# THÈSE

présentée par

**Jérôme REVILLARD**

pour obtenir le diplôme de
**DOCTEUR DE L'UNIVERSITÉ DE SAVOIE**
(Arrêté ministériel du 30 mars 1992)

**Spécialité : INFORMATIQUE**

---

# *Approche centrée architecture pour la conception logicielle des instruments intelligents*

---

Soutenue publiquement le 15 décembre 2005 devant le jury composé de :

| | | |
|---|---|---|
| **Jean-Pierre ELLOY** | Président du jury | Professeur à l'Ecole Centrale de Nantes |
| **Mireille BAYART** | Rapporteur | Professeur à l'Université des Sciences et Technologies de Lille |
| **Mourad Chabane OUSSALAH** | Rapporteur | Professeur à l'Université de Nantes |
| **Flavio OQUENDO** | Directeur de thèse | Professeur à l'Université de Bretagne Sud |
| **Eric BENOIT** | Co-encadrant | Maître de Conférences à l'Université de Savoie |
| **Sorana CIMPAN** | Co-encadrant | Maître de Conférences à l'Université de Savoie |

Préparée au sein du LISTIC
Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance

*A Claire, ma future femme…*

# Remerciements

Ce travail de doctorat a été réalisé au sein du LISTIC – Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance, un des trois laboratoires de l'ESIA – Ecole Supérieure d'Ingénieurs d'Annecy.

J'adresse mes plus vifs remerciements à M. Mourad Chabane OUSSALAH, Professeur à l'Université de Nantes et à Mme Mireille BAYART, Professeur à l'Université des Sciences et Technologies de Lille, pour m'avoir fait l'honneur d'étudier mes travaux de thèse et de les avoir cautionnés en qualité de rapporteurs.

Je tiens également à remercier M. Jean-Pierre ELLOY, Professeur à l'Ecole Centrale de Nantes, qui m'a fait l'honneur de présider le jury.

Je tiens à exprimer toute ma gratitude à M. Flavio OQUENDO, Professeur à l'Université de Bretagne Sud, pour m'avoir, tout d'abord, donné l'opportunité de réaliser cette thèse. Je le remercie également d'avoir dirigé mes travaux et de m'avoir fait part de ses conseils et son expérience.

Je tiens également à remercier M. Eric BENOIT et Mme Sorana CÎMPAN, Maîtres de Conférences à l'Université de Savoie, pour m'avoir co-encadré et soutenu durant ces trois années. Ils furent tous les deux attentifs à tout ce que j'ai fait, me donnant leurs avis éclairés chacun dans leur spécialité.

Un merci plus particulier à mes amis et collègues doctorants ou anciens doctorants qui, grâce à leur bonne humeur et à leur amitié, m'ont permis de passer trois belles années.

Mes remerciements les plus sincères vont aux différents membres du laboratoire LISTIC avec une mention toute particulière à Valérie pour l'aide qu'elle m'a apportée au quotidien.

Je remercie de tout cœur mes parents pour la confiance, le soutien et l'aide qu'ils m'ont apportés durant toutes mes études et qui m'ont, par conséquent, permis d'arriver jusqu'ici.

Enfin, un ENORME merci à Claire, qui m'a accompagné, soutenu et supporté (ce qui n'était pas une mince affaire) durant ces trois ans. Cette personne étant exceptionnelle, j'ai décidé de l'épouser.

# Table des matières

# Table des figures

# Chapitre 1 : INTRODUCTION

# Chapitre 1 : Introduction

# Introduction

Durant les dernières décennies, les domaines de la micro-informatique et de la micro-électronique ont connus un essor fulgurant. Tout a commencé dans les années 70. C'est à cette époque que les premiers calculateurs numériques ont fait leur apparition. Ces calculateurs traitaient les fonctions de contrôle/commande de façon centralisée. Ils ont très vite été supplantés par les microprocesseurs qui sont apparus dans les années 80. Ces derniers ont été à l'origine des premiers systèmes de contrôle distribué. Cependant, ces systèmes étaient toujours figés dans le sens où chaque processeur avait été programmé afin de réaliser une tâche spécifique.

La notion d'instrument intelligent est apparue au début des années 90. En effet, à cette époque, la diminution des coûts des microprocesseurs, leur banalisation et les possibilités offertes par la technologie (puissance/consommation/miniaturisation) permettent de concevoir et d'utiliser des constituants dits *intelligents* jusqu'au niveau le plus bas (capteurs, actionneurs).

Leur intelligence tient du fait que de nouvelles fonctionnalités leurs sont attribuées. Ce peut être des capacités d'auto-configuration, d'auto-diagnostic, de communication, etc. Par exemple, il devient possible de modifier une procédure de mesure d'un capteur en changeant uniquement et simplement l'algorithme qui est exécuté par son architecture matérielle. L'intégration de ces fonctionnalités requiert une bonne connaissance en informatique industrielle car concevoir et implémenter ce type d'instrument est relativement complexe. Cette complexité est accrue par le fait qu'une grande fiabilité et sûreté de fonctionnement sont nécessaires. En effet, les instruments intelligents sont utilisés dans des secteurs où la moindre défaillance peut avoir des conséquences désastreuses comme l'électronique automobile, le transport, le médical, le spatial, l'aéronautique, le militaire et bien d'autres encore. Afin de faciliter le développement des instruments intelligents et afin d'assurer la fiabilité des logiciels embarqués, il devient nécessaire d'utiliser des outils qui s'appuient sur un modèle générique.

La problématique consistant à définir, utiliser, respecter et faire évoluer un modèle de développement d'applications logicielles de même nature n'est pas propre au domaine des instruments intelligents. Elle se rapporte, au contraire, à l'ensemble des secteurs d'activité de l'ingénierie logicielle. En effet, même si, dans un premier temps, les applications logicielles étaient conçues une par une, pour répondre à des besoins spécifiques, il est aujourd'hui nécessaire, pour des raisons de coût et de temps de production, de définir et de

mettre en oeuvre des méthodes et des outils supportant le développement de familles de logiciels ayant des caractéristiques communes.

Une approche, en plein essor, traite efficacement de cette problématique, il s'agit des architectures logicielles. Cette approche permet de spécifier formellement des modèles de développement, ou styles architecturaux, et de les exploiter pour produire des applications logicielles. Ce type de conception était jusqu'à présent destiné à des spécialistes du domaine car les langages utilisés sont relativement complexes et la philosophie de développement est totalement différente de celle que l'on peut avoir en utilisant un processus de conception classique.

Dans cette thèse, ce type de conception va être mis à la portée de personnes qui ne sont pas forcément spécialistes en informatique et qui pourtant doivent concevoir des logiciels sûr : les concepteurs d'instruments intelligents.

Dans ce chapitre, nous allons tout d'abord présenter ce qu'est un instrument intelligent (section I). Dans la section II, les contraintes de conception de tels instruments seront détaillées. Enfin dans la section III, le plan du manuscrit sera donné.

# *I. Qu'est ce qu'un instrument intelligent*

Un instrument intelligent est un capteur, un actionneur ou un calculateur capable d'intégrer des fonctions tel que l'auto-configuration, la validation, l'auto-diagnostic, la communication, etc. Ces nouveaux instruments sont donc capables d'adapter leur comportement lorsque des changements se produisent dans leur environnement.

L'ensemble des fonctionnalités peut évoluer en fonction de l'architecture matérielle et fonctionnelle dont a été doté l'instrument. L'un des intérêts d'utiliser des instruments intelligents est qu'ils peuvent coopérer dans un système distribué. On trouve ce type d'instrument dans de nombreux endroits. Par exemple, dans les voitures, les avions, dans une maison (domotique), etc.

Nous allons donc présenter dans les sections suivantes l'architecture matérielle et fonctionnelle des instruments intelligents.

## I.A. Architecture matérielle

L'architecture matérielle d'un instrument intelligent doit être capable de supporter son *intelligence*. Pour ce faire, elle est généralement réalisée autour d'un système à microprocesseur qui établit un dialogue permanent avec les différents constituants à travers un bus interne. Cette architecture dépend de l'application et de l'environnement de communication et peut être associée, par exemple, à des robots ou à des automates programmables. Afin d'assumer leur intelligence, l'architecture matérielle doit regrouper :

- des *moyens de communication* avec les opérateurs et/ou le système d'automatisation ; ces moyens permettent le transit, en entrée ou en sortie, des signaux et messages utilisés par les opérateurs ou les autres équipements de l'architecture distribuée,
- des *moyens de traitement*, qui autorisent la distribution du système d'automatisation,

- des *moyens de mémorisation*, qui autorisent la distribution de ses bases de données et qui comportent, en particulier, des informations de conduite, de maintenance, de gestion technique, etc.

La figure ci-dessous (Figure 1-1), présente un modèle très simplifié, mais standard, d'architecture matérielle pour les instruments intelligents. Ce modèle est présenté dans [Beaudoin&Favennec 93] [Bayart 94]. Cette structure est devenue standard grâce à la norme IEEE1451 [Shneeman&Lee 00] qui la reprend. Cette norme a pour but de fournir un cadre cohérent et ouvert permettant la mise en relation d'appareillages de faible capacité mémoire et dont les possibilités se réduisent à opérer des prises de mesure en un point donné. La philosophie générale du développement de ce standard repose sur la création d'une volonté d'uniformiser les interfaces des appareils et ce, en ajoutant de manière incrémentale des capacités à celles déjà existantes, tout en conservant le coût de la transition accessible à tous les vendeurs de l'industrie.

