction).

Relations entre les composants/connecteurs (**entités**) et leurs **ports**.

- *Aspects structurels* : il s'agit principalement des contraintes de cardinalité éventuellement imposées par un style architectural. Par exemple, un ADL peut limiter le nombre de ports associés à une entité (c'est le cas des diagrammes d'états dans UML).
- *Aspects comportementaux* : on considère en général les conditions de création dynamique de nouveaux ports. Par exemple, le mécanisme de réflexivité est un moyen pour modifier ou ajouter des interfaces pendant l'exécution.

Connexion entre les ports (binding au sens de RM-ODP).

- *Aspects structurels* : il s'agit des contraintes régissant l'association entre ports. Par exemple, un style architectural peut imposer que seul un port de type «in» sera associé à un port de type «out».
- *Aspects comportementaux* : cela concerne la description du cycle de vie d'une connexion entre deux ports (on parlera aussi de binding).

Composition (relations entre entités composants ou connecteurs).

- *Aspects structurels* : il s'agit des contraintes de composition à appliquer pour un style architectural donné. Par exemple, on peut autoriser une structuration hiérarchique ou l'enchaînement d'entités (connecteurs et/ou composants).
- *Aspects comportementaux* : La composition d'entités ADL : cela concerne la définition des règles sémantiques de composition permettant de coller ensemble les comportements des plusieurs entités ADL. Par exemple, la composition de composants actifs (e.g. processus) communicants nécessite de préciser des règles sur la politique d'ordonnancement et sur la propagation des priorités d'exécution.

Nous étudions maintenant une sélection de langages de description d'applications logicielles à la lumière de ces critères d'analyse.

3.2.2 UML

UML **[86]** est une notation graphique adaptée à la modélisation des systèmes logiciels selon une approche Orientée Objet. Issue de l'unification de plusieurs notations tel que OMT, OOSE et Booch, UML définit neuf diagrammes pour décrire un système selon quatre *points de vue* :

- La vue **Fonctionalité gros grain** exprime, sous la forme de cas d'utilisation, les relations «gros grain» entre les fonctionnalités du système et les acteurs externes.
- La vue **Structure logique** exprime, sous la forme de diagrammes de classes et de diagrammes d'objets, l'organisation hiérarchique et structurelle du système.
- La vue **Comportement** exprime les interactions entre les composants d'un système. Quatre diagrammes permentent d'en présenter des vues complémentaires :
  - Les *diagrammes de séquence et ceux de collaboration* expriment des scénarios d'interaction entre les objets. Le premier se focalise sur une vision temporelle tandis que le second permet d'utiliser des opérateurs de langage (+ et *).
  - Les *StateCharts* représentent le comportement interne d'un objet sous la forme d'un automate.
  - Les *diagrammes d'activité* expriment le flot d'exécution des diverses activités d'un système.
- La vue **Implémentation** est une vue complémentaire décrivant l'implémentation du système. Les diagrammes de composants représentent l'architecture logicielle du système et les diagrammes de déploiement décrivent la topologie du déploiement pour une plateforme d'exécution donnée.

L'analyse de Simons **[116]** révéle plusieurs incohérences de modèlisation de la notation UML. Nous ne nous intéressons ici qu'à l'étude de cette notation par rapport aux contraintes de style architectural. Cette analyse n'inclut pas des extensions d'UML tel ROOM ou UML-RT. Ces travaux ne faisant pas partie intégrante de la norme, ils seront analysés séparément.

Composants

UML supporte la notion de composants logiciels et les considère comme des collections de classes ou de paquetages de classes fortement couplées, et donc à déployer ensemble. UML fait la distinction entre un composant et ses instances : un diagramme de déploiement peut instancier le même composant plusieurs fois sur plusieurs nœuds d'exécution indépendants. On peut dire qu'UML a tendance à considérer les composants comme des unités d'implémentation «gros grain» (fichiers, programmes exécutables etc.).

Plusieurs diagrammes permettent d'exprimer le comportement d'un système. Parmi eux, les StateCharts et les diagrammes d'activités sont fondés sur la même base formelle : les machines à états fini d'UML **[86]**. L'intérêt est d'utiliser ces diagrammes en conjonction. Les diagrammes d'interaction (i.e. séquence et collaboration) sont aussi adaptés à la description du comportement d'un système mais ils expriment moins bien le comportement interne des composants car :

- Ils ne concernent que les interactions entre les instances d'objets. De ce fait, un diagramme de séquence (ou collaboration) exprime un scénario mais pas l'intégralité des comportements possibles.
- Il est difficile de synthétiser un système dans son ensemble à partir des seuls scénarios d'interaction. Une telle opération suppose que les règles d'assemblage des scénarios soient définies au niveau de la notation UML, ce qui n'est pas le cas **[67]**. Des travaux récents tel que les Live Sequence Charts visent à donner une solution à ce probléme **[27]**.

La sémantique opérationnelle des machines à états d'UML est basée sur trois concepts de base : une *file d'événements* en entrée, un *dispacheur d'événements* et la *machine à états* en elle-même. La file est dédiée à l'enregistrement d'événements en vue de leur traitement. Le dispacheur sélectionne un par un les événements que la machine à états doit traiter. Du point de vue de la description du comportement, différents problèmes sont identifiés :

- La règle d'élection des événements par le dispatcher n'est pas précisée dans le standard, contrairement à ce que font d'autres langages comme SDL **[63]** (qui utilise une sémantique FIFO prioritaire). Les utilisateurs et les constructeurs d'outils risquent ainsi de faire des choix d'interprétation différents et souvent incompatibles.
- Les machines à états d'UML sont principalement adaptées à la modélisation de *systèmes réactifs* car l'arrivée d'un événement déclenche un traitement associé **[86]**. Ceci est une contrainte de modélisation majeure car d'autres classes de systèmes existent (agents autonomes, applications centrés sur le traitement par flots de données etc.). Il n'est pas facile de les décrire en UML **[107]**.
- Un événement sélectionné par le dispacheur est diffusé à tous les états actifs de l'automate; une transition ne pourra être franchie qu'à partir de l'un de ces états actifs et si de plus sa condition de garde est vérifiée. Cependant, la sélection d'un événement ne garantit pas l'existence d'une transition franchissable. Dans ce cas, l'événement sera perdu (si il n'est pas déféré explicitement par l'état actif sélectionné).
- UML propose une sémantique d'exécution par étapes de traitement (rounds). A chaque round, un événement est sélectionné. Une ou plusieurs actions sont alors déclenchées. Cependant, bien que plusieurs actions peuvent être exécutées en parallèle, UML 1.4 **[90]** précise que leur exécution doit être séquentialisée. Cette hypothèse simplifie la sémantique d'exécution mais introduit une limite importante : *on ne peut plus différentier une méthode synchronisée statique (non réentrante) d'une méthode statique (réentrante)*.

Les diagrammes d'activité, reposant sur les mêmes principes, souffrent donc des mêmes problèmes.

Connecteurs

UML n'offre pas d'artefact parfaitement adapté pour la description des connecteurs. Pour modéliser cette notion il faut donc soit surcharger la sémantique d'UML soit l'étendre à travers des profils comme dans **[65]** ou directement au niveau du méta modèle **[80]**. Puisque nous nous intéressons à l'évaluation d'UML dans sa version de base, nous écartons ici ces solutions.

Deux propositions de modélisation nous semblent plus naturelles :

- *L'utilisation d'associations UML*. Une association UML exprime une relation de navigabilité entre les entités logique (classes, paquetages, composants). Elle décrit donc soit une liaison logique entre classes ou paquetages d'une application, soit un canal de communication entre les nœuds d'exécution. La communication entre deux entités UML est possible seulement s'il existe un chemin d'accès.
  Pour définir la structure et le comportement d'une association, on utilise des «*classes d'association*» qui décrivent l'intermédiaire chargé de réaliser l'association. La figure 3.3 illustre cette facilité de modélisation dans un système Client-Serveur. Cette modélisation est conforme au gabarit de conception «proxy», la classe d'association est un intermédiaire entre les clients et les serveurs.

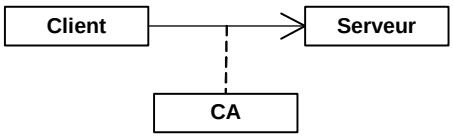


Figure 3-2 : Exemple de classe d'association en UML.

Bien qu'il soit possible de modéliser la structure et le comportement de l'association, il est impossible de spécifier comment déployer une association logique sur un canal de communication. Par exemple, dans un cas concret de *N* Clients et 1 Serveur, cette association peut se déployer sous la forme de *N* connecteurs point à point ou par l'intermédiaire d'un unique canal de communication partagé par tous les Clients. Ces deux implémentations ne sont pas sémantiquement équivalentes, l'analyse du comportement global du système ne peut donc pas être élaborée alors même que l'on dispose du comportement des entités qui le composent.

- *L'utilisation de composants dédiés à la communication*. Une deuxième proposition est la modélisation des connecteurs au moyen de composants spécialisés pour la communication. Cette proposition a une granularité «gros grain», et elle est contraire aux principes prônées par les ADL **[32]**.

Ports

UML propose la notion d'interface pour exprimer l'enveloppe externe d'une entité logicielle (classe, paquetage ou composant). Une interface est un ensemble de méthodes et de signaux (événements) qui réalisent un service. Les interfaces sont le moyen le plus naturel pour modéliser les ports de communications en UML.

Ce choix implique de fortes contraintes sur le style architectural car :

- UML ne précise pas comment interpréter et valider la signature de méthodes. Choix est laissé à l'implémentation, la signature d'une méthode sera interprétée différemment d'un langage de programmation à un autre.
  Prenons un cas caricatural : Ada calcule la signature d'une méthode en fonction de son nom, du type retourné (s'il y a lieu) ainsi que du nom et du type de ses paramètres alors qu'en C, seul le nom de la méthode est utilisé. Ainsi pour implémenter un modèle UML, il faudra dans certains cas recourir à des artifices qui ne sont pas exprimés dans la spécification en entrée.
- UML ne permet pas de modéliser des interfaces dynamiques, personnalisables pendant l'exécution. Il est en effet impossible de rajouter ou retirer des services à une interface pendant l'exécution. Un tel mécanisme est possible dans un langage réflexif tel que Python **[77]**. En conséquence, le style architectural UML se limite à la modélisation statique d'interfaces, ce qui est dommage
- Une interface ne regroupe que les services offerts par un composant, au contraire des propositions de RM-ODP ou d'ADL comme ROOM **[111]** ou ACME **[40]** qui définissent des *ports bi-directionnels*. Contrairement aux promoteurs de ces ADL, nous considérons que c'est plutôt un problème de sucre syntaxique car UML permet quand même de préciser des clauses d'utilisation «use» (voir la **Figure 3-3 a**) pour les interfaces requises par une entité logicielle.

Du point de vue comportemental, nous constatons d'autres problèmes de modélisation. Les types d'interactions offerts par UML (méthodes et signaux) ne couvrent pas toutes les possibilités d'interaction **[82]**. Par exemple, les messages synchrones que l'on trouve dans MPI **[119]** ou les flots de caractères définis dans RM-ODP ne peuvent être exprimés.

En revanche, les contrats d'interaction sont bien exprimés par les scénarios décrits dans les diagrammes de séquence et de collaboration. Hélas, ces descriptions n'ont qu'un caractère partiel car la synthèse d'un automate complet du protocole d'un objet à partir de scénarios n'est pas possible sans une heuristique de composition **[67]**. Cette dernière n'est pas précisée dans UML.

Ports des composants ou des connecteurs

En UML, bien qu'il soit possible de définir autant d'interfaces que l'on veut pour un composant, ce nombre et leurs structures sont déterminés à la modélisation. Il est donc impossible de rajouter dynamiquement de nouvelles interfaces ou services et par conséquent il est impossible de décrire des mécanis-

mes comme la réflexivité. Cette observation s'applique également aux classes d'association (les connecteurs d'UML).

Liaison des ports

UML ne distingue pas la notion de liaison entre interfaces (les ports en UML). Ces dernières sont assimilables à une partie déclarative incluse dans l'implémentation d'une autre classe (un «header»). Les connecteurs sont donc dépendants des interfaces des composants.

La **Figure 3-3** illustre ce problème de dépendance conceptuelle sur l'exemple de Client-Serveur dont la structure est donnée dans la **Figure 3-2**. En UML, la classe d'interface CA implémente le protocole entre un client et un serveur ; elle importe donc l'interface du serveur car la clause de visibilité va du client vers le serveur. Si l'on modifie ce serveur, il faut reconstruire CA car elle importe désormais une nouvelle version de l'interface du serveur (schéma de gauche).

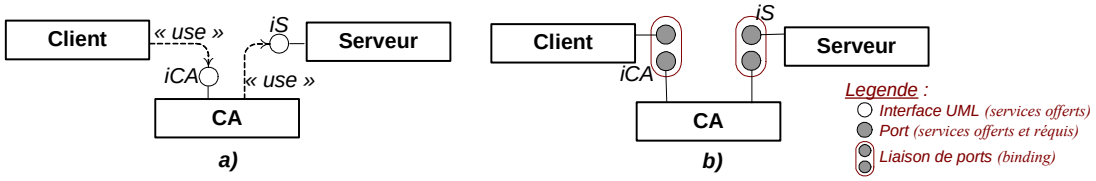


Figure 3-3 : Liaisons entre ports, la différence entre (a) UML et (b) un ADL.

Un découplage des liaisons permet de réutiliser le même connecteur. En effet, ce dernier exporte des points d'interactions («binding point» dans RM-ODP et Ports dans les ADL) qui sont associés dans un schéma d'architecture aux *points de liaison* des classes (ici, le Client, CA et le Serveur). L'association (on parle de liaison de ports ou binding) entre points d'interactions (réalisée lors de la configuration du système) est exprimée indépendamment des interfaces des composants connectés (schéma de droite). Ce problème est l'un des points principaux identifiés dans les études comparant UML et des ADL **[80]**.

Relations entre entités

UML ne permet pas la composition hiérarchique de composants. Les choix de déploiement dans UML font que ces composants ont une granularité importante, un composant étant déployé au moyen d'un fichier, d'une bibliothèque de code ou d'un programme exécutable **[80]**.

Le chaînage des connecteurs, quant à lui, soufre d'un problème similaire. En effet, une classe d'association UML (qui exprime un connecteur) ne peut pas être composée d'autres classes. La solution proposée est d'utiliser les classes pour exprimer à la fois la partie fonctionnelle d'une application ainsi que la partie protocolaire, c'est-à-dire les connecteurs. Ce choix de modélisation est très critiqué par la communauté ADL car il est considéré comme peu adéquat à la description d'architectures réparties **[80]**. Le fait reproché est de surcharger inutilement la sémantique des classes UML pour exprimer des concepts différents.

3.2.3 RM-ODP

La norme RM-ODP **[62]** définit un modèle de référence pour la conception d'applications réparties ouvertes. La spécification d'un système ODP est réalisée en cinq étapes de développement, chacune d'entre elles concernant la description d'un aspect de conception différent. Par exemple :

- La vue *Entreprise* s'intéresse aux objectifs, au domaine d'utilisation et aux politiques afférentes à l'application.
- La vue *Information* concerne la description des données manipulées par l'application et aux contraintes d'usage et l'interprétation qui correspondent à ceux-ci.

- La vue *Traitement* visé la décomposition fonctionnelle d'une application en un ensemble d'objets qui interagissent à des interfaces pour permettre de répartir le système.
- La vue *Ingénierie* est un raffinement de la vue Traitement, elle s'intéresse à la description des mécanismes et aux fonctions nécessaires pour la mise en œuvre des interactions réparties entre objets du système.
- La vue *Technologie* vise à préciser les choix et les solutions techniques d'implémentation (langages de programmation, intergiciels, modèles de gestion, etc.) capables de prendre en charge la répartition du système.

Nous portons notre attention sur l'étude des vues traitement et ingénierie car elles concernent directement l'architecture logicielle d'un système. Nous nous intéressons donc à l'étude du style architectural proposé dans RM-ODP. Une telle étude est intéressante car ODP offre plusieurs avantages :

- un style architectural neutre, adapté à la répartition,
- l'indépendance de la norme par rapport à la notation utilisée pour modéliser le système et vis-à-vis des choix d'implementation,
- une approche de spécification par raffinements au moyen de vues successives (Traitement, Ingénierie et puis Traitement),
- une conception rigoureuse basée sur la définition de *points de référence* et de *contrats* associés.

RM-ODP propose un style architectural neutre qui fait peu de suppositions sur le paradigme de programmation répartie choisi. Ainsi, un objet de traitement ODP peut avoir une granularité variable : par exemple, il peut correspondre à une simple procédure, à un objet (au sens des langagages à Objets), ou à une application. La neutralité d'ODP est également due à la séparation totale entre l'architecture d'un système et le parallélisme de son exécution. Ce n'est pas le cas d'autres langages spécialement conçus pour un paradigme (par exemple, les processus communicants dans SDL).

RM-ODP propose un cadre terminologique et un style architectural adaptés à la spécification d'application réparties ouvertes. Cependant, puisque aucune contrainte n'est précisée sur la notation, sur le processus de développement ou sur les technologies d'implémentation à utiliser. Le style architectural proposé est épuré de toute contrainte spécifique à un langage de modélisation ou paradigme d'implémentation, d'où l'intérêt d'utiliser ce style architectural comme pivot.

RM-ODP peut s'interpréter comme un processus de transformation par raffinements. Cela est particulièrement vrai pour les vues traitement, ingéniérie et technologie. On peut donc assurer une cohérence entre les spécifications de haut niveau et leurs raffinements ce qui facilite la traçabilité des spécifications. Par exemple un objet de traitement sera implémenté par un ensemble d'objets d'ingénierie auxquels on associe des contraintes d'implémentation complémentaires.

RM-ODP est adapté à l'élaboration de systèmes hétérogènes. Pour faciliter l'intégration et le test de conformité aux spécifications, la norme définit deux concepts :

- *points de référence*: Ils correspondent à des points d'interaction observables permettant de vérifier la conformité des spécifications par rapport aux contraintes de style architectural et à d'autres spécifications (e.g. situées à des niveaux de description différents) ou permettant de vérifier la conformité des implémentations par rapport aux spécifications (modéles).
- *contrat*: Il définit des contraintes interprétables comme des «obligations de preuve» statuant sur le comportement que doit respecter un ensemble d'objets. Cette notion peut être associée à celle de point de référence afin de préciser des assertions à respecter lors du raffinement des modèles.

Ces quatre caractéristiques font de RM-ODP un candidat intéressant à étudier. Nous évaluons donc en détail son style architectural.

Composants

La vue ingénierie décrit le partitionnement d'une application en vue de son déploiement. Plusieurs concepts de hiérarchisation y sont définis : objets, grappes, capsules. Parmi eux, la grappe (ou «cluster») RM-ODP exprime l'unité de déploiement et de gestion la plus fine pour les objets d'ingénierie (i.e. objets d'implémentation), elle correspond donc à un composant logiciel (voir **Définition 2-6**).

La norme ODP propose une terminologie mais sans lui associer de notation spécifique. Elle ne propose donc pas de mécanismes permettant de décrire finement la structure et le comportement d'un objet. Cela empêche de spécifier le comportement attendu d'un composant. En effet, une telle description nécessiterait la définition d'une notation, ce qui n'entre pas dans l'objectif de ODP.

Connecteurs

La fonctionnalité d'une application et le code de communication sont spécifiés séparément. Pour encapsuler les interactions réparties, la norme ODP propose les concepts d'*objet de liaison* (dans la vue traitement) et de *canal de communication* (dans la vue d'ingénierie). Ainsi, un objet de liaison correspond à un connecteur et un canal de communication à sa mise en œuvre.

La vue d'ingénierie permet de raffiner la structure d'un canal de communication sous la forme d'une liste chaînée d'objets spécialisées : *objets talons*, *objets lieurs*, *objets de protocole*, *filtres d'interception*. Cependant, cette structuration reste générale car elle ne permet pas de spécifier plus précisément ces objets. Pour ce faire, il faudrait une notation dédiée, ce qui n'entre pas dans les objectifs de la norme RM-ODP.

Ports

Le style architectural de RM-ODP est basé sur la composition d'objets répartis au moyen d'interfaces. Une interface de traitement désigne un port de communication typé via lequel un objet peut communiquer. Contrairement à UML, les interfaces dans ODP sont duales : elles définissent à la fois les services offerts et requis par un objet.

D'un point de vue structurel, une interface définit la liste des services accessibles via un moyen de communication donné. Les types d'interaction permettent de spécifier la majorité des applications existantes mais certains cas ne peuvent être pris en compte. C'est en particulier le cas des messages synchrones dans MPI **[119]**. Enfin, pour exprimer des mécanismes de communication par diffusion, il faut définir un objet de liaison dédié.

Trois types d'interfaces sont précisés :

- les *opérations* représentent la communication synchrone par invocation de procédures ou méthodes distantes ; lorsque la procédure invoquée est asynchrone, cela correspond également à l'envoi d'un message;
- les *flots de caractères* représentent le transport de données en continu (par exemple, un tube nommé UNIX);
- les *signaux* représentent des événements dont la sémantique est inspirée des signaux Unix.

Ports des composants ou des connecteurs

Aucune contrainte n'est spécifiée par la norme sur le nombre de ports des composants ou des connecteurs.

Liaison des ports

La composition d'objets est réalisée par la connexion des interfaces ODP (binding points). Plusieurs contraintes doivent alors être respectées :

- La connexion n'est possible qu'entre des interfaces du même type. Par exemple, il est impossible de connecter une interface de type opération à une interface de type flot de caractères.
- Pour connecter deux interfaces il faut à la fois que la liste des services qu'elles définissent et leur signatures soient identiques. Par exemple, on ne peut connecter une interface proposant deux services A et B à deux interfaces distinctes, l'une requérant un service A, l'autre un service B ; en effet, les interfaces sont indivisibles.
- Les services doivent être compatibles : un service offert par une interface doit être requis par l'interface duale.

Sur ce point encore, l'objectif de la norme ODP n'est pas de proposer une notation dédiée. Ainsi, le support de la composition de services reste sujet à des interprétations.

Relations entre entités

Comme indiqué dans le **Tableau 3-1**, la composition d'entités ADL concerne l'assemblage de composants (entre eux), celui des connecteurs (entre eux) et celui des composants et des connecteurs.

Composition de composants

La notion de *capsule* permet de simplifier le déploiement d'une application. Une capsule représente un conteneur d'exécution et constitue l'unité de protection de l'exécution pour les grappes (les composants logiciels dans RM-ODP). Son rôle est similaire à celui d'un processus : elle fournit un espace d'adressage protégé et partage les ressources d'un nœud de manière équitable.

Cependant, bien que la norme permette le regroupement de grappes en capsules en vue de leur déploiement, elle ne permet pas la composition hiérarchique de grappes. L'approche proposée est minimale car la définition de macro composants (grappes englobant d'autres grappes) n'est pas supportée. Cela représente une contrainte qui limite le style architectural de RM-ODP à une hiérarchie aplatie. De tels systèmes ne peuvent tirer profit des facilités de composition hiérarchique telles que celles offertes par DCOM, un intergiciel permettant l'assemblage hiérarchique de composants logiciel.

Composition de connecteurs

Un canal de communication ODP est constitué d'une chaîne d'objets de communication. Le style architectural ainsi imposé permet de paramétrer ou de configurer un canal de communication mais interdit paradoxalement la composition directe de canaux. Les objets de communication sont donc l'unité de réutilisation choisie (i.e. talons, filtres d'interception, protocoles etc.). En conséquence, il n'est pas possible de composer plusieurs canaux de communication afin d'obtenir des gabarits de communication plus complexes.

Composition de composants et de connecteurs

Le style architectural imposé par ODP oblige le concepteur à utiliser des canaux entre les composants (les grappes ODP) si ceux sont déployées à par dans des capsules differantes. Cette approche est conforme à la philosophie des ADL et, en forcant les concepteurs à se poser la question des communications, peut constituer la base d'une démarche de réalisation des systèmes répartis.

Conclusion

Bien que structurantes, les descriptions des notions couramment utilisées dans la modélisation de systèmes répartis manquent hélas de rigueur de définition et d'interprétation. A nouveau RM-ODP laisse ouvert le choix pour une notation concrète. Par exemple, on ne sait pas comment définir et interpréter la signature des services et des interfaces. De plus il est impossible de fixer la sémantique d'interaction.

La description des aspects de coordination (e.g. concurrence d'accès à un service, ordonnancement d'invocations, scénario d'interaction admissibles etc.) est complètement ignorée par la norme.

Néanmoins, pour combler ce manque de précision RM-ODP considère les interfaces comme étant des points de référence et de conformité auxquels, par l'intermédiaire d'une notation concrète, on peut associer des contrats afin de contraindre leur structure ou pour spécifier des contraintes comportementales sur l'interaction.

3.3 Langages de Description d'Architecture

L'étude des Langages de Description d'Architectures (ADL) constitue le point principal de notre analyse. La différence existante entre ces langages et les langages de modélisation classiques (e.g. UML) est que les premiers visent la description de l'*architecture de déploiement* (e.g. composants connectés par des connecteurs selon une configuration) et non la description de l'*architecture logique* d'une application (e.g. classes et objets emboîtés dans des paquetages et des composants).

3.3.1 Sélection d'ADL

Il existe de nombreux ADL correspondant à des domaines d'applications et des styles architecturaux différents **[38, 80]**. La grande majorité des ADL décrivent la structure logique d'un système de manière satisfaisante (au moins aussi bien qu'UML) et certains sont également performants pour la description des aspects comportementaux. Parmi eux, nous avons sélectionné plusieurs candidats représentatifs par rapport à plusieurs critères de sélection compatibles avec ce que nous avions proposé dans la **Section 2.4.8** : *Conclusion : critères et bases pour un langage de modélisation*. Les critères les plus importants sont :

- la capacité de description pour les architectures réparties et leur originalité,
- la formalité dans la définition et leur compatibilité avec les méthodes de vérification formelles,
- la compatibilité avec la notation UML,
- l'adéquation à l'implémentation et à la certification.

Ces candidats sont :

- **RAPIDE [66]** : développé à l'Université de Stanford, cet ADL est dédié à la simulation de spécifications et au test de conformité d'une implémentation. Basé sur un modèle de communication par événements (messages), l'originalité de Rapide reside dans la manière dont il représente les traces d'exécution (i.e. comportement des composants et des connecteurs).
  La notation proposée utilise les «Partially Ordered Set of Events Traces» (POSET) qui expriment une séquence d'événements partiellement ordonnés en fonction de leur causalité ou par rapport à une horloge locale. Ce mécanisme est utilisé pour exprimer le comportement souhaité pour les diverses entités d'un système (les composants comme les connecteurs).
- **Wright [3]** : réalisé à Carnegie Mellon University, il est basé sur le langage CSP (Communicating Sequential Processes de C.A.R.Hoare) **[114]** muni d'une notation algébrique formelle décrivant le comportement de processus séquentiels communicants. Wright considère les connecteurs comme des entités paramétrables et réutilisables. Son approche de description est ouvert : il est possible de l'étendre pour définir de nouveaux styles architecturaux. Cet ADL est particulièrement bien adapté à la vérification formelle de modèles par exploration exhaustive des états (model-checking). Ainsi Wright permet de verifier et de valider en isolement le comportement des composants: pour ce faire, il demande de préciser des automates protocolaires associés aux ports, et de synchroniser ces automates avec celui du composant.

Le processus de description proposé est par raffinements. Ces raffinements garantissent cependant la monotonie des propriétés : les assertions d'un modèle restent valables pour le modèle raffiné.

- **C2-SADL [74]** : dédié aux applications respectant le style architectural Chirion-2 (C2) **[79]**, son style architectural est particulièrement adapté à la description de systèmes dynamiques à large échelle et à l'intégration de composants. C2 est formellement défini avec Z **[122]**, une notation algébrique basée sur la théorie des ensembles et le calcul des prédicats.
  Les applications C2 sont structurées hiérarchiquement en couches de composants reliés par des connecteurs. Un composant logiciel ne dispose que de deux interfaces (ports) pour communiquer avec ses supérieurs hiérarchiques (top) et pour l'interaction avec les composants subordonnés qui sont plus proches de l'exécutif (bottom).
  Notre intérêt pour C2 n'est pas lié à son style architectural mais à son principe de conception. En effet, C2 promeut ce qu'on appelle «*le principe du couplage faible dans la conception*». Par exemple, un composant est censé ignorer les fonctions des composants subordonnés. Cela permet un redéploiement facile des applications sur des exécutifs variés car la spécification de haut niveau (les couches supérieures) ne dépend pas des couches inférieures. Pour permettre une telle indépendance, les connecteurs font office de traducteurs et d'aiguilleurs des messages entre les couches. Le modèle de communication proposé est asynchrone.
  Les caractéristiques principales de C2 sont le découplage fonctionnel des composants et la construction modulaire d'applications dynamiques. Bien que les auteurs ne l'affirment pas, ces objectifs ressemblent à ceux de la Programmation Adaptative **[72]**, c'est-à-dire la séparation des aspects conception, composition dynamique et réutilisation.
- **MetaH [136]** : développé par Honeywell pour le prototypage de systèmes embarquées critiques (avionique militaire et civile), MetaH peut être perçu comme une extension du langage Ada qui permet de greffer à ce langage les capacités de description et de composition de haut niveau spécifiques à un ADL. Le style architectural proposé respecte un modèle de conception par tâches communicantes (avec des niveaux de priorités) reliées par des règles de précédences étiquetées par des délais de communications.
  Cet ADL se prête particulièrement bien à :
  - *La description complète de l'architecture logicielle et d'exécution* (matérielle) ainsi qu'au déploiement de la première sur la seconde.
  - *L'analyse formelle des modèles* pour vérifier la faisabilité du déploiement (e.g. ordonnancement correct des tachées, délais de communications bornés, temps d'exécution etc.) ou pour évaluer les risques des pannes physiques partielles dans un système (e.g processeur défet). L'analyse d'une spécification est une étape préliminaire, elle permet d'évaluer la conception de très tôt sans avoir besoin de l'implémentation. Par exemple, il suffit de décrire les caractéristiques comportementales des tâches et celles de media de communication pour que l'analyse soit possible.
  - *La génération automatique de programmes conformes* aux spécifications et à l'intégration des programmes existants. Le code peut être instrumenté et vérifié pour conformité par rapport aux spécifications.
- **ROOM [111]** : Initialement conçu par ObjecTime, ROOM est un ADL formel pour la modélisation de systèmes temps réel selon une approche objet. Il se distingue sur les aspects suivants :
  - Le style architectural est basé sur la notion d'*Acteur logiciel*, un paradigme de programmation pour la construction d'applications réparties faiblement couplées **[2]**. Un Acteur est un objet actif. Il exprime l'unité de traitement élémentaire, encapsule un état (i.e. données non partagées), propose des services (méthodes) et dispose d'un fil d'exécution propre (contexte d'exécution). La communication avec un acteur est réalisée par échange des massages asynchrones. Chaque acteur dispose d'une adresse unique et d'une boite de messages privée.

- L'approche de modélisation est à la fois compatible et complémentaire de la notation UML. En effet, les Acteurs ROOM et les objets actifs UML sont des notions équivalentes; de plus, le comportement des Acteurs est décrit formellement à l'aide d'une variante des diagrammes d'états de Harel. Pour cette raison, cet ADL a été choisi dans le projet de profil UML-RT (UML Real-Time) **[78]**.

ROOM propose une architecture logicielle simple (mais pas simpliste) dont l'avantage est d'être formelle et facile à implémenter. Ce style architectural peut être considéré comme le plus petit dénominateur commun qu'un ADL doit supporter.

3.3.2 Composants

La **Figure 3-4** illustre la façon dont les ADL étudiés décrivent les composants logiciels. Nous observons que, dans leur totalité, les ADL analysés ne s'intéressent qu'à la description de la partie visible exposée par un composant, cette description concerne les ports de communication et le contrat d'utilisation. La modélisation de la partie interne des composants, quant à elle, est totalement ignorée.

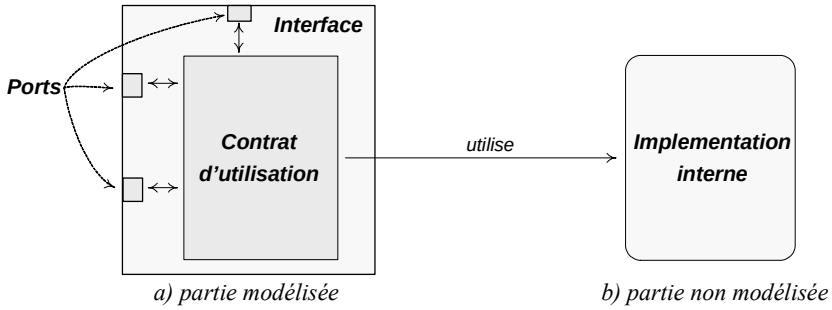


Figure 3-4 : Description des composants logiciels : l'approche classique.

Une telle démarche de description partielle limite les capacités de vérification et d'analyse formelles sur les modèles et c'est pourquoi certains ADL comme Meta-H utilisent des annotations explicites sur les modèles afin de préciser des contraintes d'implémentation (e.g. puissance et temps d'exécution nécessaire pour l'exécution d'un service, latence de transmission, gabarit mémoire occupée etc.).

L'utilisation de des contraintes permet d'effectuer une analyse de faisabilité préalable sur les modèles, cependant elle ne garantit pas la conformité des l'implémentations concrètes par rapport aux gabarits imposés. Pour faciliter cette tâche, des profils (e.g. RAVENSCAR **[30]**) et des normes (e.g. DO-178B**[112]**) de construction sont souvent imposés: leur but est de contraindre la réalisation (e.g. interdiction des d'appels cycliques, allocation statique de ressources, bornes sur le nombre de cycles pour les boucles etc.) afin de garantir des bornes sur le gabarits d'implémentation. Nous considérons ce choix comme réducteur vu qu'il limite le spectre d'applications modélisables.

Description du comportement

Nous classons les ADL retenus en fonction des choix effectués pour décrire le contrat comportemental d'un composant, c'est-à-dire la manière dont il va interagir avec les autres. Plusieurs approches se distinguent :

- RAPIDE, tout comme C2-SADL, spécifie le contrat comportemental au moyen de contraintes de causalité entre les événements en entrée d'une action et ceux engendrés en résultat. Le comportement d'un composant peut être concurrent.
- Wright décrit le comportement d'un composant à l'aide d'un automate séquentiel (processus). Au contraire de SDL qui utilise un modèle de communication asynchrone par files de messages, Wright est basée sur un modèle de synchronisation fort, basé sur les rendez-vous.

- MetaH tout comme ROOM ou SDL se trouvent à mi-chemin entre ces deux approches (concurrent et séquentiel), la concurrence d'exécution d'un composant est réalisée par la composition hiérarchique de composants séquentiels (acteurs de base ROOM).

Le choix d'un modèle de description à base d'automates séquentiels communicants, facilite donc la production automatique de programmes et la vérification. Cependant, nous considérons que les styles architecturaux utilisés pour la description de la partie visible (tâches communicantes) et l'implémentation des composants (objets ou fonctions passives, et tâches actives) ne correspondent pas d'où l'intérêt de faire la liaison entre ces deux.

3.3.3 Connecteurs

Nous allons examiner deux points importants :

- l'existence explicite dans les ADL de la notion de connecteur,
- la présence de mécanismes permettant de particulariser ces connecteurs.

ROOM tout comme UML-RT modélise les connecteurs à l'aide des associations stéréotypées. Cette solution est considérée comme partielle dans **[33]** car elle ne permet pas de modéliser leur comportement et souffre de problèmes de cohérence dans la modélisation. Par exemple, la composition d'automates pour les composants ne peut être réalisée sans considérer des heuristiques basées sur la sémantique des connecteurs ainsi que sur les règles de composition.

RAPIDE considère les connecteurs comme des caractéristiques d'assemblage. Leur description est réalisée au niveau de l'architecture globale d'un système, elle revient à une liste de contraintes de liaison. La réutilisation des spécifications est compromise puisqu'il faut redécrire tous les connecteurs pour chaque nouvelle application ou configuration.

Meta-H tout comme UNICON(1) **[141]** se contente de proposer un ensemble fixe de connecteurs types assurant des mécanismes connus et bien spécifiés (e.g. FIFO prioritaires). Les règles de composition sont claires et la réutilisation de connecteurs est assurée. Cependant, la conception de nouveaux connecteurs est soit impossible (en Meta-H) soit difficile (en UNICON) car elle demande des compétences d'expert. La solution orthodoxe consiste à spécifier les connecteurs de manière non ambiguë à l'image de ce qui est fait pour les composants.

WRIGHT se distingue comme l'ADL le plus flexible : il permet de décrire à la fois les comportements d'accès acceptables au niveau des ports et ceux des protocoles de communications associées aux connecteurs (appelés «glue»). La réutilisation modulaire et l'extension de connecteurs sont assurées. De même, les règles de composition entre les automates CSP correspondant aux diverses entités modélisées (composants, ports, connecteurs) sont formalisées.

C2 propose des connecteurs indépendants des données véhiculées. Celles-ci sont considérées comme des messages dont le contenu est opaque. Cela constitue une forme de particularisation, non sur le comportement mais sur les données véhiculées.

En conclusion, il apparaît intéressant de traiter les connecteurs comme des entités distinctes des composants. Il faut également proposer des mécanismes de communication connus et offrir un moyen permetant de particulariser les connecteurs afin de décrire des protocoles non standard. Il est également souhaitable que la description des connecteurs puisse être indépendante du type d'informations véhiculé.

(1) un ADL spécialisé dans la composition modulaire d'applications hétérogènes.

3.3.4 Ports

La sémantique donnée au concept de port dépend du style architectural choisi. Parmi les ADL étudiés deux courants contradictoires existent :

- RAPIDE propose une vision abstraite : le port n'est rien d'autre qu'une façade qui impose un contrat d'utilisation logique.
- C2, ROOM et Meta-H, SDL considèrent les ports comme des *points d'interactions* concrets. Cette approche nous semble plus adéquate car elle permet de différencier les interactions par raport aux voies de communication. Cela permet de différencier les traitements des services selon les Ports d'entrée. Par exemple, l'invocation d'un service `S` peut être admise quand elle provient d'un port `p1` ou interdite si elle se fait par l'intermédiaire d'un autre port `p2`.

Par rapport à leur structure la description des ports peut être abordée sur deux aspects :

- *Typage*. Les ports sont en général typés statiquement. La signature des services qui transitent est fixée à la conception et ne peut plus évoluer ensuite. Une telle approche a un sens pour les composants logiciels. Cependant, elle a de lourdes conséquences sur la dépendance des connecteurs vis-à-vis des services offerts par les composants (que le connecteur véhicule). Cela nuit à la réutilisabilité des connecteurs.
  De plus, les composants transmettent une information explicite aux connecteurs pour qu'un routage soit réalisé. Cela rend donc la dépendance déjà évoquée réciproque, ce qui empêche de conserver les composants lorsque un protocole d'interaction évolue. La **Figure 3-5** illustre cette dépendance cachée dans la conception des ports.
- *Nombre de ports* : de nombreux ADL permettent de déclarer un nombre arbitraire de ports.

Sur l'aspect comportemental, les ADL identifient en général les caractéristiques suivantes :

- *le modèle d'interaction* : les ADL proposent la notion de message et/ou de RPC «codés» dans des gabarits de ports prédéfinis.
- *la politique d'ordonnancement* : les ADL qui la placent au niveau d'un port (et non d'un composant spécialisé) considèrent dans leur grande majorité un comportement de file avec priorité.
- *contraintes spécifiques* : certains ADL permettent d'indiquer des contraintes complémentaires propres à un domaine d'application particulier (par exemple, la qualité de service dans MetaH).

De cette analyse, nous retenons les points suivants :

- *Sémantique* : la majorité des ADL s'accordent pour considérer les ports comme des *point d'interaction* concrets ; cette solution offre une grande souplesse de modélisation et s'adapte à différents styles architecturaux,
- *Structure* : Le typage des ports est statique. Leur nombre peut varier mais il est fixé à la conception. Nous considérons que le typage statique des ports peut introduire une interdépendance pernicieuse (illustrée dans **Figure 3-5**). Un port de composant définit les services et utilise les entêtes de messages pour contrôler le routage. De même un port de connecteur définit la structure de l'en-tête et peut-être amené à utiliser la définition des services. Le couple composant/connecteur-devient alors indissociable. .

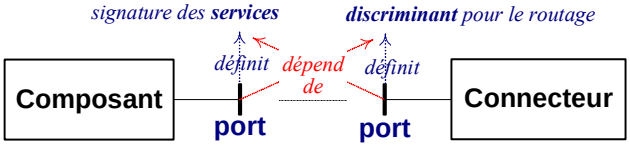


Figure 3-5 : Interdépendance cachée pernicieuse pour la conception des composants et des connecteurs.

- *Comportement* : la communication par messages ou au moyen de RPC semble suffire à la majorité

des applications réparties. Si d'autres comportement sont requis, on peut en effet les simuler via les connecteurs à condition qu'ils soient paramétrables. Il apparaît intéressant de paramétrer la politique d'ordonnancement d'un port, surtout si on assimile son implémentation à des tampons. D'autres aspects comportementaux (e.g. concurrence d'accès à un service) doivent également être étudies et assimilées dans la description des ports.

3.3.5 Relations entre entités

Liaison des ports

Du point de vue d'un ADL, l'assemblage d'applications à partir de composants et de connecteurs correspond à la configuration. Les instances de composants sont liées au moyen de connecteurs pour former une topologie de déploiement concrète. Les ports sont des points de connexion, leur liaison est soumise à des contraintes de modélisation statiques (à respecter pendant la modélisation) ou dynamiques (à respecter pendant l'exécution). Il est souhaitable de disposer des deux mécanismes : le premier (liaisons statiques) est utile dans le cas de systèmes embarqués et le second (dynamique) dans le cas de systèmes reconfigurables; il est concevable que les deux mécanismes cohabitent dans la même application.

Des solutions partielles à l'interdépendance cachée évoquée en **Figure 3-5** sont proposées dans certains ADL comme C2-SADL ou Waves **[48]**. Par exemple :

- en C2-SADL les ports des connecteurs s'adaptent dynamiquement en fonction des ports des composants auxquels ils sont associés, un connecteur C2 considère les messages d'une façon opaque.
- en Waves la conception des composants est déchargé de toute dépendance connexe au routage, un composant émetteur ne doit adresser ses messages, ceux-ci sont routés par les connecteurs selon la topologie de déploiement actuelle.

Comme illustré par la **Figure 3-6**, cela revient à supprimer les dépendances cachées sur la signature des services (pour les connecteurs) et sur la structure des entêtes (pour les composants).

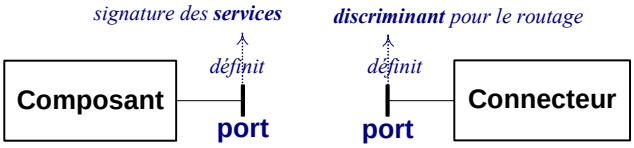


Figure 3-6 : L'élimination de l'interdépendance cachée dans la conception de composants et des connecteurs.

S'il est toujours convenable d'éliminer la dépendance conceptuelle des connecteurs vis-à-vis de la structure de l'information routée, la conception des composants ne doit pas se faire en dehors être de tout schéma d'adressage (imposée par le modèle de routage des connecteurs). En effet, nous constatons que la majorité de langages de programmation (et de modélisation) se basent sur un schéma d'adressage implicite composé par le couple <*adresse source*, *adresse de destination*> qu'il ne faut pas écarter.

Composition

Tous les ADL étudiées considèrent que les composants doivent n'être assemblés que par l'intermédiaire des connecteurs (la connexion directe entre ports de composants est interdite). Le chaînage direct de connecteurs est impossible mais des études récentes comme **[121]** suggèrent une telle approche : ils voient ce chaînage comme un moyen de construire des modèles de communications complexes à partir de mécanismes de base. Cette vison semble fort intéressante car elle correspond à des recherches nouvelles tel que Quarterware **[117]** qui prônent l'idée de la conception d'intergiciels complexes à partir d'un noyau canonique d'entités de communication simples.

Pour permettre la description de systèmes complexes, la majorité des ADL proposent également des mécanismes d'assemblage hiérarchiques : Meta-H s'appuie sur des macros, C2 sur la superposition de couches de composants, ROOM agrège des acteurs et RAPIDE basé sur la composition de modules. L'approche de WRIGHT est intéressant car ce langage définit fort peu de contraintes sur le style architectural. Cela entraîne une grande flexibilité illustrée par la représentation en WRIGHT de modèles de style C2 **[79]**.

3.4 Relation avec les approches formelles

La conception et l'implémentation d'applications réparties sont difficiles lorsque cela concerne des systèmes à haute fiabilité. En particulier, différents problèmes sont identifiés :

- UML, s'il s'agit d'une importante contribution à la modélisation d'applications, ne peut actuellement être considéré comme une solution dans le domaine des systèmes répartis, en particulier parce que les problèmes comportementaux sont mal capturés.
- Ainsi, une fois le système spécifié, l'implémentation est réalisée par des ingénieurs par des techniques de développement traditionnelles. Ces techniques ouvrent la porte à toutes sortes d'erreurs liées à des choix d'implémentation, à une mauvaise compréhension des spécifications, etc. On ne peut donc pas garantir que le programme sera l'image de la spécification dont il est issu.
- Les tests sont réalisés sur le système développé et, par conséquent, les modifications et les corrections également. Ainsi, la spécification initiale finit inévitablement par être obsolète par rapport à la spécification du système. L'impact de nouvelles modifications devient de plus en plus difficile à étudier au fur et à mesure que l'on s'éloigne de la phase de développement initial.

En ce sens, une approche de développement par prototypage constitue une bonne solution à ces problèmes. La procédure de développement s'appuie sur un modèle sur lequel des vérifications peuvent être réalisées. La liaison avec des méthodes de vérification formelles est alors intéressantes à envisager.

Mais Luqi **[75]** montre que tous les problèmes ne sont pas encore résolus. L'un des point principaux noté par les auteurs est le problème de la formation. Il faut que des ingénieurs puissent utiliser les méthodes formelles sans connaître leurs spécificités ni avoir suivi les années de formations qui en font des experts. Cela est d'autant plus important que chaque méthode formelle possède ses forces et ses faiblesse et qu'à terme, il est évident qu'il faudra en utiliser plusieurs.

Se pose donc le problème de faciliter le travail des ingénieurs vis-à-vis des méthodes formelles. Une première solution est de travailler au niveau de programmes. Certaines études suggèrent d'utiliser le langage de programmation comme base de travail pour vérifier un système (comme C avec SPIN dans **[52]**). Une telle approche, si elle est séduisante, peut poser des problèmes dans les systèmes répartis lorsque les mécanismes de communication entre composants s'appuient sur des protocoles complexes et difficile à apréhender.

Une autre approche est de greffer les méthodes formelles au niveau d'une méthodologie MDA (Model Driven Architecture). Le travail se fait sur un modèle exprimé au moyen d'un langage de haut niveau. Nous avons vu à ce titre que les solutions proposées restent partielles. Soit le langage est un standard de l'industrie (UML) et la vérification se limite en général à l'expression d'invariants, ce qui est insuffisant pour les systèmes répartis, soit les mécanismes offerts sont plus fins et la méthode formelle sous-jacente n'est pas suffisamment cachée.

Offrir un meilleur couplage entre un langage de spécification et les méthodes formelles est l'un des objectifs que nous poursuivons avec **L/P**. Il s'agit d'un travail réalisé en coopération avec la thèse de Yann Thierry-Mieg dans lequel une expérimentation est en cours avec les réseaux de Petri.

Nos travaux ont permis de bien illustrer les lacunes d'une notation comme UML dans le domaine des systèmes répartis :

- Le principal problème que nous nous proposons de résoudre est de définir de manière non ambiguë le comportement d'un système afin de pallier le principal défaut d'UML dans ce domaine : l'éparpillement de la description du comportement dans différents diagrammes dont les relations sont interprétables (et interprétées) de manières diverses.
- Un second problème est d'intégrer l'expression des propriétés dans la spécification, de manière à ce qu'elle évolue avec le modèle. En ce sens, chaque assertion peut être interprétée comme une sorte d'obligation de preuve sur le système. Si un composant est embarqué dans un autre modèle, ces assertions le suivent. Les assertions doivent permettre d'exprimer des invariants, des bornes, des exclusions mutuelles (c'est un cas particulier de borne) ou des formules de logique temporelle.