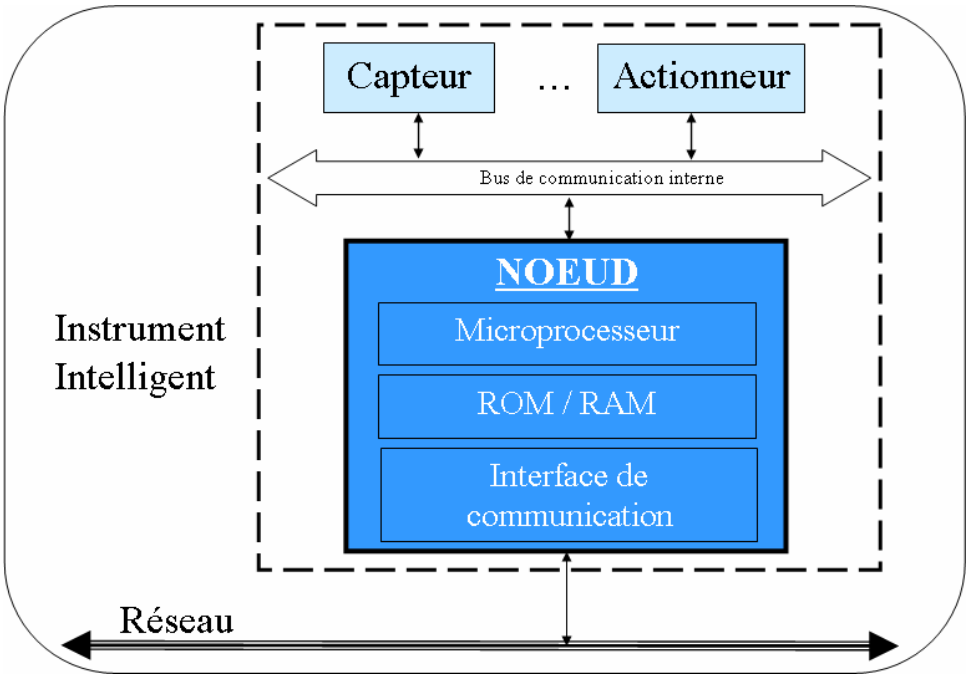



**Figure 1-1 : Architecture matérielle d'instruments intelligents**

Le standard envisagé propose également l'indépendance vis-à-vis de la couche réseau mise en oeuvre ainsi que l'indépendance vis-à-vis du type de microprocesseur embarqué sur le système. La famille 1451P consiste en la proposition et le développement de quatre standards :

- IEEE 1451.1, Network Capable Application Processor (NCAP) ;
- IEEE 1451.2, Transducer to Microprocessor and Transducer Electronic Data Sheet (TEDS) Format ;
- IEEE 1451.3, Digital Communication and Transducer Electronic Data Sheet (TEDS) Format for Distributed Multidrop Systems ;
- IEEE 1451.4, Mixed Mode Communication Protocols and TEDS Formats.

Un instrument intelligent est donc constitué de capteurs ou d'actionneurs reliés à ce que l'on appelle un *nœud* par l'intermédiaire d'un bus interne. Cet instrument peut être composé uniquement de capteurs, uniquement d'actionneurs ou des deux. Il peut aussi être composé d'un seul capteur ou d'un seul actionneur.

*Ce que l'on appelle ici **nœud** est un ensemble de composants dont les principaux sont :*

- ***Le microprocesseur** : qui permet de faire des opérations de calcul.*
- ***La ROM (ou EPROM)** : qui contient le programme permettant de rendre intelligent l'instrument (permettant de lui donner ses fonctionnalités). Ce sont des données permanentes, c'est-à-dire qu'elles ne seront pas effacées en cas de panne d'alimentation.*
- ***La RAM** : qui contient toutes les données non permanentes d'exécution (s'efface si l'instrument n'est plus alimenté).*
- ***L'interface de communication** : qui permet de gérer la réception ou l'émission de données sur le réseau.*

Une architecture matérielle construite de la sorte est capable d'intégrer les différentes fonctionnalités qui peuvent appartenir aux instruments intelligents.

## I.B. Architecture fonctionnelle

Un instrument intelligent est donc un instrument capable d'intégrer des fonctionnalités comme par exemple :

- la communication,
- l'auto-configuration,
- l'auto-contrôle.

Ces fonctionnalités ont vu le jour grâce à l'intégration des microprocesseurs. Cela a notamment permis d'augmenter la puissance de calcul et la capacité de traitement de l'information des instruments intelligents.

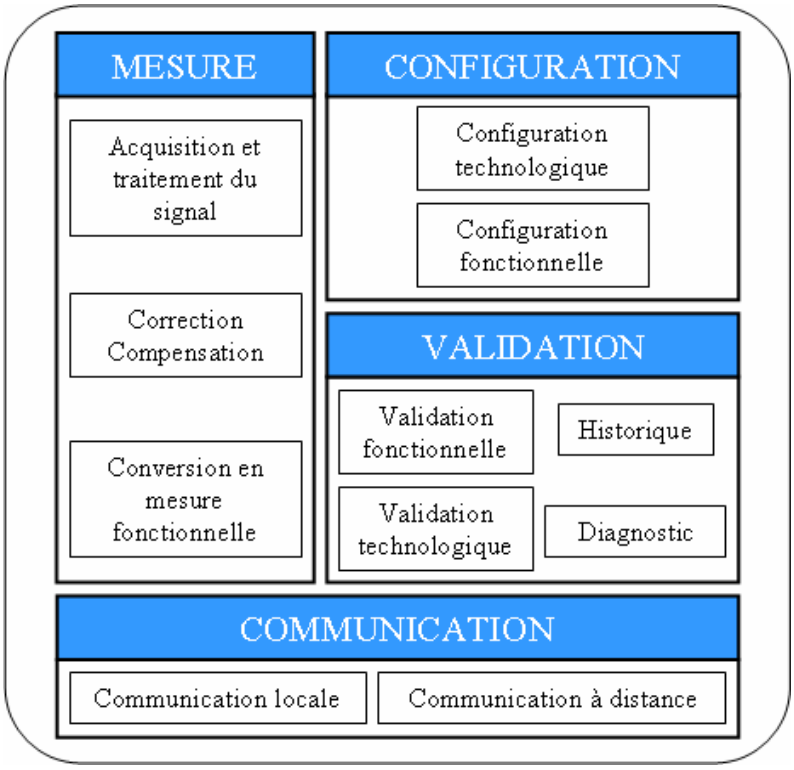



**Figure 1-2: Fonctionnalité du capteur intelligent**

La figure ci-dessus (Figure 1-2) présente les fonctionnalités génériques préconisées pour un capteur intelligent telles qu'elles ont été proposées par les membres du groupe de travail "capteur intelligent" du CIAME (**C**omité **I**nterprofessionnel pour l'**A**utomatisation et la **ME**sure) [CIAME 87].

La fonction *mesure* est l'une des fonctions primordiales d'un capteur intelligent. En effet, elle permet d'alimenter par les données qu'elle fournit toutes les autres fonctionnalités.

Grâce à la fonction *communication*, les capteurs intelligents ont tous la faculté d'interagir avec leur environnement (d'autres instruments intelligents ou des personnes physiques). Cette fonction de communication est indispensable à la réalisation d'un certain nombre d'autres fonctionnalités. En effet, la modification de la *configuration* va ainsi pouvoir être commandée depuis une communication locale et permettre, entre autre, un changement d'échelle, la communication d'une mesure sur la grandeur principale ou sur une grandeur d'influence, le choix d'un capteur adapté, le remplacement d'un capteur par un autre, le choix d'une tension d'alimentation et de référence pour un fonctionnement adaptatif, etc.

Une autre fonctionnalité préconisée dans les capteurs intelligents est celle que l'on appelle généralement la *validation* ou *l'auto-contrôle*. Cette fonctionnalité a pour but d'avertir l'instrument intelligent sur son éventuel mauvais fonctionnement, sur les défauts d'alimentation, les mauvaises conditions d'utilisation ou encore la mauvaise transmission des signaux. Cela va permettre au capteur ou à l'actionneur d'assurer une bonne crédibilité aux informations issues de la mesure ou destinées à la commande.

L'apparition des microprocesseurs a fait entrer les capteurs intelligents dans le monde du « tout numérique ». Ils ont permis tout d'abord d'améliorer leurs caractéristiques métrologiques (mesures plus précises). Ensuite, ils ont aussi permis d'améliorer la transmission des données. En effet, ce type de transmission évite toute détérioration due à la transmission de signaux analogiques. Enfin, les microprocesseurs ont permis aussi d'améliorer grandement les fonctions de filtrage ou de correction de mesure.

En ce qui concerne l'actionneur intelligent, on peut réutiliser la même architecture fonctionnelle et remplacer la fonctionnalité mesure par une autre que l'on nommera *commande* et qui aura pour but de transmettre la valeur de la commande à l'actionneur. On dit que ce type d'actionneur intelligent fonctionne en boucle ouverte car il tire ses données de l'extérieur.

Il existe aussi des instruments intelligents qui fonctionnent en boucle fermée, c'est-à-dire qui tire leur valeur de commande des actionneurs de capteurs internes. Dans ce cas là les deux fonctionnalités commande et mesure sont nécessaires.

Pour pouvoir intégrer ces fonctionnalités, un instrument intelligent doit être doté d'un logiciel qui va être implanté dans son nœud. Dans la section suivante (section II), nous allons nous intéresser à la conception de la partie logicielle de tels instruments intelligents et notamment à tout ce qui pourrait permettre de la rendre performante.

## *II. La conception d'instruments intelligents*

La conception d'instruments intelligents est un problème complexe. En effet, des compétences en physique, en mécanique et en électronique sont nécessaires pour la

conception de la partie matérielle tandis que des compétences en informatique sont nécessaires pour concevoir la partie logicielle. De plus, ces instruments intelligents sont fréquemment utilisés dans des environnements critiques. Le logiciel conçu doit alors répondre à des contraintes de sûreté de fonctionnement sévères.

La conjoncture économique actuelle oblige les industriels à être innovant, à produire vite et à des coûts réduits. Or, on voit que dans le cas des instruments intelligents, ces différents objectifs sont difficiles à atteindre. En effet, aux vues des compétences demandées, de nombreuses personnes spécialistes dans différents domaines doivent travailler en même temps sur un même produit. Ceci implique un coût de main d'œuvre considérable. Ensuite, produire vite n'est pas évident non plus car étant donné qu'un instrument intelligent doit être fiable, de nombreux tests doivent être effectués avant de le lancer sur le marché. Les instruments intelligents étant très différents les un des autres, il est, en plus, difficile de réutiliser ce qui a été fait pour un instrument sur un autre. Enfin, tout ceci joue sur les capacités d'innovation des entreprises qui n'ont plus forcément le temps ni l'argent pour développer de nouveaux produits. Une innovation peut en plus remettre en question tout le processus de conception utilisé et engendrer une refonte complète des méthodes de production.

Pour résumer, afin d'aider une entreprise à être compétitive, plusieurs points doivent être étudiés. L'un des principaux est de réduire le nombre de compétences nécessaires à la conception d'un instrument intelligent. Le niveau de compétence requis en informatique peut, par les moyens actuels, être réduit fortement par une automatisation du processus de conception. Cette automatisation n'étant que partiellement réalisable, il faut parvenir à diminuer le nombre de compétences que doit avoir une seule et même personne à la fois. Pour concevoir la partie logicielle d'un instrument intelligent, il faut avoir de très bonnes connaissances en informatique mais aussi de très bonnes connaissances dans le domaine de l'instrumentation intelligente. L'objectif n'est pas de se passer des informaticiens mais de faire en sorte que le spécialiste des instruments intelligents ne doivent pas aussi être spécialiste en informatique. Pour atteindre cet objectif, plusieurs améliorations doivent être entreprises :

1. Créer un modèle générique pour la partie logiciel des instruments intelligents. Cela signifie que ce modèle doit être valable pour n'importe quel type d'instrument intelligent.
2. Vérifier la conformité du logiciel créé par rapport au modèle le plus tôt possible dans le processus de développement. C'est-à-dire que l'on doit être capable d'effectuer, en phase de conception, toutes les vérifications nécessaires au respect du modèle de conception.
3. Avoir un modèle générique qui puisse évoluer facilement en fonction de l'apparition de nouveaux besoins dans le domaine ou de nouvelles technologies.
4. Avoir un processus de conception qui permette un passage aisé entre le modèle générique et le code source du logiciel à créer. L'idéal étant une génération automatique du code source depuis le modèle.
5. Le processus de conception ne doit pas subir de modifications importantes en cas d'évolution du modèle. Ceci concerne le processus en lui-même mais aussi les différents outils qui le compose.

L'objectif de cette thèse est de répondre à ces différents besoins d'amélioration.

# III. Organisation du manuscrit

Ce document est principalement divisé en trois parties. La première partie présente l'existant. Elle contient deux chapitres.

***Chapitre 2***

Nous commençons par présenter notre domaine de recherche, le modèle de conception des instruments intelligents et les outils basés sur ce modèle. Nous en introduisons le cadre et la terminologie. Puis, nous y définissons la problématique que nous abordons dans cette thèse.

***Chapitre 3***

Nous présentons un état de l'art des travaux concernant la conception centré architecture et plus particulièrement sur les langages de description d'architecture. Nous montrons les spécificités des différents langages et choisirons le plus approprié à la conception d'instruments intelligents en nous basant sur des critères énoncés en début de chapitre. Le langage choisi et les différents outils fournis avec seront ensuite présentés.

Une deuxième partie présente la solution apportée en trois chapitres.

***Chapitre 4***

Suite à l'étude du domaine réalisé dans le chapitre deux, différentes améliorations doivent être apportées au modèle de conception de la partie logicielle des instruments intelligents sur lequel nous nous basons. Ce chapitre a donc pour but de présenter les différentes évolutions qui ont été faites au niveau du modèle afin que tous les types d'instruments intelligents puissent être conçus.

***Chapitre 5***

Ce chapitre présente tout d'abord comment utiliser la conception centrée architecture pour la conception d'instruments intelligents. Un fois ceci présenté, la mise en œuvre est réalisée. Ceci nous amène à présenter les différents styles architecturaux qui ont été créés mais aussi les différentes actions de raffinement qui nous permettrons de concevoir d'autres éléments du logiciel qui gère un instrument intelligent.

***Chapitre 6***

Le nouveau processus de conception de partie logicielle d'un instrument intelligent sera présenté. Ce processus contiendra en majorité des outils appartenant au langage de description d'architecture qui a été choisi mais aussi des outils spécifiques. Ces derniers seront présentés. Le processus nous permet de générer de façon automatique le code source d'un instrument intelligent. Le passage des différents langages à ce code source sera présenté.

Enfin, la troisième partie présente une validation de nos travaux :

***Chapitre 7***

Ce chapitre contiendra une étude de cas qui permettra de valider nos travaux. Pour ce faire, un instrument intelligent sera conçu. Le modèle de cet instrument sera présenté puis tout le processus de conception sera déroulé afin de concevoir la partie logicielle de l'instrument intelligent choisi avec notre nouvelle approche.

Une conclusion propose une synthèse et un bilan du travail effectué, ainsi qu'un ensemble de perspectives liées à la continuité du travail.

***Chapitre 8***

La conclusion propose une synthèse et un bilan du travail effectué durant cette thèse ainsi qu'un ensemble de perspectives ouvertes par ce travail.

Plusieurs annexes sont également fournies en fin de document.

***Annexe 1***

Les différents styles créés sont listés dans cette annexe.

***Annexe 2***

Cette annexe contient le code source des différents éléments qui compose un graphe des services internes.

***Annexe 3***

Différents fichiers intermédiaires générés pendant le processus de conception d'un instrument intelligent pris en exemple se trouvent dans cette annexe.

# Chapitre 2 : L'INSTRUMENTATION INTELLIGENTE

# Chapitre 2 : L'instrumentation intelligente

# L'instrumentation intelligente

Ce chapitre a pour but d'introduire le problème que nous adressons dans cette thèse. Ce problème se situe dans le domaine de la conception de la partie logicielle des instruments intelligents. Nous allons présenter ce domaine tout d'abord en expliquant comment nous modélisons un instrument intelligent (section I). La section qui suivra (section II) aura pour but de présenter différents outils de conception qui se basent sur le modèle d'instrument que nous aurons présentés précédemment. Cette partie nous permettra aussi de nous pencher sur les avantages et les manques de ces outils. Enfin nous terminerons par une conclusion qui exposera clairement les motivations de cette thèse (section III).

## *I. Modèle de conception de la partie logicielle d'un instrument intelligent*

Plusieurs modèles génériques d'instruments intelligents ont été proposés afin de prendre en compte les nouvelles fonctionnalités présentées dans le chapitre précédent. Tous ces modèles peuvent être classés en deux catégories selon qu'ils cherchent à spécifier l'instrument intelligent par son modèle interne ou externe.

Le **modèle interne** [Géhin 94] [Luttenbacher 97] d'un instrument intelligent décrit les fonctions qu'il doit intégrer pour réaliser les services qu'en attendent les utilisateurs. Il définit la structure et la nature des traitements implantés. Dans ce sens, il s'adresse plus particulièrement au concepteur.

Le **modèle externe** [Bayart&Staroswiecki 92] [Bayart&Staroswiecki 93] [Bouras 97] d'un instrument intelligent caractérise, d'un point de vue externe, l'ensemble des services prévus par le concepteur. Ceux-ci sont commandés à l'aide de requêtes et selon un protocole de commande spécifique appartenant au modèle. La structuration de ce modèle est donc indispensable pour assurer l'interopérabilité et l'interchangeabilité d'un ensemble d'instruments constituant une application. Ce modèle s'adresse donc plus particulièrement à l'utilisateur.

Plusieurs modèles d'instruments intelligents ont vus le jour depuis quelques années. Nous en abordons quelques uns ici.

Les dernières générations d'automates programmables intègrent toutes les fonctionnalités requises pour être considérées comme des instruments intelligents (communication, configuration, mesure, etc.). Par conséquent, on peut considérer le langage Grafcet comme faisant parti des langages de programmation des instruments intelligents. Ce langage permet de modéliser aussi bien la couche externe que la couche interne d'un instrument intelligent. En effet, il décrit les services rendus à l'utilisateur par un ensemble d'actions séquentielles. Pour décrire la couche externe on utilise souvent le GEMMA (Guide d'Etude des Modes de Marche et d'Arrêt). Il s'agit d'un guide graphique structuré qui propose des modes de fonctionnement types. Selon les besoins du système automatisé à étudier on choisit d'utiliser certains modes de fonctionnement. Ce GEMMA est ensuite concrètement codé grâce au langage Grafcet. Malgré le fait que ce langage soit très utilisé, il n'est pas assez restrictif car le concepteur peut effectuer très facilement un programme faux et ne s'en rendre compte qu'au moment des tests.