La manière dont ces points sont couverts par **L*f*P** est détaillée dans le chapitre 5.

3.5 Conclusion et Décisions

Nous avons analysé UML, RM-ODP et des ADL en vue d'identifier un style architectural adapté pour la modélisation d'applications réparties. Nous avons constaté que la description d'architectures logicielles peut se faire à deux niveaux d'abstraction :

- *à un niveau logique* qui décrit l'application au moyen des entités logicielles de conception (e.g. les classes UML). Cela concerne l'*architecture logique*.
- *à un niveau concret* qui modélise l'application au moyen d'entités logicielles de déploiement (composants, connecteurs et configurations ADL). On parle d'une *architecture de déploiement*.

Cette vision est confirmée par RM-ODP qui distingue la partie logique (les vues Information et Traitement) de la mise en œuvre (les vues Ingénierie et Technologie).

UML est le standard de modélisation de facto. Il est adapté pour décrire l'architecture logique selon une approche orientée objets, mais il n'offre pas les capacités de description d'ADL, même si, par extension, on peut lui greffer des telles capacités (par exemple, avec UML-RT et ROOM). Ainsi, pour garder la compatibilité avec cette notation il faut :

- *s'appuyer sur une architecture logique orientée objets compatible UML,*
- *s'appuyer sur une démarche de modélisation complémentaire à UML*, c'est-à-dire qui ne chevauche pas les capacités de modélisation de ce langage, et *qui permette de faire le passage vers une architecture concrète d'implémentation*.

Notre analyse nous mène à deux conclusions :

- Nous pensons qu'un ADL ne doit pas se contenter de modéliser seulement la structure (les ports) et le comportement visible (le contrat) d'un composant mais qu'il doit permettre de raffiner cette description y compris au niveau de son implémentation interne.
- Le style architectural utilisé pour la description doit se baser au minimum sur un modèle de conception par tâches communicantes, mais, il est préférable de pouvoir intégrer dans la description les notions d'objet, d'opération et de variable.

3.5.1 Style architectural à utiliser pour la description d'applications réparties

L'étude des ADL et de RM-ODP nous à permis d'identifier les constructions d'une notation ADL adaptées à notre domaine. Plusieurs concepts ont été identifiés :

- *Classes d'implementation*. Le concept de composant logiciel à une connotation liée au déploiement. Au contraire, une classe d'implémentation est une notion logique. Elle exprime une entité logicielle instanciable (on parle d'objets). Par exemple, les gabarits de conception RM-ORP, ainsi que les classes d'implémentation UML incarnent ce concept. La notion de classes d'implantation est donc fort adéquate pour être utilisée comme brique de base dans la description de l'architecture logique d'une application.
- *Ports*. UML propose la notion d'interface pour exprimer un ensemble de services fortement couples du point de vue de leur utilisation. Au contraire, les ADL utilisent le concept de points d'interaction (ports) pour exprimer des points d'accès à un même mécanisme d'interaction (e.g. connecteur ADL). Puisque la notion de Port n'est pas modélisée dans UML elle doit être intégrée dans une notation ADL.
- *Connecteurs* (*Binders* et *Media*). Un connecteur exprime un mécanisme d'interaction permettant à deux composants de communiquer. Cette notion est mal prise en compte en UML ou ses extensions ROOM et UML-RT et par conséquent elle doit être intégrée.
  Les ADL étudiés modélisent les connecteurs d'une façon différente. On distingue deux approches conductrices :
  - 1) *utiliser un ensemble fixe de connecteurs normalisés*,
  - 2) *modéliser finement leur comportement au moyen d'automates*.

  Si la première approche nous semble bien convenable pour la modélisation de gabarits d'interaction directs (en local) qui s'y prêtent à la normalisation, la seconde approche paraît préférable pour la description des protocoles de communication complexes. Nous considérons donc qu'il faut distinguer les connecteurs par rapport à leur type (pour les interactions directes ou à distance) et par conséquent proposer deux artefacts de modélisation disjoints : les *Binders* et les *Media*.
- *Composants*. Un composant logiciel (voir **Définition 2-6**) est une unité binaire de déploiement et de configuration. A nos yeux, un composant n'est rien d'autre qu'une agrégation de classes d'implémentation connexes à déployer et gérer ensemble. L'approche proposée est similaire à celles de Ada95-DSA, de RM-ODP et d'UML qui proposent le même concept avec des mots différents : partition Ada95, cluster RM-ODP et de composant UML. Le support pour les composants hiérarchiques se résume à un processus de composition par MACROS similaire à ce qui est proposé par Meta-H.

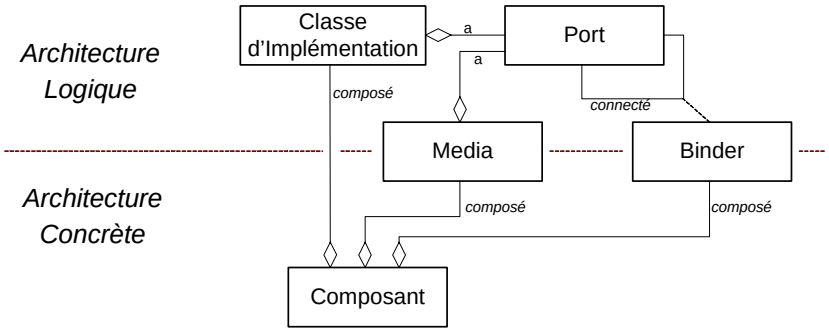


Figure 3-7 : Notre vision sur le style architectural à adopter pour la description d'applications réparties.

La **Figure 3-7** ilustre notre vision d'un style architectural à utiliser pour la description d'applications réparties. Il s'agit d'un méta modèle UML-MOF qui raffine celui qui avait été présenté dans **Figure 3-1**.

Nous faisons la liaison entre l'architecture logique et celle de déploiement. Cela constitue un point original de notre travail car aucun des ADL étudiées ne vise l'intégration entre ces deux aspects (architecture logique et architecture concrète) d'une façon orthodoxe. En effet, les **`Classes d'implémentation`**, les **`Media`**, les **`Binders`** et les **`Composants`** correspondent à des **`entités`**

ADL. Les **Media** et les **Binders** sont des cas concrets de **Connecteurs** ADL. La notion de **Binder** concrétise également la relation de **connexion** de ports. La relation de **composition** est dépliée, un composant logiciel est **composé** d'un ensemble connexe de classes d'implémentation, binders et media. Puisque les binders et les media sont présentées une seule fois sur le méta modèle, un connecteur logique est déployé dans un seul connecteur concret. Cela n'était pas possible pour les **composants** qui constituent des entités logicielles composées.

3.5.2 Nos objectifs de modélisation

Notre analyse approfondie d'UML et des ADL nous a permis aussi de déceler plusieurs insuffisances ou incohérences qui sont des obstacles pour la vérification du bon comportement des applications réparties. Elle nous amène à proposer des améliorations ou des extensions concernant l'ensemble des concepts mis en oeuvre par ces langages.

Classes d'implémentation.

Nous souhaitons décrire les classes d'implémentation aussi bien par rapport à leur aspect visible (ports d'interaction, signatures de services offerts et contrat d'utilisation) que par rapport à leur réalisation (variables membres et comportements des méthodes). Cette description doit être complète, elle nécessite de statuer sur :

- Le type, la valeur par défaut, la visibilité (privé ou publique), le caractère statique de la déclaration (persistance) ainsi que le mode d'accès (e.g. en lecture seule) des variables.
- La définition et l'interprétation de la signature pour les services offerts ou requis. Pour éviter le problème d'interdépendance caché dans la conception illustrée dans la **Figure 3-5**, nous avons décidé qu'il faut séparer la signature des services en deux parties : entête pour discriminer l'information de routage et corps pour encapsuler les donnés. Cependant, pour assurer un déploiement facile et affaiblir ce problème de dépendance caché il faut des environnements d'exécution répartis.
- Le comportement interne des méthodes.
- Les types de comportements de classes. On souhaite pouvoir décrire non seulement des comportements actifs (comme en ROOM, Wright ou Meta-H) mais également des comportements réactifs protégés, c'est à dire gardés ou des comportements passifs correspondant à du code de bibliothèque.

Media.

La conception des Media est similaire à celle d'une version allégée de classes d'implantation. Les remarques énumérées auparavant restent donc valides. Vu que notre objectif est la modélisation des protocoles de communication, nous ne nous intéressons plus à la définition de services mais seulement au contrat comportemental. Ce dernier doit supporter la définition de comportements concurrents.

Ports.

Nous avons identifié trois objectifs de modélisation:

- *Le typage statique des ports nous paraît néfaste car il induit une interdépendance entre composants et connecteurs. Il est contraire au principe de «couplage faible dans la conception» et le non respect de ce principe empêche de considérer le déploiement d'applications comme un processus de composition et de configuration ultérieures indépendamment de la conception.* La structure des ports ne doit donc pas être typée explicitement. Cela permet à la manière de C2-SADL de reduire le problème d'interdépendance caché dans la conception (voir **Figure 3-5**).
- La définition de contrats comportementaux pour les ports n'est pas nécessaire. Le comportement

d'une classe visible au travers d'un port constitue une vision partielle du contrat comportemental de la classe. Nous argumentons notre choix par un souci de sur-spécification. Remarquons d'ailleurs que UML permet de modéliser des scénarios d'interaction acceptables ou interdites au moyen des diagrammes de collaboration.

- Il faut préciser la sémantique d'interconnexion. L'assemblage d'entités logicielles (classes, binders, media, composants) dans un modèle global (l'application) est impossible si la composition de modèles partiels n'est pas formalisée par des règles de composition. Ce problème est bien connu dans les méthodes formelles comme par exemple les Réseaux de Petri pour lesquels des mécanismes de fusion de places ou de transitions sont proposés dans la littérature.

Binders.

Nous souhaitons synthétiser un noyau canonique de modes d'interactions locales. L'étude des ADL et RM-ODP nous a permis de relever plusieurs caractéristiques à étudier dans la construction de ce noyau. Par exemple, le paradigme d'interaction (e.g. par messages), l'ordonnancement d'accès concurrents à un binder (e.g. par FIFO), le sens de l'interaction, la taille des tampons de communication, l'initiative de la communication (*push* ou *pull*) sont des facteurs à prendre en compte. L'analyse des environnements d'exécution s'impose. Nous avons travaillé sur la définition de modèles réalistes pour lesquels la mise en œuvre sera facilitée.

CHAPITRE 4

Abstraction de l'environnement d'exécution pour la modélisation

4.1 Introduction

Ce chapitre étudie les problématiques concernant l'implémentation de modèles. Nous cherchons à faciliter la portabilité de solutions vers un ensemble varié d'exécutifs répartis afin de s'abstraire le plus possible des choix d'implémentation lors de la modélisation.

La **Section 4.2** justifie notre approche qui consiste dans l'élaboration d'un «*modèle canonique d'environnement d'exécution*» à utiliser pendant la modélisation. Ce besoin est suscité par ce qu'on appelle le «*paradoxe du développement des applications réparties*» **[95]** qui se manifeste par un manque perpétuel d'interopérabilité et de portabilité des applications et cela malgré les efforts consacrés à la conception des exécutifs répartis (e.g. intergiciels).

La **Section 4.3** étudie les directions à suivre pour la construction d'un tel modèle. Nous élaborons *un modèle de communication canonique* et *un ensemble de fonctionnalités de base* nécessaires pour la mise en œuvre d'applications réparties. Nous identifions également les choix de génie logiciel qui sont des obstacles à l'implémentation des modèles au dessus d'un environnement d'exécution cible (e.g. l'incompatibilité entre l'architecture logicielle de conception et celle de déploiement).

La **Section 4.4** présente les principales approches dont nous nous sommes inspirés. La diversité des approches ainsi que leur popularité ont été les deux critères de sélection. En partant des systèmes d'exploitation classiques nous avons avons centré notre analyse sur les intergiciels, qu'il s'agit des intergiciels de bas niveau (e.g. pour la communication par messages) ou de haut niveau (e.g. pour la com-

munication par RPC). Nous avons poussé notre analyse jusqu'aux incidences des langages de programmation adaptés à la répartition.

La **Section 4.5** présente nos conclusions. Elles portent sur deux aspects :

- les paramètres à considérer pour la spécification d'un modèle de communication canonique,
- les fonctionnalités de base (services) nécessaires pour la mise en œuvre d'applications réparties.

Ces conclusions ont été notre base pour la conception de *binders* et des *media* de notre langage **L*f*P** et surtout pour la spécification du runtime **L*f*P**.

4.2 Bases pour un modèle canonique d'environnement d'exécution

Malgré les avancées technologiques dans chacun des domaines des langages, des systèmes d'exploitation, des protocoles de communication et des intergiciels, l'industrie du génie logiciel connaît une crise chronique **[43]**. Cette crise est causée par l'hétérogénéité croissante des technologies d'implémentation qui empêche l'interopérabilité, la réutilisation et la portabilité des applications.

Le paradoxe du développement des applications réparties

Dans le cas des applications réparties, Pautet **[95]** assimile ce problème à un «*paradoxe de l'intergiciel*». Il constate que les intergiciels (ceux qui étaient censés être la solution technique permettant d'assurer la portabilité et interopérabilité d'applications réparties) peuvent constituer une nouvelle source d'hétérogénéité et d'incompatibilité. L'auteur exhibe l'apparition d'un nouvel enjeu : l'*interopérabilité des intergiciels*. Cet enjeu peut être extrapolé au cas général d'environnements d'exécution répartis.

Modèle d'intergiciel pivot

Les intergiciels génériques **[85, 102, 117]** se démarquent comme une solution efficace pour assurer l'interopérabilité. L'idée proposée est d'utiliser un «*modèle d'intergiciel pivot*» qui peut être paramétré pour donner lieu à des personnalités précises (e.g. CORBA Ada-DSA, Java-RMI etc.). La synthèse d'un tel modèle pivot est simplifiée pour les intergiciels appartenant à une même famille (e.g. RPC), c'est à dire ceux qui respectent un style architectural commun. Dans ce cas, les différences existantes entre les intergiciels sont plutôt d'habillage syntaxique (e.g. format d'échange des données, protocole de communication). Le paramétrage revient alors à personnaliser l'implémentation.

La faisabilité de l'approche est confirmée par des solutions opérationnelles. Parmi elles, nous rappelons celle de Quinot **[102]** qui propose PolyORB un intergiciel à comportement schizophrène, c'est-à-dire qui offre plusieurs personnalités simultanées. Comparée à des approches antérieures comme Jonathan **[85]** ou Quarterware **[117]** (des intergiciels génériques configurables), l'avantage de PolyORB vient justement de la façon dont il conçoit le modèle d'intergiciel pivot. En effet, celui-ci est basé sur la "neutralité" de son modèle de répartition intermédiaire, qui permet d'assurer l'interopérabilité des personnalités concrètes obtenues par personnalisation.

La notion de «*modèle d'intergiciel pivot*» constitue donc un gabarit de conception de référence abstrait PIM (Platform Independent Model en MDA **[90]**). Au contraire, les personnalités concrètes obtenues à partir de ce modèle pivot constituent des PSM (Platform Specific Models en MDA) qui préservent le style architectural ainsi que la sémantique du modèle pivot.

Canonisation du développement d'environnements d'exécution

Une idée complémentaire de celle de «*modèle d'intergiciel pivot*» est celle de «*canonisation*» de la conception d'intergiciels. Dans ce sens, Quarterware **[117]** propose une approche de conception de type «RISC», la construction d'un intergiciel personnalisé est réalisée par un processus de composition et paramétrage de microcomposants logiciels de base (canoniques).

Nous considérons qu'une telle démarche de conception offre plusieurs avantages :

- *conception simplifiée, évolutive.* Dans son livre dédié aux systèmes d'exploitation répartis, Tanenbaum **[130]** argumente son attachement aux approches de type micro-noyaux :
  - «*le modèle optimal (de système d'exploitation) n'est pas obtenu quand il ne reste rien à rajouter mais quand tout ce qui n'était pas strictement indispensable a été retiré du modèle*».

  Nous considérons donc qu'il est préférable de proposer un modèle d'environnement à base de micro-composants simple mais flexible plutôt qu'une approche compliquée, plus difficile à maîtriser.

  La synthèse des nouveaux mécanismes système, de plus haut niveau, peut se faire par composition modulaire. Comme dans le cas des micro-noyaux **[100]**, cette décomposition des mécanismes système facilite une conception évolutive.
- *adéquation pour la génération automatique d'implémentations conformes*. L'approche de construction par la composition de modules canoniques est avantageuse car elle se prête bien à la synthèse automatique de code. En effet, la génération d'un environnement d'exécution personnalisé se réduit à deux tâches simples :
  - implémenter chaque type de micro-composant en nombre et de complexité réduites,
  - implémenter le code d'assemblage.

  Cette démarche renforce la traçabilité de la conception car il existe une correspondance forte entre la structure des modèles et celles des programmes générés.
- *formalisation plus facile.* Dans le cas d'une approche de conception par composition de modules canoniques, la formalisation de l'environnement d'exécution est simplifiée car elle se réduit à deux tâches simples :
  - formaliser l'ensemble (basique) de micro-composants,
  - formaliser les règles de composition.
- *complétude de la vérification*. Comme expliqué en **Section 3.4**, la vérification formelle de modèles et la certification de leurs implémentations nécessitent de s'appuyer sur des modèles formels complets. De tels modèles décrivent l'intégration de l'application dans son environnement d'exécution.
- *meilleure performance d'exécution.* Contrairement aux systèmes d'exploitation de type micro-noyaux qui, malgré leur modularité, sont considérés comme étant moins performants que leurs homologues monolithiques, dans le cas des micro-modèles d'intergiciels paramétrables **[109, 110]** leur taille réduite, strictement adaptée aux besoins de l'application, permet d'optimiser les implémentations et par conséquent d'accroître les performances d'exécution. Des exemples de micro-intergiciels comme UbiCore **[109]** ou des intergiciels génériques paramétrables comme Quarterware **[117]** confirment brillament cette qualité de performance.

Bilan : Modéle canonique d'environnement d'exécution

Les intergiciels schizophrènes constituent une solution technique à l'interopérabilité des intergiciels et par conséquent à celle des applications réparties **[102]**. Cette interopérabilité est réalisée grâce à l'utilisation d'un «*modèle d'intergiciel pivot*» neutre par rapport au modèle de répartition.

C'est pourquoi l'objectif central de notre thèse est d'élaborer un modèle d'applications portables et non pas seulement d'assurer l'interopérabilité de leurs composants. Nous avons voulu intégrer l'idée de «*modèle neutre d'exécutif pivot*» dès la phase de conception. La neutralité de ce modèle nous permet d'assurer l'indépendance de la conception face aux choix d'implémentation.

Nous avons constaté que la conception d'intergiciels et celle des environnements d'exécution peut être améliorée si la démarche de développement est basée sur une approche de conception de type micro-noyau d'environnent abstrait constitué d'un ensemble de composants de base canoniques.

Nous considérons fondamental d'intégrer ces trois idées (modèle d'environnement pivot, canonisation de sa définition, complétude de vérification intégrant l'environnement) dans notre approche de modélisation. C'est pourquoi, nous adoptons le principe suivant :

> L'exploitation d'un *modèle formel canonique d'environnement d'exécution*, générique face aux choix d'implémentation, comme base dès la phase de conception, est la clé de voûte de la construction d'applications réparties évolutives dites sûres.

La définition suivante nous sert à qualifier les applications dites sûres qui respectent le principe énoncé ci-dessous.

> **Définition 4-1:** *Une Application est dite sûre* si elle :
> - *est construite sur la base d'un modèle formel,*
> - *est vérifiée de manière complète* (dans le cadre de son environnement) et,
> - *a une implémentation conforme avec son modèle*, qui préserve la structure et la sémantique des modèles .

4.3 Méthode d'analyse des environnements

La synthèse d'un modèle canonique d'environnement d'exécution nécessite d'analyser finement l'architecture des plates-formes d'exécution réparties. Pour cela, nous proposons deux aspects, que nous avons raffinés en une grille d'analyse.

Modèle d'interaction et de communication

Le style architectural proposé en **Section 2.3.4** fait référence à plusieurs types d'entités logicielles. Les *binders* et les *media* correspondent à des connecteurs ADL dédiés pour les interactions en local ou celles à distance. Nous constatons que si la description de la partie applicative (classes d'implantation et composants) est à la charge des concepteurs, la sémantique des communications est souvent imposée par l'environnement d'exécution.

C'est pourquoi nous nous sommes imposé de *virtualiser*, *canoniser* et *formaliser* la description des connecteurs. Pour cela nous avons identifié un noyau de gabarits d'interaction génériques et paramétrables pour les *binders* et les *media*. Puis nous avons confronté les environnements d'exécution répartis aux modèles d'interaction offerts en évaluant le coté standard et l'aspect canonique de chacun des mécanismes d'interaction identifiés en vue de leur *unification*.

Services

La gestion d'une application répartie nécessite la mise en œuvre de services système (e.g. référentiel, activation, ordonnancement, sécurité etc). Il y a une grande diversité dans le nombre et la sémantique de services offerts par les environnements d'exécution. Symétriquement, le besoin de services particuliers dépend du type d'application. Par exemple, le service d'activation n'est pas requis si le déploiement d'une application est statique. Nous avons identifié les principaux services absolument nécessaires pour la mise en œuvre d'applications réparties.

4.4 Les communications et les référentiels, points clés des intergiciels

Notre analyse porte sur un échantillon représentatif d'environnements d'exécution. Le choix des candidats étudiés est basé sur leur diversité de style architectural et leur renommée. Nous avons analysé trois catégories de travaux :

- *Les systèmes d'exploitation concurrents classiques* (non répartis) proposent un ensemble de mécanismes de coordination **[17, 108, 124]** susceptibles d'être normalisés et intégrés dans la conception des binders (les connecteurs pour la communication en local). De plus, ils permettent d'évaluer simplement dans un cadre local l'influence de certains choix d'implémentation (e.g. la politique d'ordonnancement système) sur la sémantique des interactions.
- *Les intergiciels* ont suscité une double analyse. D'une part, nous avons étudié et comparé les modèles de communications supportées par ceux-ci (messages **[119, 125]**, invocation transparente de services **[57, 87, 123]** ou ressources réparties partagées **[4]**) et surtout la sémantique de ces modèles. D'une autre part, nous avons recensé les services auxiliaires offerts pour la mise en oeuvre d'applications et identifié les fonctionnalités basiques (e.g. gestion des ressources, référentiel etc.) qui doivent être intégrer dans le runtime d'une application. Cette analyse a été déterminante pour la conception de notre runtime **L*f*P**.
- *Les langages de programmation concurrents adaptés pour la programmation répartie* (e.g. Orca **[11]** et Ada95 **[60]**) ont une incidence cruciale sur la répartition. Notre objectif a été d'étudier en quoi la répartition change la sémantique d'invocation et de synchronisation et en quelle mesure la gestion d'applications réparties est directement intégrée dans ces langages.

4.4.1 Systèmes d'exploitation classiques

Dans un système concurrent monolithique, il existe deux types d'interactions : *intra* et *inter-processus*. Si l'initiative d'une interaction appartient toujours à une *unité active de code* (processus ou thread) qui possède son propre fil et contexte d'exécution, le destinataire peut être également une *unité de code passive* (e.g. une fonction dans une bibliothèque) ou une *ressource passive* (e.g. une variable partagée).

Interactions intra-processus

Le changement de contexte d'exécution entre les processus dans les systèmes concurrents classiques est un problème de performance face auquel les processus légers ou threads sont une solution. Malheureusement, malgré leurs bonnes performances, la programmation d'applications concurrentes multithredées est soumise à des problèmes de communication, de synchronisation et de gestion des ressources qui sont plus difficiles à maîtriser que dans le cas des application multiprocessus.

Face à la diversité des solutions proposées, la norme POSIX-threads (*P-threads*)**[17]** propose une interface standardisée. C'est donc elle que nous avons principalement analysée.

Mécanismes de synchronisation et communication

La communication entre les threads hébergés par le même processus est privilégiée par le partage du même espace d'adresses logiques. Pour communiquer il suffit d'utiliser les variables a priori partagées par toutes les threads (au sien d'un même processus) et de synchroniser l'accès en utilisant des mécanisme de synchronisation tel que les : *mutex*, *sémaphores*, *condvars* (variables de condition) ou *signaux* **[17]**.

Si la norme est claire en ce qui concerne le fonctionnement de ces mécanismes, elle ne précise pas la façon dont les threads en attente pour la même ressource (e.g. exclusion mutuelle) seront élus si celle-ci devient à nouveau disponible. La politique d'ordonnancement varie d'une réalisation, d'une version ou d'une exécution à une autre (e.g. FIFO, par priorité etc..) ce qui change radicalement le comportement de l'application.

Parce que nous ciblons la vérification formelle des comportements, ce problème est fondamental pour nous. Pour ne pas nuire à la portabilité de la conception, nous considérons qu'il faut au minimum mettre en place une politique d'ordonnancement par défaut non déterministe qui corresponde à l'ensemble de comportements possibles. Il nous paraît également important de permettre aux utilisateurs de raffiner cette politique par rapport à une propriété (e.g. ordonnancement équitable) afin de réduire l'ensemble de comportements à un sous ensemble.

L'utilisation des sémaphores et des «condvars» (variables de condition) comme mécanismes de synchronisation standardisés nous semble utile à intégrer dans un modèle canonique d'environnement d'exécution. Au contraire, vu l'hétérogénéité de leurs implémentations, nous considérons qu'il faut éviter d'intégrer dans ce modèle les signaux.

L'emploi des variables partagées comme moyen de communication est un autre choix que nous souhaitons intégrer dans notre modèle sous réserve de préciser à la modélisation si la sémantique d'accès est atomique ou non.

Gestion de ressources

Tout processus POSIX doit comporter un thread principal. La gestion de ressources internes à un tel thread (e.g. allocation des variables) ainsi que celle du cycle de vie des threads secondaires ne peuvent pas être réalisées directement de l'extérieur du processus d'hébergement.

Cela souligne la nécessité d'intégrer dans notre modèle d'exécutif des gestionnaires pour les objets actifs (threads) partageant la même capsule de déploiement (processus). Cette remarque va dans le sens de RM-ODP qui prévoit l'existence des gestionnaires dédiés à cette tâche.

Interactions inter processus

Si la communication entre threads composant le même processus peut se faire directement, l'échange de données entre deux processus qui, à priori ne partagent pas le même espace d'adresses logiques, nécessite d'autres mécanismes tels que les IPC (*Inter Process Communications*) **[108, 124]**.

Il existe des nombreuses variantes de communications inter-processus. Parmi les contributions majeures venant du monde UNIX (System V et BSD) nous citons :

- *Tubes* (*PIPEs*) et *tubes nommés (FIFOs)*. Ceux-ci sont des tampons des données de taille fixe adaptés pour la communication sans perte par flots de caractères. Comparés à leurs homonymes, les tubes nommés ont l'avantage de pouvoir être utilisés entre des processus concurrents et pas seulement entre ceux qui sont en relation de parenté.
- *Files de messages* (*Message Queues*). Plus complexes que les tubes, les files de messages permettent la communication par messages (de taille limitée). L'envoi et la réception des messages se fait

d'une manière asynchrone. Il est possible de proposer des politiques d'ordonnancement et de multiplexage des messages par rapport à leur type qui est utilisé comme un discriminant. Un exemple classique de files de messages sont les FIFO prioritaires UNIX **[124]**.

- *Sémaphores*. Les sémaphores sont des mécanismes de synchronisation intra et inter processus servant à définir des sections de code critique pour protéger l'accès aux ressources partagées, ou pour contrôler le parallélisme et les communications.
- *Segments de mémoire partagée*. Contrairement aux mécanismes précédents (les tubes et files de messages) pour lesquels la capacité est fixée par le système, les segments de mémoire partagée sont plus flexibles car leur taille peut être contrôlée par le programme.

  Nous constatons que les segments de mémoire partagé constituent un moyen flexible pour la synthèse des mécanismes passifs de communication ayant une complexité supérieure à celle des tubes et des files de messages. En effet, ce mécanisme est associable à des sémaphores ou à des variables de condition (condvars), pour synchroniser des accès d'une manière particulière spécifiée par programme.
- *Signaux*. Principalement utilisés pour la signalisation d'événements (e.g. interruptions) système, les signaux (synchrones ou asynchrones) sont des mécanismes de synchronisation puissants. Cependant, nous avons constaté que leurs sémantiques d'implémentation varient beaucoup d'un système à un autre, voire d'une version à une autre, ce qui empêche de les considérer comme des mécanismes normalisables.

  De plus, les signaux sont évalués et traités à des moments non déterministes de l'exécution (e.g. lors de l'élection d'un processus ou thread) qui varient d'une exécution à une autre, ce qui empêche de garantir la répétitivité des comportements et, par conséquent, complique la vérification des comportements d'une application.

  Cette diversité comportementale nous empêche de considérer les signaux comme des mécanismes de synchronisation universels et par conséquence nous écartons leur intégration.
- *Ports de communication.* Utilisés comme des mécanismes de communication inter-processus unifiés, les ports de communication (e.g. les sockets BSD **[124]**, les ports Match **[130]** etc.) sont également plus complexes mais fondamentaux pour la communication en réparti. Leur objectif est d'uniformiser l'interface d'accès des services de communication et de fournir une sémantique de communication homogène en dépit de l'hétérogénéité des protocoles de communication effectivement utilisés.

  Comparés aux mécanismes précédents, les ports se démarquent par l'introduction d'un schéma d'adressage explicite qui permet la construction de topologies de communication complexes et le routage dynamique d'informations.

Sémantique des modèles de communication

L'analyse des IPC nous a permis d'identifier plusieurs aspects concernant leur description :

- **paradigmes de communication**. L'échange d'informations peut prendre diverses formes : a) *flux d'octets* (tubes), b) *messages de taille bornée* (files de messages), c) *données partagées* (segments de mémoire partagée), d) *drapeaux de signalisation* (signaux).

  Nous munissons donc notre modèle d'attributs caractérisant ces mécanismes et leur utilisation afin de pouvoir les synthétiser et les adapter lors du protoptypage, du déploiement et même de l'exécution.
- **modèle de communication**. Tous les IPC analysés utilisent un paradigme de communication de type *producteurs-consommateurs*. Les différences constatées concernent d'une part la possibilité de réceptionner l'information *avec* ou *sans consommation* et d'autre part l'obligation ou non de se

bloquer. Par exemple, les tubes, les files de messages ou les signaux sont des mécanismes pour lesquels la lecture est équivalente à la consommation d'informations. Au contraire, les ports de communication permettent aussi de ne pas consommer les informations reçues (e.g. messages) lors de la lecture.

Les segments de mémoire partagée se distinguent, à nouveau, par une flexibilité de modélisation étendue. En effet, vu que la gestion des données partagées se fait explicitement par programme il est possible d'implémenter à la fois ces deux modèles (avec ou sans consommation). On parle de modèle de communication de type lecture-écriture.

- **initiative de la communication**. A l'exception des signaux qui sont des mécanismes de communication *actifs* car c'est le *système* qui interrompt l'exécution d'un processus cible pour lui indiquer de traiter un signal non masqué, tous les autres IPC étudiées ont des comportements *passifs* : l'interaction avec le mécanisme IPC est réalisée à l'initiative des *processus* émetteur (pour envoyer) et récepteur (pour réceptionner).
- **sémantique de communication**. Pour l'intégralité de mécanismes étudiés la sémantique de communication de bout en bout est *asynchrone*, l'émission et la réception sont découplées. Pour synchroniser les communications il faut soit faire appel à des mécanismes de synchronisation système (e.g. sémaphores), soit utiliser des canaux de communication en retour (e.g. pour signaler la réception). Dans tous les cas, la synchronisation se fait explicitement par programme.
- **contraintes de capacité.** Chaque type de communication asynchrone a ses limites car il n'existe pas de tampons de taille infinie. Ce qui diffère d'un mécanisme à un autre est la manière dont cette limite est précisée. Les tubes et les files de messages ont des *capacités fixées*, celle ci dépend du système d'exploitation. Au contraire, les segments de mémoire partagée constituent des mécanismes plus flexibles vu que leur taille est *variable*. La taille d'un block mémoire peut être choisie *à l'exécution* en fonction des besoins.
- **stratégie d'appel**. Pour éviter le débordement en émission ou la famine en réception il faut prévoir des stratégies d'émission et de réception. Deux choix existent : a) les *appels bloquants*, b) les *appels non bloquants*. Dans ce deuxième cas il faut également préciser un mécanisme d'exception à entreprendre au cas où une communication n'aboutisse pas normalement. Par exemple, le débordement ou la sur-écriture (circulaire ou par fenêtre coulissante) des tampons sont des stratégies d'émission populaires.

  La stratégie de réception peut se faire soit d'une façon partielle en ne retournant que les données disponibles (e.g. pour les tubes), soit sans consommer les informations incomplètes (e.g. pour les files de messages).
- **fiabilité des communications.** A l'exception des signaux qui sont considérés comme *non fiables* dans le sens que les arrivées successives d'un même signal ne sont pas sauvegardées, les communications locales par tubes, par files de messages et par segments de mémoire partagée sont fiables, sans perte d'informations. Quant aux ports de communication, ils proposent plusieurs qualités de service (avec ou sans perte) qui sont répertoriées dans des classes de protocoles fiables ou non.
- **entrelacement** (atomicité d'accès). L'envoi d'informations provenant des plusieurs sources concurrentes ainsi que l'accès en lecture d'un même mécanisme par plusieurs consommateurs peut se faire d'une manière *atomique* (pour les files de messages) ou avec *entrelacement* (pour les tubes).
- **séquencement et multiplexage des informations**. Le séquencement de type FIFO est sans doute la politique d'ordonnancement de communications la plus connue. Si dans le cas des tubes cette politique est imposée, pour les files de messages elle peut être personnalisé car le «*type*» d'un message peut être utilisé comme discriminant pour multiplexer les messages à la reception. L'emploi des segments de mémoire partagée associée à celui des sémaphores est un moyen encore plus

flexible. Il permet de personnaliser par programme la conception des mécanismes de communication utilisateur.

- **routage.** Le routage d'informations peut se faire soit d'une manière *implicite*, directement fixé par la topologie de connexion, soit d'une façon *explicite*, c'est-à-dire par programme. Le routage explicite est possible grâce à un schéma d'adressage spécifique. Cela permet d'identifier d'une façon non ambiguë les paires intervenant dans la communication et par conséquent de discriminer la réception d'informations en fonction du tuple <*port de réception*, *adresse source*, *adresse cible*>.

  Les tubes et les files de messages correspondent à la première catégorie. Au contraire, les signaux et surtout les ports permettent le routage explicite. Pour les segments de mémoire partagée il est possible d'implémenter (par programme) des schémas d'adressage utilisateur d'une complexité raisonnable.
- **mode de connexion**. Deux modes de connexion se distinguent : *connecté* et *non-connecté*. Les tubes font partie de la première catégorie (pour émettre il faut qu'un récepteur soit présent). Au contraire, les tubes nommés, les files de messages, les signaux ou les segments de mémoire de partagée utilisent un mode de communication non connectée. Les ports de communication, quant à eux, supportent à la fois le mode connecté ou celui non connecté en fonction du type de protocole utilisé.

Type IPC **Caractéristique**	***Tubes*** *(nommées)*	***Filles de messages***	***Segments de mémoire partagée***	***Signaux***	***Ports de communication*** *(sockets)*
Paradigme	**flots de caractères**	**messages**	**données partagées** (mots de taille fixe, indexés)	**drapeaux de signalisation**	**flots de caractères, messages**
Modèle de communication	producteurs- consommateurs **réception avec consommation**	producteurs- consommateurs **réception avec consommation**	lecture et écriture des données partagées (à préciser **par programme)**	producteurs- consommateurs **réception avec consommation**	producteurs- consommateurs **réception *avec*** ou ***sans consommation***
Initiative	**processus** (communications coopératives)	**processus** (communications coopératives)	**processus** (communications coopératives)	**processus émetteur** et/ou **système**	**processus** (communications coopératives)
Sémantique	**asynchrone**	**asynchrone**	à préciser **par programme**	**asynchrone**	**asynchrone**
Contraintes de capacité	**fixées par le système**	**fixées par le système**	**capacité variable** (à précisé par programme)	**sans objet** (1 drapeau / type de signal / processus)	**oui / non** (dépend du type de port)
Stratégie d'appel.	**bloquante** et **non bloquante**	**bloquante** et **non bloquante**	non bloquante (à préciser **par programme)**	**non bloquante**	**bloquante et, non bloquante**
Fiabilité	**fiable**	**fiable**	**fiable**	non **fiable** (surécriture)	**fiable** / non **fiable** (dépend du type de port)
Entrelacement	**entrelacée** (dépend de l'ordonnanceur)	**atomique** par message	à préciser **par programme**	**sans objet**	**entrelacée / atomique** (dépend du type de port)
Séquencement et multiplexage	**FIFO sans multiplexage**	**FIFO** avec **multiplexage par** type de message	**sans séquencement** et **accès aléatoire par programme**	**sans objet**	**FIFO** avec **multiplexage par type d'information** et **urgence** (dépend du type de port)
Schéma d'adressage et routage	**sans schéma** (routage implicite)	**sans schéma** (routage implicite)	à préciser **par programme**	**par destinataire**	**routage dynamique par rapport au** <port de réception, source, destinataire>
Mode de connexion	**connecté / non connecté** (tubes / tubes nommés)	**non connecté**	**non connecté**	**non connecté**	**connectée / non-connectée** (dépend du type de port)

Tableau 4-1 : Mécanismes de communication interprocessus.

Le **Tableau 4-1** résume notre analyse sur les différents mécanismes de communication interprocessus.

4.4.2 Intergiciels

Les intergiciels sont souvent considérés comme des protocoles de communication de haut niveau (*Session*, *Présentation*, *Application* selon le modèle de référence ISO-OSI) qui ont pour but de cacher l'hétérogénéité des technologies d'implémentation sous-jacentes (e.g. langages de programmation systè-

mes d'exploitation et protocoles réseau) et d'assurer la transparence partielle ou totale face à la répartition.

Vu la diversité des solutions existantes, notre analyse est organisée en fonction de leur stratégie de communication. Nous avons identifié les catégories suivantes :

- Les *intergiciels de type MOM* (*Message Oriented Middleware*) comme MPI **[119]** et JMS **[125]** sont spécialisés *pour la communication explicite par messages*. Ils constituent des solutions de bas niveau qui n'offrent qu'une transparence partielle de l'application face à la répartition vu que le code de traitement est mélangé avec celui de communication.
- Les *intergiciels de type RPC* (*Remote Procedure Call*) comme DCE-RPC **[132]**, CORBA **[87]** ou Java-RMI **[126]** permettent l'invocation transparente d'opérations à distance. Le code applicatif est séparé de celui de communication qui est embarqué dans une couche logicielle d'adaptation générée automatiquement. Cette séparation permet donc de simplifier la conception et assure un meilleur niveau de transparence de application face à la répartition.
- Les *intergiciels de type MPR* (*Mémoire Partagée Répartie*) tel que la bibliothèque TradeMarks **[4]** permettent de cacher complètement la répartition aux yeux de l'application. Cependant, en dépit de cet avantage leur étude est paradoxalement inutile pour notre analyse vu que nous ne nous intéressons qu'à la synthèse d'un ensemble de types de communications canoniques.

4.4.2.1 Intergiciels orientés vers la communication par messages

Nous étudions deux classes distinctes d'intergiciels :

- Intergiciels de types *Files des Messages* (MQ-*Message Queuing*) et *Lecteurs-Ecrivains* (PS - Publish Subscribe). Pour étudier cette classe d'intergiciels nous avons choisit JMS **[125]**, une API Java unifiée, qui se prête particulièrement bien à la portabilité d'applications.
- Intergiciels basés sur l'échange de message (MP-*Message Passing*) comme PVM **[42]** et MPI **[119]**. L'étude de MPI est particulièrement instructive parce que cette bibliothèque excelle par le nombre et la complexité des modèles de communications supportées.

JMS

La bibliothèque JMS (*Java Message Service*) **[125]** est une interface de programmation Java pour la construction d'applications réparties de type pair-à-pair ou à messagerie centralisé. JMS offre l'avantage d'une *API de synthèse, unifiée* permettant d'accéder d'une façon transparente aux services de communication d'un intergiciel de type Files des messages ou Lecteurs-Ecrivains. Le fait que JMS soit soutenu par des nombreux acteurs industriels (IBM, Bea Systems, Talarian, Oracle, Sun Microsystems, etc.) lui confère un atout supplémentaire.

Selon les objectifs visés par notre analyse, l'étude de JMS nous semble intéressante sur deux aspects : la façon de structurer les messages, et les mécanismes d'interception et de filtrage.

Structure de messages

La structure d'un message JMS est organisée en trois parties :

- *Un en-tête fixe* utilisé par l'intergiciel sous-jacent pour router. Il regroupe de plusieurs champs (e.g. adresse source et cible, identifiant, type, mode de livraison, priorité etc) communs pour la majorité d'intergiciels MOM.
- *Un en-tête pour les propriétés optionnelles* permettant d'étendre, si besoin, la structure des entêtes. Les informations précisées dans ces entêtes peuvent être utilisées à la fois par l'intergiciel sous-jacent pour le routage ainsi que par l'application pour la réalisation de patrons d'interaction complexes (e.g. transactions) ou pour filtrer les messages.

- *Un corps* dédié au transport des données. Ce contenu peut être structuré d'une façon flexible. Plusieurs paradigmes de structuration son proposés (e.g. flots de données typées ou de caractères, messages aux données associatives <*nom*, *valeur*> etc.).

Interception et filtrage des messages

JMS permet la définition de filtres pour les messages par l'application. La syntaxe adoptée correspond à un sous-ensemble du langage SQL **[61]**, ce qui simplifie la notation en terme de lisibilité. La définition d'un filtre est exprimée par une condition de garde. Celle-ci ne peut porter que sur les propriétés précisées dans l'en-tête optionnel et non sur le corps des messages.

L'implémentation des filtres peut être déléguée par l'application au intergiciel fournisseur. Cela permet de simplifier la conception, vu que le code de filtrage n'est plus mélangé avec le code de traitement, et d'améliorer les performances d'exécution.

MPI

MPI **[119]** est une bibliothèque de communication adaptée à la construction d'applications à processus communicants. Le modèle architectural optimisé pour le calcul parallèle fait que MPI est une approche originale quand on la compare aux intergiciels de type files de messages et lecteurs-écrivains.

Le modèle de communication de MPI est proche de celui des IPC. C'est pourquoi nous l'évaluons selon les mêmes critères :

- **paradigme de communication**. Initialement conçu uniquement pour la communication par *messages*, MPI a était étendu pour permettre l'a*ccès direct aux variables partagées* des processus.
- **modèle de communication.** MPI propose deux types de communications. Pour l'échange de messages ce modèle est de type *producteurs-consommateurs*, la réception d'un message implique sa consommation. Pour l'accès direct aux variables partagées le modèle proposé est de type *lecture - écriture*.
- **initiative de la communication.** L'initiative de la communication appartient toujours à l'application. Pour les messages il s'agit d'un modèle *coopératif* : pour communiquer, toutes les parties doivent participer explicitement. En revanche, l'accès à des variables partagées est *asymétrique* : seul le processus initiateur est impliqué dans la communication.
- **sémantique de communication.** MPI supporte à la fois le mode *synchrone* (de but en en bout) et l'envoi *asynchrone* de messages. En plus, un processus émetteur peut synchroniser son exécution par rapport à un moment intermédiaire de la communication précisant si l'information est : 1) passée au runtime, 2) reçue par le destinataire, 3) reçue et son traitement est commencé. Quant à l'accès aux variables partagées, celui-ci est synchrone.
- **contraintes de capacité.** Similaire aux segments de mémoire partagées, la taille des tampons utilisés pour l'échange des messages est fixée par programme.
- **stratégie d'appel**. MPI permet aussi bien d'utiliser des stratégies d'accès *bloquantes* que *non bloquantes*. Notons que dans le cas des appels non bloquants le problème du débordement des tampons évoquée pour les IPC ne se pose plus. En effet, MPI interdit la réutilisation des tampons de messages avant la consommation totale des informations (copie par le runtime ou le processus destinataire).
- **fiabilité des communication**. Les communications MPI sont exclusivement fiables.
- **entrelacement** (atomicité d'accès). MPI garantit l'accès atomique à un tampon de messages ou à des variables partagées.
- **séquencement et multiplexage de messages.** L'ordonnancement de type FIFO n'est garanti que

pour les communications successives entre les mêmes couples émetteur(s)-récepteur(s). Cet ordonnancement ne garantit pas pour autant l'équité d'accès à un communicateur (canal de communication MPI).

Comme ce qui est proposé par les files de messages, MPI permet de multiplexer la réception des messages par rapport à leur type.

- **schéma d'adressage et routage**. Tout comme les ports de communication, MPI permet la construction des topologies de communication dynamiques. Chaque processus dispose de une ou plusieurs adresses (une par communicateur MPI), ce qui permet de discriminer les communications par rapport au tuple <*communicateur*, *processus source*, *processus cible*>.
- **mode de connexion**. En MPI les communications sont exclusivement en mode connecté. Toute initiative de communication doit être précédée par la création d'un communicateur MPI (canal de communication), celui-ci désigne une connexion pour un groupe de processus.

4.4.2.2 Intergiciels pour l'invocation transparente de services à distance

Bien qu'il soit adapté à la construction d'applications réparties à composants logiciels, le modèle de communication par messages à distance présente cependant des inconvénients :

- *Communications explicites*. Le code de communication est mélangé à celui de traitement.
- *Niveau d'abstraction faible*. Bien que l'invocation de fonctions (opérations) puisse se faire au moyen d'un échange de messages (requête - réponse), nous considérons qu'il est plus judicieux d'embarquer le code de communication séparément du code de traitement dans un mécanisme de communication abstrait.

Les intergiciels adaptés à l'invocation transparente d'opérations à distance (procédures ou méthodes) visent à palier ces inconvénients (communications explicites et de trop bas niveau).

L'idée que nous proposons est d'introduire une couche logicielle d'adaptation qui embarque le code de communication. Ainsi, bien que les communications soient toujours par messages, elles sont cachées par l'interposition des mandataires (`Stub` et `Proxy`). Cette approche est illustrée par la **Figure 4-1**

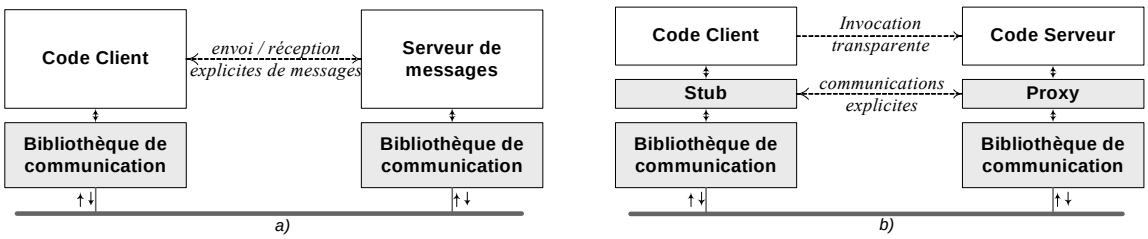


Figure 4-1 : Structure du code d'une application Client-Serveur répartis utilisant : a) un intergiciel MOM, b) un intergiciel adapté pour l'invocation transparente d'opérations à distance.