En ce qui concerne le modèle externe, l'approche USOM (USer Operating Mode) est la plus répandue [Staroswiecki&Bayart 94] [Staroswiecki&Bayart 96] [Bouras 97]. Dans ce type d'approche, l'instrument intelligent peut être considéré par un utilisateur comme une entité proposant des services, lesquels manipulent des variables et font appel à un ensemble de ressources. Ainsi, la notion de service est définie en adoptant une représentation de l'architecture matérielle de l'instrument intelligent identique à celle d'une machine informatique classique. Par ailleurs, et afin d'éviter la réalisation par l'utilisateur d'actions incompatibles, les différents services d'un instrument sont regroupés en sous ensembles cohérents dits modes d'utilisations.

En ce qui concerne le modèle interne, des modélisations formelles du concept d'instrument intelligent à l'aide de la méthode SADT sont proposées dans [Robert et al. 93] et [Géhin 94]. De telles descriptions permettent la représentation de l'architecture, des différentes activités de l'instrument, des flux de données ainsi que les moyens de réalisation des activités. Elles permettent au demandeur de mieux formuler son besoin et au fournisseur de satisfaire au mieux la demande. Néanmoins, nombre d'objectifs, besoins et contraintes, qu'ils soient quantitatifs ou qualitatifs, restent encore exprimés en langage informel, c'est-à-dire par du texte. Il est par exemple impossible de spécifier les conditions d'engagement et d'arrêt des traitements. Ce formalisme ne permet pas non plus de représenter le stockage des données qui est pourtant une fonctionnalité fortement liée au concept même de capteur intelligent. Ces descriptions génériques doivent donc être complétées par d'autres modèles de représentation. Des formalismes tels Réseaux de Pétri ou graphes d'états sont bien adaptés à la représentation des aspects temporels et de la gestion des activités.

Une autre approche pour le modèle interne a été la modélisation du capteur intelligent par une approche orientée objet à partir de la méthode OMT (Object Modeling Technique) [Rumbaugh et al. 91]. Cette approche est décrite dans [Luttenbacher 97]. Le but de cette méthode semi-formelle est de fournir trois modèles pour décrire les aspects statiques, dynamiques et fonctionnels. La variété des modèles, leur richesse sémantique et leur représentation graphique permet d'exprimer n'importe quel concept en restant très abstrait. Cette capacité d'abstraction peut être vue comme une force mais aussi comme une faiblesse. En effet, elle est source d'incohérences et va à l'encontre de certains principes de la construction du logiciel (validation dès l'analyse, automatisation de la construction).

Enfin, [Tailland 00] considère que les deux modèles internes et externes sont nécessaires et complémentaires. En effet, si le concepteur d'un instrument intelligent s'attache plus

particulièrement à l'implémentation des services de l'instrument, l'utilisateur, quant à lui, se préoccupe peu de savoir comment les traitements sont réalisés. Une fois qu'il a pris connaissance des services offerts par l'instrument, son problème est alors de pouvoir y accéder de la manière la plus simple qu'il soit. Tailland voit la couche interne d'un instrument intelligent comme un enchaînement de services appelés services internes. Ces services internes sont des entités fonctionnelles élémentaires qui ne sont pas accessibles directement par l'utilisateur. En fait, ce modèle utilise une approche client/serveur qui positionne la couche interne comme serveur de la couche externe. Pour la couche externe, le modèle USOM est utilisé. Donc, l'utilisateur voit l'instrument intelligent comme un ensemble de services (les services externes). Tailland effectue une analogie avec ce modèle en spécifiant que pour un service externe, la couche interne est constituée d'un ensemble de services internes qu'elle peut utiliser. En effet, pour effectuer un traitement qui peut être complexe (un service externe), une suite de services simples sont exécutés (services internes). Cette approche est la seule qui prenne en compte les deux modèles.

Dans les sections suivantes, l'approche de Tailland est présentée. C'est elle qui sert de base à nos travaux. Tout d'abord, nous présentons le modèle (section I.A). Ensuite, les différentes propriétés à vérifier pour respecter ce modèle sont exprimées (section I.B). Enfin, une présentation des différents acteurs qui entrent en jeu dans la conception des instruments intelligents et leurs tâches associées sera faite (section I.C).

## I.A. Modèle interne et externe : architecture client/serveur

La conception d'un instrument intelligent nécessite une description d'un point de vue interne. En effet, connaître les fonctionnalités souhaitées d'un instrument n'est pas suffisant ; encore faut-il savoir comment elles vont être réalisées si l'on veut pouvoir valider et/ou vérifier leur fonctionnement ou encore automatiser leur conception.

Rappelons que l'architecture matérielle d'un instrument intelligent s'apparente à celle d'une machine informatique classique. Ainsi, pour un utilisateur externe, un instrument peut être considéré comme une entité proposant des services qui manipulent des variables et font appel à un ensemble de ressources.

La figure ci-dessous (Figure 2-1), est le résultat des études menées dans [Benoit et al. 01]. Elle présente le modèle d'un instrument intelligent qui se base sur le modèle client/serveur de [Calvez 90].

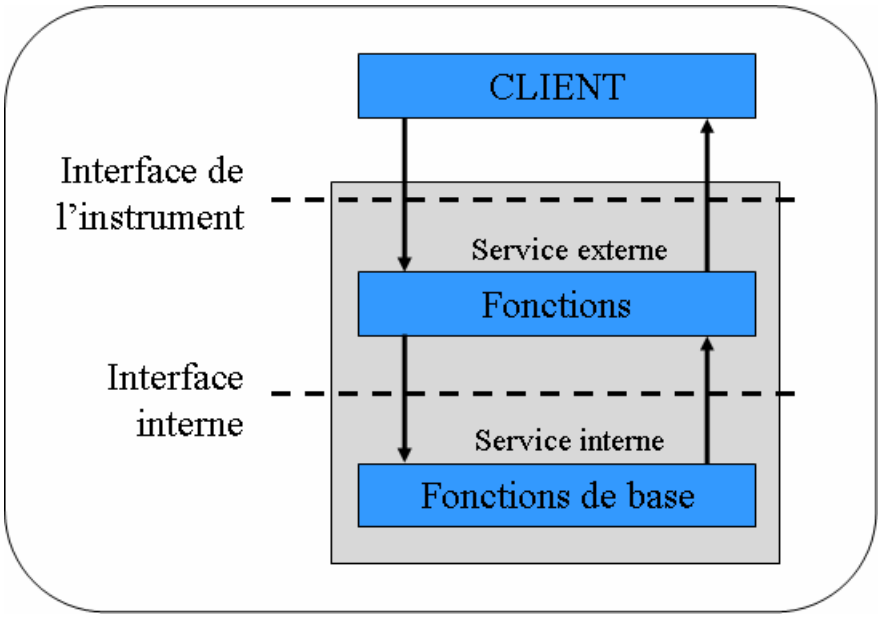



**Figure 2-1: Modèle des instruments intelligents basé sur [Benoit et al. 01]**

Ce modèle d'instruments intelligents comporte deux niveaux :

- La première relation de type client/serveur est celle qui se situe au niveau de l'interface de l'instrument. Elle permet donc d'exprimer les fonctionnalités offertes par un instrument à son utilisateur. Ce dernier est à prendre au sens large. Ainsi, il représente indifféremment, une personne physique qui, à l'aide d'une interface homme/machine, émet des requêtes, ou un autre instrument intelligent qui, à travers une demande de service joue alors le rôle d'un client. Les services permettant de réaliser ces fonctionnalités sont la représentation d'un point de vue externe des traitements que l'instrument peut effectuer.
- La deuxième relation client/serveur s'établit donc au sein même de l'instrument intelligent. Il correspond à la réalisation du service externe par l'instrument intelligent. Celle-ci nécessite l'exécution de traitements élémentaires qui peuvent être vus comme des services internes.

Pour les deux niveaux, l'échange d'information entre le client et le serveur est caractérisé par un échange d'évènements que nous allons détailler par la suite (section I.A.1). Les deux types de services (internes et externes) ainsi que leur organisation seront également présentés (sections I.A.2 et I.A.3).

## I.A.1. Les évènements

Comme nous venons de le dire, dans le modèle de type client/serveur choisi, le passage d'une couche à l'autre s'effectue par l'intermédiaire d'évènements. Deux types d'événements sont considérés [Dasarathy 85] :

- Les évènements internes qui sont produits par le système lui même.
- Les évènements externes qui sont eux produits par l'environnement du système.

### *I.A.1.a. Evènements externes*

La définition d'un évènement externe est la suivante :

> *On appelle **évènement externe** d'un instrument intelligent, tout évènement dont l'occurrence est produite par un processus externe à l'instrument et consommé par l'instrument.*

L'occurrence d'un évènement externe représente la demande d'exécution d'un service externe. Cette association entre un évènement et un service est unique, ainsi à chaque évènement externe correspond un et un seul service externe. Autrement dit, l'occurrence d'un évènement externe constitue la requête d'un unique service externe (voir section I.A.2.a). Ceci est notamment dû au fait que l'on doit pouvoir exécuter plusieurs services internes en parallèle.

### *I.A.1.b. Evènements internes*

La définition d'un évènement interne est la suivante :

> *On appelle **évènement interne** d'un instrument intelligent, tout évènement dont l'occurrence est produite et consommée par l'instrument.*

De la même manière que l'occurrence d'un évènement externe constitue une requête de service externe, l'occurrence d'un évènement interne constitue une requête de service interne. Cependant, il n'y a pas unicité entre évènement interne et service interne comme c'est le cas pour la relation évènement externe / service externe. En effet, un évènement interne peut être affecté à plusieurs services internes. Autrement dit, l'occurrence d'un évènement interne peut constituer la requête de plusieurs services internes.

## I.A.2. Les services d'un instrument intelligent

### *I.A.2.a. Les services externes*

Une définition d'un service externe a été donnée par [Bouras 97] :

> *En se plaçant d'un point de vue externe à l'instrument intelligent, un service est le résultat de l'exécution d'un traitement (ou d'un ensemble de traitements), auquel on peut associer une interprétation en terme fonctionnel.*

On voit ici que [Bouras 97] considère qu'un service externe peut être décrit en utilisant le résultat de son exécution. Il peut donc être caractérisé, par exemple, par une mise à jour de la valeur mesurée par un capteur ou par la valeur du signal de commande d'un actionneur.

Cependant, cette définition ne donne aucune information sur la façon dont est exécuté un tel service. [Tailland 00] caractérise donc un service externe de la façon suivante :

> *Un **service externe** est le résultat de l'exécution d'un ensemble de services internes de l'instrument intelligent.*

Si l'on s'en tient à cette définition, on peut représenter un service externe comme ci-dessous (Figure 2-2)

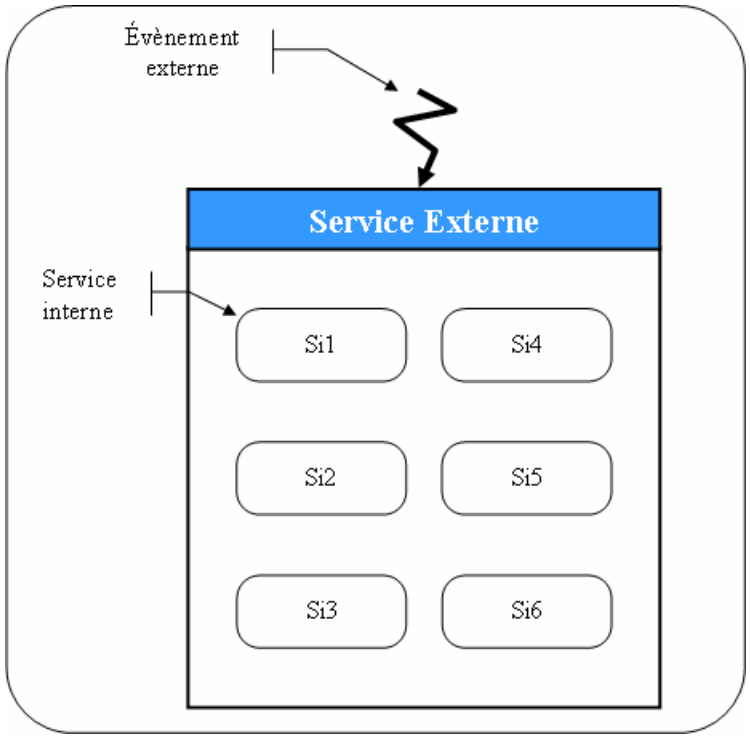



**Figure 2-2 : Exemple de représentation d'un service externe**

On retrouve bien le fait qu'un service externe est déclenché par un évènement interne unique et qu'il est composé d'un ensemble de services internes.

Il existe des liens logiques entre ces services internes qui composent les services externes. Ils se traduisent par des liens de causalité : telle opération doit s'effectuer avant ou après telle autre. Un service interne utilisant le résultat d'un autre service interne doit impérativement s'effectuer après celui-ci. Ces contraintes ou absences de contraintes causales s'expriment commodément à l'aide d'évènements internes. C'est ce que nous allons voir dans la partie suivante en présentant en détail la notion de service interne.

### *I.A.2.b. Les services internes*

Comme nous venons de le voir, les services internes sont les éléments de base qui permettent à un service externe de réaliser un travail donné. Cependant, nous ne savons encore pas précisément ce que l'on entend par service interne. Une définition d'un service interne est donnée par [Tailland 00] :

> *Un **service interne** est la plus petite entité de l'ensemble des traitements de l'instrument intelligent. Du point de vue de la conception, cette entité est un granule élémentaire des fonctionnalités intelligentes de l'instrument.*

Les services internes peuvent avoir deux types de fonctionnalités différentes :

- la première est de dialoguer directement avec la partie matérielle d'un instrument intelligent (ex : récupérer la valeur d'un capteur, actionner un vérin, etc.),
- la seconde est de transformer la valeur d'une variable afin de pouvoir la rendre utilisable soit par un autre service interne soit par le service externe qui le déclenche (ex : un filtre, un convertisseur d'unité, etc.).

D'un point de vue conceptuel, les services internes peuvent être vus comme les constituants élémentaires d'une application plus complexe. Un ensemble de services internes constitue alors une bibliothèque de composants de base. L'utilisateur peut les assembler en vue de réaliser un service externe. Ces fonctionnalités sont nommées fonctionnalités internes dans le sens où elles ne sont pas accessibles directement par l'utilisateur, mais uniquement au travers des services externes.

Afin de réaliser l'action pour laquelle il a été créé, un service interne peut avoir besoin de données spécifiques (ex : de combien de millimètres faut-il avancer un vérin ?) et de ressources matérielles (capteur, actionneur, etc.). Une fois cette action réalisée, il doit pouvoir transmettre ses résultats par l'intermédiaire de variables de sorties.

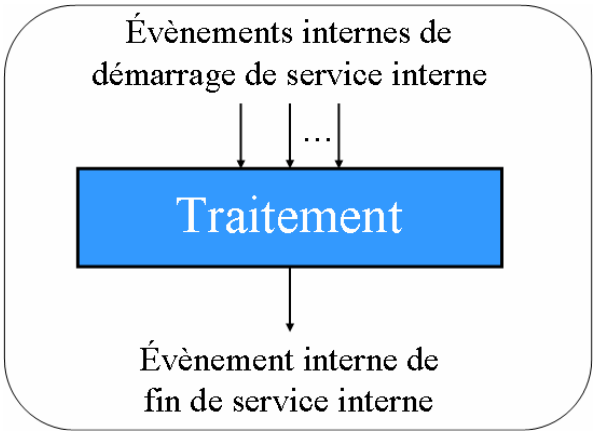



**Figure 2-3 : Représentation d'un service interne**

Comme le montre la figure ci-dessus (Figure 2-3), un service interne est vu comme une boite noire réalisant un traitement élémentaire spécifique. Pour pouvoir déclencher un service interne, un ou plusieurs événements internes peuvent être nécessaires.

Au minimum, un service interne doit avoir une entrée qui recevra un évènement interne qui le déclenchera et une sortie qui retournera un acquittement signifiant que le service est terminé.

Nous avons parlé dans la partie précédente de l'ordre d'exécution des services internes au sein d'un service externe. On voit ici que cet ordre d'exécution est déterminé dès la création des services internes en spécifiant par exemple que tel service interne se déclenche sur l'évènement interne de fin de tel autre service interne. Si l'on regroupe toutes ces informations dans un même graphe on dit que l'on construit le graphe de dépendances d'un instrument intelligent. Un exemple est donné dans la figure suivante (Figure 2-4).