Plusieurs approches existent :

- *Intergiciels proceduraux* (RPC - *Remote Procedure Call*). Historiquement les premiers, ceux-ci ont perdu leur interét pour des raissons de support industriel, structuration faible et manque d'interopérabilité. DCE-RPC **[132]**, ONC-RPC **[123]** sont des exemples que nous avons étudiés.
- *Bus logiciels à objets* (ORB - Object Request Brokers). Ayant un niveau d'abstraction supérieur, les ORB sont aptes pour la construction d'applications à objets réparties. La norme CORBA **[87]** (*Common Object Request Broker*) prônée par l'OMG constitue le standard de facto. Ceux-ci constituent un pas important vers la conception d'applications interopérables.
- *Intergiciels à composants logiciels* (COM - *Component Oriented Middleware*). Basés souvent sur une infrastructure de type ORB, les intergiciels COM visent à faciliter le déploiement et la gestion

d'applications en réparti. L'intergiciel DCOM **[57]** promu par Microsoft et l'extension CORBA-CCM **[89]** proposée par l'OMG sont deux exemples que nous avons analysés. La plate-forme EJB n'est pas détaillée ici parce qu'elle constitue une solution spécifique Java.

L'étude d'autres protocoles de communication tels que SOAP **[137]** (le successeur de XML-RPC) sont utiles seulement dans la mesure où ils sont spécialement adaptés à la représentation de structures de données complexes.

Modéle de communication

Puisque la différence majeure entre un intergiciel MOM et un intergiciel RPC (ou ORB) se trouve au niveau de la puissance d'abstraction dont l'utilisateur dispose pour la construction d'applications réparties et non au niveau du modèle de communication (toujours réalisé par échange de messages et flux de caractères), nous focalisons notre présentation sur les aspects spécifiques à l'invocation d'opérations à distance sans détailler tous les aspects concernant le modèle de communication spécifique à cette catégorie d'intergiciels.

L'invocation d'opérations à distance a une sémantique légèrement différente de celle en local. Mais c'est important parce cela change le comportement d'une application quand elle est déployée en réparti. Deux aspects entrent en jeu :

- *Sémantique d'invocations et la stratégie d'appel*. Lors d'une invocation à distance, les clients et les serveurs appelés synchronisent leurs exécutions. Il y a plusieurs manières de réaliser cette invocation :
  - *Synchrone*. Un client reste bloqué en attendant le retour d'une invocation (ou éventuellement une exception système telle que l'arrivée à l'échéance d'une horloge). Cette sémantique est la plus proche de celle d'une invocation en local.
  - *Pseudo synchrone*. Un client envoie une requête et continue son exécution. Plus tard, pour récupérer le résultat, le client se bloque dans l'attente de la réponse. Ce mécanisme peut accroître la performance d'exécution côté client.
  - *Synchrone différée*. Un client fait une requête asynchrone et continue son exécution pendant que l'intergiciel prend en charge la communication. En retour, l'intergiciel signale la réception de la réponse au client par une interruption logicielle. Le client suspend son exécution courante, termine le traitement de l'invocation et, ensuite reprend son exécution au point où il s'était arrêté. Ce mode d'exécution améliore considérablement les performances d'exécution côté client si l'exécution de celui-ci n'est pas conditionnée par le retour des résultats.
  - *Asynchrone*. Ce mode correspond à une communication asynchrone par messages. Le client fait une requête asynchrone et continue son exécution. Les appels CORBA **`one-way`** et **`asycnhronous`** de Ada-DSA entrent dans cette catégorie.
- *Fiabilité d'invocations*. Dans le cas de média de communication ayant des taux d'erreur très faibles, pour des raisons de performance de communication, il est souhaitable d'utiliser des protocoles de communication non fiables couplés éventuellement avec la retransmission d'invocations en cas de non réponse dans un délai donné. Dès lors, plusieurs politiques d'invocation existent :
  - *sans garantie*. Aucune garantie n'est donnée sur le nombre de fois qu'une invocation est traitée (zéro, une ou plusieurs fois). Ce type d'invocation peut être appliqué seulement dans le cas d'invocations unidirectionnelles qui n'ont pas besoin d'un retour explicite.
  - *au-moins-une*. Pour un service sans état, une même invocation peut être traitée une ou plusieurs fois sans que cela ait une influence sur le comportement de l'application.
  - *au-plus-une*. Dans le cas des services avec état, il n'est pas acceptable d'exécuter plusieurs fois la même invocation. En revanche, la perte d'une requête peut être parfois tolérée.

- *exactement une*. L'intersection des deux contraintes précédentes implique un comportement déterministe bijectif. Dès lors, chaque invocation est censée être exécutée une et une seule fois.

Services et fonctionnalités de base

Pour remplir leur missions, les services d'un intergiciel font appel à des fonctionnalités primaires. Prenons l'exemple CORBA : Le « service de noms », le service de « pages jaunes », ou celui de « relations » sont étroitement corrélés car ils font tous appel à une même fonctionnalité : *le référentiel commun*. Celui-ci joue le rôle d'une base de données qui permet d'enregistrer, effacer ou rechercher des références identifiant les ressources d'une application (e.g. d'objets). Cette observation est valable également pour d'autres services.

Nous avons également constaté que la spécification des services se résume trop souvent à leurs interfaces. C'est ainsi le cas de CORBA qui ne précise pas formellement la sémantique des services. Cette critique est d'ailleurs confirmée par Sy **[127]**, qui explique l'impossibilité de construire des applications interopérables dans l'absence à la fois d'une spécification formelle et d'implémentations conformes pour les services.

Ainsi, au lieu d'analyser plus profondément les services et de comparer leurs implémentations (trop hétérogènes et non formalisées) nous avons identifié des fonctionnalités de base que nous souhaitons intégrer dans le runtime d'une application répartie. Elles ont l'avantage de permettre une unification des concepts et d'être plus faciles à mettre en l'oeuvre qu'une grande diversité de services :

- *Le référentiel commun*. Dans une application repartie il est nécessaire d'assurer un schéma de noms non ambigu qui permet identifier les parties constituant une application. Au niveau gros grain cela vise l'identification de nœuds d'exécution, des capsules, celle de composants logiciels (gabarits ou instances), celle de connecteurs (média ou binders), et éventuellement celle de ports de connexion. A une granularité faible, cela peut concerner l'identification des objets d'implémentation, celle des interfaces logiques ou celle des opérations.
- *Le gestionnaire réparti*. Les applications réparties sont souvent dynamiques. Leur gestion est d'autant plus difficile car elle doit se faire en réparti. Deux aspects de gestion sont importants:
  - *gestion de ressources*. Pour une *ressource passive* (variable, objet d'implémentation, composant logiciel ou capsule) on s'intéresse à : 1) *allouer* de l'espace de stockage (e.g. mémoire) ; 2) accéder à son contenu pour *initialiser* ou *récupérer* sa valeur ; 3) *libérer* cet espace quand la ressource n'est plus requise par l'application.

    Pour une *ressource active* (tâche d'un objet actif) cela vise à : 1) *créer* une file d'exécution ; 2) l'*activer* ; 3) *arrêter* et *reactiver* l'exécution en fonction d'une stratégie d'exécution ; 4) *détruire* la file d'exécution quand son exécution est terminée.
  - *gestion de liens* (liaison). La liaison concerne la création ou la destruction de liaisons entre les instances d'objets d'implémentation. Elle est réalisable d'une façon *explicite* (à l'initiative de l'application) ou *implicite*, c'est-à-dire fixée par les règles de conception. Elle concerne la connexion directe d'objets au moyen de *binders* (connecteurs locaux) mais également la construction de connexions complexes au moyen des *media* (connecteurs à distance).

Bien que d'autres services (e.g. sécurité, temps, transactions etc.) puissent mettre en évidence d'autres fonctionnalités ou mécanismes de base (e.g. la notion de filtre d'interception), nous considérons qu' ils ne constituent pas de services de base. En effet, leur implémentation peut certes être intégrée au niveau de l'intergiciel mais il nous semble préférable de la réaliser au moyen de composants applicatifs. Par exemple, la sécurité de communications peut être assurée par l'intergiciel au moyen d'un service dédié mais elle peut être mise en œuvre au moyen d'intercepteurs (composants intermédiaires) interposés à

l'entrée des media. C'est l'exemple de GLADE **[96]** qui permet d'assurer la sécurité des communications et la compression des messages en vue d'optimiser la bande passante des communications au moyen de *filtres externes* qui ne font pas partie de son implantation.

4.4.3 Langages adaptés pour la programmation répartie

GLADE : une implémentation conforme de l'annexe DSA du langage Ada95

Le langage Ada95 et l'annexe DSA

Ada est un langage de programmation normalisé ISO (*International Standard Organisation*) qui cible le domaine d'applications sûrs (adaptées à la certification), critiques (e.g. temps-réel) de grande taille (millions de linges de code). La maintenabilité et la portabilité des programmes sont deux objectifs visés.

La norme Ada95 **[60]** constitue la version actuelle. Elle étend le langage pour supporter la programmation orientée objets sans pour autant altérer la sémantique de base du langage. Le langage est composé d'une partie centrale et une série d'annexes optionnelles dont l'annexe E (DSA - Distributed Systems Annex) qui est dédiée à la programmation en réparti.

Par sa conception, Ada95-DSA vise à faciliter la migration d'applications multitâches concurrentes existantes vers le paradigme de programmation réparti. C'est ainsi que l'annexe introduit la répartition sous la forme de directives de catégorisation **`#pragma`** à interpréter ou non par un compilateur Ada95 spécialisé.

L'avantage de l'annexe DSA, quand on la comparé à d'autres intergiciels de type RPC ou langages de programmation adaptées à la construction d'applications réparties, est qu'elle permet d'utiliser à la fois un modèle de conception basée sur :

- *un paradigme client-serveur réparti* (objets communicants). L'approche proposée est similaire à celle de Java-RMI et CORBA. L'utilisateur a la possibilité de définir soit des serveurs de services non replicables (paquetages catégorisés **`remote_call_interface`**) soit des objets répartis réplicables (paquetages catégorisés **`remote_types`**).
- *un paradigme de communication par objets répartis partagées*. L'utilisation de paragma de catégorisation **`shared_passive`** (SP) permet la définition des objets répartis partagés non réplicables. Ce mécanisme est équivalent en puissance d'expression à celui d'autres langages de programmation du même genre comme Orca **[10]** ou Linda. C'est pourquoi nous ne pressentons pas une étude détaillée de ces derniers.

GLADE

GLADE **[96]** est une implémentation complète, portable et conforme de l'annexe Ada95-DSA qui propose des améliorations d'implantation spécifiques. En effet, l'annexe DSA manque de clarté sur un certain nombre des choix d'implémentation (le langage de configuration des partitions, la délégation de priorités d'exécution etc.) que GLADE explicite. Nous présentons donc les services intégrés du runtime GLADE ainsi qu'une analyse des choix de mise en œuvre proposés.

Plusieurs services sont désormais proposés :

- *identification des unités de bibliothèque et partitions*. Comme DSA l'exige, ce service est équivalent à celui d'un service de noms pour les unités de bibliothèques (numéro de partition et de version). Ce service permet également d'attribuer à chaque partition un identifiant unique (la paire : <*protocole*, *adresse*>) qui spécifie sa localisation.
- *activation à distance de partitions*. L'activation de partitions peut-être faite manuellement ou auto-

matiquement via le service système **`rexec`** ou au moyen d'un script **`shell`**.

- *terminaison d'execution*. L'annexe DSA ne précise pas qu'elle stratégie de terminaisons utiliser pour des partitions réparties. C'est pourquoi, GLADE comble cet manque par trois politiques de terminaison : 1) globale (à l'initiative de la partition primaire) ; 2) locale (chaque partition décide son terminaison localement) ; 3) différée (sans terminaison). Cette dernière politique est utile pour la création de partitions démons.
- *tolérance aux pannes*. Caractéristique de GLADE, ce service permet de répliquer les serveurs de gestion GLADE sur toutes les partitions. Cela assure la tolérance aux pannes lorsque la partition principale devient indisponible. Pour assurer la tolérance aux pannes, GADE fait appel à deux mécanismes distincts :
  - *un système de fichiers répartis partagés* pour implémenter les partitions passives (celles qui ne contiennent que des paquetages marqués Shard_Passive) telles que NFS **[20]**.
  - *plusieurs politiques de reconnexion aux partitions.* La reconnexion à une partition peut être: 1) *interdite* (**`Reject_On_Restart`**) ; 2) *signalée* par une exception si la partition est en panne, mais elle devient de nouveau possible si la partition est réactivée (**`Fail_Until_Restart`**) ; 3) *bloquée en attente* que la partition soit réactivée (**`Wait_Until_Restart`**).
- *filtrage*. Sans que cela soit imposé par l'annexe, GLADE permet d'utiliser des filtres de communication externes en vue de sécuriser les communications ou d'optimiser l'utilisation de la bande passante.

Par sa construction, GLADE s'adresse également à l'implémentation des applications critiques, c'est pourquoi d'autres fonctionnalités de base sont inclues dans sa réalisation. Nous retrouvons :

- *un système global de priorités d'exécution pour les tâches*. GLADE propose une échelle de priorités abstraites à mapper sur celle d'un système d'exploitation. La stratégie de propagation de priorités entre les partitions client et serveur peut être également précisée. On trouve deux politiques : 1) propagation de la priorité du client sur le serveur ; 2) priorité imposé par le serveur.
- *un système de gestion de la concurrence de traitements*. Puisque Ada95-DSA ne précise pas si les invocations à un même service doivent ou non être traitées d'une façon concurrente, GLADE permet de paramétrer finement le nombre (initial, minimum et maximum) de tâches auxiliaires de traitement sur les partitions serveurs. Dans certains cas l'utilisation de tâches auxiliaires peut être même prohibée. Cela permet d'associer GLADE à des profils adaptés pour la conception d'applications critiques telles que Ravenscar **[30]** et ainsi de garantir un comportement déterministe pour l'application.

4.5 Synthèse : Spécification d'un modèle de communication canonique

Nous présentons maintenant les nombreux aspects que nous avons extraits de l'état de l'art tout au long de ce chapitre. Nous pensons que ces aspects couvrent bien tous les besoins pour la construction et le déploiement d'applications réparties. Puis nous dégagerons en conclusion les fonctionnalités basiques que nous souhaitons intégrer dans **L*f*P**.

Nous avons identifié dans les sections **3.3.4**: *Ports* et **3.3.5**: *Relations entre entités* des problèmes de dépendances pernitieuses entre unités fonctionnelles. Deux propositions nous sembles primordiales et guideront notre vision d'un modèle de communication : le média et le binder. Ces entités seront exprimées dans **L*f*P**.

Un média de communication exprime un protocole de communication entre les composants du système. Il est relié aux classes par l'intermédiaire d'un port de communication.

Un binder exprime l'association entre un couple de ports provenant respectivement d'une classe et d'un média de communication.

Ces deux entités nous permettent d'exprimer les différents types de communication que nous avons identifiés dans l'état de l'art et que nous classons maintenant en fonction de fifférents critères : paradigme de communication, format d'échange, modèles et contraintes de communication.

Paradigmes de communication

Nous avons identifié cinq paradigmes de communication importants : a) *flux d'octets*, b) *messages (asynchrones)*, c) *données partagées*, d) *invocation d'opérations* (locales ou à distance) et e) *signaux*. Parmi elles, nous retenons que quatre, l'emploi des signaux est écarté de notre modèle vu que ceux-ci :

- disposent d'une sémantique difficile à normaliser car hétérogène d'une implantation à une autre et dépendante de la politique d'ordonnancement système;
- constituent les seuls mécanismes de communication actifs (non coopératifs) : leur sémantique ne peut donc pas être homogénéisée avec celle des autres mécanismes de communication cités vu qu'elle est basée sur un principe d'interaction différent : la communication survient non quand le composant décide de l'accepter mais quand le système lui signale une interruption logicielle.

Notons également que les données partagées ne peuvent pas être considérées comme des mécanismes entièrement normalisables vu que :

- ils correspondent aussi bien à des mécanismes de communication locaux (e.g. segments de mémoire partagée) qu'à des mécanismes à distance (on parle d'objets répartis partagés et de mémoire répartie partagée);
- leur utilisation est souvent associée à d'autres mécanismes de synchronisation décrits par l'application.

Il nous paraît donc plus judicieux de les encapsuler au moyen des média protocolaires (pour lesquels on peut décrire un protocole d'accès explicite) que de les décrire par des binders normalisées.

Format d'échange canonique

L'étude de JMS montre que, d'un mécanisme de communication à un autre, ce qui diffère est surtout la façon dont l'information échangée est structurée, et qu'il est possible d'unifier ces représentations au moyen d'un format d'échange abstrait commun composé :

- d'un *discriminant* (entêtes JMS) pour le routage dépendant du modèle de programmation imposée par l'intergiciel cible.
- d'un *corps d'encapsulation* pour les données.

Notre analyse à révélé plusieurs points de conception importants :

1) Même si JMS ne vise que la communication par messages, flux d'octets ou invocations d'opérations, *nous considérons que cette structure peut être extrapolée y compris pour l'accès aux données partagées* si l'on considère que le discriminant est une clef d'accès et que le corps (s'il existe) est la valeur de la variable accédée.
2) JMS prévoit la définition de filtres d'interception. Leur portée se résume que sur le contenu des discriminants (pour router) et non sur le contenu des messages. Cela est insuffisant pour nos besoins car nous souhaitons permettre également le routage interne (par l'application) des invocations.

   Nous promouvons ainsi l'idée d'ouvrir la structure des invocations à l'application afin que celle-ci

puise mieux contrôler, si besoin, les communications. L'objectif cherché est de permettre d'implémenter des mécanismes de polymorphisme, de tissage d'aspects ou de filtrage d'invocations en fonction de leur contenu même si le langage de programmation cible ne supporte pas nativement ces mécanismes.

3)La structure du discriminant et celle du corps d'une invocation sont constituées d'une partie fixe qui peut être étendue, si besoin, par l'intergiciel (pour la structure du discriminant) ou par l'application (pour la structure du corps).

L'idée que nous promouvons est d'utiliser par défaut la structure présentée en **Figure 4-1** :

- Pour le discriminant : Une structure de type le plus petit dénominateur commun composé soit de deux champs (source et destination) soit vide. Ces deux choix permettent de mette en l'œuvre aussi bien des mécanismes de communication basés sur un schéma d'adressage explicite (e.g. l'invocation de méthodes, échange de messages) qu'implicite, c'est-à-dire sans routage (e.g. communication par flots d'octets).
- Pour le corps d'une invocation : soit une signature normalisée (**`MSG`**) soit à un block de données séquencées de taille précisée (**`BODY`**).

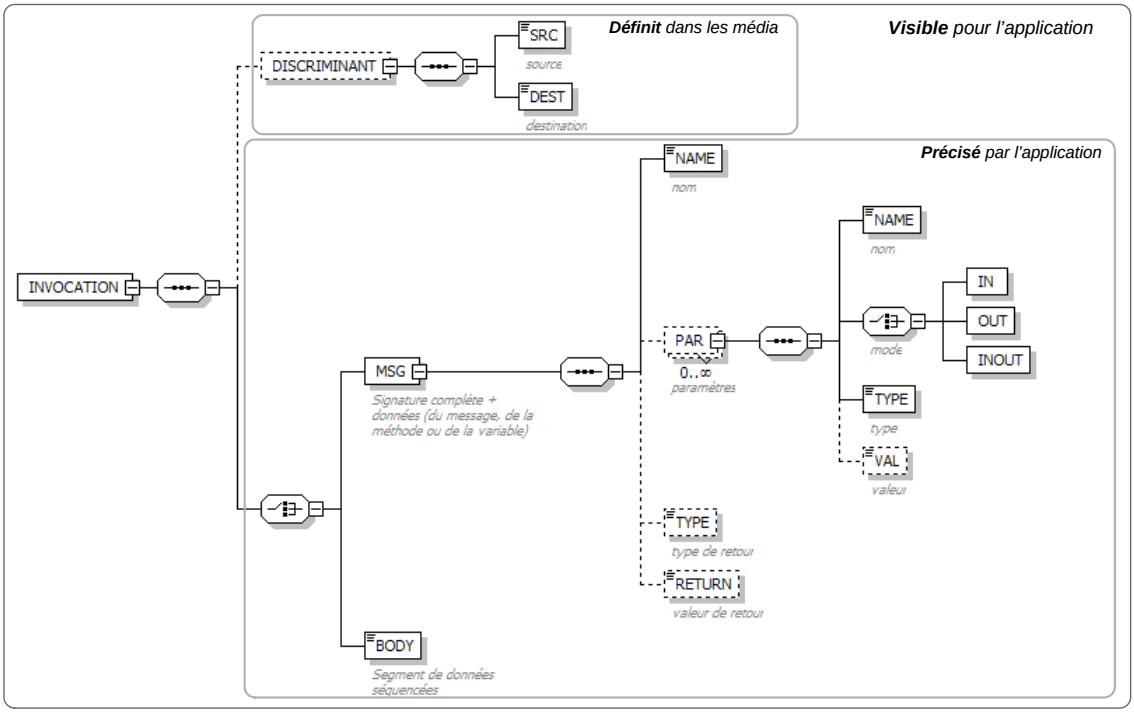


Figure 4-1 : Canonisation de la structure d'invocation.

Modèles et contraintes de communications

Nous-nous intéressons exclusivement aux modèles de communication de type *producteur-consommateur*, pour lesquels la lecture d'une information implique sa consommation. Ce choix ne constitue pas pour autant une limitation si l'on expose à l'application la structure complète des invocations. Celle-ci aura donc la possibilité d'effectuer des tests sur leur contenu (sans consommer les informations) et, le cas échéant, d'accepter la réception.

Nous identifions les caractéristiques et contraintes suivantes :

- *Initiative de la communication.* Pour les communications en local, nous avons constaté que tous les mécanismes retenus ont des comportements *passifs* car la communication se fait exclusivement à l'initiative des parties communicantes. En revanche, pour les média protocolaires nous considé-

rons qu'il faut permettre aux utilisateurs de modéliser le protocole de communication (*passif* ou *actif*).

- *Sémantique de communication*. Nous souhaitons modéliser à la fois des sémantiques de communication *asynchrones* par tampons de données que *synchrones* par *rendez-vous* ou *invocation directe d'opérations*.
- *Contraintes de capacité*. Ce qui prime pour la modélisation n'est pas le fait de pouvoir spécifier des bornes sur la taille de tampons de communication mais de prévoir des mécanismes d'exception permettant de réagir dans des situations extrêmes (e.g. débordement). Cependant, pour des raisons dues à la verification par réseaux de Petri nous sommes amenés à prévoir des bornes sur le domaine de chaque variable et par conséquent de préciser la capacité maximum des tampons de communication.
- *Strategie d'invocation*. Nous considérons que, au minimum, la stratégie d'invocation par défaut doit permettre soit 1) l'*abandon en cas d'échec et le retour d'un code d'erreur* soit 2) la *communication partielle* si la communication est par flux d'octets. D'autres stratégies d'invocation en émission comme la sur-écriture circulaire des données et le fenêtrage coulissant peuvent être imaginées. Cependant, elles ne constituent pas des modèles communs.
- *Fiabilitée des communications*. Nous considérons que toute communication bloquante en local (via un binder) doit rester fiable. Au contraire, pour les média protocolaires nous considérons qu'il faut laisser libre le choix aux concepteurs de modéliser explicitement le comportement avec ou sans pertes.
- *Entrelacement* (atomicité d'accès). Mises à part les communications par flux d'octets qui sont considérées non atomiques, nous considérons que tous les autres mécanismes de communication doivent assurer l'atomicité (locale) des échanges.
- *Séquancement et multiplexage*. Nous prévoyons de modéliser trois politiques de séquencement : FIFO (causale), LIFO (par pile) et BAG (sans ordonnancement explicite). Le rôle de cette dernière est de permettre à l'application de personnaliser par programme sa propre politique de multiplexage. Celle-ci est réalisée par le filtrage explicite d'invocation en fonction de leurs discriminants.
- *Routage*. Nous considérons que pour les binders le routage d'invocations doit se faire d'une façon implicite : un binder associée à une classe ou à un média ne reçoit que les invocations concernant cette classe. Au contraire, pour les média il nous semble utile de permettre aux utilisateurs de modifier le contenu des discriminants de routage.
- *Mode de connexion*. Nous considérons que, par défaut, le mode de communication doit être non connecté. La création de connexions est vue comme une étape ultérieure à réaliser par programme au moyen un scénario d'invocations successives de connexion.

  Le fait que les communications soient non connectées ne constitue pas un problème de cohérence pour la modèlisation vu que lors de la vérification formelle, l'envoi d'une invocation vers une destination inexistante pourra être détectée par model checking.

4.6 Conclusion

Nous avons étudié les intergiciels et les mécanismes de communication IPC en vue d'établir un modèle d'environnement canonique abstrait à utiliser pendant la modélisation. Nous avons retenu une liste de paramètres à considérer pour la construction des binders (connecteurs de communication en local) et

des média (connecteurs protocolaires pour les communications à distance) ainsi que une liste de fonctionnalités de base à inclure dans le runtime de notre langage de modélisation.

Notre analyse prouve qu'il est possible d'unifier, à la manière de JMS, les paradigmes de communication au moyen d'un format d'échange abstrait commun. Deux idées importantes ont été identifiées :

- il est utile de séparer la structure d'invocation en deux parties : a) discriminant fixé par les média et b) corps de structures paramétrable décrit par l'application. Ces deux parties représentent des aspects indépendants la communication.
- il est important de permettre à l'application l'accès aux invocations dans leur intégralité (champs du discriminant et du corps) afin d'implémenter des mécanismes d'indirection et de filtrage personnalisés.

Pour la construction des binders nous avons retenu deux classes de mécanismes de communication a) tampons de mémoire paramétrables qui peuvent servir à la fois pour la communication par flux d'octets, par messages asynchrones ou par rendez-vous synchrones et b) accès directs qui permettent de modéliser l'invocation d'opérations locales. Nous avons également identifié une liste de paramètres à considérer pour la spécification de ces mécanismes que nous considérons intégrer dans notre approche de modélisation.

L'étude de CORBA et de GLADE nous a permis d'identifier les fonctionnalités élémentaires à intégrer dans le runtime de notre langage :

- le référentiel commun, permettant d'identifier les parties constituant une application d'une façon non ambiguë.
- le gestionnaire de ressources actives (tâches) et passives (mémoire) permettant d'activer, suspendre, réactiver, ou détruire les tâches de traitement d'une application et d'allouer, initialiser et libérer les ressources mémoire nécessaire pur le stockage des contextes d'exécution.
- le gestionnaire des liens permettant de créer ou de détruire des liaisons. Puisque notre thèse vise la conception d'applications dynamiques non reconfigurables (l'architecture est implémentée dans le code, il faut au minimum reconstruire l'application en cas de changement), nous considérons que la gestion de liens doit se faire, à la lumière de ce qui est proposé dans Ada95 et DSA, d'une façon transparente pour l'application, c'est-à-dire par liaison implicite.

Le chapitre suivant présente en détail la solution que nous proposons pour modéliser une application rpartie. Elle consiste en **L*f*P**, un langage de spécification du contrôle d'une application répartie intégrant les concepts proposés dans les ADL. Nous nous sommes préoccupé de rester cohérent avec les concepts proposés dans les approches standardisées comme UML.

CHAPITRE 5

La notation **L*f*P**

5.1 Introduction

L/P vise la construction d'applications réparties selon une approche de développement par prototypage rapide. La sûreté du développement ainsi que la conformité des implémentations sont deux objectifs essentiels de sa conception. Il possède à la fois les caractéristiques d'un langage de description du contrôle d'applications réparties et celles d'un Langage de Description d'Architecture (ADL):

- **L/P** est basé sur UML. Les concepteurs d'une application répartie utilisent UML pour la phase de conception, puis passent à **L/P** dès qu'ils veulent exprimer avec précision son architecture logicielle et le comportement des composants répartis.
- **L/P** s'inspire de l'approche RM-ODP dont il reprend la séparation des aspects via les vues ingénierie, traitement et technologie. Cette démarche est bien adaptée aux applications réparties qu'il faut souvent redéployer sur des configurations matérielles et logicielles hétérogènes, très différentes sans avoir à subir des aleas et des coûts prohibitifs de modification.
- **L/P** permet d'obtenir des modèles de comportement, basés sur les réseaux de Petri de haut niveau, afin de vérifier formellement des propriétés d'une application, avant même sa réalisation. Ce qui est bien dans la ligne préconisée par l'approche Model driven Architecture (MDA) de l'OMG.

L/P se veut un langage formellement défini dédié à la description du contrôle d'une application répartie. Comme il doit à la fois permettre la vérification et la génération automatique de programmes assurant la supervision du système à développer, il possède également des caractéristiques propres à un langage de description d'arcchitecture (ADL).

L/P reste basé sur les principes d'UML afin de se rapprocher des standards et approches méthodologiques utilisées dans l'industrie. Il reprend certains éléments du modèle objet d'UML ; ainsi, la notion de diagramme d'architecture qui sera présentée en **Section 5.2.2** raffine la notion de diagramme de classe (en ajoutant des liaisons non ambiguës avec la description du comportement).

Nous nous sommes également clairement inspiré de RM-ODP et le système de vues complémentaire qui est proposé (en particulier, nous nous sommes intéressé aux vues «ingénierie» et «technologie»). Le système d'interaction entre composants logiciels (les «binding points») nous a également inspiré.

Nous présentons tout d'abord la structure générale du langage **L/P** en **Section 5.2**. Nous détaillons en **Section 5.3** les diagrammes de comportements qui est à la base de la description hiérarchique du comportement d'une entité dans **L/P**. Nous présentons alors les types disponibles dans notre langage en **Section 5.4**. Nous nous présentons enfin dans les dernières sections les notions de classes, média et binders dans **L/P**.

5.2 La structure de L/P

Pour accomplir ses objectifs, **L/P** définit une notation sans ambiguités, basée sur trois vues complémentaires, chacune étant réservée à une étape de développement différente : *Modélisation*, *Vérification*, *Implémentation*.

La vue Fonctionnelle est consacrée à la modélisation de la partie contrôle d'une application répartie et précise des points d'insertion pour le code fonctionnel. Elle contient :

- *une partie déclarative* contenant elle même les déclarations de types et constantes utilisés dans le modèle et des informations de traçabilité (entre autres les références vars les composants du modèle UML,
- *un graphe hiérachique* décrivant l'architecture du système, lui même composé de classes, corres-

pondnat aux classes instanciables d'UML, de média décrivant les communications entre classes, et de binders représentant les ports de communication entyre les classes et les média. Les média apportent une grande souplesse pour spécifier des caractéristiques de la sémantique des communications (direction, capacité, cardinalité, asynchronisme, ordonnancement). les binders spécifient des contrats de liaison entre les classes et les média.

Cette vue définit une architecture logicielle neutre, au sens du Chapitre 4, pour l'application. Ainsi elle est exploitable à la fois pour la vérification formelle et la génération automatique de programmes répartis.

La vue Propriétés permet de spécifier des propriétés sur le comportement d'une application (e.g. vivacité, invariants, assertions sur l'état d'exécution) ainsi que des prédicats logiques sur la causalité de l'exécution (e.g. traces d'exécution). Ces propriétés qui complètent la vue fonctionnelle. seront autant d'obligations de vérification. Elles seront aussi exploitées pour insérer dans les programmes générés des points de vérification à l'exécution (runtime checks).

La vue Implémentation est consacrée au déploiement et à l'implantation des modèles. Elle permet de préciser des contraintes (annotations) exploitables par un générateur de programmes ou encore des informations pour le déploiement du système. Nous nous sommes pour cela inspirés de la notion de tag dans UML. Cette vue permet donc de préciser la manière dont l'application est partitionnée, les technologie d'implémentation (langages de programmation et exécutifs sous-jacents) ainsi que d'intégrer le code fonctionnel existant.

La démarche **L*f*P** est effectivement compatible avec le modèle MDA (*Model Driven Architecture*)**[9]**. En effet, la vue fonctionnelle constitue un modèle neutre PIM (*Platform Independent Model dans MDA*) indépendant vis-à-vis des choix d'implantation ou du formalisme à utiliser pour la vérification (e.g. les réseaux de Petri). Quant aux modèles formels utilisés pour la vérification ainsi que les prototypes exécutables, ils sont des PSM (*Platform Specific Model dans MDA*) obtenus par un processus de transformation automatisable. Les vues Propriétés et Implémentation servent ainsi à guider, à consolider et à paramétrer ce processus.

La **Section 5-1** illustre les liens entre le langage **L*f*P** (ses trois vues) et les modèles ou prototypes dérivés, en explicitant leurs relations avec les concepts MDA. Les annotations de la vue Fonctionnelle introduites par la vue Propriétés permettent d'obtenir des modèles formels vérifiables tandis que celles apportées par la vue Implementation permettent d'engendre des prototypes. Ces modèles et prototypes constituent des PSM au sens de MDA. Les annotations concourent à régir le processus de transformation plus ou moins automatique de PIM en PSM.

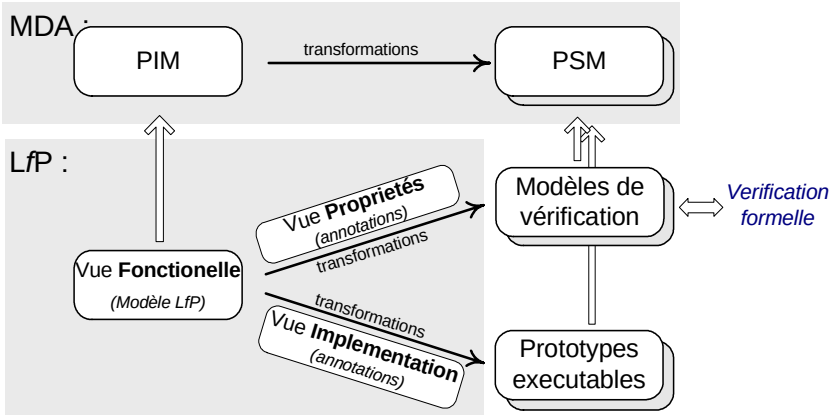


Figure 5-1 : La structure de **L*f*P** et ses relations avec MDA.

Syntaxe XML du langage L*f*P

L*f*P est une notation mixte : *textuelle* (pour les déclarations et le code séquentiel) et *graphique* (pour la description du comportement). Pour aider à la transformation de modèles **L*f*P** vers d'autres formalismes cible (e.g. les réseaux de Petri, langages de programmation) nous proposons un format d'échange intermédiaire, facilement transformable et qui est débarrassé de tout sucre syntaxique.

Pour ce faire nous avons choisi d'utiliser le langage XML, et de proposer une grammaire abstraite exprimée au moyen d'un XML schéma **[138]**. Elle a une forme arborescente dont la hiérarchie de description correspond à celle d'un modèle **L*f*P** (on parle de description par raffinements). La représentation XML d'un modèle **L*f*P** correspond donc à un arbre d'expression XML abstrait. Cette représentation hiérarchisée est utilisée tout au long de ce chapitre afin d'illustrer la structure logique des diverses constructions **L*f*P**.

5.2.1 La vue fonctionnelle

La vue fonctionnelle **L*f*P** vise la modélisation de l'architecture logicielle d'implementation d'une application répartie aussi bien d'un point de vue structurel que comportemental. Notre approche de modélisation est complémentaire de celle d'UML : Au lieu de décrire l'application à l'aide d'un ensemble hétérogène de diagrammes nous préférons rassembler cette description dans un diagramme unique de structure modulaire hiérarchique qui permet de d'assurer la cohérence sémantique de la description. Cette démarche est similaire à celle d'autres ADL tels que Wright **[3]** et ROOM **[111]**.

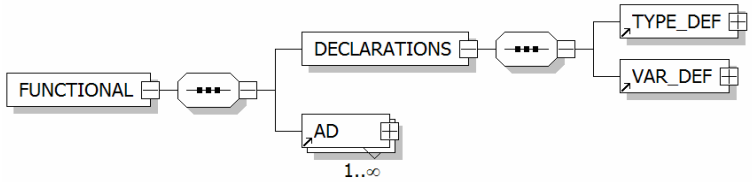


Figure 5-2 : Structure logique de la vue fonctionnelle.

La **Figure 5-2** exhibe la structure logique de la vue fonctionnelle **L*f*P**. Comme illustré, les déclarations globales à une application (types et constantes) sont regroupées directement dans la vue fonctionnelle. Celle-ci précise également une liste de diagrammes d'architecture (au moins un). Chacun d'eux précise une partie de l'architecture d'implémentation du système modélisé.

5.2.2 Le Diagramme d'Architecture

Le Diagramme d'Architecture (AD) constitue le pivot d'une spécification **L*f*P**. Il décrit l'architecture logicielle concrète d'une application au moyen d'un graphe connexe d'entités logicielles (les Classes **L*f*P**) reliées par des entités de communication décrivant des protocoles (les média **L*f*P**). L'assemblage des Classes et des média est réalisé par la «connexion» de points d'interaction (les Ports **L*f*P**). Cette connexion est contrainte par des contrats de liaison normalisés (les Binders **L*f*P**).

L'ontologie proposée (Classes, Ports, Binders, média) s'appuie sur les conclusions que nous avons exposées dans la **Section 3.2** *Description des applications réparties*. Elle s'inspire à la fois des concepts d'RM-ODP, d'UML et des ADL. Ainsi :

- Les ***Classes* L*f*P** décrivent la partie fonctionnelle de l'application. Elles constituent des unités d'encapsulation pour les données (les attributs de la classe) et les traitements associés (les méthodes LfP) qui précisent un mode d'emploi (le contrat d'utilisation de la classe), des contraintes d'accès aux services ainsi que des détails de mise en oeuvre (les contrats comportementaux des méthodes). Une classe **L*f*P** précise également des détails ADL (les Ports d'accès).

Une classe **L/P** raffine donc la description d'une classe d'implémentation UML. Les concepts de classe et d'instance de classe **L/P** sont équivalents à ceux de gabarit de conception et respectivement d'objet RM-ODP (information, traitement ou d'ingénierie).

- Les ***média* L/P** sont des connecteurs ADL complexes pour lesquels il est possible de décrire un protocole de communication au moyen d'un automate.
  Lors de l'implémentation, les média correspondront à l'infrastructure de communication utilisée pour le déploiement de l'application (e.g. un protocole de communication concret, l'intergiciel etc.).
- Les ***Ports* L/P** sont des points de connexion physiques (Ports ADL) pour les Classes et les média. L'interaction de l'extérieur avec une classe ou un média ne peut se faire que par l'intermédiaire d'un port, celui-ci exprime un point de «soudure».
  Le concept de Port **L/P** est équivalent à un point de liaison RM-ODP. Ce concept n'est pas équivalent à celui d'interface UML qui exhibe une connotation logique.
- Les ***Binders* L/P** sont des objets de liaison permettant de typer la connexion des Ports par des contraintes d'interaction simples, normalisées et portables. Lors de l'implémentation, les binders **L/P** seront traduits dans des mécanismes de communication en local (e.g. IPC, appels directs de procédures etc.).
  On peut associer plusieurs binders à une classe pour spécifier un protocole complexe. Ainsi **L/P** est plus général que le modèle événementiel des statecharts UML qui n'offe qu'un unique point d'entrée.
  Dans la terminologie ADL, un binder **L/P** correpond à un *connecteur normalisé exprimant une interaction directe*.

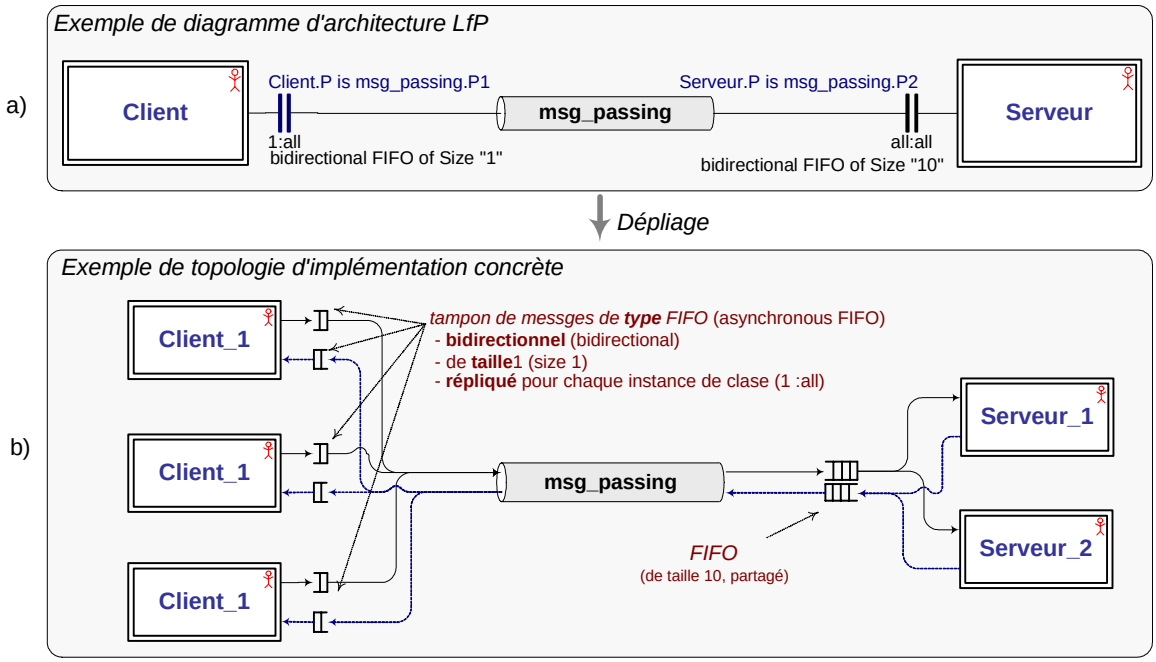


Figure 5-3 : Diagramme d'architecture **L/P** : Un exemple de modélisation.

Exemple de modélisation

La **Figure 5-3** illustre un exemple classique d'application Clients-Serveurs répartis dont les Clients et les Serveurs sont des acteurs (Classes **L/P** actives) connectés par l'intermédiaire d'un média (nommé `msg_passing`). Cette connexion est réalisée par la liaison des ports `P` (des clients et des serveurs) aux

ports `P1` ou `P2` (du média). Pour préciser la sémantique de connexion on interpose des binders (de taille, multiplicité et sémantique d'interaction précisées par des annotations).

Le diagramme d'architecture présenté dans la **Figure 5-3**-a) correspond à un gabarit de conception qui contraint la topologie de déploiement réelle. Ainsi, la **Figure 5-3**-b) illustre une version dépliée de ce diagramme qui correspond à une application concrète (3 Clients et 2 Serveurs). Les binders sont instanciés par rapport à leur multiplicité. Chaque client dispose de ses propres tampons de communication (multiplicité `1:all`). Au contraire, tous les serveurs partagent (multiplicité `all:all`) les mêmes tampons de communication (on parle d'un pool de serveurs).

5.3 Les diagrammes de comportement (L*f*P-BD)

Pour modéliser le contrat comportemental imposé par une Classe, le protocole de communication exhibé par un Média ainsi que le flot d'exécution propre à une Méthode (dans une Classe ou un Média), le langage **L*f*P** propose une notation commune : les diagrammes de comportement **L*f*P**-BD (**L*f*P** Behavior Diagram).

Ce diagramme constitue la plaque tournante de notre langage. C'est un moyen pour spécifier (formellement) le comportement d'une application selon une *approche de description par composition modulaire hiérarchique*.

Définition 5-1: Un diagramme de comportement **L*f*P**-BD décrit un gabarit de comportement composé d'Etats et de Transitions **L*f*P** connectées par des Arcs orientés dont :
- Les *Etats* **L*f*P** correspondent à des points d'exécution importants, ils marquent des étapes dans le comportement d'un fil d'exécution.
- Les *Transitions* **L*f*P** modélisent une suite d'actions séquentielles permettent à un fil d'exécution de passer d'un état à un autre.
- Les *Arcs* de liaison décrivent des chemins d'exécution acceptables, le fait qu'une transition soit liée à un état au moyen d'un arc exprime une contrainte sur le flot d'exécution.

La construction de ce graphe doit respecter les règles suivantes :

- C'est un graphe biparti, c'est-à-dire que les nœuds de ce graphe alternent : les voisins d'une transition sont des états et vice versa.
- C'est un graphe connexe et, plus précisement tel qu'il existe toujours au moins un chemin élémentaire (sans circuit) de l'état initial à tout nœud, et toujours au moins un chemin élémentaire de tout nœud à l'état terminal. Ceci n'exclut pas l'existence de circuits dus aux boucles d'itération.
- Ce diagramme est un graphe totalement connecté, c'est-à-dire quel qu'il soit `N1` et `N2` deux nœuds il existe un chemin qui permet de parcourir le graphe de `N1` à `N2` (sans prendre en compte le sens des arcs).
- Chaque transition n'a qu'un seul arc en entrée et un seul en sortie.

La **Figure 5-4** présente la structure logique d'un diagramme **L*f*P**-BD sous la forme d'un schéma XML. Celle-ci consiste en un ensemble de d'entités de modélisation (voir **Définition 5-1**) : Etats, Transitions et Arcs **L*f*P**. La structure de chacune d'elles est raffinée dans les sections suivantes.

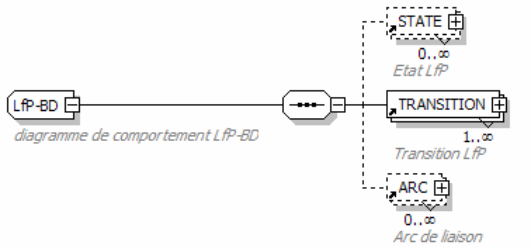


Figure 5-4 : Structure logique d'un diagramme de comportement **L*f*P**-BD.

5.3.1 Etats L*f*P

Un état **L*f*P** correspond à un point d'exécution dans le comportement d'un fil d'exécution. Tout état **L*f*P** peut être étiqueté. **`BEGIN`** et **`END`** sont des noms réservés, qui indiquent l'état initial et respectivement l'état final d'un **L*f*P**-BD. Il existe un unique état initial **`BEGIN`**, et un unique état final **`END`**, sans arc en sortie.

La déclaration d'états dans un **L*f*P**-BD n'a un sens que si le comportement modélisé par celui-ci est séquentiel (e.g. le contrat comportemental d'une classe active, celui d'une classe protégée, le flot d'exécution d'une méthode **L*f*P** etc.). C'est ainsi que dans le cas d'un **L*f*P**-BD exprimant un comportement concurrent (e.g. le protocole exhibé par un média) la présence d'états n'est pas obligatoire (à condition que le protocole soit séquentiel).

Les deux états **`BEGIN`** et **`END`** sont également utilisés pour la composition hiérarchique de sous diagrammes, ils constituent des points de soudure permettant l'assemblage modulaire de diagrammes. Ce point est abordé en détail dans la **Section 5.3.6** : *Composition hiérarchique de blocs*.

Exemple de modélisation

La **Figure 5-5** présente un exemple de diagramme de comportement **L*f*P**-BD décrivant le contrat d'utilisation d'une classe. Cet automate est constitué de trois états (**`BEGIN`**, **`S1`** et **`END`**) et trois transitions (**`P:M1()`**, **`P:M2()`** et **`T`**). **`P:M1()`** et **`P:M2`** réglementent l'accès aux services **`M1()`** et **`M2()`** offerts par la classe. **`T`** est une transition interne qui permet de changer l'état d'exécution de **`S1`** à **`END`**.

Ce diagramme propose deux branches d'exécution alternatives. Pour celle de gauche : **`BEGIN`** et **`S1`** sont les états d'avant et d'après l'invocation de la méthode **`M2()`** (déclanchée lorsque une invocation arrive via le port **`P`**). L'état **`BEGIN`** est à la fois prédécesseur et successeur de l'invocation du service **`M1()`** (visible sur le port **`P`**).

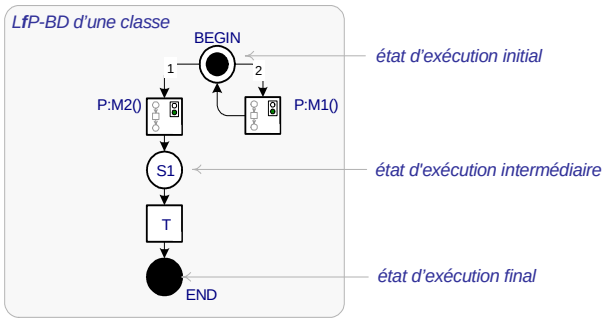


Figure 5-5 : Modélisation d'états **L*f*P**.