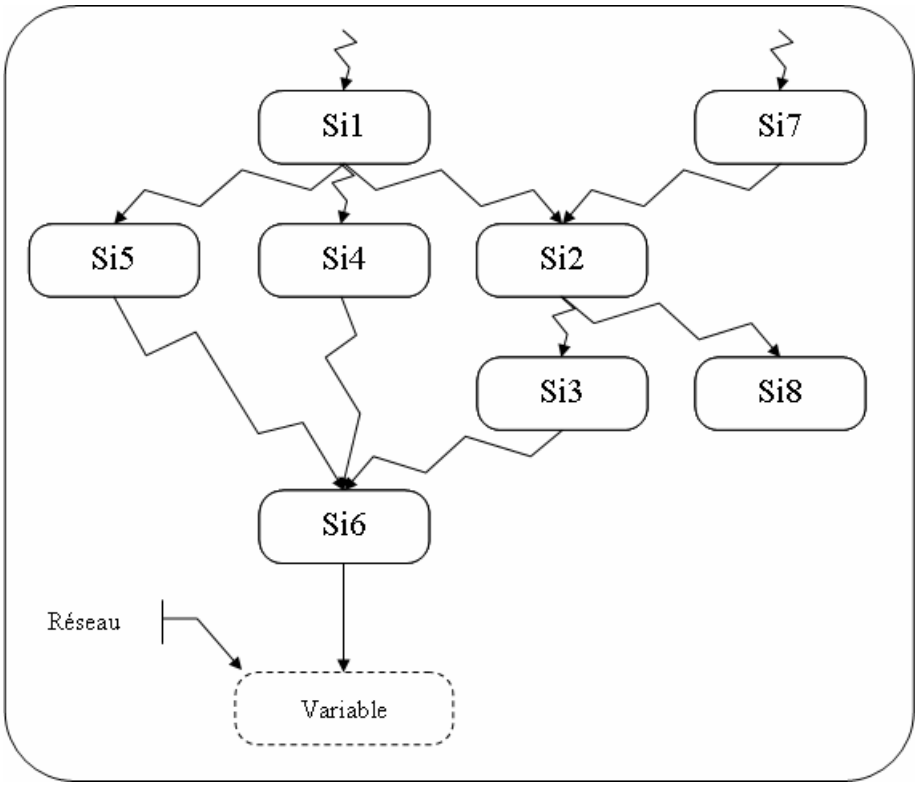



**Figure 2-4 : Exemple de graphe de dépendances**

Si le service externe de la figure 2-4 de la section précédente se base sur ce graphe de dépendances, il exécutera Si1 puis Si2 et Si4 en parallèle. Une fois Si2 exécuté, il exécutera Si3. Pour ce qui est de Si6, il nous manque une information. En effet, Si3 ***ET*** Si4 ***ET*** Si5 sont-ils nécessaires pour l'exécuter ou s'agit-il de Si3 ***OU*** (Si4 ***ET*** Si5) ou encore de (Si3 ***OU*** Si4) ***ET*** Si5 etc. Cette information est en fait codée à l'intérieur du service interne. Ici, nous considèrerons qu'il faut que les trois services internes soient terminés pour que Si6 soit exécuté.

Cet exemple montre que les services internes Si6 et Si8 sont particuliers. En effet, ils ne produisent pas d'évènement en fin d'exécution mais ils effectuent, au plus, un export de variable. Pour Si6, la variable exportée est appelée *variable externe*. Rappelons qu'un service externe peut être caractérisé par le résultat de l'exécution de son traitement. Or si par exemple, le but de ce dernier est de récupérer la valeur d'un capteur, elle doit être transmise à l'utilisateur (une personne physique ou un autre instrument). Pour ce faire, la variable qui contient cette valeur doit être mise à disposition sur le réseau. C'est le travail du service interne Si6. Le résultat du traitement d'un service externe peut aussi être la mise à jour d'une variable interne comme par exemple la modification de la commande d'un actionneur. Dans ce cas, le service interne qui réalise cette mise à jour ne délivre pas d'évènement de fin de service interne. C'est le cas de Si8.

Il n'y a donc jamais d'évènement qui nous indique qu'un service externe est terminé puisqu'il se terminera toujours par ces services internes particuliers.

## I.A.3. Organisation des services d'un instrument intelligent

En résumé, on voit que les services externes sont le résultat de l'exécution d'un ou de plusieurs services internes ordonnés de façon séquentielle et/ou de façon parallèle. Ces

services externes sont directement accessibles par l'utilisateur (un homme ou un autre instrument), ce qui n'est pas le cas des services internes. Ces derniers ne sont exécutables qu'à la demande des services externes.

Cette organisation permet de donner un niveau d'abstraction qui aide l'utilisateur dans la manipulation d'un instrument intelligent mais ne le protège pas encore contre une mauvaise utilisation. En effet, il se peut que certains services externes doivent absolument être précédés par d'autres pour fonctionner. Par exemple, un capteur intelligent ne peut offrir ses services de mesure qu'après une phase de configuration ; un actionneur intelligent ne peut être en maintenance et en même temps proposer des services liés à la production. Ceci a pour conséquence qu'à un instant donné, un utilisateur ne doit pas pouvoir accéder à l'ensemble des services externes disponibles de l'instrument. De manière similaire, dans un état donné, tel service interne sera exécuté à la place de tel autre car une ressource est indisponible.

Ainsi, afin de simplifier les conditions d'activation des services externes et des services internes, il a été convenu de les organiser en sous-ensembles cohérents qui engendrent, comme décrit dans les paragraphes suivants, les notions de modes externes et de modes internes.

### *I.A.3.a. Les modes externes*

La solution qui consiste à organiser les services externes en sous-ensembles cohérents a été introduite dans [Bayart&Staroswiecki 94]. Une définition en a été donnée dans [Bouras 97] :

> *Un **mode externe** est un sous-ensemble de l'ensemble des services externes de l'instrument intelligent. Un mode externe comprend au moins un service externe et chaque service externe appartient à au moins un mode externe.*

Lorsque les services externes sont structurés de la sorte, l'instrument intelligent n'accepte que les requêtes des services inclus dans le mode dans lequel il se trouve. La notion de mode externe répond donc au problème mentionné précédemment. Par contre, l'inconvénient d'une telle approche est que la structuration des services externes en modes externes est complètement arbitraire. En effet, bien qu'il existe une classification relativement générale des types de fonctionnement possibles permettant d'organiser les services en sous-ensembles cohérents, cette organisation dépend, en définitive, de l'instrument et de l'utilisation qui en est faite.

Chaque mode d'utilisation doit comporter un service externe spécifique de changement de mode. Le mode d'origine étant le mode courant, la requête de changement de mode doit indiquer le mode destination. Cependant, on ne peut pas rejoindre n'importe quel mode à partir de n'importe quel autre pour des raisons de sécurité et de cohérence de fonctionnement. Par exemple, la requête de passage en mode automatique devra être rejetée si les paramètres de l'instrument intelligent n'ont pas été configurés. C'est pourquoi, il convient de définir les conditions logiques qui permettent le passage d'un mode d'utilisation à un autre. La donnée de l'ensemble des modes et des conditions de passage définit entièrement la gestion des modes d'utilisation de l'instrument intelligent et peut être décrite par un graphe d'état (voir Figure 2-5 ci-dessous).

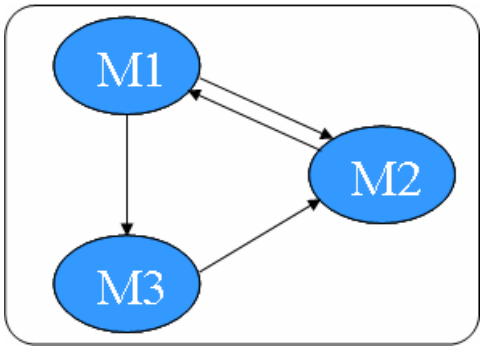



**Figure 2-5 : Exemple de graphe de changement de mode externe**

On remarque dans cet exemple que si l'on est dans le mode *M1*, on a accès à *M2* et *M3*. Par contre, une fois dans *M3*, on ne peut retourner dans *M1* qu'en passant auparavant par *M2*.

Dans ce modèle de la même manière que les services externes sont rangés dans des modes externes, nous allons voir par la suite que les services internes sont organisés dans des modes internes.

### *I.A.3.b. Les modes internes*

Les services internes sont eux aussi organisés dans des modes de la même manière que le service externe bien que dans un but différent.

> *Un **mode interne** est un sous-ensemble de l'ensemble des services internes de l'instrument intelligent. Un mode interne comprend au moins un service interne et chaque service interne appartient à au moins un mode interne.*

Cette notion de mode interne a été introduite afin de gérer la notion de service dégradé. En effet, les services internes sont des entités qui utilisent des ressources et si l'une d'elles n'est plus disponible à un instant donné, le service interne qui en a besoin ne va pas pouvoir s'exécuter. De ce fait, un service dégradé pourrait être automatiquement déclenché à la place du service demandé et ceci de façon totalement transparente pour l'utilisateur. Ce genre de concept suppose bien entendu que l'instrument dispose d'une entité capable de surveiller l'état de ses ressources.

Lors de nos travaux, nous avons décidé d'abandonner complètement cette notion de mode interne. En effet, cette notion est discutable à différents niveaux.

Tout d'abord, la notion de mode interne appartient, comme son nom l'indique, à la couche interne de l'instrument intelligent. Par définition, cette notion devrait donc être totalement transparente à l'utilisateur, or, il est tout à fait impossible de concevoir qu'un instrument fasse, par exemple, une mesure avec un capteur plutôt qu'avec un autre sans que l'utilisateur en soit informé. On pourrait très facilement imaginer, par exemple, que la précision de la mesure demandée soit une chose primordiale pour l'utilisateur et que, malheureusement, le nouveau capteur choisi ne soit pas capable de mesurer avec la précision demandée. Mais, ce qui est le plus grave est le fait que l'utilisateur final ne le saura même pas puisque pour lui la couche interne d'un instrument intelligent est inaccessible !

La deuxième raison pour laquelle nous n'utilisons plus cette notion de mode interne est tout simplement que, suivant les cas d'utilisation, si une ressource n'est plus disponible, par exemple un capteur, l'utilisateur pourrait préférer le changer plutôt que de passer dans un mode interne dégradé.

Dans la suite de ce manuscrit, lorsque nous parlerons de mode, nous ne préciserons donc plus interne ou externe. La seule notion que nous gardons est celle de mode externe.

Dans la partie suivante, nous allons présenter les différentes propriétés à respecter pour construire un instrument intelligent de façon correcte.

## I.B. Les vérifications à faire

Dans les paragraphes précédents, nous avons introduit différents concepts :
- les services internes,
- les services externes,
- les modes.

Ces différents concepts doivent être conçus par les personnes qui vont participer à la création des instruments intelligents. Il va donc de soit que, aussi bien pour le modèle interne que pour le modèle externe, différentes propriétés de construction doivent être respectées.

Nous allons donc lister les propriétés que vont devoir respecter chaque concept afin que ce dernier réponde bien aux exigences de conception des instruments intelligents. Ces propriétés proviennent toutes de [Benoit et al. 01]. Elles doivent impérativement être vérifiées lors de la conception d'un instrument intelligent bâti sur le modèle que nous avons présenté dans ce chapitre.

Dans cette section, nous allons utiliser les notations suivantes :

- $es$ est un service externe,
- $is$ est un service interne,
- $m$ est un mode,
- $\mathcal{ES}$ est un ensemble de services externes,
- $\mathcal{IS}$ est un ensemble de services internes,
- $\mathcal{EE}$ est un ensemble d'évènements externes,
- $\mathcal{M}$ est un ensemble de modes destinés à l'utilisateur (USOM),
- $fes$ est une application de $\mathcal{ES}$ dans $\mathcal{EE}$ qui lie les services externes et les évènements externes,
- $\mathcal{EG}$ est une relation sur $\mathcal{M}$ x $\mathcal{M}$ qui représente l'ensemble des transitions possibles entre les modes.
- $feg$ est une fonction qui représente le lien entre les évènements externes et les transitions entre les modes.

Dans la suite du manuscrit, ces propriétés vont être modifiées (car le modèle va évoluer). Par conséquent, pour éviter de les confondre avec les suivantes, la numérotation des propriétés inclut le numéro de chapitre dans lequel elles se trouvent.

### I.B.1. Propriétés des services externes

**Propriété 2.1**
*Un instrument intelligent comporte un ensemble fini $\mathcal{ES}$ de services externes.*
$\mathcal{ES}$ est un ensemble fini

**Propriété 2.2**
*Un service externe est déclenché sur l'occurrence d'un évènement externe unique.*
*fes* est une fonction bijective de $\mathcal{ES}$ *dans* $\mathcal{EE}$

**Propriété 2.3**
*Un service externe est constitué d'un ensemble non vide de services internes.*
$\forall es \in \mathcal{ES}, \exists s \mid s \in es-\{\varnothing\}$

**Propriété 2.4**
*Un service externe est constitué d'un ensemble de services internes atteignables.*

### I.B.2. Propriétés des services internes

**Propriété 2.5**
*Un instrument intelligent comporte un ensemble fini $\mathcal{IS}$, de services internes.*
$\mathcal{IS}$ est un ensemble fini

### I.B.3. Propriétés des modes

**Propriété 2.6**
*Un mode externe est un ensemble non vide de services externes.*
$m \in \mathcal{M} \Leftrightarrow m \in \mathcal{P}(\mathcal{ES}) - \{\varnothing\}$
*où $\mathcal{P}(\mathcal{ES})$ désigne l'ensemble des parties de $\mathcal{ES}$*

**Propriété 2.7**
*Un service externe appartient au moins à un mode externe.*
$\forall es \in \mathcal{ES}, \exists m \in \mathcal{M} \mid es \in m$

**Propriété 2.8**
*Chaque transition entre les modes répond à un unique évènement externe.*
*feg* est une fonction injective de $\mathcal{EG}$ dans $\mathcal{EE}$

**Propriété 2.9**
*Le service externe associé à l'évènement externe qui gère la transition entre le mode* m1 et m2 appartient au mode m1.
$\forall (m1, m2) \in \mathcal{EG}, feg^{-1}(feg((m1, m2))) \in m1$
*($feg^{-1}$ est la fonction inverse de feg)*

**Propriété 2.10**
*Un instrument peut atteindre et quitter n'importe quel mode.*
$\forall m1 \in \mathcal{M}, \exists m2 \in \mathcal{M}, (m1, m2) \in \mathcal{EG}$

## I.C. Les Acteurs du système

La conception d'instruments intelligents est un problème complexe car des compétences en physique, mécanique et informatique sont nécessaires et de nombreuses contraintes doivent être respectées. La conception rapide de tels instruments n'est donc pas une chose aisée. En effet, il est difficile pour une entreprise de trouver une personne qui ait de telles compétences. Il faut donc réussir à diviser le travail à réaliser en fonction des compétences de chacun. La figure ci-dessous (Figure 2-6), reprend le modèle qui nous intéresse. On remarque que deux rôles entrent en jeu dans la conception de la partie logicielle de tels instruments.

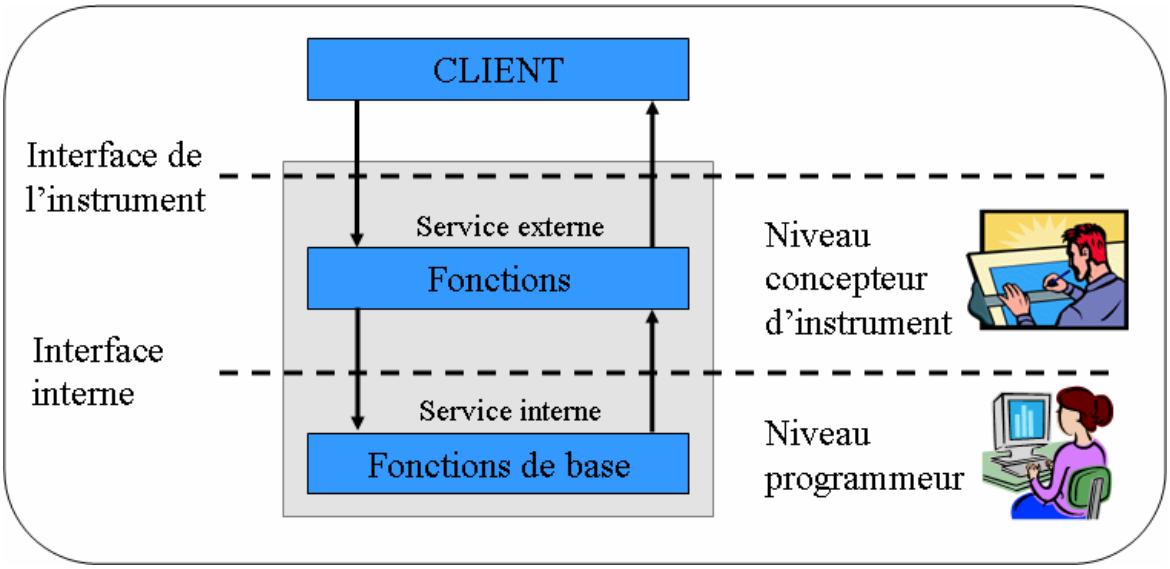



**Figure 2-6 : Modèle des instruments intelligents avec les acteurs associés**

Le « programmeur » a à sa charge la création des services internes. La création de ces fonctions de base ne peut être confiée qu'à une personne spécialiste en informatique car ce sont des fonctions de bas niveau qui pour la plupart sont directement liées à la partie matérielle des instruments. Une fois ces briques de base créées, elles doivent être vérifiées et validées une à une afin d'éliminer tous les bugs qui ont pu être introduits.

La deuxième personne qui entre en jeux dans la conception des instruments intelligents est celle que l'on nomme le « concepteur d'instruments ». Cette personne est spécialiste du domaine de l'instrumentation intelligente et est, dans la plupart des cas, celle qui conçoit aussi leur partie matérielle. Dans la majorité des cas, cette personne n'est pas spécialiste en informatique et doit néanmoins être capable d'agencer les services internes créés par le programmeur afin de créer les services externes. Une fois les services externes créés, le concepteur d'instruments doit aussi être capable d'organiser ces services externes dans différents modes de fonctionnement. Enfin, il doit fournir à l'utilisateur potentiel de l'instrument intelligent une Interface Homme/Machine (IHM) spécifique au pilotage de l'instrument.

C'est en partant dans l'optique de simplifier le travail du concepteur d'instruments que différents outils ont été réalisés par Tailland [Tailland 00]. Nous allons les présenter dans la partie suivante.

# II. Outils de conception d'instruments intelligents

La nécessité d'avoir des outils de conception performants pour la partie logiciel des instruments intelligents est partie d'un constat très simple : les contraintes techniques conjuguées à une réduction constante des délais et à un accroissement de la qualité requise, font que le concepteur est à la recherche d'outils capables de l'aider durant toutes les phases de la vie du produit : spécification, implantation, validation, maintenance. Un des handicaps majeurs dans notre cas est qu'en plus, la personne responsable de la majeure partie de la création de ce logiciel n'est pas spécialiste en informatique.

Les outils de conception qui ont été créés ont pour but d'aider le concepteur d'instruments à concevoir les services externes, les modes mais aussi de générer automatiquement le code de l'application associé. Il va de soit que lors de la conception, toutes les propriétés intrinsèques au modèle doivent être vérifiées.

Une première version d'une telle boite à outils a été réalisée il y a quelques années au sein du laboratoire [Tailland 00]. Cet ensemble d'outils de conception se nomme CAPTool. La description d'un instrument intelligent a été prévue de manière textuelle à l'aide d'un langage pivot nommé CAP. Ce langage servait de support au modèle d'instrument développé dans la section I et offrait une syntaxe qui reprenait les termes utilisés dans le domaine de l'instrumentation intelligente.

Dans un premier temps, nous allons présenter brièvement le langage CAP. Ensuite nous présenterons le processus conception d'un instrument intelligent et les différents outils qui l'accompagne.

## II.A. Le langage CAP

Le langage CAP est un langage textuel qui permet de décrire les fonctionnalités d'un instrument intelligent selon le modèle présenté précédemment. De ce fait, ce langage doit permettre la déclaration de services internes, de services externes, de modes, de transitions entre les modes, d'événements etc.