5.3.2 Transitions L*f*P

L*f*P se base sur une approche de conception descendante. La notion de transition **L*f*P** permet d'organiser ce raffinement au moyen de modules structurés hiérarchiquement.

Une transition **L*f*P** encapsule une suite d'actions séquentielles permettant à un fil d'exécution de passer d'un état en entrée à état en sortie.

Nous définisons deux classes de transitions :

1) **transitions simples**. Elles décrivent une suite d'actions séquentielles en pseudo code **L*f*P** (embarquées dans le corps de la transition).

2) **transitions** hiérarchiques. Il s'agit de l'encapsulation d'autres comportements exprimés à l'aide d'un sous diagramme **L*f*P**-BD. Les transitions hiérarchiques permetent la reutilisation modulaire. Deux types de transitions hiérarchiques sont proposés :
   - *transitions-bloc*. Ce sont des artefacts graphiques (macros) permettant de faciliter la conception en cachant à l'utilisateur un sous **L*f*P**-BD borné par deux états (un en entrée et un en sortie).
     Un bloc **L*f*P** peut être utilisée à divers fins de conception :
     - pour *accroître la lisibilité des modèles.* La conception d'un **L*f*P**-BD est allégée de tout détail jugé comme sans importance pour le niveau de description donné.
     - *pour encapsuler un sous-réseau commun* référencé plusieurs fois dans un diagramme.
     - *pour décrire un rôle*. Dans le contrat comportemental d'une classe un rôle dénote une partie du protocole d'interaction qui correspond à un scénario d'utilisation. Par exemple, une classe modélisant un serveur actif peut jouer deux rôles : rôle de daemon ou rôle de traitement.
   - *transitions-méthode*. Ce sont des interfaces permettant de réglementer l'accès à un service offert par la classe (méthodes publics). Une telle transition précise les conditions de visibilité permettant l'invocation d'une méthode ainsi que des gardes qui doivent être satisfaites pour que l'exécution d'une méthode soit possible.

   Une transition-méthode empile un nouveau contexte d'exécution local. Cela n'est pas le cas de transitions-bloc.

Structure représentation graphique des transitions L*f*P

La représentation graphique et la structure logique correspondant à une transition LfP sont indiqués dans la **Figure 5-6**. A gauche de la figure, on représente la transition et les attributs qui la décrivent. A droite, on illustre sous la forme d'un XML schéma comment ces informations sont structurées.

Les informations suivantes y figurent :
- *Un nom* (optionnel) indique soit une étiquette soit, dans le cas des transitions encapsulant des méthodes, il correspond au couple <*ports d'accès*, *signature de la méthode*>.
- *Plusieurs annotations* précisent si la transition est *protégée* (`protected`), si elle est *partagée* au niveau de la classe (`shared`) et si elle précise un sous-réseau *hiérarchique* (e.g. une méthode, un bloc ou un rôle **L*f*P**).
- *Une garde* exprime un prédicat logique à satisfaire avant d'exécuter la transition. Comme nous l'avons argumenté en **Section 3.3**: *Langages de Description d'Architecture* il est possible, en **L*f*P** de "garder" les invocations non seulement par rapport à l'état interne de la classe mais également par rapport au contenu des informations échangées. L'évaluation de ce prédicat est atomique.
- *Un corps* n'est présent que dans le cas des transitions simples (non hiérarchiques). Il précise une suite d'instructions en pseudo code **L*f*P**. L'exécution de ces instructions est séquentielle et non

atomique.

- *Une post condition* précise une assertion sur l'état de la classe à verifier en fin d'exécution, c'est-à-dire dans l'état suivant la transition.

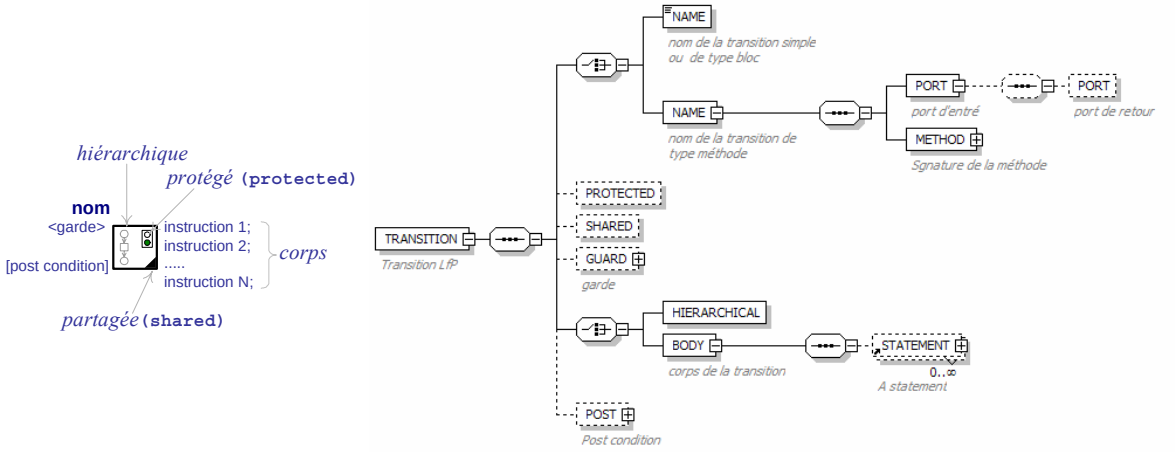


Figure 5-6 : Représentation graphique et structure logique d'une transition **L*f*P**

Exemple de modélisation

La **Figure 5-7** illustre un exemple de modélisation simple : le cas d'un serveur de traitements concurrents (une classe **L*f*P** active) qui peut exécuter des actions à l'initiative de clients. Cette figure ne décrit que les contrats comportementaux de la classe et celui d'un service : la méthode **`init`.** Nous l'utilisons pour exhiber la manière dont une transition **L*f*P** est modélisée.

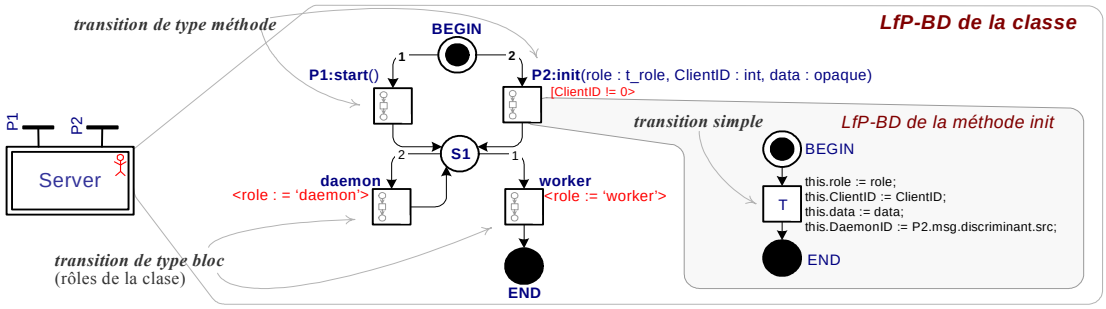


Figure 5-7 : Modélisation de Transitions **L*f*P**.

La classe **`Server`** dispose de deux ports d'accès (**`P1`** et **`P2`**) : **`P1`** est dédié à l'interaction avec les clients et **`P2`** à la gestion des instances en interne.

Le contrat comportemental de la classe comporte plusieurs transitions :

- **`daemon`** et **`writer`** sont des transitions de type bloc. Elles encapsulent des rôles alternatifs que la classe peut jouer. Chacun d'eux correspondant à un scénario d'exécution disjoint (ils ne partagent pas d'états intermédiaires). Ces transitions sont hiérarchiques. Leur exécution est gardée par un prédicat permettant de choisir un rôle ou l'autre.
- **`P2:init()`** identifie une transition - méthode. Celle ci correpond à un service public, une méthode d'instance. Le nom de cette transition est composé de deux parties : l'une précisant les ports d'accès (**`P2`**) et l'autre identifiant le nom du service invocable (la méthode **`init`**). Le port **`P2`** est à la fois un port d'entrée et de retour.
  Cette transition est protégée (marquée par un feu) et gardée par un prédicat. L'exécution de la

méthode init est possible que si le paramètre **`ClientID`**, l'identifiant du client, n'est pas null.

- **`P1:start()`** est une autre transition-méthode. Elle n'est visible que sur le port **`P1`**.

Le contrat comportemental de la méthode **`init`** consiste d'une unique transition simple : **`T`**. Celle-ci précise une suite d'actions ayant pour but de sauvegarder les paramètres d'invocation (**`role`**, **`ClientID`** et **`data`**) dans des variables d'instance (**`this.role`**, **`this.ClientID`** et **`this.data`**). La dernière instruction précise de stocker l'identifiant du serveur **`daemon`** (précise dans le discriminant de l'invocation) dans une variable locale.

5.3.3 Arcs L*f*P

Un arc **L*f*P** permet de lier les transitions et états afin de préciser les chemins d'exécution possibles.

Un arc peut être étiquetée par une priorité d'évaluation qui impose un ordre d'exécution entre plusieurs transitions en conflit issues d'un même état en entré. Ce choix permet de rendre déterministe l'exécution lorsque plusieurs transitions sont déclanchables simultanément. Cette priorité dénote un ordre d'évaluation. Si deux arcs ont la même priorité, le déclenchement des transitions cible est non déterministe. Par exemple, dans **Figure 5-7** les transitions-méthodes P1:start() et P2:init() sont en conflit (elles sont déclanchables simultanément). C'est pourquoi il faut étiqueter les arcs en entrée par des priorités d'évaluation.

5.3.4 Composition hiérarchique de blocs

La composition hiérarchique des transitions-bloc se fait par remplacement de la transition par son diagramme **L*f*P**-BD. On se base donc sur la fusion d'états : les états initial (**`BEGIN`**) et final (**`END`**) du diagramme de comportement du bloc sont fusionnés avec les états en entrée et en sortie de la transition-bloc.

Deux règles de compositions sont définies :

- La garde de la transition-bloc, lorsqu'elle existe, est reportée sur toutes les transitions d'entrée (ayant comme état d'entrée **`BEGIN`**) du **L*f*P**-BD du rôle.
- La priorité de l'arc en entrée est reportée sur les arcs des transitions en entrée du sous **L*f*P**-BD. Cette transformation se fait en préservant l'arbre de décision fixée par les priorités hiérarchiques. Un exemple est donné par la suite.

Exemple de composition hiérarchique de blocs

La **Figure 5-8** reprend l'exemple de la classe présentée en **Figure 5-7**, page 79. Elle illustre le contrat comportemental de la classe et deux sous diagrammes **L*f*P**-BD correspondantes aux rôles (**`daemon`** et **`worker`**) que la classe peut interpréter.

Pour le schéma de gauche : le rôle **`daemon`** est constitué de deux chemins d'exécution alternatifs. L'exécution de la méthode **`notify`** est privilégiée (pointé par un arc de priorité **`1`**) par rapport à elle de **`exec`** (pointé par un arc de priorité **`2`**) car elle permet de libérer des ressources lorsqu'une instance de traitement worker termine un traitement. Ces deux méthodes sont visibles sur deux ports distincts : **`P1`** et **`P2`**.

Pour le schéma de droite : le rôle **`worker`** est constitué d'une transition simple. Le corps de cette transition précise une suite d'actions en pseudo code **L*f*P** :

- **`code:=user_exec(data)`** constitue un appel local à une opération privée. Le résultat de cette opération est récupéré par la variable **`code`**.
- **`P1:ClientID.return(code)`** constitue la réponse à l'invocation **`exec`** faite par un client au **`daemon`**. Ce retour est réalisé par l'intermédiaire du port **`P1`**. Notons que cette réponse est simpli-

fiée car elle ne précise pas à quelle invocation elle correspond. Cela suppose que le client soit bloqué dans une attente synchrone. Ainsi, afin de construire une réponse complexe (corps et discriminant de routage), l'utilisateur doit manipuler explicitement la structure d'invocation.

- **`P2:notify()`** notifie le serveur de la terminaison d'un traitement. Suite à cette action, l'instance de classe est terminée car elle arrive dans l'état final (**`END`**).

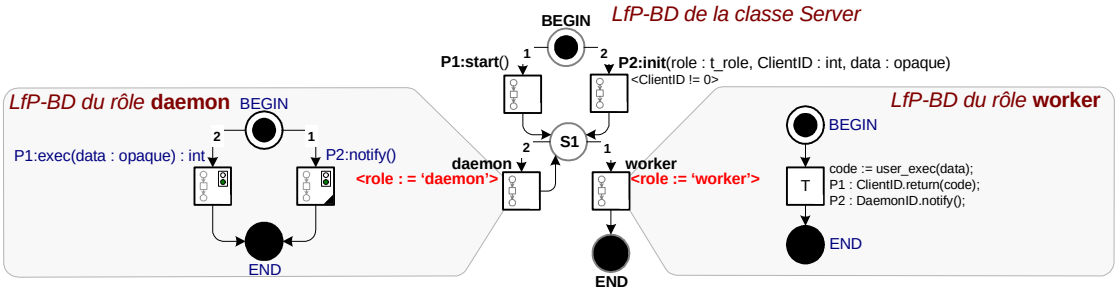


Figure 5-8 : L*f*P-BD d'une transition-bloc : Raffinement des rôles `daemon` et `worker`

La **Figure 5-9** illustre une version aplatie du contrat comportemental de la classe Server obtenue après la composition des rôles **`daemon`** et **`worker`**.

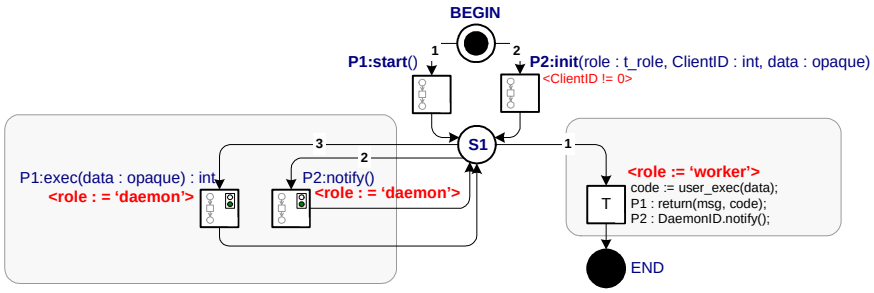


Figure 5-9 : Inclusion des rôles worker et daemon dans le **L*f*P**-BD de la classe Server.

On observe que :

- les états **`BEGIN`** et **`END`** du sous **L*f*P**-BD des deux rôles sont fusionnés avec **`S1`**, et **`S1`** et **`END`** du **L*f*P**-BD de la classe.
- les gardes qui protègent l'exécution des rôles (**`<role='daemon'>`** et **`<role='worker'>`**) ont été déportées sur chacune des transition d'entrée du sous **L*f*P**-BD correspondant.
- les priorités d'exécution sur les arcs des transitions ont été recalculées afin de préserver l'arbre décisionnel. C'est ainsi que :
  - toute transition en entrée du rôle **`worker`** reste prioritaire à toute transition en entrée du rôle **`daemon.`**
  - l'exécution de **`notify`** du rôle **`daemon`** reste prioritaire par rapport à **`exec`**.

5.3.5 Exécution des L*f*P-BD

On dit qu'une transition **L*f*P** *est exécutable* si et seulement si:

- l'état en entrée de la transition est actif,
- éventuellement, la garde de la transition est satisfaite,
- éventuellement, la transition est prioritaire par raport à ses voisines,
- éventuellement il existe au moins un des ports d'attente qui est activé par une invocation (c'est le cas pour pour l'exécution de services),

- éventuellement, si la transition est protégée, il est possible d'acquérir l'accès exclusif à l'instance courante ou à la classe (si la transition est déclarée statique).

Nous précisons ces règles :

1) ***Etats d'entrée.*** La notion d'état **L*f*P** a une connotation différente en fonction du type de comportement modélisé. Ainsi :
   - Pour une *classe passive* : La notion d'état d'exécution n'a pas de sens vu que l'accès aux services est concurrent et non protégé. Dans ce cas on considère que l'état d'entrée de cette transition est toujours marqué, c'est-à-dire qu'elle est toujours exécutable.
   - Pour une *classe protégée* : Un état correspond à une variable utilisée pour discriminer l'exécution. On dit que l'état en entrée d'une transition est marqué si la valeur de cette variable correspond à son nom. Cela correspond à une construction de type "`entry` ....`when etat` = '*nom_de_l'état*'" d'un objet de type protégé Ada95 qui permet de n'accepter une invocation que si l'objet se trouve dans le bon état.
   - Pour une *classe active* : Un état correspond à un point d'exécution important. On dit qu'un état est actif si le fil d'exécution de l'instance de classe se trouve dans ce point d'exécution.

2) ***Evaluation des gardes.*** Une garde est exprimée par un prédicat **L*f*P**. Chaque terme de ce prédicat exprime soit :
   - *un test* sur la valeur d'une variable visible,
   - *une demande d'acquisition sur un sémaphore*.
     Par exemple la garde `<a = 5 AND v1.p() AND v2.p()>` est vraie si la valeur de `a` (une variable) est `5` et si les opération `p()` sur les deux sémaphores `v1` et `v2` sont possibles ensemble.

3) ***Priorité.*** Par défaut, **L*f*P** rend déterministe tout choix d'exécution. La priorité d'exécution est calculée par raport à deux informations :
   - *La priorité de l'arc en entrée.* Dans un état donné, l'ordre d'évaluation des transitions en conflit est fixée par la priorité des arcs en entrée (voir **Section 5.3.3**).
   - *L'ordre d'énumération des ports* dans le nom de la transition-méthode.

4) ***Activation de ports*** : On dit qu'un port d'entrée est activé pour une transition-méthode si celui-ci contient une *invocation éligible* compatible avec la signature de la méthode.

   L'éligibilité d'une invocation est fixée par la politique d'ordonnancement du binder de liaison connectée au port. Par exemple, pour un binder de type FIFO, seule l'invocation correspondant au premier message dans la file est éligible. En revanche, pour un binder de type BAG (sans ordonnancement) toute invocation stockée dans le binder est considérée comme éligible. Une description détaillée des binders **L*f*P** est donnée en **Section 5.7**.

5) ***Synchronization*** : Le fait qu'une transition soit protégée se traduit par une tentative d'acquisition sur le sémaphore de l'instance de classe ou sur celui de la classe.

L'évaluation de ces conditions respecte plusieurs principes :

1) *Atomicitée globale*. Pour une transition donnée, l'évaluation des règles d'exécution respecte un schéma de type «tout ou rien» garantissant l'atomicité de l'évaluation de l'ensemble de conditions. Par exemple, on ne peut pas acquérir le sémaphore d'instance (d'un service déclaré protégé) et se bloquer en l'attente d'une invocation.

2) *Evaluation par état*. L'évaluation du franchissement d'une transition a lieu quand le système arrive dans un nouvel état d'exécution (non terminal).

3) *Gestion des conflits*. L'évaluation de ces règles se fait sur l'ensemble des transitions accessibles à partir de l'état d'exécution courant. En cas de conflit (plusieurs transitions exécutables simultané-

ment), on utilise les règles de priorité afin de rendre déterministe le choix de la transition à exécuter.

4) *Evaluation permanente*. Pour un état actif donné, si aucune transition n'est exécutable les conditions d'exécution sont réévaluées systématiquement soit à l'aide d'un mécanisme de test actif (polling) soit au moyen d'un mécanisme de surveillance passif (moniteurs).

5.3.6 Principes de raffinement de L*f*P en réseaux de Petri

Une spécification **L*f*P** constitue une abstraction de haut niveau d'un réseau de Petri Coloré bien Formé d'une structure particulière. La traduction d'une spécification **L*f*P** dans un en réseaux de Petri respecte deux principes :

1) Chaque construction R*d*P, que cela soit au niveau de description gros grain (classes, média, binders, ports **L*f*P**) ou à une granularité fine (états, transitions et arcs dans un diagramme de comportement **L*f*P**-BD), trouve un correspondent équivalent en réseaux de Petri; On parle de patrons de transformation génériques ou schéma de traduction paramétrables. En effet, pour obtenir le sous-réseau de Petri correspondant à une à une construction R*d*P il faut paramétrer le patron correspondant.

2) La structuration modulaire hiérarchique d'une spécification **L*f*P** est préservée. Cela se traduit par un processus d'assemblage hiérarchique et de composition modulaire de sous-réseaux de Petri. Chaque sous-réseau correspond à une instance personnalisée de patron de traduction. Ce processus d'assemblage est basé sur des règles de composition précises que nous détaillons dans les sections suivantes.

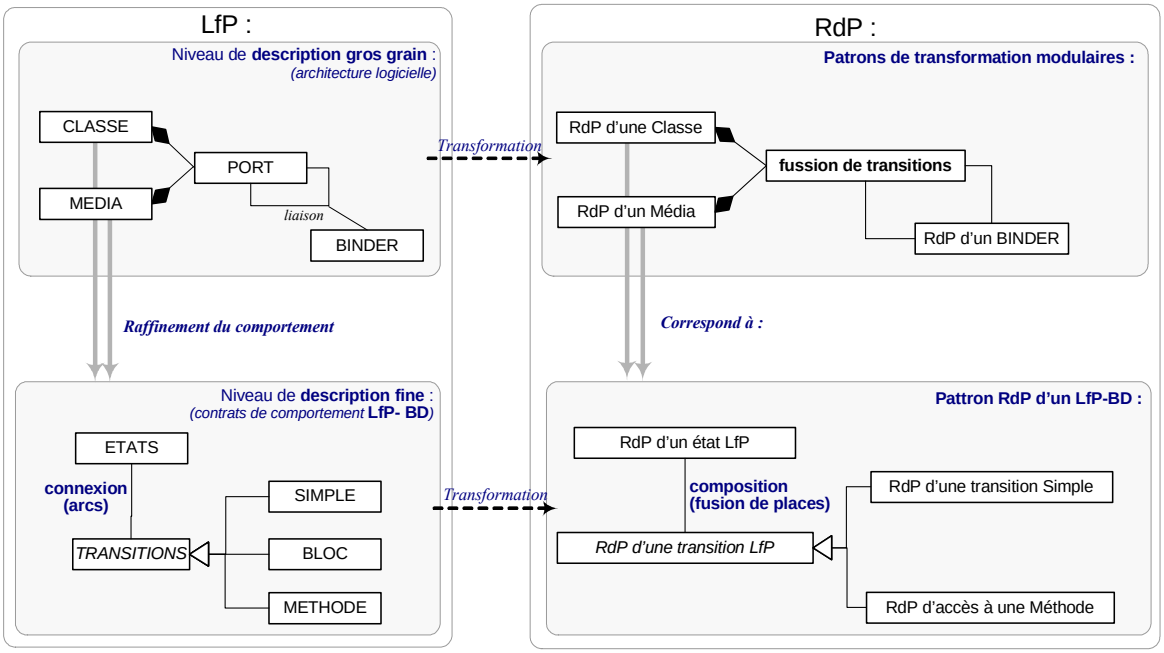


Figure 5-10 : Approche de raffinement des modèles **L*f*P** en réseaux de Petri.

La **Figure 5-10** illustre notre approche de traduction. Ce schéma exhibe les relations existantes entre les entités de modélisation **L*f*P** (sur le schéma de gauche) et la façon dont ces entités sont traduites en réseaux de Petri (sur le schéma de droite).

Nous faisons deux remarques importantes :

- Au niveau gros grain (en haut sur la figure), chaque entité de modélisation **L*f*P** (classes, média et binders) se traduit par un sous-réseau de Petri (module). L'assemblage de ces modules est réalisé par ***fusion de transitions***. Ces transitions correspondent à la traduction R*d*P de Ports **L*f*P**. Nous considérons que ce mécanisme de composition par fusion de transitions est plus adéquat pour la

modélisation des interactions directes synchrones.

- La traduction d'un **L*f*P**-BD (en bas de la figure) raffinant le comportement d'une classe ou d'un média se fait à l'aide de trois patrons de traduction RdP :
  - un pour les états,
  - un pour les transitions simple,
  - un pour les transitions de type méthode.

  Le procédé de composition utilisé ici est par ***fusion de places***. Nous considérons que ce mécanisme est préférable si l'on veut raffiner hiérarchiquement la description du comportement. On parle de raffinement de transitions **L*f*P**.

La traduction en réseaux de Petri (R*d*P) d'un diagramme de comportement **L*f*P**-BD concerne donc quatre aspects :

- la *modélisation des états* **L*f*P**,
- la *modélisation des transitions simples*,
- la *modélisation des transitions de type méthode. Cela concerne l'assemblage hiérarchique* de diagrammes de comportement **L*f*P**-BD.
- *l'assemblage modulaire* des contrats comportementaux (classes, média, binders) par l'intermédiaire des ports.

Nous détaillons chacun de ces points dans les sections suivantes.

5.3.6.1 Modélisation d'états

L'équivalent en réseaux de Petri d'un état **L*f*P** est une place simple.

Domaine de couleurs

Le domaine de cette place correspond au contexte d'exécution visible dans l'état **L*f*P**. Deux cas se distinguent :

- **`<class_ID,instance_ID>`**. Le domaine d'une place correpondent à un état **L*f*P** située dans le **L*f*P**-BD de la classe (du média) est composé d'un couple de deux couleurs dont **`class_ID`** permet de discriminer les classes entre elles, et **`instance_ID`** les instances d'une classe donnée.
- **`<class_ID,instance_ID,invocation_ID>`**. Le domaine d'une place correspondante à un état **L*f*P** interne au **L*f*P**-BD d'une méthode est étendu par une couleur supplémentaire : **`invocation_ID`**. Celle-ci permet de discriminer entre elles les invocations à une même méthode. Cela est nécessaire vu que **L*f*P** permet l'invocation circulaire ou récursive des méthodes.

Marquage

Une telle place R*d*P peut ou non être marquée par des jetons. Chacun d'eux indique l'état d'exécution courant d'une instance de classe. Ces jetons sont tous différents, sans doublons, car chaque identifiant d'instance est unique pour la classe et tout identifiant de méthode est spécifique à une invocation donné.

Notons que :

- La création d'une instance de classe **L*f*P** (active ou protégée) ou celle d'un média à comportement séquentiel correspond à la création d'un jeton dans la place **`BEGIN`** du **L*f*P**-BD de la classe (du média).
- La création d'instances impose également de créer le contexte d'exécution associé (les variables membres d'instance ou de méthode).
- L'enlèvement d'un jeton marquant la place **`END`** du **L*f*P**-BD d'une classe (ou média) correspond à la

destruction d'une instance et du contexte associé.

Exemple

La **Figure 5-11** illustre un exemple simple de raffinement. La partie de gauche présente deux diagrammes **L*f*P**-BD : celui d'une classe active et celui du service `M1()` de la même classe. La partie de droite présente la traduction équivalente en R*d*P. Pour simplifier, la structure des sous-réseaux correspondantes aux transitions `T`, `T1` et `T2` est ignorée (elle est remplacée par un nuage). Notons que :

- Chaque place **L*f*P** donne lieu à une place R*d*P homonyme. Les états `BEGIN` et `END` de la méthode `M1` sont fusionnés avec les états `BEGIN` et `S` de la classe.
- Les domaines des places `BEGIN`, `S` et `END` correspondent au premier scénario de traduction. En revanche celui de la place `S1`, interne à la méthode `M1()`, correspond au modèle second.

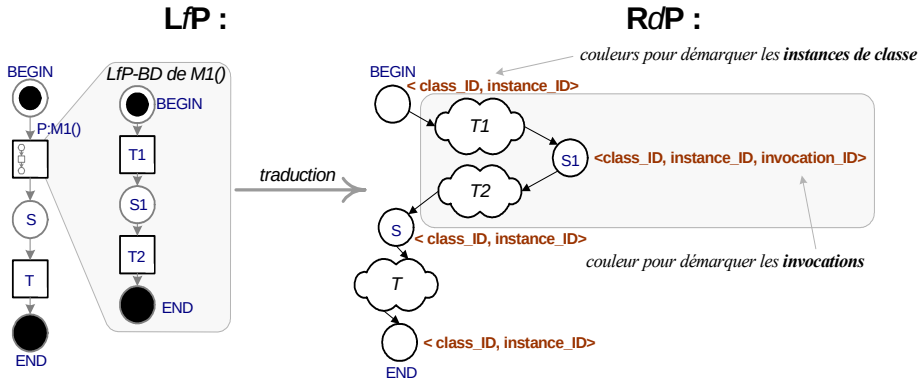


Figure 5-11 : Traduction en réseaux de Petri d'un état **L*f*P**.

5.3.6.2 Principes de raffinement pour les transitions simples

Une transition-simple identifie une suite d'actions en pseudo-code **L*f*P**.

Le raffinement en réseau de Petri d'une transition simple correspond à un sous-réseau R*d*P séquentiel borné par deux places dont la place en entrée correspond à l'état **L*f*P** précèdent l'exécution et la place en sortie à l'état **L*f*P** d'après l'exécution.

Ce sous-réseau est composé d'un ou plusieurs segments enchaînés dont :

1) *Un segment de garde (le premier) qui protège l'exécution*. Celui-ci est composé d'une simple transition R*d*P. Son rôle est de mettre en l'œuvre les règles de franchissement exposées en **Section 5.3.5**.

2) *Zéro ou plusieurs segments de traitement (suivants)*. Chacun d'eux modélise une instruction **L*f*P**. Celle-ci est raffinée par un morceau du sous-réseau de Petri dont :
   - les transitions correspondent à des actions élémentaires (e.g. accès à une variable, application d'un opérateur, test et changement d'état etc.),
   - les places correspondent soit à un état d'exécution intermédiaire soit à l'état précédant (ou suivant) l'exécution de l'instruction.

3) *Un segment de test en sortie (en dernier)*. Celui-ci a pour rôle de vérifier la véridicité des post conditions **L*f*P**. La post condition ne peut pas référencer des sémaphores vu qu'elle exprime une simple assertion.

La structure du sous-réseau R*d*P obtenu par raffinement d'une transition **L*f*P** simple est schématisée dans la **Figure 5-12**. Plusieurs précisions sont nécessaires :

- Au minimum, cette structure se réduit à une simple transition R*d*P (la partie de synchronisation) qui permet, tout simplement, de faire avancer l'état d'exécution.

- L'évaluation de la garde et de la post condition **L*f*P** ainsi que l'exécution de chaque instruction de base sont atomiques (une par une) en raison de l'atomicité systématique des transitions du R*d*P.
- Les domaines de couleurs des places R*d*P dépendent de l'endroit de déclaration de la transition **L*f*P**. Si déclaré dans le contrat d'exécution d'une classe **L*f*P** ce domaine correspond au couple <*class_ID, instance_ID*> démarquant l'instance courante. Si déclaré dans le **L*f*P**-BD d'une méthode ce domaine correspond à une invocation <*class_ID,instance_ID,invocation_ID*>.

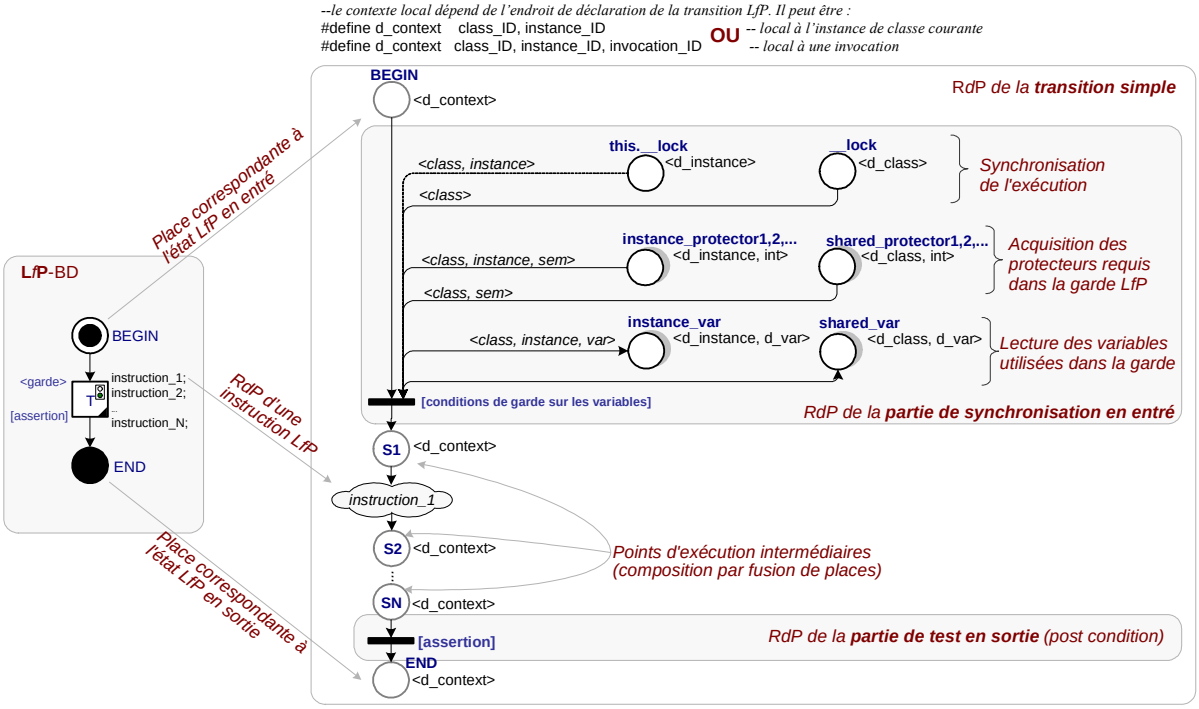


Figure 5-12 : Le patron de traduction R*d*P d'une transition - simple.

Principes de raffinement des instructions L*f*P

Les instructions **L*f*P** n'ont pas toutes la même importance. En effet, certaines (e.g. les boucles) constituent des facilités de modélisation qui peuvent être obtenues aussi par modélisation directe à l'aide d'un **L*f*P**-BD. L'extension du langage est facilitée vu que chaque nouvelle instruction **L*f*P** s'insère par macro composition dans le réseau correspondant à la transition.

Raffinement de l'instruction 'new' pour la création d'instances

Nous détaillons le raffinement des instructions d'instanciation (**`new`**) qui correspond à un cas spécial. La **Figure 5-13** présente la structure du sous-réseau R*d*P obtenu par raffinement de l'instruction **`new`**, l'instruction d'instanciation.

Cette opération consiste à réaliser (d'une façon atomique) quatre actions :

- récupérer un identifiant d'instance non utilisé (**`target_instance`**),
- initialiser le **L*f*P**-BD de l'instance cible si celle-ci est active ou protégée,
- initialiser le contexte local de l'instance cible,
- sauvegarder l'identifiant de l'instance créée dans la variable locale (ici **`test`**).

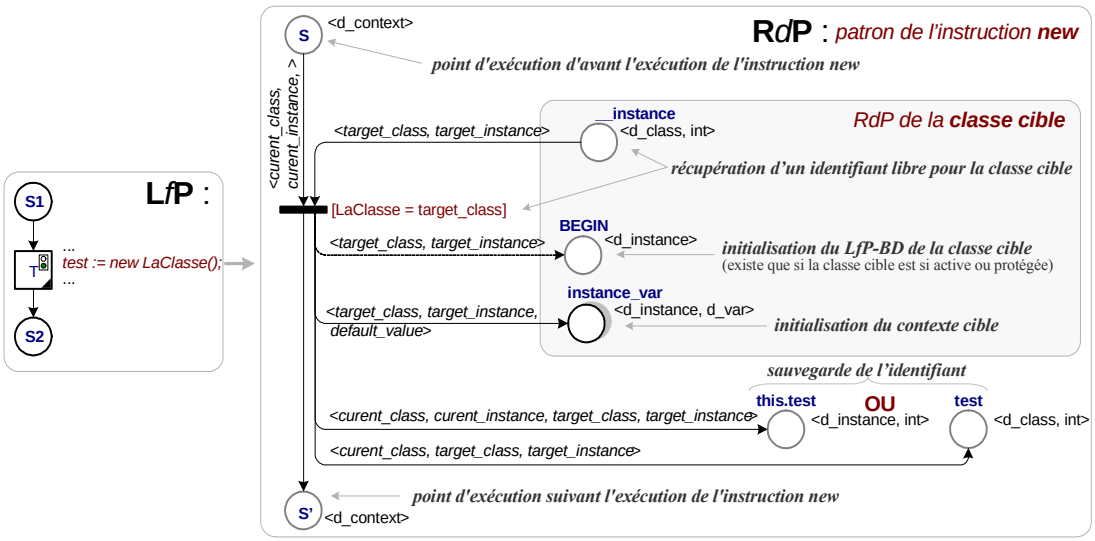


Figure 5-13 : Patrons de modélisation R*d*P pour la création d'instances.

5.3.6.3 Principes de raffinement pour les transitions de type méthode

Une transition-méthode indique un point d'accès gardé à un service public.

Ce type de transition se raffine en un sous-réseau de Petri séquentiel borné par deux places (une en entrée et une en sortie) qui correspondent à l'unique état en entrée et à l'unique état en sortie de la méthode. Cette structure est organisée en trois couches (voir la **Figure 5-14**) :

1) *Une partie de synchronisation en entrée*. Elle correspond à l'acceptation d'une invocation si les règles d'exécution mentionnées en **Section 5.3.5** sont satisfaites.

   Cette partie est répliquée pour chaque tuple *<état* **L*f*P** *en entrée, port en entrée, garde>*. Elle correspond à une *transition* R*d*P *partagée* (un rendez-vous) entre le sous-réseaux de la classe (le média) et le réseau du binder de liaison. L'assemblage modulaire de classes, de média et de binders est basé sur une hypothèse de composition forte, par fusion de transitions.

   Dans le sous-réseau de la classe, cette transition R*d*P fait la liaison entre la place R*d*P en entrée correspondant à l'état **L*f*P** précédant l'invocation) et le corps de la méthode (l'état **L*f*P** précédant son exécution). Ce dernier est unique.

2) *Une partie de traitement*. Elle correspond à un sous-réseau de Petri séquentiel obtenu par la traduction du **L*f*P**-BD de la méthode. Cette partie est commune (partagée) pour toutes les invocations que cela soit en local ou de l'extérieur de la classe, c'est-à-dire provenant d'un port.

3) *Une partie de synchronisation en sortie* (optionnelle). Elle permet de synchroniser l'appelant en fin de traitement. Cette partie est absente dans le cas de méthodes asynchrones (triggers **L*f*P**). Une description détaillé de méthodes **L*f*P** est donné en **Section 5.5.2**.

   Sa structure est quasi identique à celle de la partie de synchronisation en entrée. Cette structure correspond toujours à une transition R*d*P d'interfaçage partagée entre la classe (le média) et le binder de liaison. Cette transition est répliquée pour chaque trinôme *<état* **L*f*P** *en sortie, port en sortie, post condition>*. Elle fait la liaison entre l'état après l'exécution du corps de la méthode et l'état suivant le retour.

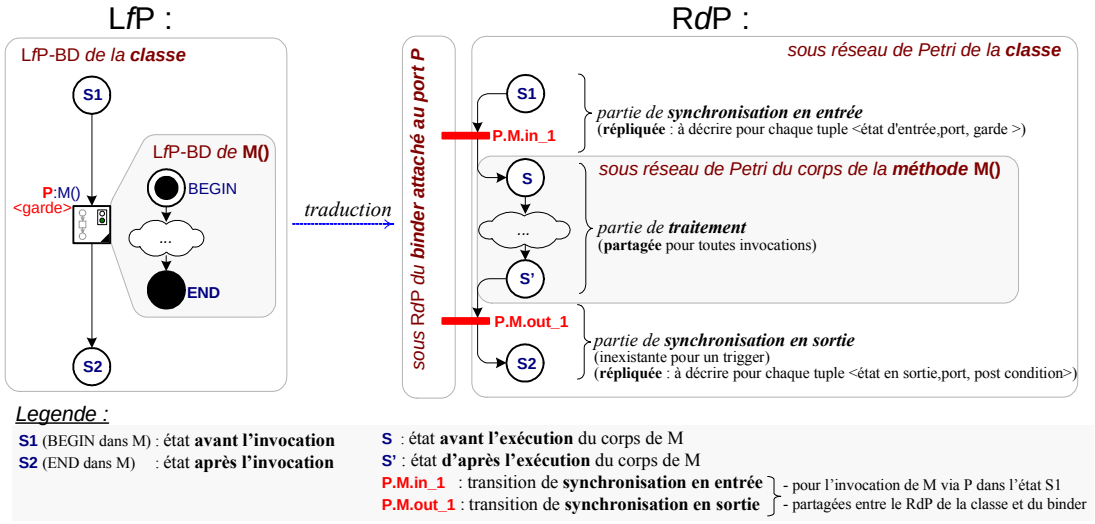


Figure 5-14 : Le patron de traduction R*d*P d'une transition-méthode.

La **Figure 5-15** raffine la structure de la partie de synchronisation en entrée. Ce raffinement respecte plusieurs règles de transformation :

1) *Places en entrée*. La transition de synchronisation est connectée en entrée à plusieurs places R*d*P. Celles-ci sont :
   - la place correspondant à l'état **L*f*P** de la classe précédant l'invocation (`S`).
   - la place correspondant à l'état **L*f*P** du binder précédant l'invocation (`SB`).
   - la place `this.__lock` (le sémaphore de synchronisation de l'instance de classe) ou `__lock` (le sémaphore de synchronisation de la classe) si la méthode est déclarée protégée (`protected`) ou si elle est déclarée protégée et partagée (`protected shared`).
   - si la garde **L*f*P** précise des sémaphores : la place R*d*P du sémaphore requis.
   - si la garde précise des tests sur des variables : la place R*d*P correspondant à la variable **L*f*P**.
   - la place `__Invocations`. Celle-ci joue le rôle d'un conteneur de jetons disponibles, à prendre comme « badge » pendant l'invocation d'une méthode afin identifier l'invocation courante.

2) *Places en sortie*. Deux places sont précisées : l'une pour la classe et l'autre pour le binder associé. Ceux-ci seront marquées lorsque l'invocation est accepté (la transition de synchronisation est franchie). Elles correspondent à l'état précédent l'exécution (dans la classe) et suivant l'acceptation de l'invocation (dans le binder).

3) *Garde*. Elle précise plusieurs conditions à satisfaire d'une façon simultanée. Ces conditions portent sur :
   - la valeur du discriminant : Ne sont acceptées que les invocations qui ciblent l'instance courante.
   - la signature de la méthode (nom).
   - les valeurs des variables **L*f*P** ou celui des arguments en entrée.
   - la priorité d'évaluation par rapport aux autres entrées partant du même état d'entrée. Cette priorité est calculée en appliquant les règles énoncées en **Section 5.3.5**.

Nous précisons deux remarques de conception importantes :
   - Le domaine `d_invocation` de la place en entrée (côte binder) modélise un conteneur capable d'encapsuler toutes les invocations acceptables sur le port donné. Cela revient à maximiser la signature de services invoqués au moyen d'une structure commune. Celle-ci est obtenue par l'union

des domaines de chacune des signatures véhiculées.

- Les couleurs **`c_caller`** et **`c_port`** sont utilisées pour la gestion d'invocations. Elle permettent identifier la provenance d'une invocation afin de restituer (démultiplexeur) en fin d'exécution le résultat aux initiateurs. Pour une invocation en local on n'utilise que l'identifiant de l'appelant et on ignore le port. Au contraire, pour une invocation externe provenant d'un port, on sauvegarde son identifiant et on ignore l'identifiant de l'appelant qui n'a aucun sens.

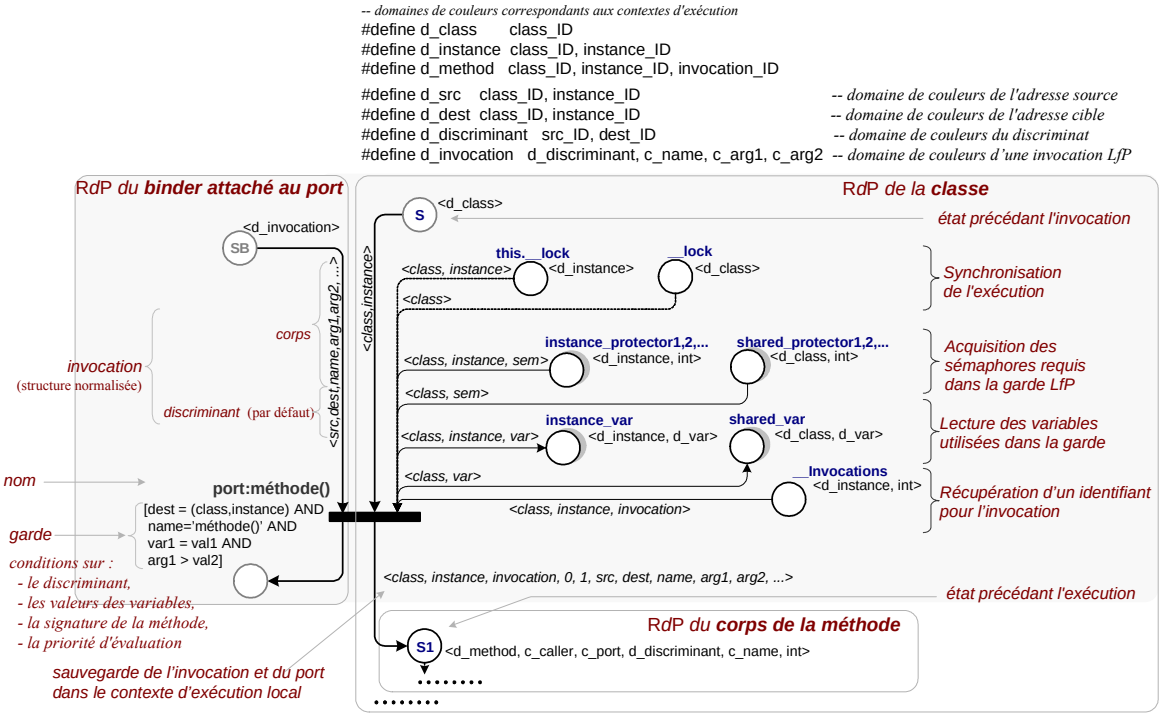


Figure 5-15 : Le patron de traduction de la partie de synchronization en entrée.

Le patron de traduction pour la transition de synchronisation en sortie à une structure quasi identique à celle de la transition en entrée. Sa description n'est donc pas détaillée içi.

5.3.6.4 Assamblage modulaire de L*f*P-BD

Comme précisée dans l'introduction de cette section, l'assemblage des modules R*d*P correspondantes aux classes, aux média et aux binders **L*f*P** se fait par fusion de transitions. Ces transitions correspondent aux Ports de communication. On considère donc une hypothèse de composition forte adéquate à l'invocation directe d'opérations.

5.4 Types de données et variables L*f*P

5.4.1 Système de typage

Pour le système de typage **L*f*P** nous nous sommes particulièrement inspirées de :

- Promela **[55]** constitue un excellent point de départ. Ce langage est muni d'un vérificateur efficace SPIN par model-checking. Il offre un système de typage proche de celui du langage C, mais il semble moins souple pour la modélisation des types de données de plus haut niveau car son système d'encapsulation est purement structurel. Ainsi, Promela permet la déclaration de struc-

tures sans pour autant permettre d'imposer des contrats d'utilisation.

- UML **[90]** et CORBA-IDL **[89]** proposent des systèmes de typage souples, orientés objets, faciles à implémenter dans un langage de programmation cible. Cependant, ces approches sont inadaptées à la vérification formelle car ils souffrent de deux inconvénients : l'un de spécification et l'autre de faisabilité. Primo, UML et IDL ne formalisent que la partie statique du système de typage (hiérarchie de types, domaines des valeurs, signatures des opérateurs prédéfinis) sans préciser pour autant les aspects sémantiques sur l'accès aux données (e.g. atomicité, concurrence d'accès). Secundo, les types proposés sont souvent trop complexes pour permettre la vérification formelle. La taille importante des domaines peut rendre prohibitive l'emploi des certaines techniques de model-checking en raison de l'explosion combinatoire de l'espace d'états lors de la vérification.
- Ada95 **[60]** se distingue par un système de typage fort qui permet, entre autres, la déclaration de types génériques, paramétrables. Cette facilité ainsi que le typage syntaxique (deux types définis d'une façon identique sont incompatibles) nous intéressent.

Notre proposition, qui s'inspire des approches précédents, repose sur une modélisation à deux niveaux d'abstraction :

- *Un noyau de types prédéfinis paramétrables* adaptés aux méthodes de vérification formelles,
- *Un système d'encapsulation de plus haut niveau* basé sur les Classes **L*f*P**. Ce système permet, si besoin, la modélisation des types de données complexes.

Notre idée est de formaliser à la fois les aspects *statiques* (structuraux) et *dynamiques* (comportementaux) dans les définitions de types. Cela revient à préciser :

- Un *domaine de valeurs admissibles* (constantes symboliques),
- Une *liste d'opérateurs et de fonctions prédéfinies*. Cela vise la définition de signatures, de fonctions de transfert (applications entre des domaines sources et des domaines cibles) et de la sémantique d'accès aux les données.

5.4.1.1 Types prédéfinis

La **Figure 5-16** présente la hiérarchie de types prédéfinis en **L*f*P** au moyen d'un diagramme de classes UML.

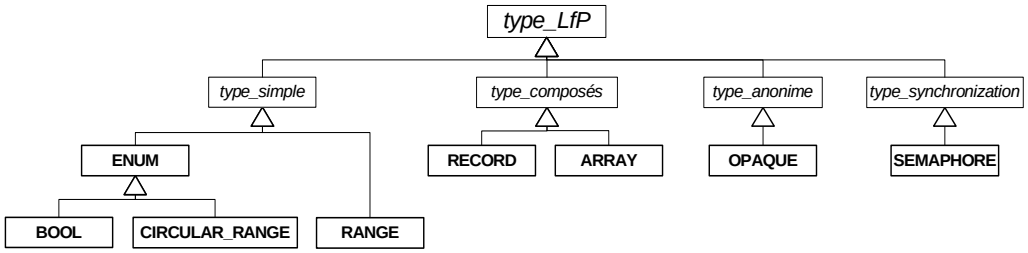


Figure 5-16 : La hiérarchie de types prédéfinis en **L*f*P**.

L*f*P est dédié à la modélisation de systèmes ayant un espace d'états d'exécution fini. C'es pour cela que tout type **L*f*P** est de *domaine discret fini* constitué *d'éléments* (littéraux d'énumération) *sans doublons*.

Nous proposons les catégories suivantes :

- *Types simples*. Ils sont utilisés pour la déclaration de variables discrètes monovaluées. Comme une majorité des langages de programmation, **L*f*P** supporte la définition de types énumérés (**`ENUM`**), du type booléen (**`BOOL`**) ainsi que celle d'intervalles d'éléments circulaires (**`CIRCULAR_RANGE`**) ou non circulaires (**`RANGE`**).
- *Types composés*. Ils servent pour la construction de structures des données et celle de tableaux

associatifs. Ces derniers sont vus comme des fonctions discrètes mutables, c'est-à-dire qu'il est possible de modifier dynamiquement la valeur retournée (la fonction de transfert).

- *Types anonymes*. Le type **`OPAQUE`** et un conteneur qui permet d'encapsuler les données d'une manière opaque. Il permet d'ignorer localement le type d'une variable si celle-ci n'a pas d'influence sur le contrôle de l'application.
  Cela permet également de faire la liaison entre la partie contrôle d'une application répartie et les aspects de calcul. Les variables utilisées pour le calcul ne peuvent pas intervenir dans le contrôle de l'application et la partie calcul ne peut ne peut pas modifier les valeurs de variables de contrôle ce qui est crucial pour garantir la pertinence de la vérification.
- *Types de synchronization*. Le type **`SEMAPHORE`** est l'unique type de synchronisation. Sa sémantique est proche à celle des sémaphores POSIX. Cependant, au contraire de ces derniers, un **`SEMAPHORE`** **L/P** permet de fixer à la modélisation (et non à l'implémentation) la politique d'accès à une ressource partagée (e.g. section critique ou variable statique).

Opérateurs

A part les fonctions de lecture (**`get`**) et d'écriture (**`set`**), l'opérateur d'affectation (**`:=`**) est le seul opérateur prédéfini.

Le type énuméré

Le type **`ENUM`** modélise un type énuméré Ada95. Il précise un *ensemble discret et fini d'éléments* (littéraux d'énumération) *pourvu d'une relation d'ordre total*. La fonction successeur (**`succ`**) est implicitement fixée par l'ordre d'énumération, elle est circulaire, c'est-à-dire que la borne minimum (**`first`**) est le successseur de la borne maximum (**`last`**).

Ce type est générique, l'utilisateur à l'obligation de paramétrer explicitement le domaine d'énumération (**`domain`**) ainsi qu'une valeur d'initialisation pas défaut (optionnelle).

Plusieurs opérateurs sont prédéfinis (voir **Figure 5-17**). On y trouve des opérateurs de test, de comparaison ainsi que des fonctions pour le calcul du successeur, du prédécesseur, du minimum ou du maximum. La fonction **`rand`**, quant à elle, est utilisée pour générer une valeur aléatoire dans l'intervalle. La distribution de valeurs est uniforme.

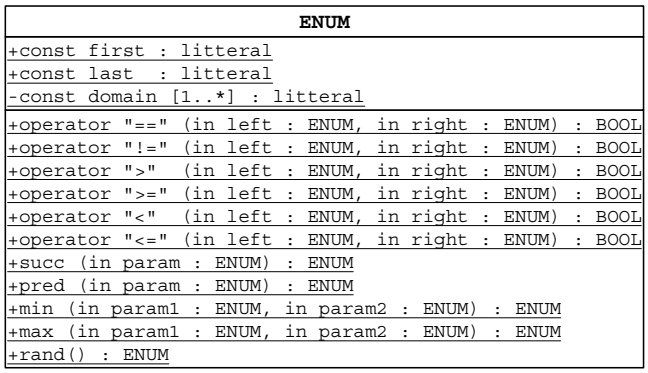


Figure 5-17 : Le type énuméré.

La **Figure 5-18** donne un exemple de déclaration pour un type énuméré concret. La syntaxe utilisée, est similaire à celle du langage Ada95. Dans l'exemple, **`JOUR`** constitue un type énuméré de domaine de **`lundi`** à **`dimanche`** dont **`lundi`** constitue la valeur pas défaut.

```
type JOUR is {lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche}:= lundi;
```

Figure 5-18 : Déclaration d'un type énuméré concret.

Le type booléen

Le type booléen (**BOOL**) est utilisé pour représenter des valeurs en logique binaire. Ce type constitue un type concret et, par conséquence, il peut être instancié au moyen des variables.

Son domaine est restreint à deux littéraux d'énumération {**FALSE, TRUE**} dont **FALSE** est la valeur d'initialisation par défaut.

Dérivé du type énuméré, le type booléen propose plusieurs opérateurs logiques classiques utilisables pour le calcul des prédicats (voir **Figure 5-19**).

BOOL
+operator "NOT" (in right : BOOL) : BOOL
+operator "OR" (in left : BOOL, in right : BOOL) : BOOL
+operator "AND" (in left : BOOL, in right : BOOL) : BOOL
+operator "XOR" (in right : BOOL, in left : BOOL) : BOOL

Figure 5-19 : Le type booléen (**BOOL**).

Les types intervalles circulaires d'entiers

Le type **CIRCULAR_RANGE** constitue le support pour la déclaration des types énumérés ayant comme domaine l'intervalle fermé d'entiers contigus [**first**, **last**]. Comme pour le langage C, le domaine des valeurs est considéré comme cyclique, c'est-à-dire que la fonction successeur permet de passer de la borne maximum à la borne minimum.

Ce type est générique, l'utilisateur souhaitant préciser un type concret doit préciser les bornes minimum et maximum de son domaine ainsi qu'une valeur d'initialisation par défaut.

Comparé au type «entier universel» proposé par Ada95, le type **CIRCULAR_RANGE** est plus souple. En effet, ce type n'impose pas des restrictions sur la taille du domaine qui se limite en Ada95 à un support sur 32 bits.

Ce type étend le type énuméré au moyen de plusieurs opérateurs arithmétiques (voir **Figure 5-20**). Parmi eux, l'opérateur modulo n'est défini que pour le cas d'intervalles incluant l'origine (zéro). La division, quant à elle, n'est pas supportée par ce type vu qu'il n'est pas possible d'en détecter les erreurs de calcul.

CIRCULAR_RANGE
+operator "+" (in left : CIRCULAR_RANGE, in right : CIRCULAR_RANGE) : CIRCULAR_RANGE
+operator "-" (in left : CIRCULAR_RANGE, in right : CIRCULAR_RANGE) : CIRCULAR_RANGE
+operator "-" (in right : CIRCULAR_RANGE) : CIRCULAR_RANGE
+operator "*" (in left : CIRCULAR_RANGE, in right : CIRCULAR_RANGE) : CIRCULAR_RANGE
+operator "**"(in left : CIRCULAR_RANGE, in right : CIRCULAR_RANGE) : CIRCULAR_RANGE
+mod(in param1 : CIRCULAR_RANGE, in param2 : CIRCULAR_RANGE) : CIRCULAR_RANGE

Figure 5-20 : Le type intervalle circulaire d'entiers (**CIRCULAR_RANGE**).

Figure 5-21 présente un exemple de déclaration d'intervalle personnalisé representant les jours du mois.

```
type JOUR_DU_MOIS is CIRCULAR RANGE 1..31;
```

Figure 5-21 : Déclaration d'un type intervalle circulaire concret.

Le type intervalle étendu d'entiers

Le type intervalle d'entiers étendu (**RANGE**) a été élaboré afin de permettre la détection par model-checking des éventuelles erreurs de conception (e.g. valeurs interdites, variables non initialisées) ou de calcul (e.g. débordement d'intervalle, division par zéro etc.). Ce type permet de simplifier la description des mécanismes d'exception explicites.

L'idée proposée est de modifier légèrement le type **`CIRCULAR_RANGE`** pour modéliser les conditions d'erreur. Deux modifications sont apportées :

- le domaine contigu d'entiers [**`first`**, **`last`**] est étendu par des littéraux d'énumération signalant des valeurs d'erreur;
- les fonctions de transfert de chaque opérateur ou fonctions membres sont redéfinies afin de prendre en compte les conditions d'erreur.

Ainsi, à l'exception de la division qui est nouvelle, ce type exhibe une interface identique à celle proposée par le type **`CIRCULAR_RANGE`**. Cependant, malgré leur compatibilité structurelle, ces deux types sont sémantiquement incompatibles. En effet, le type **`RANGE`** ne peut plus être considéré comme dérivé d'une énumération vu que la fonction successeur n'est plus définie d'une façon circulaire.

Pour faciliter la conception, **L/P** propose plusieurs valeurs d'erreur :

- **`ndef`** modélise la valeur non définie, considérée comme valeur d'initialisation par défaut.
- **`div`** spécifie une erreur de division par zéro (diviseur nul).
- **`oflow`** précise une erreur de débordement vers la borne maximum.
- **`uflow`** explicite une erreur de débordement vers la borne minimum.
- **`error`** exprime une condition d'erreur sans pour autant l'expliciter son type (division, non initialisation, dépassement de borne). Cette valeur est obtenue lorsqu'on propage l'erreur d'une opération à une autre (e.g. 1 + ndef => error).

Ce mécanisme peut être étendu par des étiquettes utilisateur. Le prix à payer pour cette facilité est la nécessité de redéfinir explicitement la fonction de transfert de chaque opérateur.

La **Figure 5-22** présente deux déclarations. Le type **`JOUR_DU_MOIS`** préalablement défini comme circulaire dans **Figure 5-18** est redéfini. Son domaine [**`1`**, **`31`**] est étendu par la valeur **`ndef`** qui modélise la valeur d'initialisation par défaut. Le deuxième exemple définit un type externe,

```
type JOUR_DU_MOIS is RANGE 1..31 := ndef;
```

Figure 5-22 : Déclaration d'un intervalle étendu.

Type dérivés

Pour faciliter le développement, **L/P** prédéfinit plusieurs types dérivés concrets (voir **Figure 5-23**). Ceux-ci correspondent à des entiers signés ou non ayant le support codé sur 8, 16 ou 32 bits.

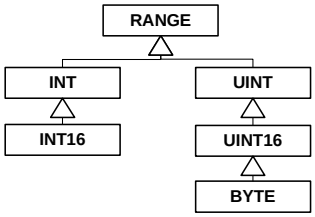


Figure 5-23 : Types **`RANGE`** prédéfinis par **L/P** : Les entiers sur 8, 16 et 32 bits (signés ou non).

Le type RECORD

Le type **`RECORD`** définit un mécanisme d'encapsulation de base : il définit une agrégation statique de champs (i.e. variables types). Ce type est similaire à un **`RECORD`** Ada95 ou à une structure C.

Sa définition est donné en **Figure 5-24**. Le domaine d'un type **`RECORD`** est *composé* : il est constitué par le produit cartésien des sous domaines correspondent aux champs encapsulés. Quatre opérateurs sont prédéfinis. Ainsi, les test d'égalité et d'inégalité sont par valeur. L'opérateur «.», quant à lui, permet de qualifier l'accès à un champ dans la structure. La définition de ce dernier est double, cela se justifie par

son utilisation «à gauche» ou «à droite» de l'affectation permetant l'assignation et respectivement la lecture d'un champ.

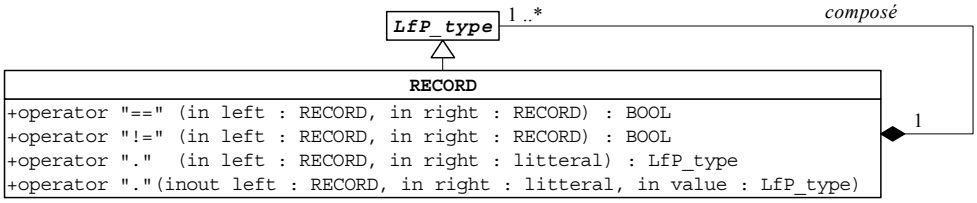


Figure 5-24 : Le type **RECORD**.

La **Figure 5-25** illustre deux exemples de déclarations équivalentes. Le type **POINT** est une structure composée de deux entiers **x** et **y** initialisées par défaut à zéro.

```
type POINT is RECORD {
   INT x := 0;
   INT y := 0;
};
```

OU

```
type POINT is RECORD {
   INT x;
   INT y;
} := (0,0);
```

Figure 5-25 : Déclaration d'un type RECORD concret.

Le type tableau (ARRAY)

Le type **ARRAY** est la base pour la déclaration des variables multivaluées (tableaux associatifs). Il définit une application entre un domaine source (une liste d'index) et un domaine cible (un type **L*f*P**). Ce type est sémantiquement compatible avec les tableaux Ada95.

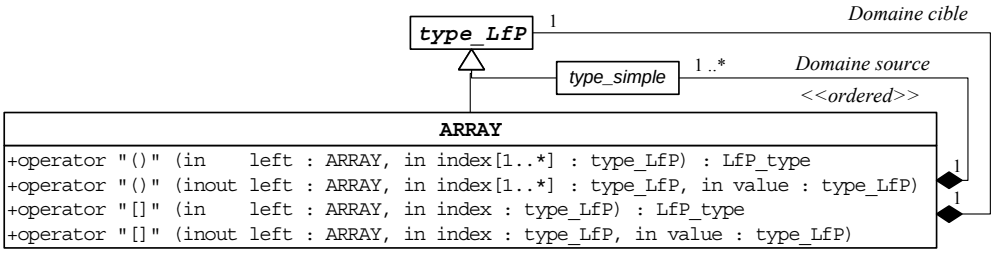


Figure 5-26 : Le type ARRAY

La **Figure 5-26** ilustre la définition statique du type **ARRAY** au moyen d'un diagramme de classes UML. On utilise les deux opérateurs usuels : les crochets et les parenthèses. Ces deux opérateurs sont équivalents, ils permettent l'accès indexé aux enregistrements d'un tableau. L'emploi des parenthèses permet de voir un tableau comme une fonction «mutable» (fonction pour laquelle on peut modifier l'application x->y en cours d'exécution). La définition de ces opérateurs est double afin de prendre en compte leur utilisation «*à gauche*» ou «*à droite*» de l'affectation, pour écrire ou lire la valeur d'un élément.

Le dépassement du domaine d'indexation par une variable d'indexation ne pose pas de problèmes de conception vu que toute valeur erronée (qui sort du domaine d'indexation) est détectée lors de la vérification formelle. C'est à la charge de l'utilisateur de prévoir soit de mécanismes d'exception explicites ou d'utiliser celles qui existent dans les langages de programmation ciblés pour l'implémentation. Ce problème concerne la génération automatique de programmes.

La **Figure 5-27** illustre plusieurs exemples de déclarations de tableaux. Cet exemple montre également le mode d'emploi pour les opérateurs d'indexation (les crochets) et application (les parenthèses).

```
--déclarations
type t_tab is ARRAY (INT) of INT;                  -- un type tableau : 2^32 éléments de type entier
tab1 : ARRAY (INT) of {"A", "B"};                  -- un tableau non initialisé
tab2 : ARRAY (BYTE RANGE 0..1) of BOOL := ((0,FALSE),(1,TRUE)); -- un tableau initialisé
tab3 : ARRAY (JOUR, INT RANGE 1..10) of POINT; -- un tableau bidimensionnel non initialisé

--utilisation des operateurs [] et () pour lire ou écrire la valeur d'un enregistrement
tab1[3] := "A";                    <-- notations équivalentes -->    tab1(3) := "A";
variable := tab3["lundi"][1];                                        variable := tab3("lundi",1);
```

Figure 5-27 : Déclarations de tableaux associatifs.

Le type OPAQUE

Le type **`OPAQUE`** est un type d'encapsulation anonyme qui permet d'ignorer localement le type effectif d'une variable si celle-ci n'a pas d'influence directe sur le contrôle de l'application. Par exemple, dans un média, le corps d'un message routé peut être déclaré comme opaque.

L'introduction du type opaque est justifiée par deux constats :

- *La réutilisation des modèles et la modélisation sont facilitées si l'on considère une approche de modélisation modulaire à base de modules faiblement couplés par leur conception*. Cela permet de concevoir une application par une approche de composition de modules. Tout module logiciel (Classe, média, Opération) est conçu en isolement. La construction d'une application est vue comme un processus d'assemblage et configuration ultérieur.
- *La vérification des modèles peut être simplifiée si l'on ignore les aspects de modélisation qui ne concernent pas le contrôle de l'application*. L'idée est de séparer la partie contrôle par rapport au traitement sur les données et de se focaliser sur la vérification formelle de la première.

Ce type n'introduit pas d'opérateurs spécifiques et il ne peut pas être initialisé car son domaine est non spécifié. Le seul opérateur applicable est donc l'affectation, l'opérateur commun à tout type **L*f*P**.

Figure 5-28 presente un exemple de déclaration de type opaque (**`msg`**).

```
type msg is OPAQUE;
```

Figure 5-28 : Déclaration d'une variable **`OPAQUE`**.

Le type SEMAPHORE

Le type **`SEMAPHORE`** est un type de synchronisation. Il permet de définir des mécanismes de synchronisation pour l'accès concurrent à une ressource partagée (variable statique ou section de code critique). Sa définition est présentée en **Figure 5-29**.

SEMAPHORE
-value : UINT = capacity
+p(in value : UINT, in blocking : BOOL = True) : BOOL +v(in value : UINT)

Figure 5-29 : Le type **`SEMAPHORE`**

Les primitives définies par un **`SEMAPHORE`** sont similaires à celles d'un sémaphore POSIX : **`P`** permet d'acquérir l'accès à une ressource partagé, **`V`** permet de libérer en sortie l'accès. La sémantique d'invocation de **`P`** peut être bloquante ou non, celle de **`V`** est toujours atomique et non bloquante.

Cependant, au contraire d'un sémaphore POSIX **[17]** qui dépend dans son comportement de la politique d'ordonnancement système, et qui reste donc à préciser à l'implémentation (comme évoqué dans la **Section 4.4.1** *Systèmes d'exploitation classiques*), **L*f*P** formalise dès l'étape de modélisation le compor-

tement d'un **SEMAPHORE**. Cette spécification précoce nous est apparue nécessaire parce que la vérification formelle ne peut pas se faire sans considérer cet aspect.

La politique d'ordonnancement par défaut est *non déterministe*. Ce choix se justifie pour les raisons suivantes :

- Elle couvre l'ensemble des comportements possibles quelque soit la politique d'ordonnancement réelle.
- Elle s'exprime facilement en réseaux de Petri. Par exemple, si on n'accède à un **SEMAPHORE** qu'en mode bloquant, son expression en R*d*P revient à une simple Place partagée.
- Lors de la vérification formelle il sera possible de synchroniser le modèle de l'application avec celui d'un ordonnanceur spécifique (e.g. équitable).

5.4.1.2 Mécanismes d'encapsulation de plus haut niveau

L'expérience, par exemple celle issue de Modula, de Promela, ou des réseaux de Petri de haut niveau, montre que l'utilisation d'un système de typage basique (composé d'entiers, de types énumérés, de tableaux et de structures de données) est suffisant pour décrire la partie contrôle d'une majorité d'applications en vue de la vérification et l'implémentation. Cependant, puisque les capacités d'extension d'une telle approche sont limitées (elle permet, au plus, de définir des structures des données) nous proposons des mécanismes de typage de plus haut niveau, à la fois formels et extensibles.

C'est ainsi que la notion de classe **L*f*P** peut être utilisée à uns double fin : d'une part pour décrire la partie contrôle d'une application, et d'une autre part pour encapsuler des nouveaux types de données utilisateur. Notons que cette facilité à un prix. En effet, la construction de types de données complexes peut s'avérer pernicieuse pour la vérification formelle vu qu'elle accroît la taille et la complexité des modèles.

La description des classes **L*f*P** est présentée dans la **Section 5.5**.

5.4.1.3 Raffinement d'opérateurs L*f*P en réseaux de Petri

L'application d'un opérateur ou d'une fonction associée à un type prédéfini est répartie sur trois étapes d'exécution successives :

- *lecture des paramètres en entrée*,
- *traitement : application d'une fonction de transfert*,
- *écriture du résultat en sortie*.

Toutes ces étapes ne sont pas obligatoires pour chaque opérateur : certains opérateurs ne précisent pas de fonction de transfert et par conséquent leur exécution se réduit à une simple copie de variables (lecture de paramètres en entrée et écriture du résultat en sortie). Par exemple, l'opérateur d'affectation (:=) correspond à ce cas, s'il n'y a pas de conversion implicite de type, son exécution consiste à copier la valeur de la partie droite dans la partie gauche. Un autre exemple est l'opérateur de qualification (.), qui permet de lire un champ d'une variable structurée et de stocker sa valeur dans une variable intermédiaire.

Dans cette section nous visons exclusivement la modélisation en réseaux de Petri de l'étape de traitement (application de la fonction de transfert). La modélisation R*d*P des étapes d'accès, quant à elle, concerne la manipulation de variables **L*f*P** et est détaillée dans la **Section 5.4.2**: *Variables LfP*.

La modélisation en réseaux de Petri d'un type **L*f*P** ne concerne donc que deux aspects :

- *la déclaration d'un "domaine de couleur"* associé au type **L*f*P**,
- *la modélisation des fonctions de transfert* spécifiques aux opérateurs et aux fonctions prédéfinies

pour le type.

Chaque fonction de transfert est modélisée au moyen d'une place R*d*P. Le nom, le domaine et marquage de cette place correspondent au nom, à la signature et respectivement à l'application définie par l'opérateur modélisé.

La **Figure 5-30** présente un exemple de traduction. La place **`intervalle.plus`** modélise l'opérateur d'addition du type **`intervalle`**. Cet exemple montre également la manière dont cet opérateur peut être appliqué dans les modèles R*d*P, la transition **`t`** modélise l'addition de deux variables **`a`** et **`b`**.

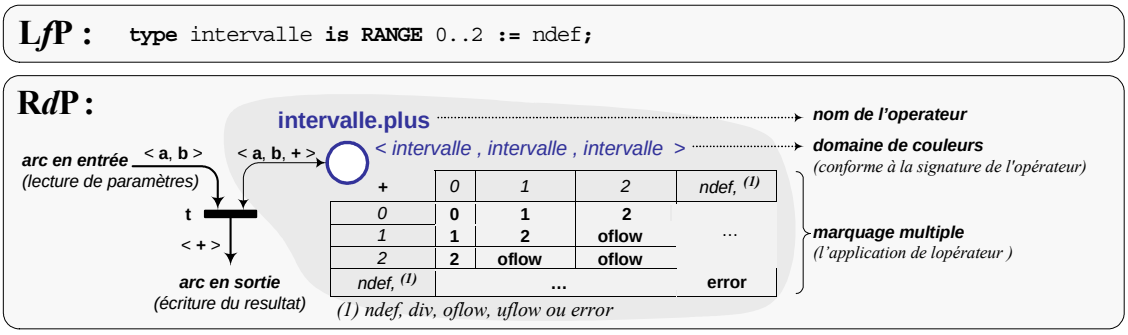


Figure 5-30 : Modélisation R*d*P d'un opérateur **L*f*P** : Un exemple de traduction.

5.4.2 Variables L*f*P

5.4.2.1 Déclaration

Contrairement à d'autres langages de programmation, **L*f*P** interdit la déclaration directe de variables globales. Cette interdiction se justifie pour deux raisons :

- L'emploi de variables globales constitue une mauvaise discipline de programmation qui empêche à la réutilisation modulaire de modèles.
- Le partage direct de variables en reparti introduit des hypothèses implicites sur la sémantique de partage. Cela est incompatible avec l'objectif de **L*f*P** qui impose de modéliser explicitement les interactions à distance au moyen des Média.

La déclaration de variables est autorisée à l'intérieur de Classes et de Média, le rôle de ces derniers est d'imposer des contrats d'utilisation protocolaires et des points d'interaction (Ports **L*f*P**) permettant de réglementer l'accès aux données.

Variables membres (Attributs)

Comme UML, le langage **L*f*P** permet la déclaration d'attributs (variables membres) pour les Classes et les Média. Ceux-ci sont utilisés pour modéliser soit des *variables d'instance* soit des *variables de classe* (marquées **`shared`**) partagées par toutes les instances. Ce partage n'est que local à un composant logiciel et par conséquent aucune hypothèse implicite n'est faite sur la sémantique de communication répartie, qui reste toujours à préciser au moyen des média. Nous imposons que les composants logiciels soient des enveloppes de déploiement et d'exécution indépendantes qui ne partagent pas de ressources (e.g. la mémoire).

Pour protéger l'accès à une variable de classe, **L*f*P** permet de lui associer un verrou (**`SEMAPHORE`** binaire) lors de sa déclaration. Ainsi, la déclaration d'une variable de classe marqué **`protected`** permet de garantir l'atomicité d'accès quel que soit son type. Cette facilité de modélisation permet de simplifier la conception. Ce comportement est similaire à une déclaration *synchronisée* en Java mais pour les variables.

Variables de méthodes

UML ne décrit les classes que par rapport à leur spécification : la structure et le comportement d'une classe sont exprimés au moyen d'une liste de déclarations d'attributs et de méthodes, et d'un automate de Harel simplifié (*StateChart UML*).

La description interne de méthodes soit est limitée à des annotations informelles attachées aux déclarations (on parle des étiquettes UML), soit est modélisée dans le StateChart UML de la classe à l'aide d'états composés. Si la première solution est totalement inadaptée car informelle, la deuxième soufre de deux inconvénients de modélisation :

- La description du comportement des méthodes est agglomérée dans un seul diagramme avec celle du contrat comportemental de la classe. Cela nuit à la lisibilité et empêche la réutilisation modulaire de méthodes.
- UML ne prévoit pas de moyens pour exprimer le contexte local (variables locales) spécifique aux méthodes (voir UML1 .4 la page 3-139 **[90]**).

L*f*P vise à palier ces inconvénients et ainsi, au contraire de UML, il propose de modéliser les méthodes séparément du contrat comportemental de la classe. Cette description est complète car elle exprime non seulement l'automate modélisant le flot de contrôle de la méthode mais également son contexte local.

En **L*f*P** une variable de méthode peut être marquée comme persistante (`static`) ou partagée pour toutes les instances (`shared`). Le mot clef `static` présente une connotation sémantique différente de celle donnée à ce terme par les langages C++ et Java. Celui-ci indique la persistance d'une variable sans pour autant impliquer son partage.

5.4.2.2 Visibilité et cycle de vie

La visibilité d'une variable correspond à une zone du modèle où celle-ci peut être référencée. Ce problème peut être abordé selon deux angles de vue différents : à «l'intérieur» et à «l'extérieur» de la Classe ou du Média d'encapsulation.

Visibilité externe

Par défaut, la visibilité d'un attribut membre d'une Classe ou d'un Média **L*f*P** est privée. L'accès de l'extérieur à sa valeur est cependant possible si l'on définit des méthodes d'accès publiques en lecture (`get`) et respectivement en écriture (`set`).

L'accès aux attributs d'une classe peut être paramétré dynamiquement. En effet, le *contrat d'utilisation* d'une classe permet de varier la visibilité de méthodes publiques (services) et par conséquent l'accès aux attributs. Cette problématique est détaillée dans la **Section 5.5.2**: *Méthodes LfP*.

La **Figure 5-31** illustre un exemple de modélisation concret : une classe protégée (qui propose que des services synchronises) disposant de deux ports `P1` et `P2`. La visibilité de `V`, un attribut d'instance, est assurée par l'intermédiaire de deux services d'instance : `get_V` et `set_V`. Le contrat d'utilisation de la classe précise les conditions de visibilité (voir la **Section 5.5**). Plusieurs restrictions sont précisées :

- L'accès à la variable `V` se fait en exclusion mutuelle car le caractère protégé de la classe interdit tout traitement concurrent. La déclaration d'un contrat d'utilisation n'est possible que dans le cas des comportements séquentiels, c'est-à-dire protégés.
- La variable `V` n'est visible que si la classe se trouve dans l'état `E1`.
- `V` est accessible en lecture quel que soit le port d'invocation (`P1` ou `P2`). En revanche, l'accès en écriture est réservé aux seules invocations provenant du port `P2`.
- Les opérations d'accès ne sont pas privilégiées les unes par rapport aux autres, elles ne bénéficient

que d'une même faible priorité (marquée 1 sur les arcs).

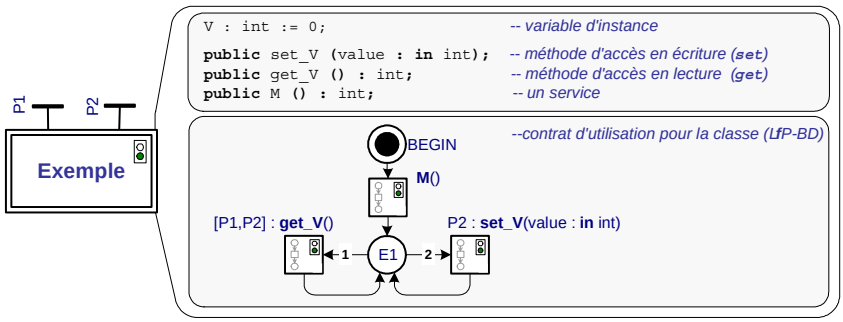


Figure 5-31 : Utilisation de méthodes **L*f*P** pour assurer l'accès de l'extérieur à une variable **L*f*P**.

Visibilité interne et cycle de vie

A l'intérieur d'une Classe ou d'un Média, la visibilité et la cycle de vie d'une variable dépendent à la fois de l'endroit où elle est déclarée (Classe, Média ou Méthode) et du fait qu'elle soit partagée (**`shared`**) ou persistante (**`static`**).

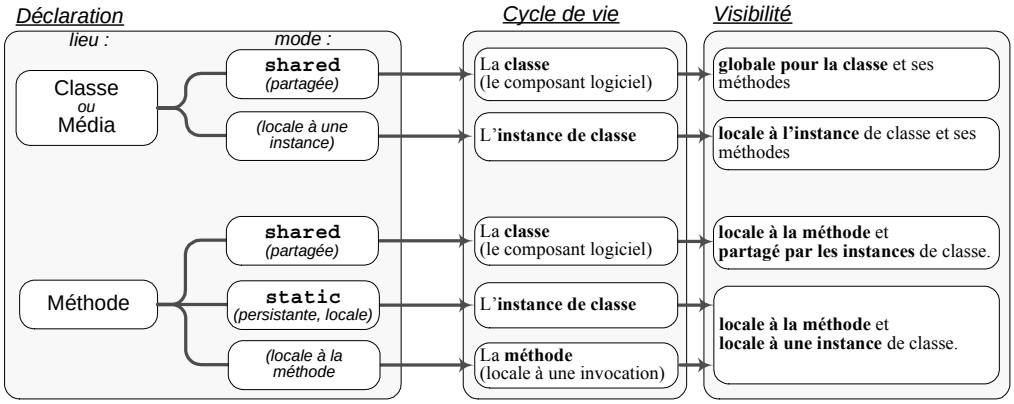


Figure 5-32 : Visibilité interne et cycle de vie d'une variable **L*f*P**.

La **Figure 5-32** présente l'ensemble des cas de visibilité et de durée de vie. Il exhibe plusieurs choix de modélisation que nous avons fait :

1) *Une variable* **L*f*P** *est déclarée strictement dans la zone où elle sera utilisée*.

La Classe, le Média ou la méthode d'encapsulation. **L*f*P** vise donc à faciliter non seulement la réutilisation des Classes et des Média mais également celle des méthodes.

Pour illustrer cette idée nous proposons de comparer l'approche de **L*f*P** à celui de C++ au moyen d'un exemple simple : la déclaration d'une variable persistante **`v`** locale à une méthode **`M`**. Cet exemple est

illustré dans la **Figure 5-33**.

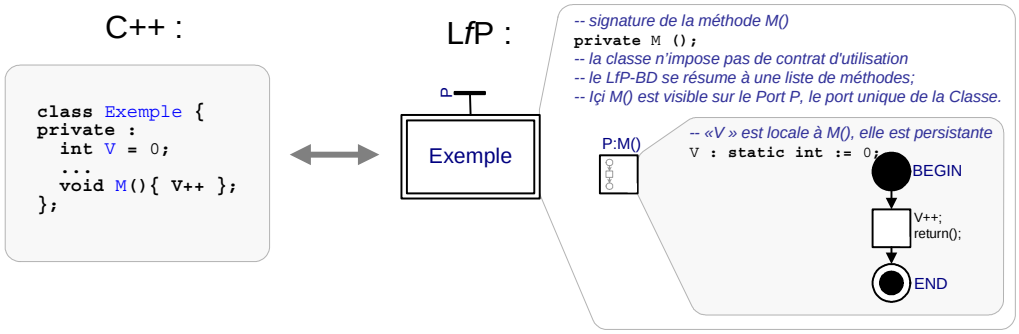


Figure 5-33 : Déclaration d'une variable persistante, locale à une méthode dans une instance de classe.

En C++ : La variable **`V`** se traduit par un *attribut d'instance*. Bien que la persistance de sa valeur soit assurée par sa déclaration externe à la méthode **`M`**, cette déclaration est peu convenable pour deux raisons :

- **`V`** n'est pas locale à la méthode **`M`** et par conséquent elle peut être référencée accidentellement dans une autre méthode membre.
- pour réutiliser la méthode **`M`**, en tant que gabarit de conception, il faut redéclarer la variable **`V`** ce qui détruit la modularité.

En L*f*P : La variable **`V`** est déclarée localement dans la méthode **`M`**. La déclaration **`static`** permet d'assurer la persistance de sa valeur sans pour autant impliquer son partage au niveau de la classe. L'utilisation accidentelle de la variable **`V`** à l'extérieur de la méthode **`M`** est évitée et la réutilisation de la méthode **`M`** comme gabarit de conception est facilitée : les deux inconvénients présentés par C++ disparaissent donc :

2) *Le cycle de vie d'une variable dépend exclusivement de sa déclaration*.

Lors de sa déclaration, une variable **L*f*P** est rattachée à un contexte d'exécution : *global pour la classe*, *local à une instance de classe* ou *local à une invocation de méthode*. La cycle de vie d'une variable est liée à celle de son contexte. Les règles d'appartenance sont illustrées dans **Figure 5-32**, elles sont similaires à ce qu'on trouve dans le langage Java.

Exemple

Pour illustrer la mise en œuvre de ces règles de visibilité et de gestion du cycle de vie des variables nous proposons un exemple pédagogique. La **Figure 5-34** montre le cas d'une classe passive (sans contrat d'utilisation) pourvue de plusieurs déclarations de variables.

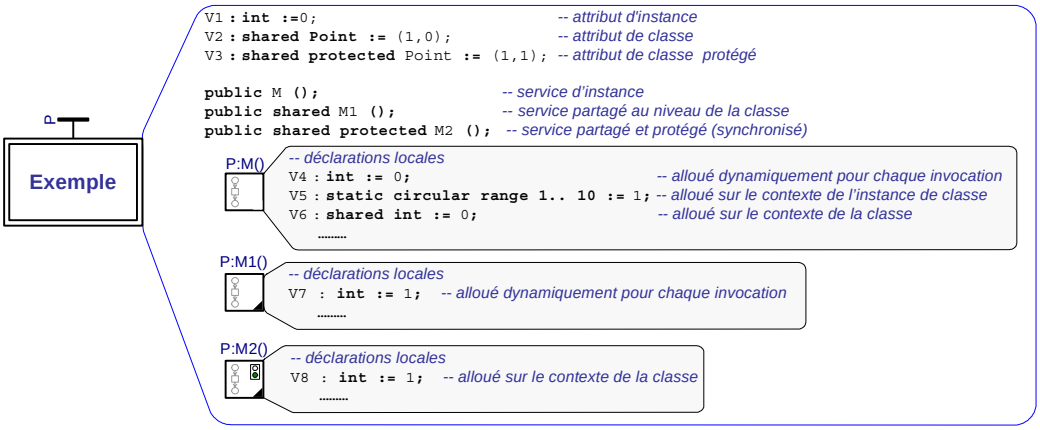


Figure 5-34 : Variables **L*f*P** : Exemples de déclarations

On y trouve :

1) *des attributs de la classe* :

- `V1` est un attribut d'instance. Sa visibilité est assurée dans toutes les méthodes d'instance de la classe. Tout comme dans C++, l'emploi de `V1` dans une méthode partagée (statique en C++) est interdit.
- `V2` et `V3` sont des attributs de classe, elles font partie du contexte partagé de la classe. Leur visibilité est globale pour toutes les instances de classe. Au contraire de `V2`, la variable `V3` est déclarée protégée (entre les instances non réparties d'une même classe) ce qui garantit l'atomicité des accès.

2) *des variables locales à une méthode membre* :

- `V4` est une variable de méthode (non persistante et non partagée) accessible uniquement à l'intérieur de `M`. Cette variable fait partie du contexte local de la méthode `M`, elle est allouée dynamiquement à chaque invocation.
- `V5` est une variable de méthode. Sa déclaration persistante oblige, malgré sa déclaration locale à `M`, à allouer cette variable une seule fois dans le contexte d'exécution de l'instance de classe.
- `V6` est une variable de méthode, partagée par toutes les instances de `M`. Bien que visible exclusivement à l'intérieur de cette méthode. La déclaration partagée de `V6` fait que celle-ci est allouée dans le contexte commun de la classe.
- `V7` est une variable de méthode. Cette variable fait partie du contexte local de la méthode `M1`, et malgré la déclaration partagée de son conteneur, elle est allouée dynamiquement à chaque invocation.
- `V8` est une variable de méthode. Similaire en déclaration avec `V7`, cette variable fait partie du contexte local de la méthode `M2`. Cependant, parce que la déclaration indique que `M2` est partagée et protégée, le contexte de cette méthode est unique et global pour la classe. C'est pourquoi la variable `V8` n'est pas allouée dynamiquement mais une seule fois car elle fait partie du contexte de la classe.

5.4.2.3 Sémantique d'accès

L'accès à une variable **L*f*P** peut être atomique ou non. Ce problème d'accès ne se pose que dans le cas des classes passives et celui des média qui exhibent des comportements non sychronisés. Dans ces cas, l'atomicité d'accès n'est garantie que si la variable est :

- de type simple (énumération, booléen ou entier) ou sémaphore,
- de type tableau ou structure déclarée comme protégée (`protected`),
- locale à une méthode non protégée, et si elle n'est pas persistante (`static`) ou partagée (`shared`).

L'application de ces règles est illustrée dans la **Figure 5-35** au moyen de plusieurs exemples de déclarations. Pour chaque déclaration de variable nous interprétons (à droite) la sémantique d'accès.

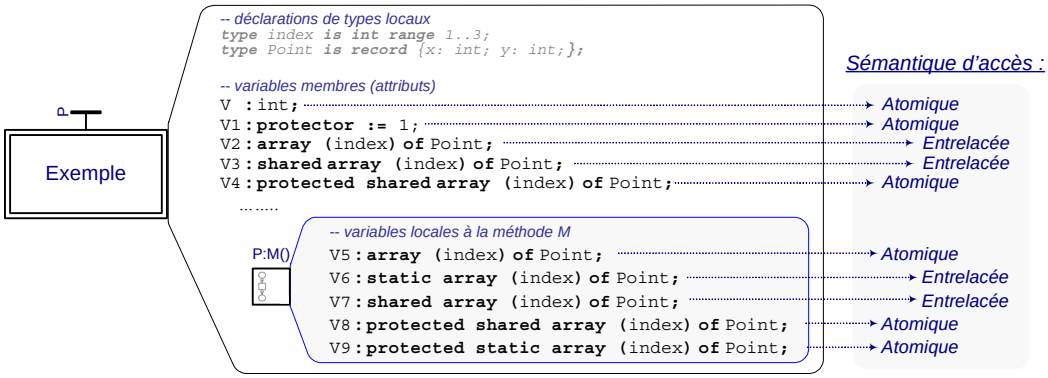


Figure 5-35 : Sémantique d'accès aux variables **L*f*P**.

5.4.2.4 Modélisation en réseaux de Petri

Cette section présente uniquement la modélisation en R*d*P des variables **L*f*P** d'un type de base (prédéfini ou dérivé d'un type prédéfini). La modélisation en R*d*P des classes **L*f*P** sera abordée à part dans la **Section 5.5**: *Classes LfP*.

La modélisation en R*d*P d'une variable **L*f*P** concerne deux aspects de modélisation :

- l'*expression d'un conteneur* permettant de stocker la valeur de la variable,
- la *modélisation des opérations* d'accès de lecture et respectivement d'écriture.

Conteneur d'une variable L*f*P

En R*d*P le conteneur d'une variable **L*f*P** est modélisé au moyen d'une Place marquée. Le domaine de cette place est déterminé par le couple <*contexte, type*> dont le *contexte* exprime une couleur optionnelle qui permet de discriminer entre elles les instances d'une variable. Le *type*, quant à lui, correspond au type de la variable.

La valeur d'une variable **L*f*P** est exprimée par un ou plusieurs jetons de marquage. Ce marquage sera simple dans le cas de variable monovaluées (de type simple, structure ou sémaphore). Au contraire, dans le cas de tableaux le marquage sera multiple, chaque jeton sera associée à un et un seul élément dans le tableau.

La **Figure 5-36** illustre deux exemples de modélisation :

- `V1`, une variable de type entier déclarée au niveau de la classe.
- `V2`, un vecteur d'éléments de types composé (`Point`), celui-ci exprime une variable d'instance.

Dans cette figure le domaine de la place `V1` est constitué d'une seule couleur, celle-ci correspond au type entier, le type de la variable. Au contraire, le domaine de la place `V2` est composé de quatre couleurs dont la première (`class_instance_ID`) constitue un discriminant qui permet de multiplexer les instances de `V2`. Les autres trois couleurs restantes correspondent au type d'indexation (`int8`) ainsi qu'aux champs `X` et `Y` du type `Point`, le type des enregistrements.

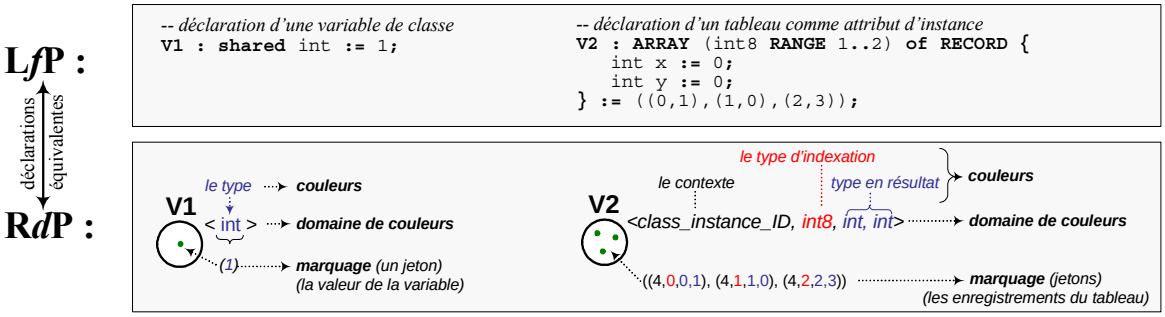


Figure 5-36 : Variables **L*f*P** : Modélisation du conteneur ent R*d*P.

Opérations d'accès

Que cela soit en lecture ou en écriture, la modélisation en R*d*P des opérations d'accès dépend du caractère atomique ou non de l'accès. C 'est pourquoi nous proposons deux schémas de spécification.

- *Dans le schéma d'accès atomique* : Le modèle de traduction se réduit à une simple transition.

- *Dans le schéma d'accès non atomique* : Les modèles d'accès correspondent à des sous-réseaux complexes clos par des transitions (une en entrée et une autre en sortie). Les opérations d'accès sont raffinées dans une série d'actions atomiques (transitions intermédiaires), chacune de ces actions vise l'accès atomique à un champ élémentaire. L'ordre d'exécution est non déterministe. Cela permet de couvrir l'ensemble de comportements possibles dans un système réel.

La complexité du modèle d'accès entrelacés peut engendrer une explosion de l'espace d'états du graphe d'exécution. Il est donc souhaitable, autant que possible, de limiter, lors de la modélisation, les accès concurrents non atomiques.

Exemple

La **Figure 5-37** illustre la construction de ces modèles à l'aide d'un exemple de modélisation concret. Cet exemple concerne la modélisation R*d*P d'une variable `V` de type vecteur d'éléments de type structure (voir l'exemple de `V2` de la **Figure 5-36**). La place `V` modèlise le conteneur de cette variable.

Sur les modèles R*d*P, les opérations de lecture et respectivement d'écriture sont le sous-réseau à gauche de la place `V` et le sous-réseau à droite de la place). **`a-a`**' et **`b-b`**' expriment des points de connexion en entrée et en sortie. Ces points correspondent aux extrémités des arcs.

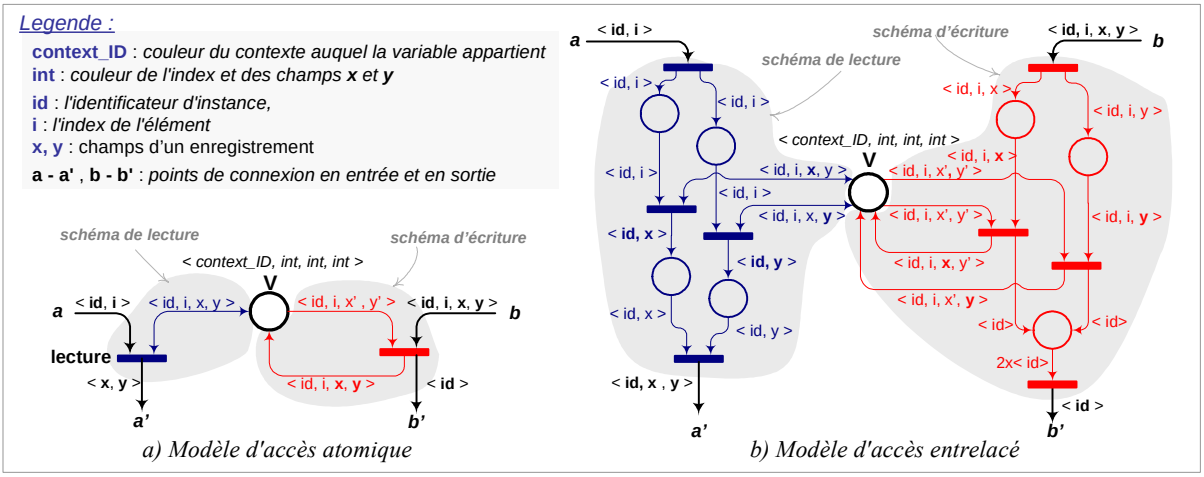


Figure 5-37 : Raffinement en R*d*P des opérateurs d'accès à une variable **L*f*P**.

5.5 Classes L*f*P

L*f*P est un langage orienté objets. Une classe **L*f*P** (voir **Définition 5-2**) précise la description d'une classe d'implémentation UML par rapport à trois aspects :

- la spécification des ports de communication physiques,
- la description d'un contrat d'utilisation pour les services,
- la spécification détaillée du flot de contrôle des méthodes.

Définition 5-2: Une *classe* **L*f*P** exprime un *gabarit de conception instanciable* qui décrit une unité d'encapsulation réutilisable pour la modélisation et l'implémentation.

Présentation

Dans un diagramme d'architecture, une classe **L*f*P** est modélisée graphiquement au moyen d'un classificateur stéréotypé pourvu de zéro ou plusieurs ports de communication. Ce classificateur constitue un conteneur graphique réutilisable permettant de cacher les détails d'implémentation internes. Les ports de communication sont des points de connexion. Un diagramme d'architecture **L*f*P** est un graphe connexe de Classes et média connectés au niveau de leurs ports par des Binders.

a) classe active avec trois ports *b) classe protégée avec deux ports* *c) classe passive avec un port unique*

Figure 5-38 : Déclaration de classes **L*f*P**

La **Figure 5-38** illustre trois exemples de déclarations. On retrouve l'exemple d'une *classe active* (un acteur) disposant de trois ports de communication, celui d'une *classe protégé* ayant deux ports, et celui d'une *classe passive* pourvue d'un seul port.

5.5.1 Description et structure interne

La **Figure 5-39** illustre la structure logique d'une classe **L*f*P** au moyen d'un schéma XML. Cette représentation sert également comme patron syntaxique utilisé lors de la sérialisation des modèles **L*f*P**.

La spécification d'une classe **L*f*P** est organisée en deux parties :

- *Une partie déclarative* précise des paramètres généraux pour la classe et sa structure interne. On peut définir de la sorte le nom et le type de comportement exhibé par la classe (*actif*, *protégé* ou *passif*) ainsi que la liste des variables et des méthodes membres.
- *Un contrat comportemental* précise les actions qu'une instance de classe peut exécuter et définit une *signature comportementale* pour la classe et ses instances. Cette signature décrit la façon de se servir d'une classe (le protocole) sans se préoccuper de l'implémentation détaillée des méthodes.

Il est modélisé au moyen d'un diagramme de comportement **L/P**-BD (**L/P** *Behavioral Diagram*) .

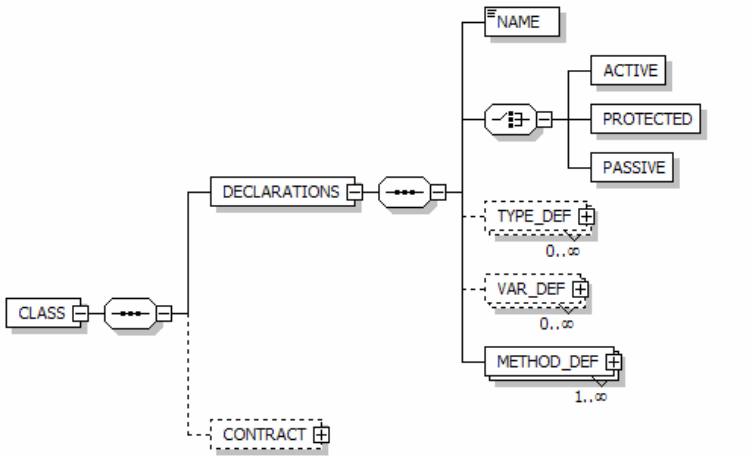


Figure 5-39 : Structure logique d'une classe **L/P**.

La description des méthodes et du contrat d'utilisation d'une classe **L/P** sont illustrés dans les sections suivantes.

5.5.2 Méthodes L/P

Comme tout langage orienté objet, **L/P** permet de structurer la conception des Classes et de Média au moyen des méthodes. Celles-ci constituent des mécanismes d'encapsulation modulaires pour les traitements.

La description d'une méthode **L/P** est séparée en deux parties :

- *Une partie déclarative* statue sur :
  - la *signature*,
  - la *visibilité*,
  - la *cycle de vie de son contexte*,
  - la *sémantique d'invocation et*
  - *la sémantique de synchronization* de la méthode.

  Cette déclaration s'intègre dans celle de son conteneur (la Classe ou le Média). Elle consiste en une liste des déclarations textuelles et des contraintes sur le contrat d'utilisation de la classe (le média) d'hébergement.
- *Un contrat comportemental* précise une signature comportementale pour la méthode. Ce contrat est modélisé au moyen d'un diagramme de comportement **L/P**-BD. Son utilité est double :
  - Lors de la vérification formelle il sera employé pour valider le comportement de la méthode et celui de l'application ;
  - Lors de la génération automatique des programmes il peut servir de modèle.

5.5.2.1 Déclaration

UML réglemente la syntaxe à utiliser pour la déclaration de méthodes sans préciser pour autant comment l'interpréter (voir **Section 3.2.2**). Cet aspect sémantique est fixé tardivement à l'implémentation lors du choix du langage de programmation cible. Hélas, ce manque d'interprétation dans la modélisation est incompatible avec le principe du prototypage rapide par modélisation **[70]**. En effet, l'existence des modèles exécutables ainsi que la possibilité de pourvoir utiliser les techniques de vérification for-

melles nécessite de formaliser complètement la modélisation, c'est à dire formaliser à la fois la *syntaxe* et la *sémantique statique* et *dynamique* de la notation utilisée pour la modélisation.

Signature

L/P se propose donc de corriger le manque de précision d' UML et par conséquent d'attacher à la syntaxe de déclaration une interprétation permettant d'évaluer la signature de méthodes. Pour ce faire, nous nous sommes inspirés de UML pour sa syntaxe, et de Ada95 pour l'interprétation des signatures.

La signature d'une méthode **L/P** précise donc deux aspects :

- *la structure d'appel* : le nom de la méthodes, la liste de paramètres formels (nom, type et une éventuelle valeur par défaut), et le type du résultat en retour (optionel). Mise à par le symbole d'affectation (`:=` au lieu de `=`) utilisé pour préciser la valeur par défaut, la syntaxe de déclaration est identique à celle de la notation UML.
- *la sémantique associée* permettant d'interpréter les signatures qui sont les mêmes que celles du langage Ada95.

Règles de visibilité

Une méthode **L/P** peut exprimer soit un *service public* soit une *opération privée* (fonction ou procédure). Comme en UML, un service exprime une fonctionnalité offerte par la classe, accessible (dans certaines conditions) depuis l'extérieur ; une opération privée, quant à elle, décrit correspond à une fonction ou une procédure auxiliaire invocable que de l'intérieur de la classe ou le média de déclaration.

La déclaration de services est réservée aux seules Classes **L/P**. En effet, pour les Média la déclaration des services ne se justifie pas si l'on considère que ceux-ci sont dédiés à la description des protocoles de communication et non à la modélisation des services de traitement.

Comme pour les variables **L/P**, la visibilité des méthodes peut être abordée selon deux vues différentes : *de l'extérieur* du conteneur de déclaration (classe ou média **L/P**), et *de l'intérieur* du conteneur de déclaration, c'est-à-dire entre les objets (instances de classe) d'une même classe et entre les invocations multiples à une même méthode.

Visibilité externe

La déclaration publique (`public`) de services n'implique pas pour autant leur visibilité permanente. En effet, **L/P** est un ADL et par conséquent il permet de varier dynamiquement l'accès aux services.

L'originalité de notre approche, par rapport aux ADL, vient du fait que l'accès à un service peut non seulement être paramétré par le couple <*Port d'entrée*, *Etat interne de la classe*> mais aussi déterminé en fonction du <*contenu de l'invocation*> elle-même (e.g. la valeur d'un paramètre, l'émetteur, la priorité d'invocation etc.).

Cette capacité de pouvoir *déterminer les invocations* en fonction de leur contenu est similaire au mécanisme de multiplexage offert par les files UNIX et les Sockets.

Cependant, **L/P** généralise ce mécanisme sur trois points :

- *Le mécanisme de multiplexage* **L/P** *est indépendant du modèle de communication.* Modélisé au moyen d'une condition de *garde*, ce mécanisme peut s'appliquer aussi bien sur des messages que sur des invocations directes. Cette dernière facilité est similaire à celle du langage Eiffel **[83]** qui permet la définition de *gardes* sur l'invocation de méthodes.
  Cette indépendance est possible parce que **L/P** sépare bien la description de l'*aspect fonctionnel* d'une application (embarquée dans les Classes) de l'*aspect communication* (modélise au moyen de Binders et de Média). La conception de ces entités se fait en isolement. L'assemblage des composants d'une application est alors du ressort d'un processus de configuration ultérieur.

- *La définition de gardes sur les services peut porter sur l'ensemble des informations présentes dans une invocation.* Cela n'est pas possible dans le cas de files de messages UNIX ou des sockets (voir **Section 4.4.1**) qui ne permettent pas de distinguer la réception de messages que par rapport à un champ fixe.
  Bien plus, au contraire de JMS qui ne permet pas de multiplexer les messages que par rapport à leur entête, **L*f*P** étend la définition de gardes jusque dans le corps des invocations, c'est-à-dire qu'elles peuvent porter surd le nom de la méthode invoquée et sur les valeurs des paramètres passées.
- *Le filtrage de services est dynamique.* Pour un Port donné, une garde peut être redéfinie en fonction de l'état interne de la classe. Il est donc possible de limiter l'accès à un service par rapport à sa provenance (le Port) et de la disponibilité momentané du service chez le fournisseur (la classe offrant le service).
  Le mécanisme de multiplexage **L*f*P** peut être employé également à d'autres fins que celui de filtrer la l'accès aux méthodes. Dans les Média il permet d'intercepter et de router les invocations.

Remarque 1 : La construction de Média d'interception ouvre la voie vers une démarche de *conception par aspects*. Un Média **L*f*P** peut être utilisée pour intercepter et ensuite pour tisser les aspects de conception d'une façon totalement transparente pour le service invoqué. La conception des services est donc simplifiée, elle ne concerne que la partie fonctionelle de l'application. Cela simplifie également l'intégration de services existants parce qu'il n'est pas requis de modifier leurs implémentations. Cette facilité s'avère particulièrement utile si on ne dispose pas des sources.

Remarque 2 : La construction de Média d'adaptation (proxy) qui interceptent, adaptent et ensuite dispatchent les invocations entre des Ports, permet à la manière de C2 ou de Waves de stimuler le principe du couplage faible dans la conception. La conception des classes **L*f*P** peut se faire donc en isolement sans disposer des interfaces utilisées. Un Média **L*f*P** peut être utilisé, comme un connecteur C2, pour adapter les invocations d'une façon transparente pour les classes.

Visibilité interne

A l'intérieur d'une Classe ou d'un Média **L*f*P**, les méthodes et les attributs membres respectent les mêmes règles de visibilité. Celles-ci ont déjà été illustrées dans la **Section 5.4.2**: *Variables LfP*. Ainsi, le corps d'une méthode déclarée shared sera visible dans toutes les instances de la classe. Au contraire, l'implémentation d'une méthode d'instance sera répliquée localement pour chaque instance.

Cycle de vie

Comme nous l'avons expliqué en **Section 5.4.2**: *Variables LfP*, toute variable **L*f*P** est rattachée à un contexte d'exécution. Celui-ci peut être global pour la classe, local à une instance ou local à une invocation de méthode. Les paramètres d'invocation d'une méthode font partie de ce dernier cas, leur création se fait donc dynamiquement sur le contexte local de la méthode à chaque invocation.

Sémantique d'invocation

En tant qu'ADL, **L*f*P** sépare la modélisation de la *partie fonctionnelle* d'une application de l'aspect *interaction* et `communication`. Pour ce faire, **L*f*P** propose les notions de *Classe* et respectivement celles de *Binder* et de *Média*.

La déclaration d'un service dans une classe ne réglemente que la sémantique d'interaction directe entre la classe et les binders connectés. La sémantique d'invocation de bout en bout, quant à elle, est assimilée au résultat des synchronisations entre l'ensemble de mécanismes d'interaction et de communication qui s'interposent entre les parties communicantes.

Pour ce faire, notre langage **L*f*P** prône deux sémantiques d'interaction directes :

- *synchrone*. Ce modèle correspond à une invocation bloquante de fonction avec retour, il correspond à une méthode disposant de paramètres en sortie et/ou qui retourne un résultat.
- *trigger*. Un service marqué **`trigger`** a la sémantique d'une barrière de synchronisation par *rendez-vous* qui précède l'exécution du service invoqué. La déclaration de triggers empêche l'utilisation des paramètres en sortie, cela se justifie si l'on pense que le traitement n'aura lieu qu'après la synchronisation. L'exécution d'un trigger se fait par délégation, c'est pourquoi ce type de service peut être déclarée que dans une classe active qui possède son propre fil d'exécution (e.g. tâche).

Sémantique de synchronisation

L*f*P permet de synchroniser l'accès aux méthodes. L'exécution d'une méthode peut être protégé par rapport à toute exécution concurrente. Pour ce faire, nous proposons plusieurs modèles de synchronisation basiques :

- *Synchronisation globale* pour la classe.
- L'exécution de la méthode se fait en exclusion mutuelle quel que soit l'instance de classe invoquée (ou la classe elle-même). Ce modèle correspond à une déclaration de méthode de type «**`static sinchronized`**» en Java.
- *Synchronisation locale* par rapport à l'instance courante (objet).
- L'exécution de la méthode est réalisée en exclusion mutuelle avec les autres méthodes de d'instance et par raport à elle-même, aucun autre fil d'exécution ne peut entrer dans l'objet. En Java, cela correpond à une méthode d'instance déclaré synchronized. Ce type de synchronisation correspond également à celui d'une entrée (entry) ou d'une procédure exhibée par une variable de type protégée Ada95.
- *Sans synchronization*.
- L'accès concurrent à la méthode est permis. Cela correspond à une déclaration de méthode normale, non synchronisée.

Une méthode protégée **L*f*P** peut être invoquée d'une manière récursive ou circulaire par le même fil d'exécution sans avoir besoin de réacquérir la clé d'accès (le **`SEMAPHORE`**). En effet, comme en Java la synchronisation sur un objet ou une classe se fait une seule fois, lors de la première entrée.

Rappelons que ces mécanismes ne sont pas pour autant les seuls moyens de synchronisation en **L*f*P**. En effet, la déclaration de sémaphores associée avec celle des gardes sur les transitions du contrat d'utilisation d'une classe permet de définir des mécanismes de synchronisation utilisateur plus réstricitifs.

Exemple de déclarations

La **Figure 5-40** présente un exemple de modélisation didactique : une *classe active* **L*f*P** (tâche) pourvue de plusieurs méthodes.

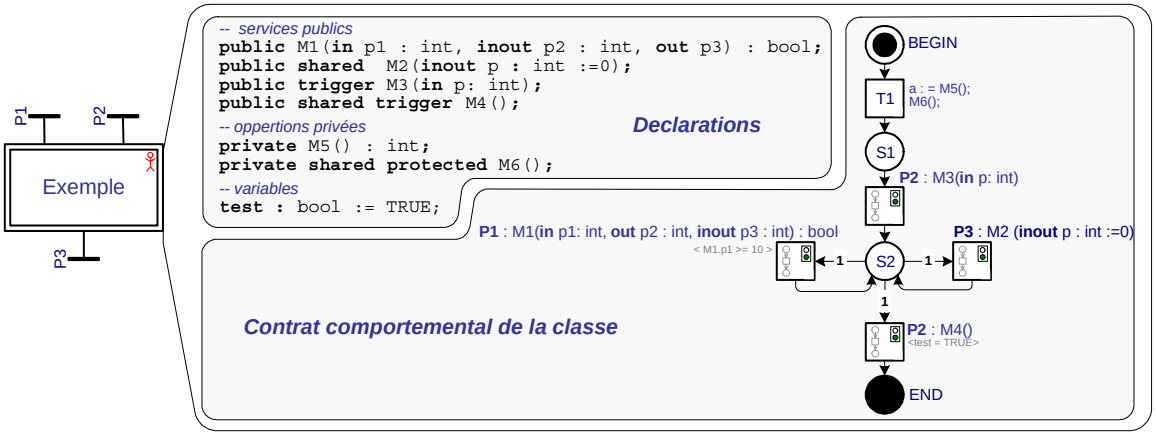


Figure 5-40 : Déclaration des méthodes **L*f*P** : Exemples.

Nous détaillons un par un les aspects de modélisation de cet exemple :

1) *Signature*.

- **`M1`** est la seule méthode qui retourne de résultat. Elle correspond donc à une fonction.
- **`M2`** est une procédure, son unique paramètre **`p`** est passé à la fois en entrée et en sortie. La valeur par défaut de **`p`** est zéro.
- **`M3`** correspond à une procédure qui ne prend pas des paramètres en sortie.
- **`M4`**, **`M5`** et **`M6`** sont des procédures sans paramètres.

2) *Visibilité*.

Visibilité externe. Au contraire, **`M5`** et **`M6`** qui sont des opérations internes, **`M1`**, **`M2`** **`M3`** et **`M4`** correspondent à des services publics. La visibilité externe de ces derniers est contrainte par le contrat comportemental de la classe. Ainsi :

- **`M1`** n'est visible que sur le port **`P1`** et que si l'objet cible se trouve dans l'état **`S2`**. L'invocation de ce service est conditionnée par une garde portant sur la valeur de l'un des paramètres d'invocation. Le non respect de cette garde empêche de traiter l'invocation. Notons que, au contraire des StateCharts UML, une invocation bloquée n'est pas perdue mais elle peut être traitée plus tard (e.g. dans un autre état).
- **`M2`** n'est accessible que par le port **`P3`** et que si l'état d'exécution courant est **`S2`**.
- **`M3`** n'est visible que pour les invocations provenant du port **`P2`** et que dans l'état **`S1`**.
- **`M4`**, tout comme **`M1`** et **`M2`**, n'est accessible que si l'état courrant est **`S2`** et que par le port **`P2`**. Son traitement est gardé par la valeur d'une variable d'instance (**`test`**). La priorité 1 sur l'arc d'entrée de cette méthode précise que son traitement est non privilégié par rapport à ceux de **`M1`** ou de **`M2`**.

Visibilité interne.

- **`M1`** **`M3`** et **`M5`** sont méthodes d'instance.
- Au contraire, **`M2`**, **`M4`** et **`M6`** sont partagées (**`shared`**) au niveau de la classe.

3) *Cycle de vie.*

Mis à part la déclaration publique des services **`M1`**, **`M2`**, **`M3`** et **`M4`** le contrat comportemental de la classe impose plusieurs contraintes de visibilité. Ainsi, le contexte local de chacune de ces méthodes est crée dynamiquement pour chaque invocation. Cela est nécessaire car rien n'empêche d'invoquer récursive-

ment chacun de ces services une fois que l'accès est acquis.

4) *Sémantique d'invocation*.

Mis à part **`M3`** et **`M4`** qui sont des triggers toutes les autres méthodes utilisent un modèle d'invocation synchrone.

5) *Sémantique de synchronisation*. .

- Tous les services précisés (**`M1`** à **`M4`**) sont protégés et par conséquent l'accès à ceux-ci est synchronisé. Cela est du au caractère actif de la classe qui, à la manière d'une tâche Ada95, empêche de déclarer des services non synchronises. Pour **`M1`** et **`M3`** la synchronisation est locale à chaque instance. Au contraire, pour **`M2`** et **`M4`** celle-ci est réalisée globalement pour la classe
- La méthode **`M5`** est non protégée. Cela se justifie vu que l'accès à cette méthode n'est que local.
- Ce n'est pas le cas pour **`M6`** qui est partagée, c'est pourquoi nous déclarons explicitement cette opération comme protégée (**`protected`**).

5.5.2.2 Contrat comportemental

L*f*P permet de modéliser en détail le corps des méthodes. Cette description est partagée en deux parties:

- *une partie déclarative* précisant des déclarations locales pour la méthode (e.g. types et variables).
- *un diagramme de comportement* **L*f*P**-BD (voir la **Section 5.3**: *Les diagrammes de comportement (LfP-BD)*) qui précise le flot d'exécution (séquentiel) de la méthode. Cet automate a une structure particulière, il ne peut contenir que des transitions simples (du code séquentiel) ou de type bloc (pour encapsuler un sous-réseau complexe). L'emploi de transitions de type méthode est donc interdit. En effet, il n'est pas possible d'accepter une autre invocation lorsqu'on est dans le corps d'une méthode.

Exemple

La **Figure 5-41** présente un exemple de modélisation simple : une méthode qui calcule **`n!`** (la factorielle d'un entier). Le contexte local de cette méthode consiste d'une seule variable (non persistante et non partagée). Son **L*f*P**-BD précise trois flots d'exécution possibles à déterminer (à l'aide des gardes sur les transitions) en fonction de la valeur de n. Pour les entiers négatifs, on retourne la valeur error. Les factorielles de 0 et de 1 sont prédéfinies à 1. Pour les entiers positifs, on invoque d'une façon récursive la même fonction. Cet appel étant local, il n'est pas nécessaire de préciser de Port en sortie.

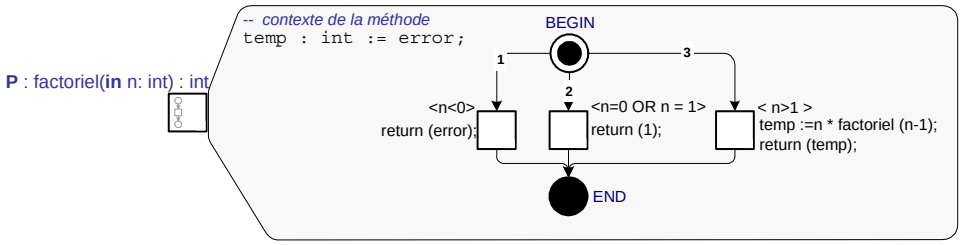


Figure 5-41 : Corps d'une méthode **L*f*P** : exemple de modélisation.

5.5.3 Contrat d'utilisation d'une classe

Le contrat d'utilisation d'une classe **L*f*P** permet d'imposer une signature comportementale sur l'ordre d'exécution de méthodes, la concurrence d'accès aux services ainsi que sur les conditions d'acceptation des invocations.

Ces aspects de conception ont été décrits d'une manière approfondie dans la **Section 5.5.2** dédiée à la description de méthodes **L/P**. C'est ainsi que, mis à part une remarque d'ordre général, nous considérons qu'il n'est pas nécessaire de revenir sur ce point.

Notons que la structure d'un contrat d'utilisation est soumise à des contraintes de conception imposées par le type de comportement qu'on veut modéliser pour la classe. Trois cas se distinguent :

- pour une classe passive : le contrat d'utilisation est absent. En effet, la notion d'état d'exécution n'a pas de sens dans puisque l'exécution des services n'est pas gardée par des conditions de synchronisation systématiques (pour tous les services). Elle est initiée par un appel émanant directement ou indirectement d'une classe active qui «prête» son compteur ordinal.
- pour une classe active : le **L/P**-BD de la classe n'est contraint par aucune restriction.
- pour une classe protégée : le contrat d'utilisation ne peut référencer des transitions simples. En effet, une classe protégée ne disposant pas d'un fil d'exécution propre ne peut pas exécuter d'actions en dehors d'une invocation. La définition d'un automate **L/P**-BD est cependant possible puisque que tout accès à une méthode est synchronisé.

5.6 Média L/P

Une média **L/P** est un connecteur ADL pour lequel il est possible de modéliser explicitement le protocole de communication à l'aide d'un diagramme de comportement **L/P**-BD. Il est possible de les utiliser comme élément de communication de base, ou de les composer pour former des schémas de communication plus complexes.

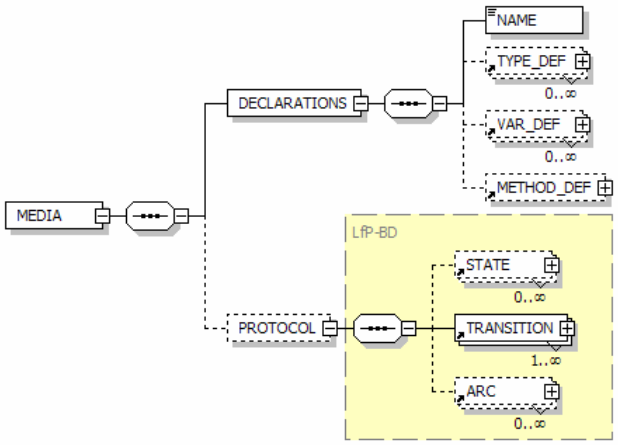


Figure 5-42 : Structure logique d'un média **L/P**.

La **Figure 5-42** illustre la structure logique d'un média. Elle est identique à celle d'une classe aux différences suivantes près :

- le comportement d'un média n'est plus catalogué en *actif*, *protégé* ou *passif* ; il est directement déduit de la structure du diagramme **L/P**-BD.
- l'utilisation de transitions-méthodes dans le **L/P**-BD d'un média est interdite. Effectivement, la différence de modélisation majeure entre une classe et un média **L/P** est que le média ne peut pas offrir de service.

Ces points sont détaillés en **Section 5.6.2**.

5.6.1 Définition de discriminants personnalisés

Comme évoquée en **Section 4.5**, nous considérons qu'un média de communication doit permettre la définition des discriminants de routage pour les invocations. Cette redéfinition n'est possible que par rapport à un port de communication, toutes les invocations échangées via un port doivent respecter le même discriminant. La notion de discriminant joue donc le rôle d'un contrat de liaison devant être respectée par les classes connectées au média.

Un discriminant est d'un type structuré (**`RECORD`**) de nom identique à celui du port de communication. Chaque champ de cette structure constitue un terme du discriminant.

Exemple

La **Figure 5-43** corresond à un média **L*f*P** disposant de deux ports de communication (**`P1`** et **`P2`**). Au contraire du port **`P1`** qui laisse la structure du discriminant par défaut inchangée (définie en **Section 4.5**), le port **`P2`** étend cette structure par un champ additionnel (**`sequence_no`**) identifiant un numéro de séquence.

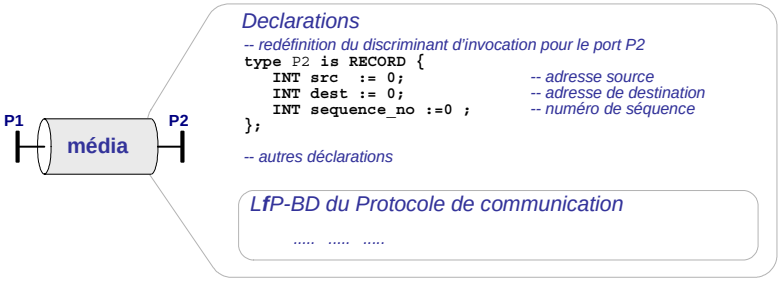


Figure 5-43 : Définition de discriminants personnalisés.

5.6.2 Protocole de communication d'un média LfP

Du point de vue de la modélisation, un média correspond à une version simplifiée de classe qui ne dispose pas de méthodes publiques (services). L'utilisation de transitions-méthode dans le **L*f*P**-BD d'un média est donc interdite ; l'interception des invocations provenant d'un port est réalisée directement au niveau des transitions simples. Le nom de la transition d'interception correspond à une liste de ports en entrée.

Le comportement d'un média LfP peut être soit :

- *séquentiel avec état*. Dans ce cas, la structure du **L*f*P**-BD est identique à celle d'une classe active. Ce type de comportement permet la déclaration de protocoles de communication avec état. La concurrence des communications peut être réalisée par l'instanciation répétée de plusieurs média identiques (on parle d'un pool de média).
- *concurrent sans état*. Dans ce cas, le **L*f*P**-BD du média se résume à une liste de transitions simples (non connectées) permettant de relayer les invocations entre les ports. Ce type de comportement interdit la déclaration de variables et d'états dans le média.

Exemple

La **Figure 5-44** présente deux exemples de modélisation :

- Le **L*f*P**-BD de gauche correspond à un média implémentant un protocole séquentiel de type requête-réponse. Les invocations reçues sur le port **`P1`** sont transférées sur le port **`P2`**. Le protocole se bloque ensuite dans l'état **`WAIT`**. Pour relayer une autre requête il faut qu'une réponse (de signa-

ture identique) soit routée en sens inverse du port `P2` vers `P1`.

- Le **L*f*P**-BD de droite correspond à un protocole concurrent sans état qui permet de router les invocations provenant d'un port `P1` soit sur le port `P2` soit sur le port `P3` en fonction de leur priorité (`priority`), un champ du discriminant.

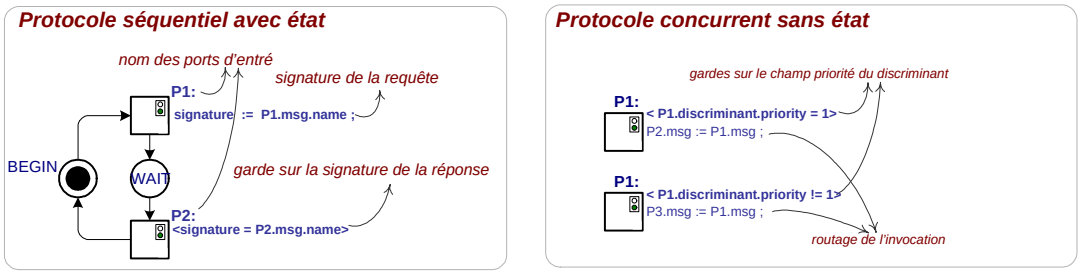


Figure 5-44 : Protocoles de communication d'un Média **L*f*P**.

5.7 Binders L*f*P

Un Binder définit un contrat d'interaction simple et normalisée, utilisé pour typer la liaison des ports. Lors de l'implémentation, il correspond à un mécanisme de communication simple portable sur différents types d'environnement d'exécution (système d'exploitation + middleware).

La conception des binders est fondée sur les conclusions de la **Section 4.5** (consacrée à la synthèse d'un modèle de communication canonique). Deux types de mécanismes d'interaction ont été retenus :

- *tampons de données paramétrables*. Ceux-ci peuvent être utilisées aussi bien pour :
  - la communication par flux d'octets (messages de taille fixe sans nom et sans discriminant),
  - la communication par échange asynchrone de messages structurés,
  - invocation d'opérations par rendez-vous synchrone (e.g. RPC).
- *mécanisme d'invocation directe* à un sous programme (opération) local.

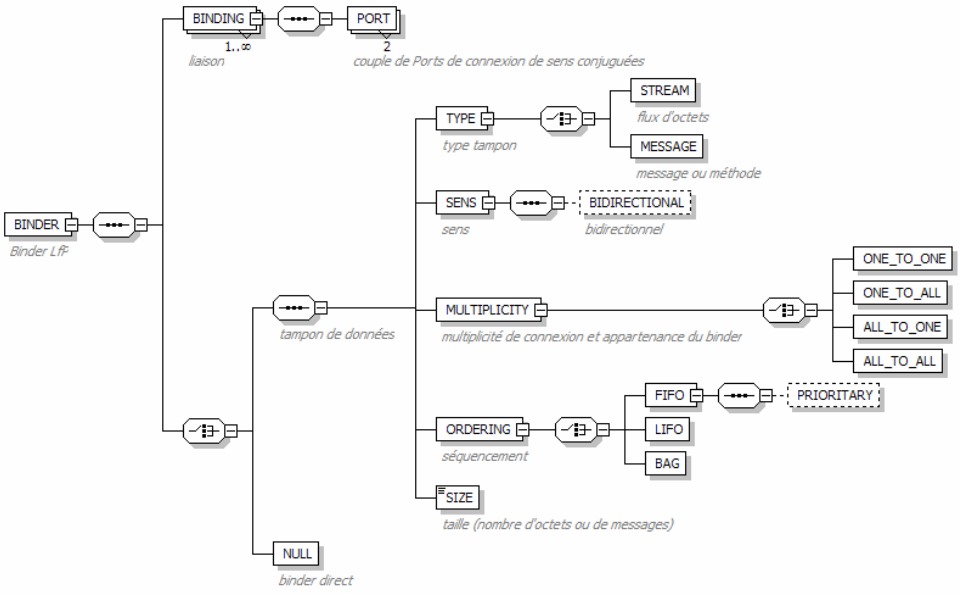


Figure 5-45 : Structure logique d'un binder **L*f*P**.

La **Figure 5-45** présente la structure logique d'un Binder **L*f*P**. Cette structure fait référence à plusieurs points de paramétrage déjà évoqués en **Section 4.5** :

- *Liaison*. Un binder peut être utilisée pour lier plusieurs couples de ports. Chaque couple identifie un port source (le premier) et un port cible (le second) appartenant à une classe ou un média.
- *Type (paradigme de communication)*. Un binder peut designer soit un *tampon de données* paramétrable utilisée pour la communication par flux d'octets (**`STREAM`**), par messages asynchrones ou rendez-vous synchrones (**`MESSAGE`**) soit *un mécanisme de liaison synchrone directe* (**`NULL`**) non paramétrable.
- *Sens*. Un binder peut être unidirectionnel ou bidirectionnel. Dans le second cas, les tampons de communication sont sont créés par paires (un dans chaque direction de communication).
- *Multiplicité*. Elle définit comment instancier les tampons de communication, c'est-à-dire leur nombre et leur affinité à l'une des entités liée (classe ou média).
- *Politique de séquencement*. Trois politiques classiques sont proposées **`FIFO`**, **`LIFO`** et **`BAG`** (non déterministe). Pour les files de messages **`FIFO`** ou **`LIFO`**, la politique de séquencement peut-être raffinée pour permettre le multiplexage de messages par rapport à un champ de priorité à choisir par l'utilisateur parmi les champs du discriminant ou la signature du message.
- *Taille*. Elle spécifie la capacité de stockage en nombre d'octets (ou messages) du binder.

5.7.1 Multiplicité et appartenance des binders

La multiplicité d'un binder **L*f*P** concerne son instanciation. Elle précise si un nouveau tampon de communication est crée pour chaque instance de classe ou média qu'il connecte ou si un tampon unique est partagé entre toutes les instances. La multiplicité d'un binder désigne également le propriétaire du tampon de communication. Comme évoquée en **Section 4.5**, nous considérons que le cycle de vie et la liaison de binders doivent être implicitement déduits à partir de leur déclaration.

La **Figure 5-46** illustre la façon dont la multiplicité d'un binder doit être interprétée. Trois cas sont possibles :

- **`1:1`** (en bas et à droite de la figure) est la multiplicité par défaut. Ce cas correspond à ce qu'on trouve dans d'autres ADL comme ROOM ou dans la langage Ada95 où chaque classe active (acteur ROOM, tâche Ada95) ne possède que les files de réception (nommée entry en Ada95).
- **`1:all`** (en haut à gauche et à droite de la figure) est une multiplicité asymétrique, c'est-à-dire elle est interprété par rapport à une direction de lecture. Un binder marqué ainsi est crée dynamiquement. La multiplicité 1 dénote la classe ou le média d'appartenance.
- **`all:all`** (en bas et à gauche de la figure) dénote un tampon unique, partagé par toutes les instances de classes (ou de média) qu'il connecte.

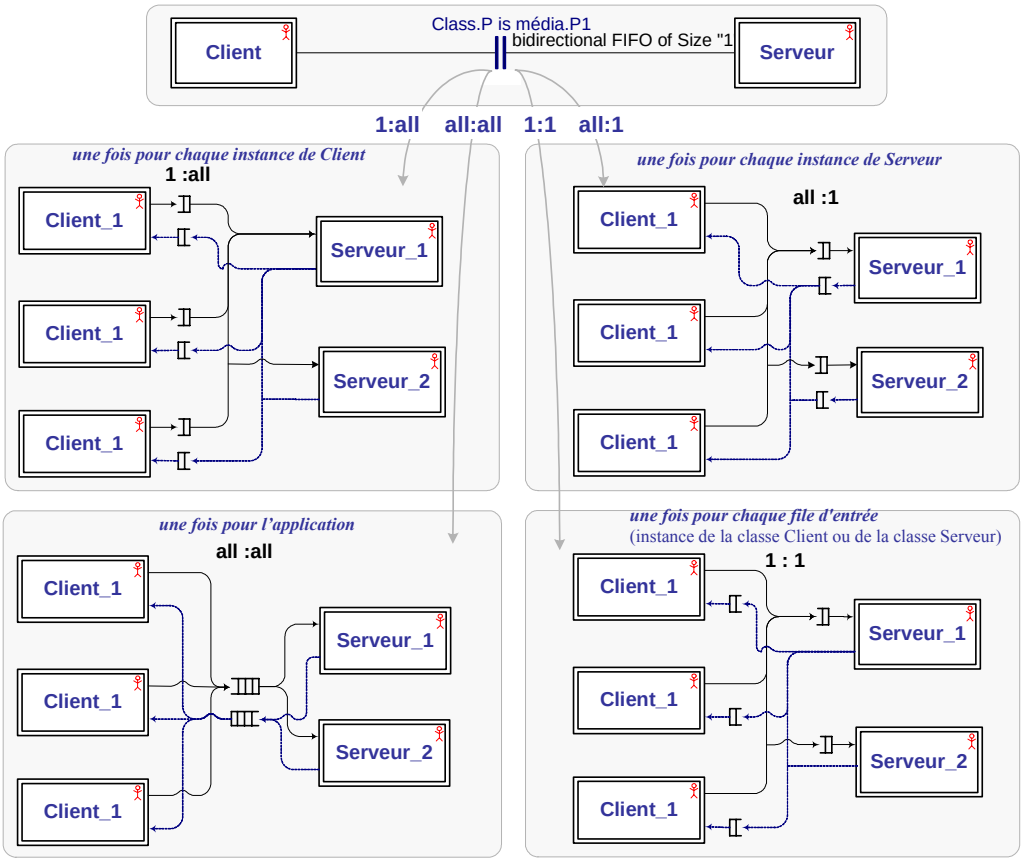


Figure 5-46 : Schéma de dépliage de la multiplicité d'un binder **L/P**.

5.7.2 Raffinement en réseaux de Petri

Comme nous l'avons vu en **Section 5.3.6**, un binder **L/P** est directement traduit en réseaux de Petri par sans passer par l'intermédiaire des **L/P**-BD. Nous proposons quatre patrons de modélisation, chacun dédié à un type ou politique d'ordonnancement différente.

La **Figure 5-47** illustre le schéma de traduction pour un binder de type **`NULL`**, utilisé pour représenter des appels locaux. Ce patron est constitué d'une simple transition d'interface partagée (fusionnée) ensemble avec les transitions de synchronisation correspondant aux ports de communication **L/P** qu'il connecte (voir **Section 5.3.6.4**).

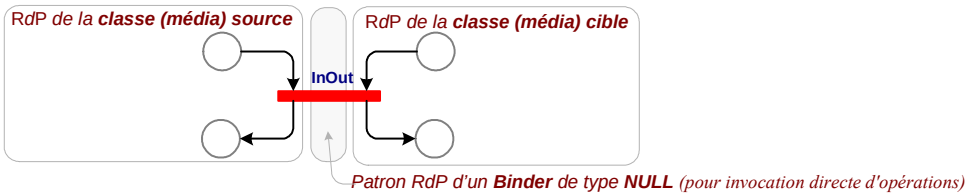


Figure 5-47 : Patron de traduction en réseaux de Petri d'un binder NULL.

Le patron R*d*P d'un binder de type **`BAG`** est illustré en **Figure 5-48**. Il constitue une base pour la traduction de binders de type **`FIFO`** et **`LIFO`** qui sont présentés par la suite.

Ce patron est constitué de deux transitions de synchronization **`in`** et **`out`** et deux places **`BAG`** et **`CAPACITY`**. Les transitions in et out sont partagées, elles correspondent aux ports d'entrée et de sortie. La

place **BAG** modélise le tampon de données dans lequel sont stockées les invocations. La place **CAPACITY** est un compteur indiquant la capacité de stockage disponible.

Deux couleurs composées sont référencées :

- **d_context** correspond à une couleur composée permettant de distinguer les instances d'un binder. Cette couleur existe que dans le cas de binders de multiplicité **1:1** et **1:all**. Elle dénote l'instance de classe (ou de média) à laquelle le binder appartient. Dans ce cas, le marquage de cette place est crée (détruit) de la même façon que celui de la classe d'appartenance (voir **Section 5.3.6.2**).
- **d_discriminant** et **d_invocation** sont deux couleurs correspondant respectivement au discriminant et aux invocations.

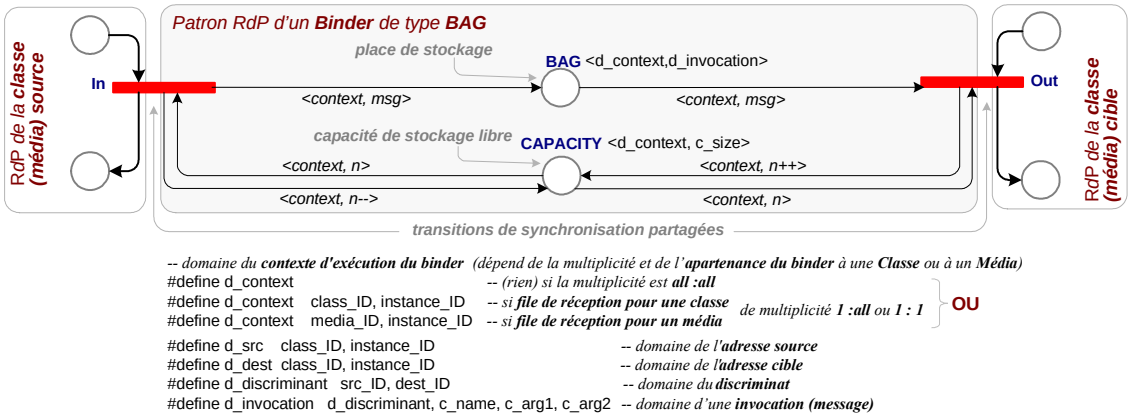


Figure 5-48 : Patron de traduction en réseaux de Petri d'un binder de type BAG.

La **Figure 5-49** illustre le schéma de traduction d'un binder de type FIFO avec priorité. Ce schéma constitue une modification du précédent qui prend en compte la politique de séquencement FIFO et éventuellement le multiplexage par priorités. Deux modifications sont apportées :

- les places **HEAD** et **TAIL** correspondent à des compteurs permettent d'introduire et de retirer les messages respectivement en fin et en tête de la file.
- la priorité est représentée par une couleur supplémentaire permettant de ranger les messages par priorité et par ordre.

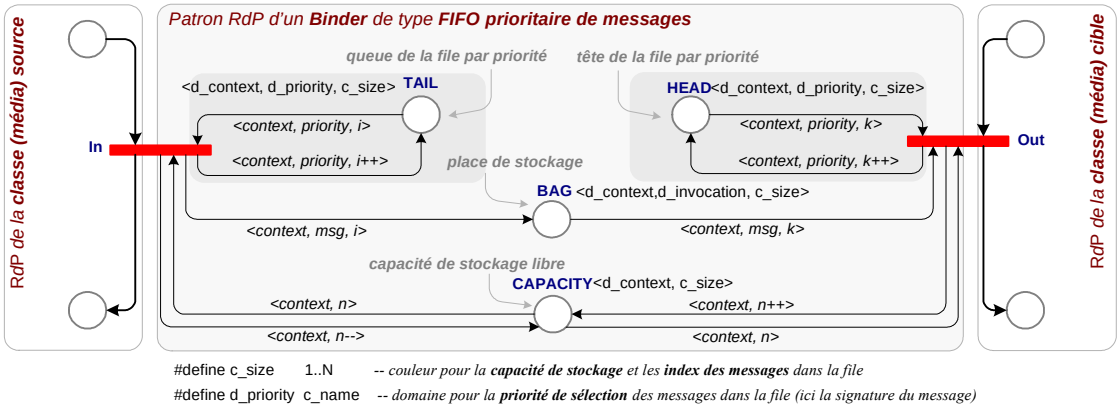


Figure 5-49 : Patron de traduction en réseaux de Petri d'un binder de type FIFO.

Finalement, la **Figure 5-50** illustre le patron de traduction d'un binder de type **LIFO**. Celui-ci ne diffère des précédents que par une place. En effet, **TAIL** n'a plus de sens car les messages sont introduits et retirés du binder toujours à partir de la tête.

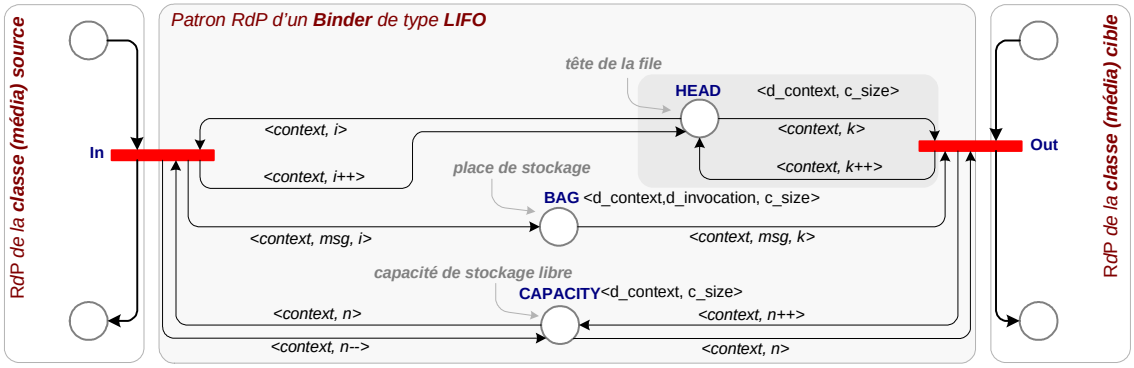


Figure 5-50 : Patron de traduction en réseaux de Petri d'un binder de type LIFO.

5.8 Conclusion

Dans ce chapitre, nous avons présenté **L*f*P**, notre langage de descriptioon de haut niveau du contrôle d'applications logicielles réparties. Ce langage possède les caractéristiques suivantes :

1) Il est compatible avec une approche basée sur UML et conserve les mécanismes de structuration statiques de cette notation,
2) Il permet de capturer la sémantique d'exécution des différents composants du système sous la forme d'un contrat comportemental non ambigu,
3) Il intègre des fonctions que l'on trouve dans les langages de description d'architecture (ADL) pour prendre en compte l'aspect déploiement dans un processus de génération automatique de programmes.

Comparé à l'état de l'art, notre langage se distingue sur les points suivants :

- une sémantique pour les systèmes faiblement couplés (coûts de communication élevés par rapport au temps de calcul),
- l'intégration dans un même langage de concepts formels et de concepts issus des AD,
- une approche méthodologique.

Sémantique pour les systèmes faiblements couplés

La sémantique de **L*f*P** est résolument dédiée à des applications réparties asynchones. En effet, nous visons une exécution sur des architecture réparties, donc faiblement couplées (typiquement des architectures multi-machines).

Notre approche se distingue donc dans son principe des approches synchrones ayant des objectifs similaires. Ces languages sont en général basés sur le paradigme «flot de données» comme Signal **[5]** ou Lustre **[23]** et plus rarement sur le paradigme objet comme COOPN **[13]**. De tels langages visent en général des systèmes fortement couplés (par exemple, des systèmes embarqués sur «Systems on Chip» avec plusieurs unités travaillant en parallèle) car la sémantique fortement synchrone s'accomode mal des conditions d'exécution des systèmes répartis à large échelle.

Intégration de concepts formels et d'ADL

L*f*P vise un type de problème difficile : la mise en œuvre de la supervison d'applications réparties. Dans ce domaine, il existe à l'heure actuelle peu d'approches susceptibles d'aider les développeurs de

système. La liaison avec des méthodes formelles est un atout pour maîtriser la complexité de tels systèmes.

Pour ces aspects formels, nous avons défini la sémantique de **L*/*P** en nous appuyant sur les réseaux de Petri parce qu'ils étendent naturellement au calcul parallèle, les automates d'UML en offrant une base théorique solide. Notre langage propose cependant une encapsulation des concepts réseaux de Petri pour :

- offrir une vision simplifiée aux concepteurs de systèmes (qui peuvent en effet ne pas connaître ce formalisme),
- se baser sur des mécanismes simples à implémenter et à composer formellement : les automates d'états,
- masquer la sémantique compositionnelle des modules de notre langage (fusion de transitios ou fusions de place) qui est choisie en fonction des règles de composition (en particulier les caractéristiques des binders) ainsi que des propriétés que l'on cherche à vérifier.

Nous avons intégré les principaux concepts proposés dans les ADL courants :

- Les classes **L*/*P** permettent d'établir une correspondance entre une notation orientée objet comme celle d'UML et le concept tel qu'il est offert par les ADL. Nous réglons ainsi le problème de déploiement potentiel des classes UML (qui peuvent contenir du parallélisme, ce que nous interdisons) mais nous sommes plus riches que la majorité des ADL en permettant de spécifier qu'une classe est passive ou protégée.
- Les ports **L*/*P** sont des points de liaison et non des interfaces logiques. Nous éliminons ainsi les dépendances pernitieuses évoquées dans la **Section 3.3.4** en séparant la description des invocations en deux parties (le discriminant et le corps d'un message) traitées et imposées par des entités différentes (les média pour les discriminants, les classes pour le corps des messages).
- Les binders sont une mise en œuvre canonique des contrats de liaison simples et portables identifiés en **Section 4.5**, ils permettent de faciliter la liaison entre entités dans **L*/*P**.
- Similairement à ce qui est proposé dans un ADL comme WRIGHT, on peut associer des modèles de comportement complexes à un média de communication. Cela permet de supporter de manière distincte des protocoles à mémoire, au contraire de ROOM **[111]** ou SDL **[63]** qui n'offrent que des mécanismes de base peu paramétrables.

Ces deux caractéristiques font de **L*/*P** un langage apte à supporter une approche formelle comme une approche de génération automatique de programmes. Ainsi, notre langage se pose comme un pivot entre les spécifications de haut niveau et différentes formes d'implémentation.

Approche méthodologique de modélisation

La notation **L*/*P** s'accompagne d'une philosophie de conception basée sur le prototypage par raffinements. A ce titre, notre langage a une caractéristique importante : il se veut un pivot entre les spécification de très haut niveau (UML) et l'implémentation (des spécifications formelles ou un langage de programmatuion).

La méthodologie est liée au langage et vice-versa. En effet, **L*/*P** propose trois vues complémentaires qui doivent être renseignées au fur et à mesure que la spécification se rapproche de l'implémentation. Nous espérons ainsi combler le fossé qui subsiste entre la phase de conception et la phase de réalisation dans les approches de développement traditionnelles.

CHAPITRE 6

Conclusion Générale

6.1 Bilan

L'objectif de nos travaux était de définir un langage de spécification adapté à la conception de la supervision dans les systèmes répartis. Ce langage devait être conçu pour supporter une démarche de développement basée sur le prototypage par raffinements défini comme un ensemble d'opérations :

- de modélisation du système à développer,
- d'évaluation de ce système en promouvant l'usage des méthodes formelles,
- de génération automatique de programmes en vue d'obtenir une implémentation de la supervision conforme au modèle analysé.

Pour que nos travaux soient acceptables par des développeurs, nous avons essayé de les placer dans le contexte d'UML. Pour bien couvrir les trois ensembles d'opérations que nous avons énumérés, nous proposons de procéder par vues complémentaires.

La première vue s'appuie sur UML (les aspects modélisation). Cependant, la version actuelle du standard UML (2.0) ne supporte pas les aspects dynamiques. Nous avons donc analysé de nombreuses propositions issues de la communeauté scientifique allant dans ce sens en vue d'en faire la synthèse dans un langage de spécification répondant à nos objectifs : **L*f*P** (Language for Prototyping).

Deux autres vues complémentaires permettent de préciser les propriétés attendues (pour la vérification) et des annotations nécessaires pour la génération de code. Ces nouvelles informations sont à croiser : un générateur de programme tirera avantage d'informations structurelles comme des bornes de buffer ; de même certaines informations relatives à l'implémentation, comme l'existence de priorités entre composants du système, permettent de réduire la complexité d'analyse (en réduisant les entrelacements dans le cas d'une étude par model checking).

Ce langage de modélisation a été pensé afin de respecter à la fois les principes posés dans les méthodologies à base d'UML, la possibilité de liaison avec des techniques d'analyse formelles dédiées à l'analyse des systèmes répartis (les réseaux de Petri) et en gardant à l'esprit l'implémentabilité du langage en vue d'une génération automatique de programmes s'appuyant sur les middleware actuels.

Notre apport consiste en les points suivants.

Apport méthodologique

L'analyse de l'état de l'art dans le domaine de la spécification de systèmes répartis à montré que deux aspects complémentaires sont à considérer :

- la spécification du système en lui-même, qui s'appuie actuellement sur UML mais aussi parfois sur des extensions (ou d'autres notations) afin de capturer sans ambiguïté les aspects dynamiques,
- la description de l'architecture, qui se base sur des langages dédiés (ADL ou Architecture Description Languages) et vise à capturer les informations permettant de déployer le système sur une architecture cible.

Si la spécification du système permet de décrire le système sans ambiguïté, alors on peut envisager une liaison avec les méthodes formelle pour le vérifier formellement (et non le valider par simulation seulement). La description de l'architecture enrichit la configuration du générateur de code et permet de fournir une aide au déploiement du système.

La **Figure 6-1** illustre le type de démarche que permet un langage de description comme **L*f*P**. Ce langage est actuellement le point d'entrée d'une démarche de développement par prototypage et nous verrons plus loin qu'il peut être construit par enrichissement d'une spécification UML. Il sert de base à la génération automatique de programmes et, dans son état actuel, permet d'envisager la production des spécifications formelles à des fins d'analyse.

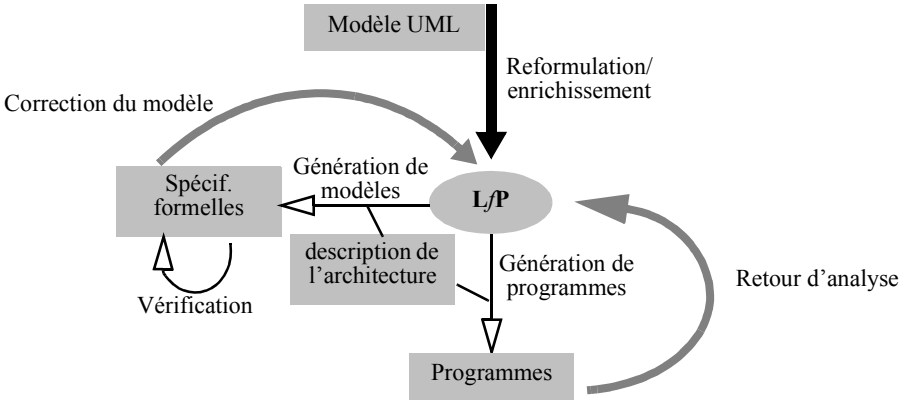


Figure 6-1 : Approche de prototypage par raffinement avec **L*f*P**.

Nous avons ainsi deux cycles principaux. le premier permet de corriger le modèle sur la base de son analyse et le second permet, par étude de l'exécution du prototype généré, de raffiner certaines stratégies de génération de code et les choix d'implémentation définis dans la vue implémentation.

L'intérêt de notre langage est de regrouper dans une seule spécification les informations requises aux différents stades du développement d'un logiciel mais aussi de fournir un support pour la méthodologie. Les ingénieurs ne peuvent négliger les problèmes de liaisons et de communications entre les composants d'une application répartie : ils doivent explicitement intégrer leurs choix, ainsi que les particularités d'une architecture cible dans la spécification. L'expérience montre que ces choix peuvent avoir un impact dramatique sur les objectifs d'une application. Comme ils sont saisis dans le modèle (même si c'est plus tard dans le cycle de développement), il reste possible d'analyser leur impact sur les propriétés attendues.

Apport technique

L*f*P répond aux objectifs que nous nous sommes fixés dans le cadre d'un prototypage par raffinements. Sur les aspects vérification, le langage force le concepteur d'un système à intégrer les bonnes informations sans être un spécialiste des techniques formelles sous-jacentes. Il le contraint également à tenir

compte de toutes les hypothèses et contraintes exprimées au fur et à mesure du processus de conception. L'étude de cas présentée dans **[71]** l'a clairement caractérisé : l'analyse d'une spécification formelle générée (manuellement en respectant pourtant les règles de traduction que nous avions fixées) a démontré que les conclusions d'une analyse précédente, effectuée à partir d'un modèle conçu directelent avec des réseaux de Petri était fausse.

Le problème de cette première étude était méthodologique : il venait d'une optimisation malencontreuse du réseau de Petri réalisée après la phase de spécification (un assemblage de modules conçus séparément par fusion de transitions). Cette optimisation, ayant pour objectif de réduire la complexité de l'espace d'états sur une spécification de grande taille avait été réalisée dans le «feu de l'action» mais ses promoteurs avaient perdu de vue que les communications entre certains des composants ainsi fusionnés était asynchrones. L'analyse, bien que pertinente, reposait sur une spécification violant une hypothèse de base du système.

L'étude menée avec **L*f*P** n'a pas posé ce problème car l'information issue du cahier des charges ne pouvait être malencontreusement perdue : la transformation, systématique, n'obéissant plus aux aléas de «choix d'implémentation».

Sur les aspects génération de code, **L*f*P** offre une grille d'analyse permettant de déduire une architecture type **[44]**. Cette architecture repose sur un exécutif qui virtualise l'environnement d'exécution en définissant l'ensemble de services minimum requis pour supporter l'exécution d'une spécification **L*f*P**. On se situe bien dans le contexte MDA au niveau d'un PIM (Platform Independant Model) que l'on peut dériver en PSM (Platform Specific Model) par intégration de la vue implémentation.

Enfin, nous avons utilisé **L*f*P** dans le cadre de plusieurs étude de cas :

- une étude sur un système de convoyeurs **[105]** dont les conclusions ont déjà été évoquées,
- l'analyse d'une station-service automatique qui est présentée en annexe de ce document et a fait l'objet d'une publication dans une conférence **[106]**; nous la présentons en annexe de ce mémoire,
- l'analyse du contrôle de vitesse dans les rames du BART (Bay Area Rapid Transit District de San Francisco) qui fait l'objet d'un chapitre de livre collectif à paraître en 2004.

6.2 Perspectives

Nos travaux ouvrent des perspectives dans les domaines suivants.

Aspects méthodologiques

Un travail intéressant concerne la liaison entre UML et **L*f*P** (la flèche noire de la **Figure 6-1**). Ce travail est nécessaire si l'on veut envisager l'utilisation d'un tel langage dans un contexte industriel. La notion de profil UML introduite dans la version 2.0 du standard ouvre des perspectives dans ce domaine.

L'idée est de définir un profil UML intégrant des informations complémentaires permettant de produire automatiquement la spécification **L*f*P**. Le langage devient alors un pivot vers la vérification formelle et la génération de code. Ce pivot pose les limites de ce qui peut être supporté par la méthode dans le domaine de la vérification et de la génération de code. Il constitue alors un langage intermédiaire au sens d'un compilateur : il définit les possibilités opératoires du PIM.

Une telle étude est en cours dans le cadre du projet RNTL MORSE, labelisé en 2002 et démarré en Juillet 2003.

Génération automatique de programmes

La constructions de générateurs automatiques de programmes est en cours d'étude. Dans ce domaine, deux types de travaux nous semblent intéressants :

- La définition de générateurs de code «clefs en main» prenant à leur charge les choix d'implémentation pour un modèle de répartition donné. A ce titre, l'utilisation de plusieurs générateurs de code pour des cibles différentes constitue un défi afin de montrer que l'on peut viser des systèmes répartis hétérogènes.
- L'exploitation d'environnements hautement configurables, comme le middleware PolyORB **[102]** afin d'expérimenter comment des paramètres de déploiement et de configuration peuvent être remontés au niveau du langage dans des termes facilitant la compréhension d'ingénieurs compétents mais non spécialistes de la répartition.

Certains de ces aspects sont étudiés dans le cadre de la thèse d'université de Frédéric Gilliers et du projet RNTL MORSE.

Vérification formelle

Nous avons conçu **L*f*P** comme un langage permettant d'encapsuler des méthodes formelles. L'objectif est de fournir, à travers son sucre syntaxique, une méthode d'utilisation raisonnable des méthodes formelles. Des mécanismes comme les types opaques (données véhiculées mais qui ne peuvent être manipulées) et la structuration des messages (sous la forme d'un discriminants + un corps non manipulable au niveau d'un média) permet de simplifier le processus de génération d'une spécification formelle.

Ainsi, à l'optimisation «en aval» de la vérification (comme les techniques de représentation optimisée de l'espace d'état en model checking ou les réductions dans les réseaux de Petri) s'ajoute une optimisation «en amont» similaire à celle réalisée par un spécialiste de la modélisation qui exclut de la spécification d'un système les éléments risquant de compliquer la preuve d'une propriété donnée. L'intégration de ce type de connaissances dans un outil automatisable est un atout pour la diffusion des méthodes de vérification formelles.

Ces aspects sont en cours d'étude dans le cadre de la thèse de Yann Thierry-Mieg.

Enfin, un objectif plus immédiat consiste à implémenter les règles de traductions de **L*f*P** vers les réseaux de Petri afin de fournir une base pour des études sur des systèmes plus complexes que ceux étudiés précédemment. Cette première phase de traduction constituera une base pour l'expérimentation des optimisation «en amont» évoquées plus haut.

ANNEXE

Etude de cas : La Station service

1 Introduction

UML est le standard largement adopté par l'industrie pour la conception de systèmes informatiques. Cependant, il reste mal adapté au domaine des applications réparties, à fortiori dans le domaine de l'embarqué. En effet, la conception orientée objet nécessite l'existence d'un modèle de répartition sous-jacent, généralement fourni par des intergiciels (tels que CORBA **[87]**). Cependant, ces intergiciels sont difficiles à utiliser dans des cadres contraints tels ceux des applications embarquées réparties à cause des impératifs liés d'une part à la validation qui doit aussi valider l'intergiciel, et d'autre part des contraintes pesant sur l'environnement d'exécution qui requièrent un bon niveau de performance.

Les ADL (Langages de Description d'Architecture Logicielle) proposent une solution adaptée à ce type de systèmes. Ils permettent de caractériser une application en termes de *composants* (i.e. programmes s'exécutant sur des sites) reliés par des *connecteurs* selon une *configuration* (communications entre les

composants) et soumis à des *contraintes* (signatures de méthodes, enchaînements de méthodes, etc.). Ils sont utilisés pour décrire des architectures logicielles de manière formelle ou semi-formelle **[80]**. Par rapport aux langages de programmation, les ADL permettent de structurer la réflexion et de gagner en pouvoir d'abstraction. Ils s'appuient sur la séparation entre la description des composants, celle de leurs interfaces, et celle de leur interconnexions.

Dans ce contexte, des approches orientées modèles, telles que MDA (Model Driven Architecture), prônée par l'OMG **[93]**, s'avèrent intéressantes. On les appelle aussi "prototypage par raffinements" (*evolutionary prototyping*) **[76]**. Le principe est de concevoir le système en utilisant un modèle exprimé dans un langage dédié au domaine d'application visé (ici, les systèmes répartis). Ce modèle sert ensuite à la génération automatique de programmes conformes à la sémantique du comportement spécifié. Il sert également de base à la vérification de propriétés qui doivent être respectées.

Nous proposons un langage de spécification, **L*f*P** (Language for Prototyping) **[105]** pour décrire un système réparti, appuyé par une méthodologie de prototypage par raffinements. **L*f*P** propose une structuration en classes inspirée d'UML, et présente les caractéristiques d'un ADL. De plus, **L*f*P** promeut une approche orientée modèle permettant la vérification formelle du système à développer et la synthèse automatique du squelette de contrôle réparti de l'application.

L*f*P répond aux besoins des applications réparties. Son objectif est de fournir des abstractions adaptées à la modélisation des mécanismes de contrôle réparti et de proposer une correspondance sur des architectures variées comme CORBA, Java/RMI **[126]**, Ada/Distributed System Annex **[60]**, ou des bibliothèques de communication. Pour permettre une telle variété, nous nous sommes focalisés sur la définition des relations entre l'application, l'exécutif (les bibliothèques qui assurent la sémantique opérationnelle de **L*f*P**) et l'environnement d'exécution (système d'exploitation et architecture matérielle).

Nous positionnons notre approche par rapport à la notation UML et à d'autres ADL, puis nous présentons les grandes lignes de notre méthodologie et du formalisme **L*f*P** associé en **Section 2**. La **Section 3** présente un exemple classique, et sa modélisation à l'aide de **L*f*P**. Enfin, la **Section 4** présente les techniques utilisées pour valider la spécification par transformation en réseaux de Petri et Model-checking.

1.1 Aspects comportementaux et architecturaux d'un système réparti

UML est le standard *de facto* pour décrire des applications objets. Cependant, malgré l'existence de nombreux environnements de qualité et d'un soutien industriel indéfectible, UML souffre de lacunes rendant délicate son utilisation dans le contexte des systèmes répartis et/ou embarqués:

- UML **[90]** est un langage semi-formel qui définit correctement les aspects statiques d'un système, mais dont la sémantique est imprécise dans sa globalité.
- Le modèle de répartition favorisé par UML est naturellement basé sur des notions de bus logiciels dont l'implémentation est lourde.
- Il est reconnu qu'UML ne permet pas de décrire une architecture logicielle au sens ADL du terme **[80]**, dans la mesure où il ne permet pas une séparation de la spécification des connecteurs par rapport aux composants.

1.2 Formalisation des aspects dynamiques en UML

La sémantique d'UML décrit parfaitement la structure d'un système. Ce n'est toutefois pas le cas des aspects dynamiques (ou comportementaux) qui sont exprimés au moyen de différents diagrammes (séquence, collaboration, activité, états) sans que la cohérence entre eux puisse être assurée **[21]**. Récemment, des groupes tels que pUML **[36]** ont proposé une sémantique rigoureuse pour la version

2.0 d'UML (à l'aide du langage OCL) mais ces extensions n'abordent toujours pas les aspects dynamiques.

Les aspects dynamiques concernent non seulement les diagrammes décrivant le comportement mais également leurs relations. Dans ce sens, des travaux récents basés sur la théorie des automates d'états, comme les réseaux de Petri Objets/Hiérarchiques **[31, 113]**, ou sur Promela **[55]** ont montré qu'il était possible d'assembler de manière cohérente l'information répartie dans les différents diagrammes de comportement d'UML à des fins de vérification formelle, notamment par model checking **[74, 46]**. Ces travaux mettent en avant un procédé de synthèse automatique pour les spécifications formelles.

L'utilisation de notations de haut niveau (par exemple, orientées objets) présente l'avantage de fournir des mécanismes de composition et de structuration plus simples à manipuler et fournit une base intéressante pour la traçabilité des informations dans le système. D'autres notations, basées sur l'algèbre des processus (E-LOTOS **[24]**) ou sur des langages d'assertions (Object-Z **[7]**), sont également utilisées pour détecter des incohérences et des ambiguïtés dans les modèles UML au début du processus de développement.

Bien que la formalisation d'UML permette l'utilisation de méthodes de vérification formelles, les outils correspondants requièrent une expérience importante, considérée comme prohibitive pour la plupart des projets industriels. Il est dans ce cas souhaitable de cacher la complexité d'utilisation de ces outils aux développeurs en l'encapsulant dans les services rendus par un AGL **[75]**. C'est un des objectifs de la notation **L*f*P**: la vérification de propriétés d'un modèle **L*f*P** se fait par traduction vers des réseaux de Petri, puis model-checking. Les éventuelles traces d'erreurs seront ensuite remontées au niveau du modèle **L*f*P**, sous la forme de traces simulables sur le modèle, cachant ainsi une large partie de la complexité des techniques mises en oeuvre pour la vérification.

1.3 Description d'architecture logicielle

UML n'est pas réellement adapté à la description d'architectures **[81]**, car il ne propose pas d'abstraction correspondant aux connecteurs, qui composent le coeur d'un ADL. Pour pallier à cela, une possibilité est d'étendre la notation UML à l'aide de profils **[65]**, permettant ainsi de définir un mapping d'un ADL vers UML afin de pouvoir utiliser cette notation qui est plus connue dans un contexte industriel. Cependant, on ne peut espérer créer un profil qui permette d'inclure les capacités de tous les ADL, et si UML possède un pouvoir d'expression égal ou supérieur à la majorité des ADL, les abstractions qu'il fournit sont mal adaptées aux besoins de la conception de systèmes répartis.

L'objectif d'un langage est de fournir des notations adaptées à la représentation des concepts manipulés par les experts du domaine. UML n'atteint pas cet objectif en termes de description d'architecture car on est forcé d'encapsuler certains concepts (comme les connecteurs) dans d'autres abstractions (des classes), rendant plus difficile la modélisation. De plus, les modèles ainsi spécifiés, bien qu'effectivement décrits en UML, n'ont plus la même sémantique: une classe portant le stéréotype connecteur n'a que peu de rapport avec une classe ``classique'', limitant la réutilisabilité et la communication sur la base d'un langage commun, qui est la grande force d'UML.

Notre approche s'inspire du framework ODP **[62]**, qui fournit un standard ouvert dédié au développement d'applications réparties, tout en étant compatible avec la notation UML. De cette manière nous gardons un lien étroit avec la notation UML, mais nous définissons des abstractions (médias, binders) qui dépassent les concepts orientés-objet de classe, et qui sont mieux adaptés au domaine visé.

1.4 UML, intergiciels et validation

La conception d'un système à l'aide d'UML s'appuie sur un découpage de l'application en classes. Les mécanismes de communication entre instances sont implicitement laissés à la charge de l'exécutif, et sont le plus souvent implémentés à l'aide d'intergiciels (i.e. CORBA **[90]**). Or, le développement d'application certifiées doit respecter des normes strictes imposant la validation de toutes les composantes de l'application. Si un intergiciel est utilisé, il doit être entièrement validé pour chaque utilisation particulière ou changement d'architecture. Chaque validation doit inclure la totalité du code qui sera embarqué et les fonctions inutilisées de l'intergiciel (avec le code correspondant) doivent être clairement identifiées. Dans les cas les plus restrictifs, la présence de code mort n'est même pas tolérée et il devient nécessaire d'adapter finement l'intergiciel à l'application considérée, ce qui est en contradiction avec la démarche classique des intergiciels qui visent un haut niveau de généricité. De plus, ces intergiciels sont difficiles à utiliser dans des cadres tels que ceux des applications embarquées réparties à cause des contraintes pesant sur l'environnement d'exécution qui requièrent un bon niveau de performance.

Pour permettre le développement de logiciels *certifiables* (c'est-à-dire conformes à une norme de référence pour un domaine d'application donné), l'ADL utilisé doit décrire non seulement la structure et la sémantique de l'application, mais aussi les caractéristiques de son environnement d'exécution: protocoles de communication, gestion de ressources, et algorithmes d'ordonnancement. Néanmoins, construire une application dédiée à un exécutif cible n'est pas recommandé car cela limite la portabilité de la solution proposée.

L'approche que nous proposons intègre l'environnement dans des composants externes définissant des gabarits de comportement ce qui permet de construire l'application de manière plus générique. Ainsi en séparant clairement l'application de son environnement d'exécution, nous facilitons son déploiement sur des plate-formes différentes, et nous favorisons la réutilisation des composants existants. Nous intégrons de la sorte l'exécutif à la spécification du système sous la forme d'éléments de configuration. Si les paramètres de cette configuration sont bien maîtrisés, les pertes en performances sont négligeables **[117]**. Néanmoins, chaque instanciation de l'application sur un runtime cible doit donner lieu à la certification du runtime et de l'application. Cette solution constitue un bon compromis entre certification et coût de développement.

2 Présentation du langage L*f*P

Cette section présente notre proposition de langage pour la modélisation d'applications réparties. Dans un premier temps nous présentons la méthodologie associée à notre approche, puis nous décrivons succinctement le formalisme **L*f*P**.

2.1 Méthodologie d'utilisation de L*f*P

Notre méthodologie de développement par prototypage est adaptée aux systèmes répartis embarquables. Notre objectif est de lier fortement la spécification du système, la preuve des propriétés énoncées et les programmes correspondants. Cette approche s'appuie sur un *modèle pivot* permettant la vérification formelle (au moyen de réseaux de Petri Colorés) et la génération automatique de programmes réalisant le contrôle réparti du système. Cette démarche permet de travailler au niveau du modèle puisque les programmes sont obtenus automatiquement à un coût faible. Ce modèle sert également comme base pour la vérification formelle.

La partie contrôle des systèmes répartis (synchronisation entre les composants, respect de l'intégrité des ressources, etc.) est la plus délicate à réaliser. Elle doit bénéficier d'une attention particulière lors de la modélisation et de la vérification. La génération automatique de programmes permet à un non-spécialiste d'obtenir, à partir du modèle pivot, le squelette de contrôle de l'application conforme à la spécification vérifiée. Ce squelette est ensuite enrichi par du code séquentiel[1] puis déployé dans un environnement d'exécution donné.

L*f*P, le formalisme proposé pour le modèle pivot, peut être vu comme un diagramme complémentaire d'UML regroupant les informations suivantes :

- Les données issues des différents diagrammes d'UML (classe, collaboration, séquence, états) sont utilisées pour construire l'ossature d'une spécification **L*f*P**.
- L'usager intègre ensuite les propriétés attendues dans le système (sous la forme d'invariants, de formules de logique temporelle, etc.).
- L'usager ajoute enfin des directives d'implémentation (langage cible, référence à des composants logiciels préexistants, morceaux de code séquentiel, etc.) et des «directives de déploiement» (références à des éléments de l'architecture cible d'exécution) qui seront utilisées par des générateurs de programmes.

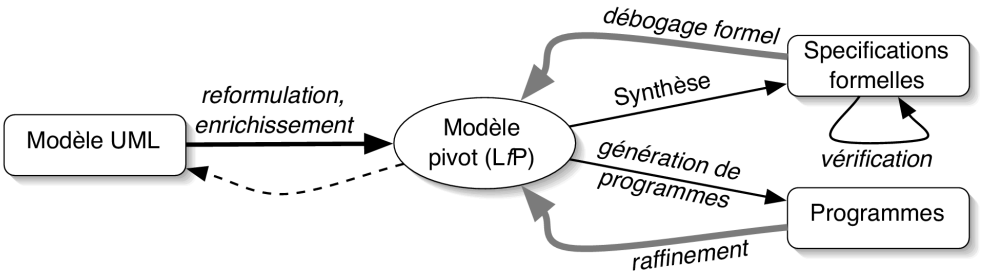


Figure 1 : Vue générale de la méthodologie.

La **Figure 1** illustre notre méthodologie. Un diagramme **L*f*P** est construit par raffinement et enrichissement d'une spécification UML. À partir des propriétés identifiées dans le modèle, un processus de synthèse produit des spécifications formelles sur lesquelles sont appliquées des preuves. La sémantique des propriétés à vérifier est utilisée pour réduire la taille du modèle ainsi généré en supprimant les éléments de spécification n'intervenant pas. On est donc amené à générer autant de modèles que de propriétés à vérifier. On effectue ainsi par raffinements successifs le déboggage de la spécification au niveau du modèle pivot, jusqu'à ce que ce dernier vérifie toutes les propriétés énoncés.

Une fois les propriétés d'un diagramme **L*f*P** démontrées, on obtient une implémentation conforme du système par génération de code. Le diagramme **L*f*P** peut à nouveau être modifié en fonction des observations effectuées lors des exécutions ou en réponse à une évolution du cahier des charges. On obtient alors un cycle de production par raffinements de la spécification **L*f*P** qui comprend les étapes suivantes: synthèse, vérification, correction, génération de programmes, exécution, optimisation, et évolutions.

2.2 Le formalisme L*f*P

L*f*P est un langage formellement défini permettant de décrire le contrôle d'une application répartie. Il possède à la fois les caractéristiques d'un ADL et celles d'un langage de coordination:

- Il reste lié à une approche basée sur UML. Les concepteurs d'un système utilisent UML pour la phase de conception du système puis passent à **L*f*P** dès qu'il faut exprimer précisément l'architecture logicielle et le comportement des composants répartis.
- Il s'inspire de l'approche RM-ODP **[62]**. Nous nous sommes en particulier intéressés aux vues

ingénierie, traitement et *technologie*. ODP nous apporte une démarche d'identification des caractéristiques d'une application particulièrement adaptée aux systèmes répartis.

- Il est défini formellement en se basant sur les réseaux de Petri **[45]**. Cela rend possible la vérification formelle d'un système dès les premières phases de réalisation, ce qui est dans la ligne de l'approche *Model Driven Architecture* (MDA), prônée par l'OMG **[93]**.

Pour remplir ces objectifs, **L*f*P** définit une notation non ambiguë adaptée aux systèmes répartis basée sur trois vues orthogonales: la vue *fonctionnelle*, la vue *propriété* et la vue *implémentation*.

La vue fonctionnelle décrit l'architecture et le comportement du système. Elle contient:

- une partie déclarative contenant des informations de traçabilité (par exemple, les références vers des composants du modèle UML d'origine) et la déclaration des types et constantes utilisés dans le modèle,
- un graphe hiérarchique décrivant l'architecture du système, ce graphe est composé de *classes*, *média* et *binders*.

Une classe **L*f*P** correspond à une classe instanciable UML. Les média permettent de décrire les schémas de communication (protocoles) entre classes. Les relations d'association, d'agrégation, et de composition d'UML peuvent être capturées au moyen de média. Les binders représentent des points d'entrée (ports de communication) entre les classes et les média.

Les binders **L*f*P** connectent des ports de communication (à laé manière de ce qui se fait dans d'autres ADL) et sont placés entre classes et médias. Ils décrivent des caractéristiques simples de la sémantique d'interaction: direction des communications, (a)synchronisme, ordonnancement (FIFO, LIFO), capacités, cardinalités. La possibilité d'associer plusieurs binders à une classe permet de spécifier des protocoles d'interaction complexes, à la différence de l'unique point d'entrée FIFO du modèle événementiel des statecharts UML.

La vue propriété explicite les propriétés à vérifier sur le système. Ces propriétés peuvent être considérées comme des obligations de preuve (au sens des assertions dans B **[1]**). Ces propriétés sont exprimées en complément de la vue fonctionnelle. Elles prennent la forme d'invariants (pour exprimer une exclusion mutuelle ou une condition sur un ensemble de variables), de formules de logique temporelle (pour exprimer une causalité entre des événements), ou de post-conditions sur les actions. Les informations de cette vue sont exploitées pour la vérification formelle ainsi que pour l'insertion de runtime-checks dans les programmes générés.

La vue implémentation définit les contraintes d'implémentation du système et de l'environnement d'exécution ciblé, le langage utilisé par le générateur de code, ou encore des informations de déploiement du système. Nous nous sommes inspirés de la notion de tag dans UML, ces derniers permettent d'associer toute sorte d'informations ponctuelles aux entités de **L*f*P**. Les informations sont exploitées pendant la génération automatique de programmes.

3 Un exemple : la station service

Cette section présente un exemple simple, la station service introduite dans **[54]**, destiné à illustrer les principales caractéristiques de **L*f*P**. Cet exemple a été souvent réutilisé pour comparer l'efficacité de techniques de vérification, comme dans **[25, 142]**. Dans la suite nous nous appuyons sur cet exemple pour exhiber les caractéristiques du langage **L*f*P**.

3.1 Présentation de l'exemple

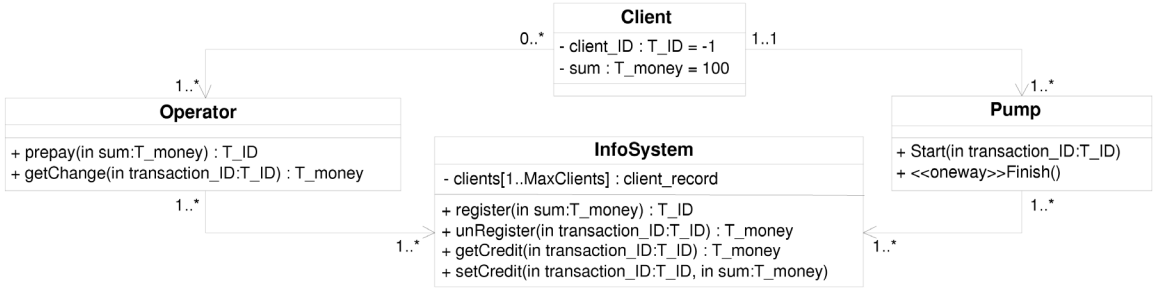


Figure 2 : Le diagramme de classe UML de la station service.

Considérons le comportement simplifié d'une station service dont le diagramme de classe UML est donné en **Figure 2**. Le système est composé de quatre entités: l'opérateur (**`operator`**), les clients (**`client`**), les pompes (**`pump`**) et le système d'information de la station (**`infosystem`**). Lorsque le client entre dans la station, il prépaye l'opérateur, et reçoit un *ticket* qui comporte son *identifiant*. Le client se rend ensuite à une pompe, insère son *ticket* et remplit son réservoir en étant limité par la somme d'argent prépayée. Enfin il retourne voir l'opérateur pour récupérer sa monnaie avant de quitter la station.

Les informations concernant les clients sont centralisées dans le système d'information de la station (**`infosystem`**). L'opérateur enregistre le client (opération **register**) lorsque celui-ci effectue son prépaiement (**prepay**). La pompe récupère le plafond (**getCredit**) attribué à ce client quand il fait **start** avec son ticket, et met à jour le crédit du client (**setCredit**) lorsque le message **finish** est envoyé par le client. L'opérateur retire le client du système d'information (**unRegister**) lorsque celui-ci vient récupérer sa monnaie (**getChange**).

3.2 Le diagramme d'architecture de la station service

Les classes (représentées par des rectangles) du schéma d'architecture d'un modèle **L*f*P** sont complétées par des média (les tubes) décrivant les protocoles de communication entre les composants du système, correspondant aux associations UML.

Les média sont connectés aux classes via des *binders* (double barre noire) correspondants à des points d'entrée ou de sortie de messages. Le média décrit le comportement d'un lien de communication (est-il fiable? préserve-t-il l'ordre des messages ?). Le binder modélise l'interaction entre un média et la classe: il permet d'indiquer s'il y a une file d'attente par instance de classe (cardinalité *1* du coté de la classe) ou une file d'attente unique pour l'ensemble des instances de classe (cardinalité *all*). Un *binder* peut également contenir des informations comme la capacité maximale de la file d'attente correspondante ou le sens de circulation des messages (unidirectionnel, bidirectionnel).

Le diagramme d'architecture **L*f*P** de ce système est présenté en **Figure 3**. Quatre classes **L*f*P** correspondent aux quatre classes UML. Une classe **L*f*P** dispose d'un flot d'exécution propre exprimé au moyen d'un automate. Le concept est clairement inspiré d'un `task type` en Ada **[60]**: chaque instance de classe dispose de son propre compteur ordinal et correspond à un processus ou une thread.

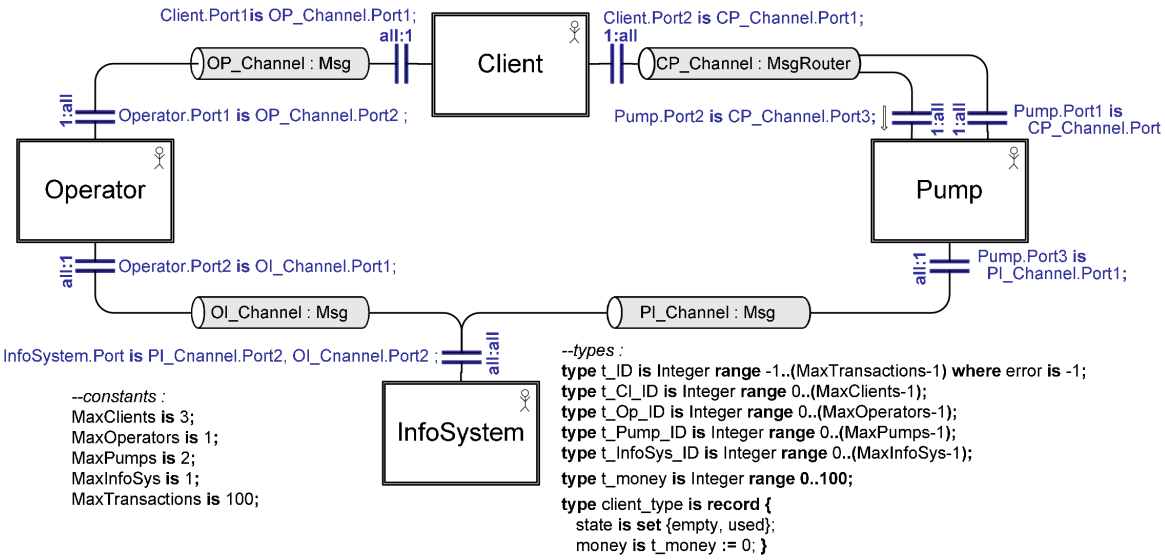


Figure 3 : Le diagramme d'architecture logicielle **L*f*P** de la station service.

Dans notre exemple, les classes communiquent à travers des canaux bidirectionnels; les média auront donc un schéma unique décrivant les caractéristiques de ce type de communication. La seule exception est le média connectant les clients aux pompes CP_channel : selon la nature du message que le client adresse à la pompe, il sera aiguillé sur l'un des deux ports (binders) de la pompe.

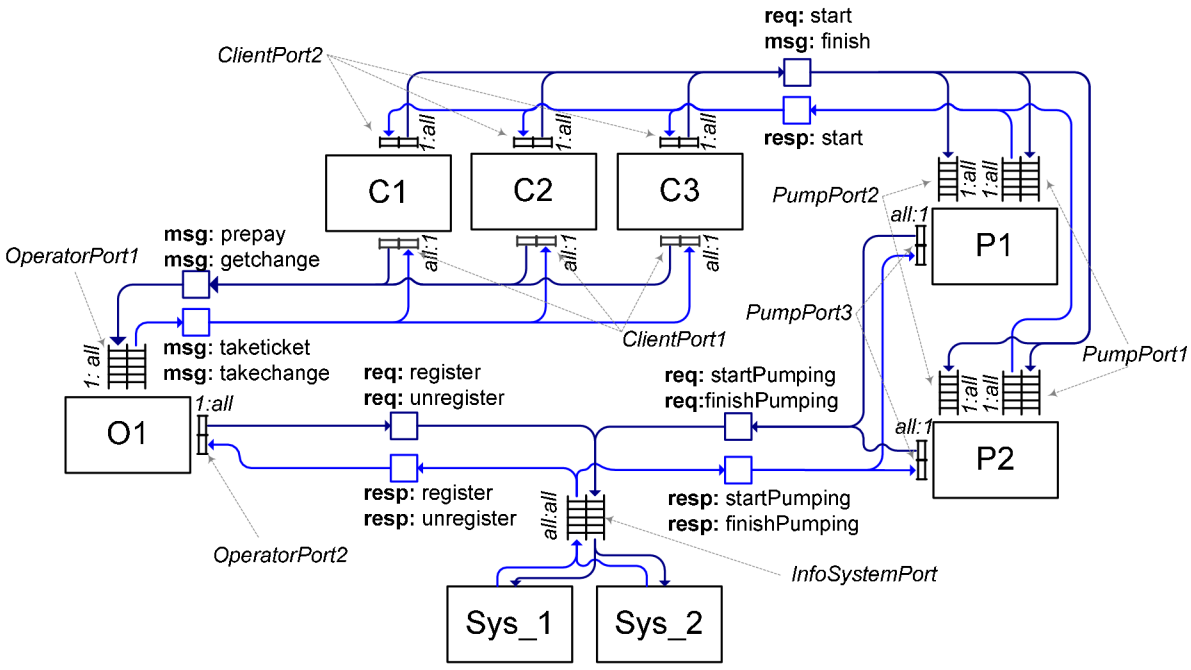


Figure 4 : Diagramme d'architecture déplié, pour 3 clients, 2 pompes, 2 InfoSystem et un opérateur

La **Figure 4** présente le dépliage du diagramme d'architecture pour trois clients, deux pompes, un opérateur et deux systèmes d'information. Les classes et média **L*f*P** seront détaillés plus tard, nous précisons ici la signification des cardinalités portées par les binders à travers un diagramme de déploiement **L*f*P**. Les binders de cardinalité `1:all` se traduisent par deux files de messages (entrants et sortants) pour chaque instance de classe connectée à un média commun partagé par tous. Le binder `all:all` qui

lie les systèmes d'information au reste du système correspond à une file unique partagée par les deux instances *I1* et *I2* et commune aux médias OI_channel et PI_channel représentés sur la **Figure 3**. Le fonctionnement de cette file est similaire à celui d'une file d'attente: dès qu'un guichet se libère (**`info-system`**) le premier client de la file est servi (requête d'un opérateur ou d'une pompe). Les communications d'une pompe sont réalisées à travers deux binders bidirectionnels et un binder unidirectionnel.

3.3 Description d'une classe

Le diagramme de comportement (**L*f*P**-BD ou *Behavioral Diagram*) est une représentation graphique hiérarchique du comportement d'une classe. Il exhibe les actions que peut exécuter une instance de classe ou de média en fonction de son état interne. Le **L*f*P**-BD permet ainsi de spécifier la façon de se servir d'une entité **L*f*P**, contraignant ainsi l'ordre des appels aux services. On définit de la sorte le comportement interne d'une classe **L*f*P** mais aussi le protocole (i.e. enchaînement de méthodes) qui doit être respecté pour en garantir un bon fonctionnement. Cette notion augmente la réutilisabilité des composants définis en **L*f*P** car on peut vérifier qu'un nouveau composant s'intègre correctement dans un système préexistant (i.e. le protocole qu'il offre est compatible). En contraignant les possibilités d'utilisation du composant, on prévient son usage abusif par l'environnement. On s'assure donc à travers cette *signature comportementale* que, quel que soit le comportement de l'environnement, le composant préserve sa cohérence interne.

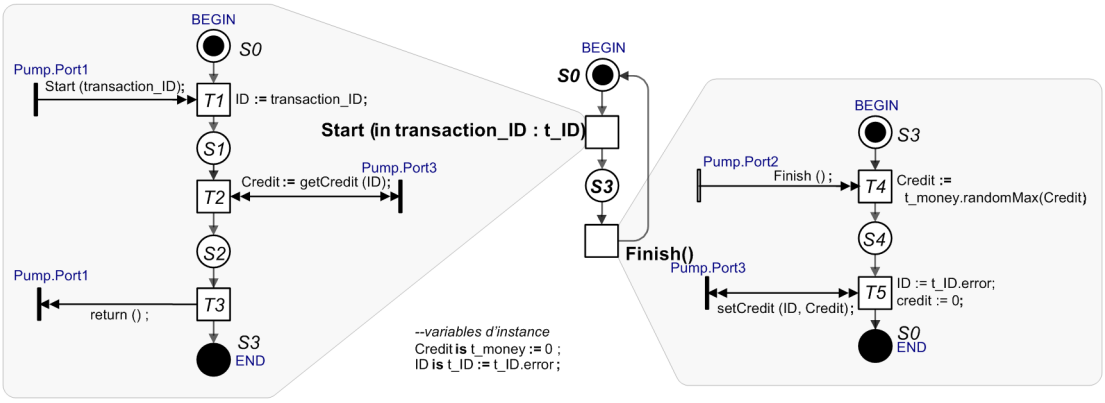


Figure 5 : Le diagramme de comportement (**L*f*P**-BD) déployé de l'objet **`pump`**.

La **Figure 5** exprime le comportement de la pompe à travers son **L*f*P**-BD. Au centre le **L*f*P**-BD principal (main-**L*f*P**-BD) de la classe décrit l'ordre dans lequel les appels aux services offerts par le composant doivent s'effectuer. Il exprime que la méthode *start* doit précéder la méthode *finish*. De part et d'autre, les **L*f*P**-BD des deux méthodes, sont déployés avec les instructions et les communications associées.

- Les **déclarations** de types et de variables prennent la forme d'annotations sur le diagramme. La visibilité des déclarations est limitée au diagramme contenant la définition ainsi qu'aux diagrammes qu'il contient hiérarchiquement. Le contexte d'une classe **L*f*P** contient des variables locales (une copie par instance de classe), des variables partagées entre les instances d'une classe (notées <<static>>). Toute variable **L*f*P** doit être initialisée, mais on peut définir une valeur par défaut à la déclaration du type de la variable.
  La pompe dispose de deux variables locales, **ID** et **Credit**, qui permettent respectivement de mémoriser l'identité du client en cours de traitement, et le plafond qui lui est associé.
- Les **états** représentent les étapes de l'exécution du comportement d'une classe. Deux états spéciaux sont distingués: `BEGIN` et `END` qui correspondent respectivement à l'état initial et à l'état final de l'exécution. Les **L*f*P**-BD n'ont qu'un seul état initial. Pour une méthode l'état final correspond à

l'état suivant du main-**L*f*P**-BD, l'état final du main-**L*f*P**-BD correspond à la destruction de l'instance.

Ainsi l'état initial `S0` est le premier état de la méthode *start*, et l'état `S3` intermédiaire est à la fois l'état final de *Start* et l'état initial de *Finish*.

- Les **transitions** représentent les actions à effectuer en fonction de l'état courant d'une instance de la classe. Une transition peut référencer un rôle ou une méthode d'une classe. Elle doit donc être décrite par un sous-**L*f*P**-BD (hiérarchique). Une garde spécifie les préconditions à satisfaire pour qu'elle puisse être tirée. Le début de la hiérarchie est prédéfini: les transitions du main-**L*f*P**-BD référencent soit des descriptions de méthodes, soit la description de rôles, qui référencent eux-mêmes la description de méthodes. Ces différents niveaux de hiérarchie sont définis pour offrir différentes vues sur le comportement d'un système, selon la granularité de description souhaitée.
- On peut associer aux transitions un **invariant** ou **post-condition** utilisé pour la vérification ou pour définir un runtime check à insérer lors de la génération du squelette du programme.
- On associe du **pseudo-code séquentiel** aux transitions. Ce pseudo-code peut modifier les variables accessibles à ce niveau hiérarchique au moyen d'un jeu d'instructions prédéfini spécifique à chaque type manipulé (lecture et écriture des variables simples, accès indexé aux tableaux, comparaisons, additions ...). **L*f*P** dispose de types prédéfinis ainsi que d'opérateurs associés (types énumérés et numériques discrets, tableaux, multi-ensembles...). Les utilisateurs peuvent également décrire leurs propres types par composition des types prédéfinis ainsi que les opérations associées. Les types ainsi définis doivent être discrets et les opérations non ambiguës (e.g. s'exprimer sous la forme $f(v_1, ... , v_n) = valeur$). On fournit de plus quelques structures de contrôle courantes: boucles et conditionnelles. Par exemple, la transition `T4` (**Figure 5**) fait appel à une opération prédéfinie RandomMax, qui permet d'obtenir une valeur aléatoire sur un domaine majoré par l'argument. Cette modélisation permet de prendre en compte toutes les valuations possibles de dépense d'essence, et correspond lors de la génération de programme à un appel à une primitive de service de la pompe (module *externe*), liée au programme. Il faut s'assurer que cette primitive a bien ce comportement pour que le modèle soit fidèle à la réalité.

L*f*P propose des **sémaphores**. Les sémaphores binaires protègent des sections critiques.

- On dispose également de **constructeurs** de classes **L*f*P**, qui créent une nouvelle instance de classe (dans l'état BEGIN du main-**L*f*P**-BD), et initialisent son contexte. Le nombre d'instances maximum présent dans le système doit cependant être borné pour conserver le caractère fini du système, nécessaire pour certaines techniques de vérification. Cette borne peut être définie dans la vue implémentation. Si tel n'est pas le cas, l'approche associe à ce constructeur une obligation de preuve afin de s'assurer qu'il existe bien une borne structurelle finie correspondant au nombre maximum de processus crére dés.

3.4 Description d'un média

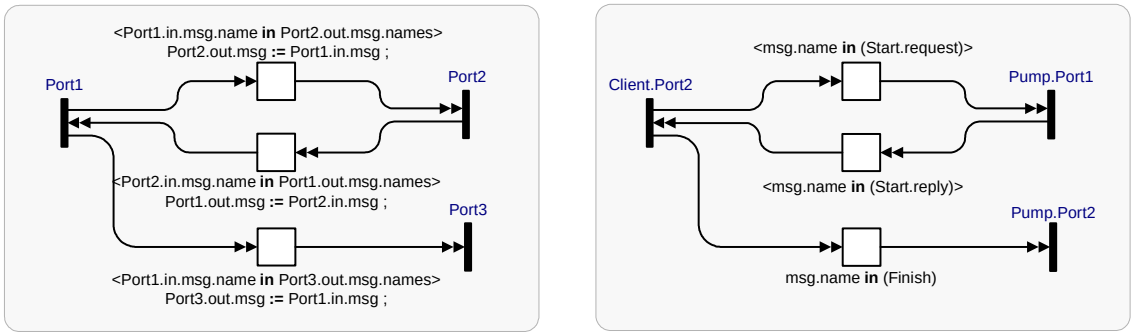
Les médias décrivent les protocoles de communication entre les classes du système. Il est possible de les utiliser comme élément de base, ou de les composer pour former des schémas de communication plus complexes. Les médias sont déduits des associations, agrégations, ou compositions UML. Ils sont déclarés de façon générique, mais leur instanciation est paramétrée par les classes frontières. Un média consiste en :

une partie déclarative similaire à celle d'une classe. Elle spécifie les éventuels types et variables locales au média.

un **L*f*P**-BD qui exprime le protocole de communication. Moins complexe que le contrat comportemental d'une classe, le **L*f*P**-BD d'un média ne comporte pas de méthodes. Un média peut être avec ou sans mémoire, dans ce dernier cas il ne comporte ni états ni variables persistantes.

des ports d'interaction spécifiant les binders **L*f*P** auxquels un média peut se connecter. Ces ports sont similaires aux *bindings points* ODP.

Un message **L*f*P** est composé d'une en-tête de taille variable et d'une partie qui véhicule les données. La structure de l'en-tête est définie dans le média et est utilisée pour le routage des messages. Elle est composée de plusieurs champs implicites (source, destination, et nom du message) et d'autres paramètres optionnels à définir par l'utilisateur.



a) MsgRouter (template) *a) CP_Channel (instance)*

Figure 6 : L*f*P-BD du modèle de média MsgRouter (a) et son instanciation dans le système (b).

La **Figure 6**.a montre le patron générique du média MsgRouter qui relie les **`client`** aux **`pump`**. Ce média est doté de 3 ports de communication: *Port1* et *Port2* connectables à des *binders* bidirectionnels (pour des requêtes RPC), et *Port3* unidirectionnel (pour des requêtes sans réponse). Le diagramme d'architecture logicielle (**Figure 3**) spécifie comment ces ports seront connectés aux binders des classes frontières. Ce média assure le routage de l'information explicitement en fonction des noms des messages, et implicitement en fonction des destinataires spécifiés dans leur en-tête. Cette dernière information est implicite car, à la définition du média le nombre et le type des instances de classes connectées n'est pas connu.

La **Figure 6**.b montre l'instanciation *CP_Channel* de ce média dans le système. Ses ports sont liés effectivement au *Port2* du **`client`**et aux *Port1 et Port2* de la **`pump`**. La connexion aux classes frontières permet de définir quels messages peuvent être effectivement acceptés par les différents binders. En l'occurrence, ce média permet de véhiculer les requêtes *Start* (RPC) de la classe **`client`** sur le *Port1* de la classe **`pump`** et les messages *Finish* asynchrones (car spécifiés <<oneway>> sur le diagramme de classe UML de la **Figure 2**) sur le *Port2* de la classe **`pump`**

Ce média exhibe deux caractéristiques intéressantes:

- La généricité de définition d'un média permet de réutiliser des protocoles complexes déjà spécifiés. Le **`client`** connecté au *Port1* ne connaît pas explicitement l'existence des différents ports de la **`pump`**, et le média dans sa forme générique ne fait aucune hypothèse sur le contenu des messages qu'il va véhiculer.
- La possibilité pour une classe de disposer de plusieurs points de connexion de sémantiques différentes, ce qui permet d'élaborer facilement des synchronisations et traitements répartis entre classes **L*f*P**. Notre démarche se distingue du modèle événementiel d'UML où tous les messages sont délivrés à travers une unique file de taille non-spécifiée. Nous adoptons une description qui s'apparente à celle de tâches communicantes (à la manière d'Ada95), ou d'objets UML actifs.

4 Éléments pour la vérification avec L*/*P

Nous nous appuyons sur les réseaux de Petri (RdP) pour vérifier des modèles **L*/*P**. Les réseaux de Petri sont particulièrement adaptés pour exprimer le parallélisme et sont employés aussi bien en analyse d'algorithmes répartis, qu'en automatique ou en évaluation de performances. Nous utilisons ici les réseaux de Petri colorés, un modèle de haut niveau permettant de décrire de façon concise des systèmes comportant des éléments répliqués.

Un réseaude Petri est un graphe biparti composé de places et de transitions. Une place, symbolisée par un cercle, est un élément de stockage qui peut contenir des jetons, elle est alors marquée. Une transition, symbolisée par un rectangle, est un élément actif représentant les évolutions du système. Le franchissement d'une transition consomme de façon atomique des jetons des places en amont de la transition (places précondition), et en restitue dans les places en aval (places post-condition). Une transition est dite franchissable si ses places préconditions sont marquées. Le choix de la transition à franchir est indéterministe parmi les transitions franchissables du réseau.

On parle de réseaux colorés lorsqu'une "couleur", ou donnée, est associée aux jetons. Une place ne peut contenir que certaines couleurs, qui forment son domaine ou son type. Un domaine peut également être crére cré à partir de la composition d'autres domaines de couleurs. Les arcs, reliant les places aux transitions possèdent une fonction (dite fonction de couleur), permettant de spécifier la nature des jetons pris ou mis dans une place lors du franchissement de la transition. Pour une présentation formelle de ce formalisme voir **[45, 31]**.

Les informations comportementales de **L*/*P** peuvent être utilisées pour créer un réseau de Petri. Les différentes classes et médias sont traduites séparément, puis reliées pour former un modèle unique sur lequel les vérifications vont pouvoir être effectuées. La suite de cette section présente l'application de certaines des règles de traduction que nous avons développées, afin de générer une spécification en RdP à partir du modèle exprimé en **L*/*P**. Plutôt que d'énumérer ces règles, nous allons présenter leur application sur l'exemple de la classe **`pump`**. Ces règles systématiques sont applicables à un modèle **L*/*P** quelconque.

4.1 Exemple de traduction : la classe `pump`

La **Figure 7** est la traduction en réseaux de Petri colorés de la classe **`pump`** présentée en **Figure 5**. Chaque instance de pompe est modélisée par un jeton qui porte la valeur des deux variables locales de la classe (**ID** du client, plafond **Credit**), ainsi qu'un identifiant unique permettant d'assurer le routage entre les instances de classes (nommage global des entités du système). Le domaine des jetons représentant une pompe est donc PumpTok composé de <*entities,client,money*>, correspondant aux variables d'instance <*monID,IDclient,Credit*>. Les jetons représentant les pompes sont initialement placés en S0, initialisés avec les valeurs par défaut (<*P1,error,0*>,<*P2,error,0*>).

La structure de l'automate de contrôle du **L*/*P**-BD est conservée: on retrouve les états `S0` à `S4` et des transitions `Ti.x` correspondant à la traduction de la transition **L*/*P** `Ti`, décomposée en pseudo-instructions élémentaires. Cette décomposition introduit des états intermédiaires notés `Si.x`.

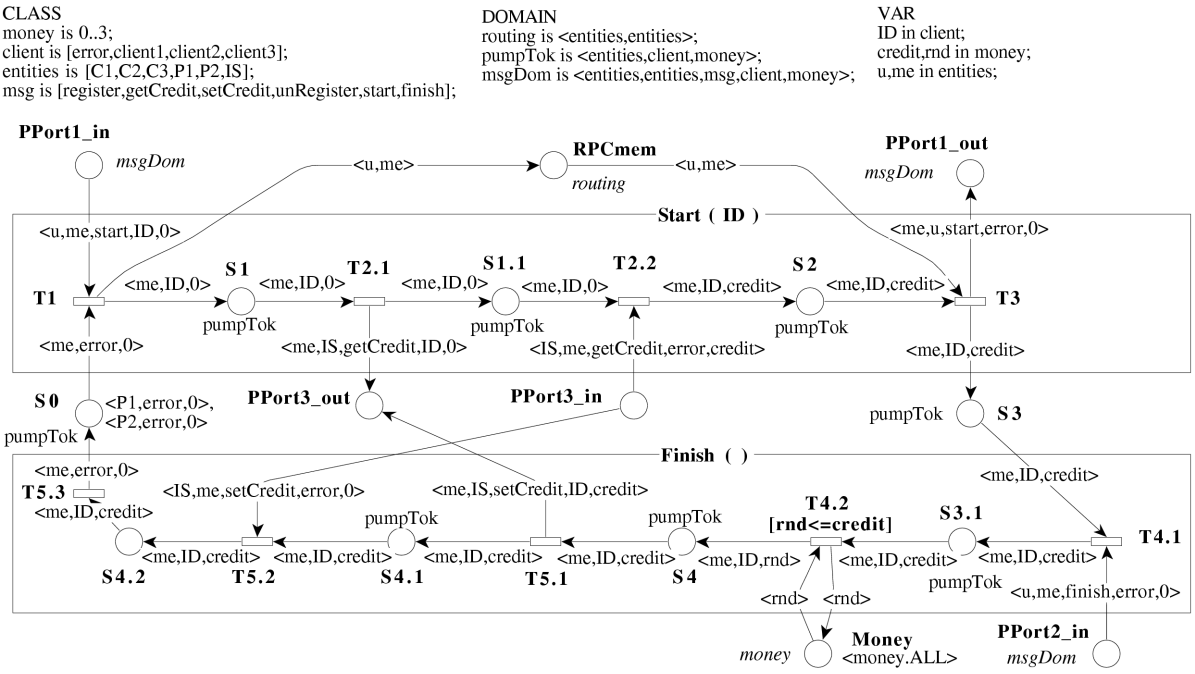


Figure 7 : Modèle réseau de Petri de la classe **L*f*P `pump`**

Les binders sont ici représentés par des places d'où arrivent les messages des autres entités. Un mécanisme FIFO peut être implémenté mais n'est pas représenté ici pour éviter de surcharger la figure. Les places représentant les binders correspondent à des places d'interface avec les autres entités du système, et permettent par fusion de connecter les parties du modèle. Comme les RdP sont fortement typés et que les medias doivent permettre de véhiculer différents types de messages, on peut soit multiplier le nombre de places générées (chacune gérant un type de message), soit uniformiser le type des informations à transmettre. C'est cette deuxième solution, plus compacte, que nous avons choisie ici. Le domaine des places binder est donc construit pour véhiculer tous les messages possibles, et est composé de *<source,destination,nom du message,argument1,argument2>*. Le mécanisme RPC permettant un *return()* anonyme sur l'arc liant `T3` à *Pump.port1* dans la **Figure 5** est implémenté par la place **RPCmem**. Elle permet de stocker les identificateurs des requêtes en cours, et de retrouver l'émetteur pour lui adresser la réponse.

Une pompe **P** peut démarrer l'exécution de *Start* dès qu'une requête à son attention arrive par **PPort1_in**. On s'assure que le destinataire du message est bien celui qui le consomme en `T1` (variable *me* identique sur les deux arcs). **P** lance alors la requête *getCredit* par **PPort3_out** et attend la réponse en `S1.1`. Sur réception de la réponse (`T2.2`), **P** est libre de franchir `T3`, ce qui renvoie un acquittement au client à l'origine de la requête par **PPort1_out**, et place **P** dans l'état intermédiaire `S3`.

L'exécution de la méthode *Finish* est déclenchée par la réception (`T4.1`) du message par **PPort2_in**. `T4.2` correspond à l'instruction *Credit=RandomMax(Credit)*. Son franchissement prend aléatoirement dans la place Money une valeur *rnd*. La valeur choisie doit être inférieure à **credit**, comme l'indique la garde portée par `T4.2` : **[rnd<=credit]**. La valeur *rnd* est affectée à **credit** (troisième élément du jeton), puis replacée dans la place Money afin de préserver son marquage (<money.ALL> soit un jeton par valeur du domaine). La transition `T5` est décomposée en trois pseudo-instructions: envoi de la requête (`T5.1`), réception de la réponse (`T5.2`), et réinitialisation des variables **ID** et **Credit** ici fusionnées dans `T5.3`. Cette affection parallèle est autorisée car chaque instance de **`pump`** est la seule à avoir l'accès à ses variables d'instance.

4.2 Exemple de vérification : détection de comportement déviant

Nous nous sommes intéressés dans un premier temps à vérifier la vivacité du système. Cette propriété très simple assure que le système ne comporte pas d'états bloquants (e.g. aucune transition n'est franchissable).

4.2.1 Vérification modulaire : événements observables.

Afin de réduire la taille du modèle tout en préservant la propriété de vivacité, nous avons utilisé plusieurs techniques de réduction dont l'agglomération inspirée de **[12, 50]** qui préserve les transitions observables du point de vue de l'ensemble du système. Les états intermédiaires augmentent fortement la taille du graphe des marquages accessibles (GMA) car ils introduisent des entrelacements. Pour déterminer si deux transitions `T1` et `T2` sont agglomérables, éliminant l'état intermédiaire `S` qui les sépare, il faut vérifier que :

- les variables modifiées par le franchissement de `T2` ne sont pas accessibles en lecture par le franchissement d'une quelconque autre transition du réseau pendant que `S` est marquée ;
- réciproquement, les variables lues par `T2` ne sont pas modifiables par le franchissement d'une quelconque autre transition du réseau pendant que `S` est marquée ;
- `T2` est la seule transition franchissable par un jeton placé en `S`.

Ces critères ont des conséquences simples qui peuvent s'interpréter comme suit:

- `T1` et `T2` peuvent être fusionnées si elles ne manipulent que des variables locales.
- De plus, si les files de réception ne sont pas partagées entre les entités (`1:all`) elles peuvent être considérées comme des variables locales. Cependant comme les messages sortants agissent potentiellement sur le reste du système, les files d'émission sont alors considérées comme des variables globales.
- Une action qui modifie l'environnement peut être agglomérée avec les actions locales qui la précédent en conservant l'ordre des opérations. Elle peut également être agglomérée avec les actions locales qui lui succèdent si celles-ci ne font pas d'appels à l'environnement (en particulier pas de lecture sur une file de réception).

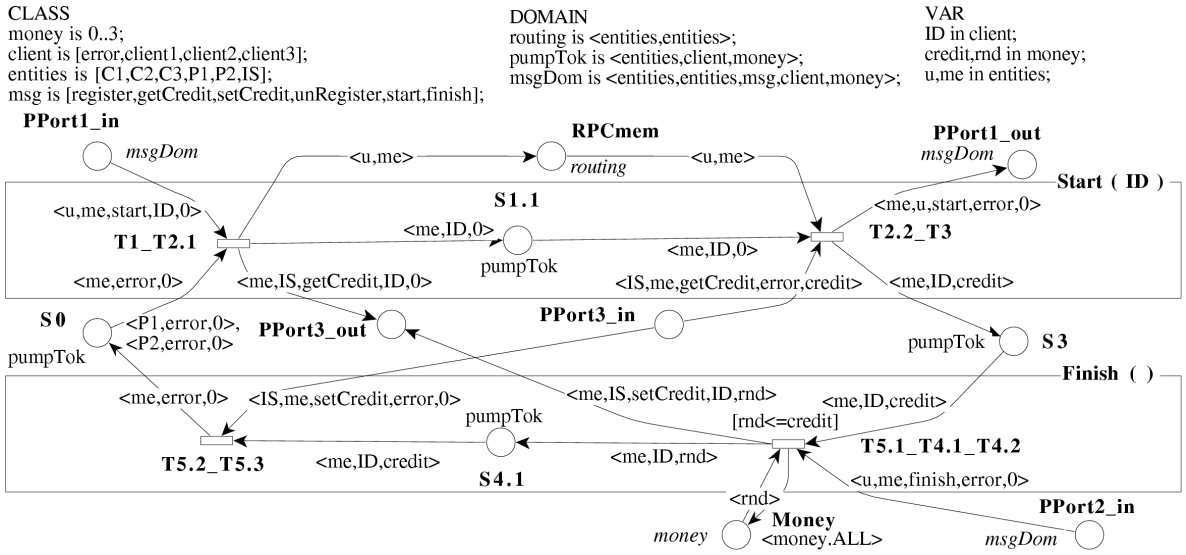


Figure 8 : Modèle réseau de Petri de la classe **L*f*P `pump`** après agglomération

La **Figure 8** montre le modèle en RdP de la pompe après agglomération :

- Les transitions `T1` et `T2.1` sont agglomérables selon les règles énoncées ci-dessus, car il s'agit d'une réception dans une file non partagée (*PPort1_in*) suivie d'une émission. Cette opération diminue l'entrelacement et donc le nombre d'états, mais préserve l'indéterminisme global du système. En effet, la place `S1` permettait l'entrelacement suivant : **(a)** `T1(P1) -> T2.1(P1) -> T1(P2) -> T2.1(P2)` ou **(b)** `T1(P1) -> T1(P2) -> T2.1(P1) -> T2.1(P2)` ou **(c)** `T1(P1) -> T1(P2) -> T2.1(P2) -> T2.1(P1)`, ainsi que les scénarios symétriques en tirant `T1(P2)` en premier. Sur le modèle réduit on ne permet que le scénario **(a)** et son symétrique. Du point de vue du reste du système, la seule action observable est l'émission du message en franchissant `T2.1`. En conséquence, cette réduction préserve bien l'ensemble des scénarios observables *du point de vue du système*, et permet d'optimiser la vérification. Les transitions `T2.2` et `T3` sont agglomérables pour les mêmes raisons.
- L'agglomération est poursuivie jusqu'à stabilisation : `T4.1` et `T4.2` sont agglomérées (actions locales) en `T4.1_T4.2`, elle même agglomérée avec `T5` (actions locales précédant une émission). Bien que la table **Money** permettant le choix d'une valeur aléatoire soit une variable partagée entre les instances de pompes, son marquage reste fixe, ce qui fait de `T4.2` une action locale.
- Le marquage initial de `S0`, non nul, interdit de supprimer cette place.

Le gain de cette opération est combinatoire par rapport aux nombre d'instances de l'entité considérée.

4.2.2 Analyse des résultats

L'étude du modèle obtenu montre l'existence d'un interblocage qui arrête le système. Celui ci est dû à l'appel *finish* qui est asynchrone non-bloquant. Le marquage bloquant est obtenu quand le premier message de la FIFO *ISPort_in* est *setCredit(ID,sum)* et que l'**`infosystem`** ne peut le consommer car le client identifié par *ID* a déjà quitté la station service. Ceci est dû au scénario dans lequel le client enchaîne les actions *Finish* et *getChange* suivi du *unregister* de l'opérateur (**`operator`**) avant que la pompe (**`pump`**) n'ait eu le temps d'effectuer un *setCredit*.

Ce scénario est possible, car en construisant le **L*f*P**-BD de l'**`infosystem`** à partir de la spécification UML, nous n'avons pas ajouté de contraintes sur l'ordre des appels à ses méthodes. Une correction consiste donc à spécifier le caractère obligatoire de l'enchaînement des appels *register, getCredit, setCredit, unRegister* à l'aide d'un **L*f*P**-BD, et à permettre un traitement correct en donnant au binder **IS_Port** une sémantique de *multi-ensemble (bag)*, ne préservant pas l'ordre des messages.

Cette correction effectuée, le réseau devient vivant (7 millions d'états accessibles, 31 millions d'arcs). Les bornes structurelles nous permettent alors de dimensionner les tailles des files de messages. Ainsi *IS_Port* a une taille bornée par *Nombre d'instances de* **`pump`** + *Nombre d'instances d'*__`operator`__, *Operator_Port1* et *P_Port1* sont bornées par *Nombre de* **`client`** et *P_Port2* est bornée par une taille de 1.

5 Conclusion

Nous avons présenté une approche de prototypage par raffinements pour les systèmes répartis embarqués. Notre approche s'inspire de MDA (Model Driven Architecture), prônée par l'OMG **[93]**. Le système est décrit au moyen d'un modèle **L*f*P** que l'on peut construire à partir d'une spécification UML. **L*f*P** est un langage de spécification ayant les caractéristiques d'un ADL (Architecture Description Language) et permettant de capturer de manière non ambiguë les interactions entre composants d'un système réparti. Le modèle **L*f*P** sert de base a la vérification formelle du système à développer par la

définition de règles de traduction en RdP automatisables, et à la synthèse automatique du squelette de contrôle réparti de l'application.

Avec une telle approche, les squelettes de contrôle générés automatiquement sont conformes à leur spécification (propriété garantie par le générateur de code) et les propriétés vérifiées sur le modèle sont préservées. Cela permet non seulement de focaliser l'effort de conception sur un modèle dédié à la vérification et à la synthèse de programme, mais d'offrir également un haut niveau de fiabilité dans la phase de maintenance, cette dernière pouvant s'effectuer sur le modèle **L*f*P**.

L'application "à la main" des techniques présentées dans cet article a donné des résultats intéressants sur plusieurs études et permis de produire rapidement des spécifications formelles analysables. Une étape importante en cours est de construire les outils permettant d'automatiser l'usage de ces techniques pour en permettre une utilisation ne nécessitant pas la connaissances des méthodes formelles sous-jacentes.

Remerciements: Nous tenons à remercier Isabelle Mounier et Emmanuel Paviot-Adet pour leurs conseils dans le traitement de l'étude de cas présentée dans cet article.

Bibliographie

[1] Abrial J.-R., "The B Book - Assigning Programs to Meanings", Cambridge University Press, 1996.

[2] Agha G., Kim W., "Actors : A Unifying Model for Parallel and Distributed Computing", Technical report, EuroMicro Society, 1999.

[3] Allen R.J., "A Formal Approach to Software Architecture". Technical report, Carnegie Mellon University, 1997. PhD Thesis CMU-CS-97-144.

[4] Amza C., Cox A.L., Dwarkadas S., Keleher P., Lu H., Rajamony R., Yu W., Zwaenepoel W., "TreadMarks: Shared Memory Computing on Networks of Workstations", IEEE Computer, vol. 29, no.2, pages 18-28, 1996.

[5] Amagbégnon P., Besnard L., Le Guernic P. "Implementation of the data-flow synchronous language SIGNAL", in Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, 163 - 173, ISSN:0362-1340, 1995.

[6] Andreopolous W. : "Defining Formal Semantics for the Unified Modelling Language (UML)", Technical report, University of Toronto, 2000.

[7] Araújo J., Moreira A., "Specifying the Behaviour of UML Collaborations Using Object-Z". In Proceedings of the Association for Information Systems, Americas Conference on Information, Systems (AMCIS), 2000. Object-Oriented Software Development, Mini-Track.

[8] Arnaud R., "Design an Simulation of Real-Time Systems wirth Object Geode", Verilog Document reference : F/GPSM/FA/320/840, 1998.

[9] Architecture Board ORMSC, "Model Driven Architecture (MDA)", Technical report,OMG, 2001. Document number ormsc/2001-07-01.

[10] Bakker A., Van Steen M., Tanenbaum A. S., "From Remote Objects to Physically Distributed Objects", In Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems, Cape Town, South Africa, pages 47-52, 1997.

[11] Bal H. E., Kaashoek F. M., Tanenbaum A. S., "Orca: a language for parallel programming of distributed systems", IEEE Transactions on Software Engineering, vol. 18, no. 3, 1992.

[12] Berthelot G., "Checking Properties of Nets Using Transformations", LNCS: Advances in Petri Nets 1985, 222, 1986, 19--40, Springer-Verlag.

[13] Biberstein O., Buchs D., Guelfi N., "Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism", Editors G. Agha and F. De Cindio and G. Rozenberg, Lecture Notes in Computer Science, vol. 2001, pages 70-127, Springer-Verlag, 2001.

[14] Blair G.S., Coulson G., Andersen A., Blair L., Clarke M., Costa F., Duran-Limon H., Fitzpatrick T., Johnston L., Moreira R., Parlavantzas N., Saikoski K., "The Design and Implementation of OpenORB" v2. IEEE Distributed Systems Online, 2001. Special Issue on Reflective Middleware.

[15] Blanc X., Gervais M.-P., Le Delliou R. "Using the UML Language to Express the ODP Enterprise Concepts", In Proceedings of the 3rd International Enterprise Distributing Object Computing Conference (EDOC'99), pages 50–59. IEEE Press, 1999.

[16] Boerger E., Cavarra A., Riccobene E., "An ASM Semantics for UML Activity Diagrams", In Proceedings of Algebraic Methodology and Software Technology (AMAST 2000), volume 1816 of Lecture Notes in Computer Science. Springer, 2000.

[17] Bradford N., Buttlar D., Proulx F. J., "Pthreads Programming", O'Reilley & Associates, ISBN 1-56592-115-1, 1996.

[18] Breg F., Diwan S., Villacis J., Balasubramanian J., Akman E., Gannon D., "Java RMI performance and object model interoperability : experiments with Java/HPC++", Concurrency : Practice and Experience, vol. 10, no. 11, pages 941–955, 1998.

[19] Booth C.J., Kurpis G.P., "The New IEEE Standard Dictionary of Electrical and Electronics Terms [Including Abstracts of All Current IEEE Standards]", fifth ed. New York: IEEE, 1993.

[20] Callaghan B., Pawlowski B., Staubach P., "NFS Version 3 Protocol Specification", Sun Microsystems, IETF RFC-1813 Standard, 1995.

[21] Caillaud B., Talpin J.-P., Jézéquel J.-M., Benveniste A., Jardy C., "BDL : A Semantics Backbone for UML Dynamic Diagrams", Technical report, INRIA - Rénnes, 2000. Projet Pampa, Epatr.

[22] Calì A., Calvanese D., Giacomo G., Lenzerini M. A., "A Formal Framework for Reasoning on UML Class Diagrams", In Proceedings of the 13th Int. Sym. on Methodologies for Intelligent Systems (ISMIS 2002), volume 2366 of Lecture Notes in Articficial Inteligence, pages 503–513. Springer, 2002.

[23] Caspi P., FMaud D., Halbwachs N., Plaice J. A., "LUSTRE: a declarative language for programming synchronous systems", In 14th ACM Symposium on l'Principles of Programming Languages, pages 178-188, 1987.

[24] Clark R., Moreira A., "Use of E-LOTOS in Adding Formality to UML", Journal of Universal Computer Science, 6, 11, 2000, 1071--1087.

[25] Corbett J., "Evaluating Deadlock Detection Methods for Concurrent Software", Software Engineering, 22(3), 1996, 161--180.

[26] Coulouris G., Dollimore J., Kindberg T., "Distributed Systems : Concepts and Design", Addison-Wesley, 3 edition, 2000.

[27] Damm W., Harel D., "LSCs: Breathing Life into Message Sequence Charts", Formal Methods in System Design, Kluwer Academic Publishers, Vol. 19 pages 45–80, 2001.

[28] DeMichiel Linda G. and all. "Enterprise JavaBeans 2.0 Documentation", Technical report, Sun Microsystems, 2001.

[29] Diagne A., "Une Approche Multi-Formalismes de Spécification de Systèmes Répartis : Transformation de Composants Modulaires en Réseaux de Petri", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Juin 1997

[30] Dobbing B., Burns A., "The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In Proceedings of ACM SIGAda Annual International Conference (SIGAda '98)", ISBN-1-58113-033-3, 1998.

[31] Elkoutbi M., Keller R., "Modeling Interactive Systems with Hierarchical Colored Petri Nets", 1998 Conference on High Performance Computing, April 1998.

[32] Egyed A., Medvidovic N., "Extending Architectural Representation in UML with View Integration", In Proceedings of the 2nd International Conference on the Unified Modeling Language (UML'99), pages 2–16, 1999.

[33] Engels G., Küster J.M., "Enhancing UML-RT Concepts for Behavioral Consistent Architecture Models", In Proceedings of the 1st ICSE Workshop on Describing Software Architecture with UML, 2001.

[34] Eshuis R., Wieringa R., "A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models", 2001. CTIT Technical Report 01-04.

[35] ESTEREL Technologies, "SCADE Suite for Safety-Critical Software Development", 2002.

[36] Evans A., Stuart K., "Core Meta-Modelling Semantics of UML: The pUML Approach", In Proceedings of UML'99 conference, IEEE Computer Society Press, 1999.

[37] Francu C., Marsic I., "An Advanced Communication Toolkit for Implementing the Broker Pattern", In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, page 458, 1999.

[38] Fuxman A.D., "A Survey of Architecture Description Languages". Technical report, University of Toronto, 2000.

[39] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[40] Garlan D., Monroe R. T., Wile D., "Acme : Architectural Description of Component-Based Systems". Cambridge University Press, 2000.

[41] Garlan D., Kompanek Andrew J., "Reconciling the Needs of Architectural Description with Object-Modeling Notations", In Kent, S. and Evans,A., editor, Proceedings of the Third International Conference on the Unified Modeling Language, volume 1939 of Lecture Notes in Computer Science, page 498. Springer, 2000.

[42] Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V., "PVM: Parallel Virtual Machine", A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, Editor Janusz Kowalik, 1994.

[43] Gibbs W. W., "Software's Chronic Crisis", Scientific American pp 72-81, September 1994.

[44] Gilliers F, Kordon F., Regep D., "Proposal for a Model Based Development of Distributed Embedded Systems", In Proceedings of the Monterey Workshop on Enginneering Automation for Intensive System Integration 2002, pages 178–191, 2002.

[45] Girault C., Valk R. and all., "Petri Nets for Systems Engineering", ISBN 3-540-41217-4 Editors Girault, C. and Valk, R., Springer-Verlag, 2003.

[46] Gnesi S., Latella D., Massink M., "Model Checking UML Statechart Diagrams using JACK", In Raymond Paul and Catherine Meadows, editor, Fourth IEEE International Symposium on High Assurance Systems Engineering, 1999.

[47] Gordon V. S., Bieman James M., "Rapid Prototyping : Lessons Learned. In Legacy Systems", volume 12, pages 85–95, 1995.

[48] Gorlick M.M., Razouk R.R., "Using Weaves for Software Construction and Analysis", InpProceedings of Int'l Conf. Software Eng. (ICSE'91), IEEE CS Press, Los Alamitos, Calif., 1991, pp. 23-34.

[49] Grimshaw A. S., "The LegionVision of a Worldwide Virtual Computer". Communications of the ACM, 40, 1997.

[50] Haddad S., "A Reduction Theory for Coloured Nets", Lecture Notes in Computer Science; Advances in Petri Nets 1989, 424, 1990, 209--235, Springer-Verlag.

[51] Harel D., Naamad A., "The STATEMATE semantics of statecharts", ACM Transactions on Software Engineering and Methodology, 5 :293–333, 1996.

[52] Havelund K., Visser W., "Program Model Checking as a New Trend", 7th International SPIN workshop, Stanford, August 2000.

[53] Heineman G.T., Councill W.T., "Component-Based Software Engineering : Putting the Pieces Together", AddisonWesley Professional, 2001.

[54] Helmbold D., Luckham D., Debugging Ada Tasking Programs, IEEE Software, 2(2), 1985, 47--57.

[55] Holzmann G. J., "The SPIN model checker", Addison-Wesley, ISBN 0-321-22862-6, 2003.

[56] Homburg P., van Steen M., Tanenbaum A.S., "Distributed Shared Objects as a Communication Paradigm", Proc. Second Annual ASCI Conference, pp. 132-137, Belgium, June 1996.

[57] Horstmann M. and Kirtland M., "DCOM Architecture", Technical report, Microsoft Corporation, 1997.

[58] Huber F., Schäatz B., "Rapid Prototyping with AutoFocus", In Wolisz, A. and Schieferdecker, I. and Rennoch, A., editor, Formale Beschreibungstechniken für verteilte Systeme. GMD Verlag (St. Augustin), 1997.

[59] International Organization for Standardization Open Systems Interconnection "Basic Reference Model", Technical report, ISO/IEC 7498-1 standard, 1994.

[60] International Organization for Standardization "Ada Reference Manual", Technical report, ISO/IEC 8652 :1995 Standard, 1997.

[61] International Organization for Standardization Information Technology - Database Languages - SQL, Technical report, ISO/IEC 9075:1992 Standard, 1992.

[62] ITU-T, Open Distributed Processing, recommandation X.901, X.902, X.903 and X.904, Technical report, ITU-T, 1997.

[63] ITU-T, "Specification and description language (SDL)", recommandation Z.100, Technical report, ITU-T, 1999.

[64] El Kaïm W., "Structuration, Placement et Exécution de Composants Logiciels dans les Applications Réparties ou Parallèles : mise en œuvre avec des applications construites selon le paradigme client-serveur sur des architectures matérielles hybrides", Thèse de l'Université Pierre & Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, Décembre 1997.

[65] Kandé M. M., Strohmeier A., "Towards a UML Profile for Software Architecture", In Andy Evans and Stuart Kent and Bran Selic, editor, Proceedings of the Third International Conference on the Unified Modeling Language (UML2000), volume 1939 of Lecture Notes in Computer Science, pages 513–527. Springer, 2000.

[66] Kenney J.J., "Executable Formal Models of Distributed Transaction Systems based on Event Processing", Stanford University, PhD Thesis, 1996.

[67] Khriss I.l, Elkoutbi M., Keller R. K., "Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams", In Jean Bézivin and Pierre-Alain Muller, editor, The Unified Modeling Language, UML'98 - Beyond the Notation, pages 115–126, 1998.

[68] Kilov H., Rumpe B., Simmonds I., "Behavioral Specifications of Businesses and Systems", Kluwer Academic Publishers, 1999. Chapter 17 : 30 Things that go wrong in object modelling with UML 1.3

[69] Kon F., Román M.l, Liu P., Mao J., Yamane T., Magalhães C. L., Campbell R.H., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB", In Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), 2000.

[70] Kordon F., Luqi, "An Introduction to Rapid System Prototyping". IEEE Transactions on Software Engineering, 28 :817–821, 2002.

[71] Kordon F., Mounier I., Paviot-Adet E., Regep D. "Formal verification of embedded distributed systems in a prototyping approach", in proceedings of the International Workshop on Engineering Automation for Software Intensive System Integration , Monterey, June 2001.

[72] Lieberherr K.J., "Adaptive Object-Oriented Software : The Demeter Method with Propagation Patterns", PWS Publishing Company, 1996.

[73] Lilius J., Paltor I. P., "Formalising UML State Machines for Model Checking", In Proceedings of the seccond International Conference on the Unified Modeling Language (UML1999) - UML'99, volume 1723 of Lecture Notes in Computer Science, pages 430–445. Springer Verlag, 1999.

[74] Lilius J., Paltor P. I., "vUML : a Tool for Verifying UML Models", In Proceedings of the 14th IEEE International Conference on Automated Software Engineering, 1999. TUCS-TR-272.

[75] Luqi, Goguen J., "Formal Methods: Promises and Problems", IEEE Software, vol 14, no. 1, pages 75-85, IEEE Press, 1997.

[76] Luqi, Berzins V., Shing M., Riehle R.Nogueira J., "Evolutionary Computer Aided Prototyping System (CAPS)", Technology of O-O Languages and Systems, 2000.

[77] Lutz M., "Programming Python", O'Reilly Associates, 2001. 2nd Edition.

[78] Lyons A., "UML for Real-Time Overview", 1998. Rational Whitepaper.

[79] Medvidovic N., "Formal definition of the Chiron-2 software architectural style", Technical report, University of California at Irvine, 1995.

[80] Medvidovic N., Taylor, Richard N., "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Software Engineering, 26 :70–93, 2000.

[81] Medvidovic N., Taylor R.N., Whitehead Jr. E. J., "Formal Modeling of Software Architectures at Multiple Levels of Abstraction", In Proceedings of the California Software Symposium. University of Southern California, 1996.

[82] Mehta N. R., Medvidovic N., Phadke S., "Towards a Taxonomy of Software Connectors", In Proceedings of the 22nd international conference on Software engineering, pages 178 – 187. ACM Press, 2000.

[83] Meyer B., "Eiffel, Le langage", Masson, 1994, ISBN 2-7296-0525-8.

[84] Minsky N., Pal P., "Imposing The Law of Demeter and Its Variations", In Proceedings of the TOOLS Conference (TOOLS'96), 1996.

[85] ObjectWeb Consortium, "Jonathan", Technical report, France Télécom R&D, 2002. Whitepaper.

[86] OMG, "Unified Modelling Language specification", Technical report, OMG, 2000. Chapter 2.3.3.

[87] OMG, "Common Object Request Broker Architecture (CORBA/IIOP)", Technical report, Object Management Group, 2001. version 2.6 (formal/2001-12-35).

[88] OMG, "Real-Time CORBA 1.0 Specification", ftc/00-09-02 document, Tehnical report, 2000.

[89] OMG, "CORBA Component Model, v3.0", Technical report, OMG, 2002.

[90] OMG, "Unified Modelling Language (Action Semantics)", Version 1.4, Technical report, 2002.

[91] OMG, "Meta Object Facility (MOF) Specification", Version 1.3, Technical report, 1999.

[92] OMG, "XML Metadata Interchange (XMI)", version 1.2. Technical report, 2002.

[93] OMG, "Model Driven Architecture", ormsc/2001-07-01 document, 2001.

[94] Övergaard Gunnar., "A Formal Approach to Collaborations in the Unified Modeling Language". In France, Robert and Rumpe, Bernhard, editor, Proceedings of UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, volume 1723 of Lecture Notes in Computer Science, pages 99–115, 1999.

[95] Pautet L., "Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition", Technical report, Université Pierre et Marie Curie - Paris VI, 2001. Habilitation à diriger des recherches.

[96] Pautet L., Tardieu S., "GLADE Users Guide", Technical Report, AdaCore Technologies, 2000.

[97] Petriu D., Sun Y., "Consistent Behaviour Representation in Activity and Sequence Diagrams", In Andy Evans and Stuart Kent and Bran Selic, editor, Proceedings of the Third International Conference on the Unified Modeling Language (UML2000), volume 1939 of Lecture Notes in Computer Science. Springer, 2000.

[98] Pfister C., Szyperski C., "Why Objects are Not Enough", In Proceedings, International Component Users Conference. SIGS, 1996.

[99] Point G., "AltaRica Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement", LaBRI, Université Bordeaux I, PhD Thesis, 2000.

[100] QNX team, "System Architecture Guide - QNX RTOS v6", Technical report, QNX Software Systems Ltd., 2002.

[101] Quartel D., Van Sinderen M., Pires L. F., "A model-based approach to service creation", In Seventh Workshop on Future Trends of Distributed Computing Systems. IEEE Society Press, 1999.

[102] Quinot T., Kordon F., Pautet L., "From functional to architectural analysis of a middleware supporting interoperability across heterogeneous distribution models", In Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01), Rome, Italy, September 2001.

[103] Quinot T., "CIAO : Opening the Ada 95 Distributed Systems Annex to CORBA Clients", In Ada France 1999, 2000.

[104] Regep D., Kordon F., "Using MetaScribe to prototype an UML to C++/Ada95 code generator", in proceedings of the 11th IEEE International Workshop on Rapid System Prototyping, pp 128-133, Paris, June 2000.

[105] Regep D., Kordon F., "LfP : A specification language for rapid prototyping of concurrent systems", in proceedings of the 12th IEEE International Workshop on Rapid System Prototyping, Monterey, June 2001.

[106] Regep D., Thierry-Mieg Y., Gilliers F, Kordon F., "Modélisation et vérification de systèmes répartis :une approche intégrée avec LfP", in proceedings of AFADL'2003 conference, January 2003.

[107] Regio G., Astesiano E., "An Extension of UML for modelling the non Purely-Reactive Behavior of Active Objects", In Proceedings of the Monterey Workshop on Enginneering Automation for Intensive System Integration 2001, 2001.

[108] Rifflet Jean-Marie, "La communication sous UNIX", 2ème édition, Ediscience International, Paris, 1995, ISBN 2-84074-106-7.

[109] Roman M., "Universally Interoperable Core", 2002. The Universally Interoperable Core is being used by the University of Illinois at Urbana-Champaign, University of Lancaster and University of Oslo.

[110] Roman M., Campbell R.H., "Unified Object Bus : Providing Support for Dynamic Management of Heterogeneous Components", Technical report, Universiy of Illinois at Urbana-Champaign, 2001.

[111] Rumpe B., Schoenmakers M., Radermacher A., Schürr A., "UML + ROOM as a Standard ADL?", In F. Titsworth, editor, Proceedings of Engineering of Complex Computer Systems ICECCS'99. IEEE Computer Society, 1999.

[112] RTCA, Software Considerations in Airborne Systems and Equipment Certification : "Advisory Circular D0-178B", RTCA, Inc., Washington D.C.

[113] Saldhana J., Shatz S.M., "UML Diagrams to Object Petri Net Models : An Approach for Modeling and Analysis", In Proceedings of the Int. Conference on Software Engineering and Knowledge Engineering (SEKE), pages 103–110, 2000.

[114] Schneider S., "Concurrent and Real-time Systems : The CSP Approach", John Wiley and Sons, 1999. Part 1: The language of CSP.

[115] Shtil S., Bhatia V., "Rapid Prototypong Technology Cuts System Verification Time and Cost by More Than Half", in proceedings of the Design Conference, January 1998.

[116] Simons A.J.H., Graham I., "30 Things that go wrong in object modelling with UML 1.3", volume 523 of The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1999. Chapter 17.

[117] Singhai A., Sane A., Campbell R., "Quarterware for Middleware", In Proceedings of the 18th International Conference on Distributed Computing Systems - ICDCS'98, page 192, 1998.

[118] Sinha P.K., "Distributed Operating Systems : Concepts and Design", John Wiley and Sons, 1997.

[119] Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J., "MPI : The CompleteReference", MIT Press, 1996.

[120] Somervillle I., "Software Engineering (6th edition)", Addison-Wesley, 1999.

[121] Spitznagel B., Garlan, D., "A Compositional Approach for Constructing Connectors", In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001). IEEE Press, 2001.

[122] Spivey J. M., "The Z Notation - A Reference Manual", Technical report, University of Oxford - Programming Research Group, 1988.

[123] Srinivasan R,. "RPC : Remote Procedure Call Protocol Specification Version 2", Technical report, Sun Microsystems, 1995. IETF RFC- 1831 Standard.

[124] Stevens W. R. "UNIX Network Programming", Prentice Hall, 1990, ISBN 0-13-949876-1.

[125] Sun Microsystems, "Java Message Service", Technical report, Sun Microsystems, 2002. version 1.1.

[126] Sun Microsystems, "Java Remote Method Invocation", Technical report, Sun Microsystems, 2002. version 1.4.

[127] Sy Ousmane., "Sécification Comportementale de Composants CORBA", Technical report, Laboratoire d'Interaction Homme-Systèmes Université Toulouse 1, 2001. PhD Thesis.

[128] Sy Ousmane, Bastide R., Palanque P., Duc-Hoa L., Navarre D., "PetShop - a tool for the formal specification of CORBA systems", In Proceedings of the 20th International Conference on Applications and Theory of Petri Nets - ICATPN'99, 1999.

[129] Szyperski C., "Component Software : Beyound Object Oriented Software", Addison Wesley, 1998.

[130] Tanenbaum A. S., "Distributed Operating Systems". Prentice Hall, 1995.

[131] Tanenbaum A. S., Kaashoek M.F., Van Renesse R., Bal H., "The Amoeba Distributed Operating System-A Status Report", Computer Communications, 14 :324– 335, 1991.

[132] The Open Group, "DCE 1.1: Remote Procedure Call", CAE Specification, Document C706, Technical Report 1997.

[133] TNI Valiosys, "SILDEX", 2000. Version 5.

[134] U.S. National Communication System Technology & Standard Division, "Telecom glossary", Federal Standard 1037C. Technical report, National Communication System Technology & Standard Division, 1996.

[135] Van Steen M., Homburg P., Tanenbaum A.S., "Globe : A Wide-Area Distributed System", IEEE Concurency, 7 :70–78, 1999. no. 1.

[136] Vestal S. : "MetaH User's Manual", Technical report, Honeywell Inc., 1998.

[137] W3C - World Wide Web Consortium, "SOAP", Version 1.2 Part 0: Primer, Working Draft, 2002.

[138] W3C - World Wide Web Consortium, "XML Schema Part 1: Structures", Tehnical report, W3C Recommendation, 2002.

[139] Wagner, "Interoperability", ACM Computing Surveys, vol. 28, no. 1, 1996.

[140] Whalen, M. W. and Heimdahl, M. P.E. : "On the Requirements of High-Integrity Code Generation", In 4th IEEE International Workshop on High Assurance in Systems Engineering, 1999.

[141] Zelesnik G., "The UniCon Language Reference Manual", Technical report, Carnegie Mellon University, 1996.

[142] Zhao J., "A Slicing-based Approach to Extracting Reusable Software Architectures", CSMR, 2000, 215-223.

Résumé

Le développement rapide d'applications réparties sûres par leur conception correcte et leur implémentation conforme constitue un sujet de recherche délicat qui nécessite des solutions dans chacun de ces domaines : 1) Description d'architectures logicielles ; 2) Vérification et validation de la conception ; 3) Implémentation conforme des modèles ; 4) Méthodologies de développement.

Nos objectifs ont été de définir un langage de spécification adapté aux applications réparties afin de capturer leurs éléments caractéristiques, de les développer dans le contexte du développement par prototypage et de les déployer de manière transparente sur des architectures hétérogènes. Nous avons eu constamment à l'esprit les contraintes suivantes :

- Permettre la génération automatique des éléments qui assurent le contrôle réparti puisqu'il s'agit du point le plus délicat de ces systèmes,
- Rester compatible avec les démarches de développement de l'industrie et en particulier avec des standards comme UML (Unified Modeling Language), afin d'apporter une aide aux ingénieurs sans perturber leur travail,
- Supporter l'évaluation des prototypes par simulation mais aussi la vérification de leurs propriétés par utilisation des méthodes formelles,

Dans cette thèse nous proposons comme solution **L*f*P** (A langage for Prototyping) un langage de spécification ayant les capacités de description d'un ADL (Architectural Description Language) formel, compatible et complémentaire avec la notation UML. ce langage est facilement intégrable dans une démarche de développement par prototypage évolutif.

Abstract

The rapid development distributed systems sure by their design and conformant implementation is a difficult research subject that needs solution in each of these fields: 1) Software architectural description; 2) Design verification and validation; 3) Conformant implementation of models; 4) Development methodology.

Our goals have been to define a specification language adapted to the description of distributed systems. It is able to capture their specific elements, to implement them according to a rapid prototyping development methodology, and to transparently deploy them on heterogeneous systems. The design of this language was guided by several goals:

- Permit the automatic code generation of elements encapsulating the distributed control of an application since aspect is considered difficult to design and implement.
- Stay compatible with the existing industrial modeling notations such as UML (the Unified Modeling Language), in order to help the software practitioners to integrate our approach into their development process.
- Support the simulation of executable prototypes as well as the formal verification of their design properties by means of formal verification methods,

In this Thesis, we propose **L*f*P**, a Language for Prototyping having the characteristics of a formally defined ADL (Architectural Description Language), which is compatible and complementary to the UML notation. Our language tends to be easily used according to a rapid, evolutionary development methodology.