Ce langage a, en définitive, été créé de façon totalement ad hoc. Néanmoins, il a pour but de faciliter la conception des instruments intelligents. Il utilise donc des termes spécifiques au domaine de l'instrumentation intelligente. La finalité de ceci est de fournir au concepteur d'instruments un langage qui dispose d'une grammaire facilement assimilable pour une personne du domaine.

L'avantage de ce type de langage est que le concepteur d'instruments intelligents n'aura plus à connaître un langage de programmation classique (Java, C, C++), qu'il ne maîtrisera pas la plupart du temps. Il concevra ces instruments intelligents de façon presque naturelle puisqu'il utilisera, pour ce faire, des termes qui, pour lui, sont courants.

L'annexe A de [Tailland 00] présente la grammaire détaillée du langage CAP[1], cependant, nous en présentons les principales instructions ici (toutes les possibilités du langage CAP ne sont pas exposées).

[1] <> : Variables
[] : Valeur optionnelle
[]* : Valeur pouvant apparaître zéro ou plusieurs fois

La première chose qu'un concepteur d'instruments doit faire est de donner un numéro à l'instrument sur le réseau. Pour ce faire il utilise l'instruction suivante :

```
.numero = <numéro> ;
```

Ensuite, le concepteur a la possibilité de faire des inclusions de la même manière qu'on les ferait en langage C :

```
#include <fichier> ;
```

Ici, une bibliothèque du langage C peut être incluse car, par exemple, elle pourrait servir dans la définition d'un ou de plusieurs services internes.

Les déclarations suivantes concernent les variables externes utilisées par l'instrument de la même manière que si on les déclarait en langage C :

```
var <C-type> <variable>;
```

Ensuite, on déclare des liens de communication en importation et en exportation. Ces liens remplissent deux fonctions. D'une part, ils permettent de spécifier par quel protocole s'effectue un envoi ou une réception de variable et, d'autre part, ils sont l'unique moyen de manipuler un événement externe :

```
link export <lien> [, <lien>]* ;
link import <lien> [, <lien>]* ;
```

La partie suivante concerne directement la couche externe. On y déclare les modes, les transitions entre les modes, le mode par défaut dans lequel se trouve l'instrument intelligent au démarrage et les service externes :

```
mode <mode> [, <mode>]* ;
transition <mode_départ> to <mode_arrivée> on <lien_import> ;
default mode <mode> ;
service <service> on <lien> [in <mode>]
{
     Uses <service_interne>, <service_interne>, ... ;
}
```

On peut voir qu'un service est déclaré comme se déclenchant sur un « lien », c'est à dire sur un événement externe particulier. On voit aussi qu'il est composé par un ensemble de services internes. Enfin, on remarque que la déclaration du mode auquel appartient le service externe est optionnelle. Cela signifie que, si rien n'est spécifié, le service externe déclaré appartient à tous les modes.

Enfin, la dernière partie concerne la couche interne. On y trouve la déclaration des évènements internes et des services internes :

```
ievent <event> [, <event>]* ;
iservice <service> on end(<iservice>)
{
      <Déclaration du service interne en C>
}
```

On peut remarquer ici que la déclaration du cœur des services internes s'effectue à ce niveau. Par conséquent, la séparation entre le travail du programmeur et du concepteur d'instruments n'est pas très nette et peut entraîner des disfonctionnements si, par exemple, le concepteur d'instruments modifie par inadvertance le code créé par le programmeur.

Nous venons de voir comment est conçu un instrument intelligent en langage CAP, maintenant nous allons voir le processus de conception qui va nous permettre de passer de ce fichier CAP au code source de l'instrument intelligent.

## II.B. Le processus de conception

Le processus de conception présenté dans la figure ci-après (Figure 2-7) est celui que doit suivre le concepteur d'instruments. Nous ne parlons donc pas ici de la partie amont qui concerne le travail du programmeur. En effet, une fois la partie matérielle de l'instrument intelligent conçue, le programmeur doit créer les services internes qui lui sont associés. C'est seulement une fois ceci réalisé que le travail du concepteur d'instruments au niveau du logiciel commence.

Le processus est assez simple du point de vue du concepteur d'instruments. En effet, une fois l'instrument intelligent décrit en langage CAP, le concepteur d'instruments doit lancer le compilateur CAP. Par cette simple action, le concepteur va pouvoir générer le code exécutable de l'instrument intelligent qui sera chargé en ROM et son interface Homme/Machine qui pourra être utilisée par une personne physique pour interagir avec l'instrument.

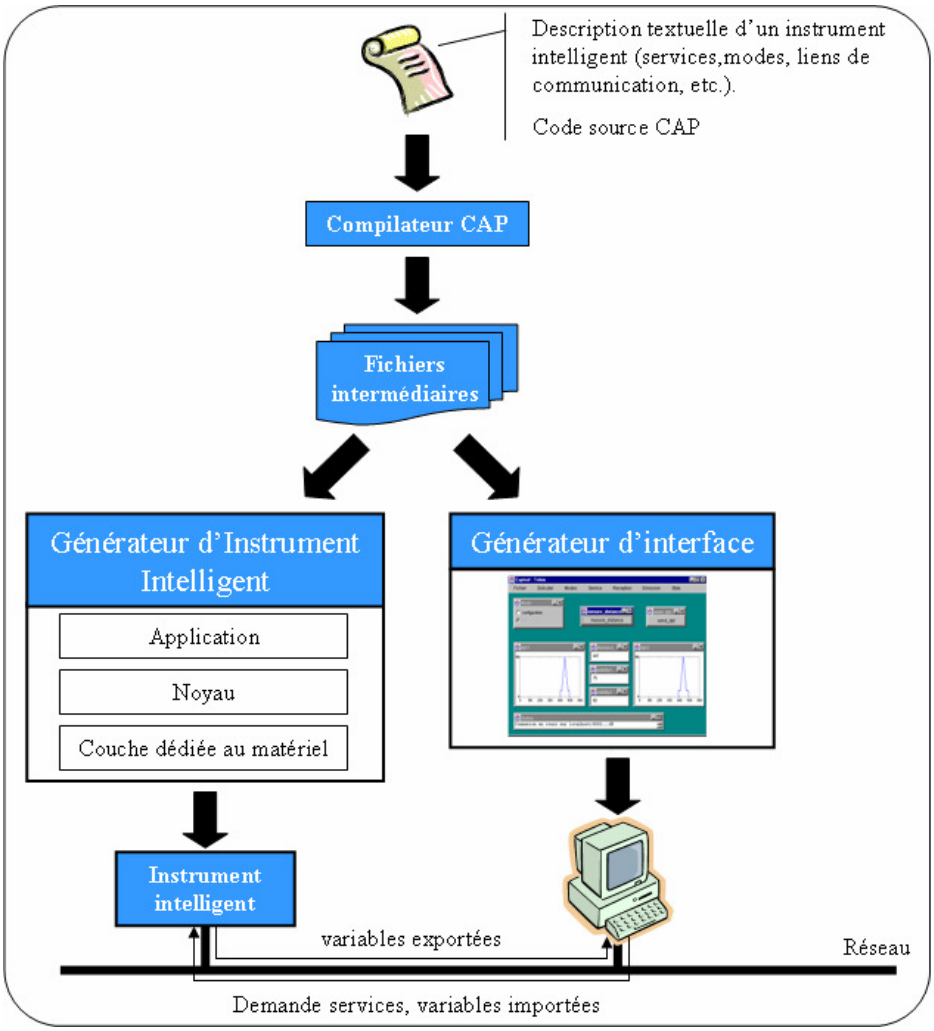



**Figure 2-7 : Processus de conception de la partie logiciel d'un instrument intelligent [Tailland 00]**

## II.C. Le compilateur CAP

Le compilateur du langage CAP a été réalisé à l'aide d'un générateur d'analyseur syntaxique YACC [Johnson&Murray 75] et d'un générateur d'analyseur lexical LEX [Lesk&Schmidt 75]. Ce compilateur est structuré en plusieurs parties :

- l'analyseur lexical ;
- l'analyseur syntaxique ;
- l'analyseur sémantique ou contextuel ;
- le générateur de code.

L'analyseur lexical a pour fonction d'inspecter le programme source, caractère par caractère, d'éliminer les caractères superflus, de reconnaître les unités lexicales (tokens), de construire les lexèmes à partir de ces caractères et de les passer à l'analyseur syntaxique.

L'analyseur syntaxique ou parseur effectue ensuite une analyse syntaxique du programme source en se basant sur les unités lexicales que lui fournit l'analyseur lexical. L'analyse syntaxique consiste à vérifier si le programme est syntaxiquement correct, c'est-à-dire si la séquence des symboles dans le programme source respecte les règles de grammaire du langage CAP.

L'analyseur sémantique ou contextuel est une partie très importante de ce compilateur car il vérifie si le fichier CAP, qui est déjà syntaxiquement correct, est aussi sémantiquement bon. Ainsi, il procède à toutes les vérifications des propriétés du modèle d'instrument intelligent.

Enfin, le générateur de code permet de générer un code intermédiaire (XML) qui sera ensuite utilisé par un générateur d'interface et un autre générateur de code (générateur du code source de l'instrument intelligent).

## II.D. Le générateur d'interface (GI)

Comme nous venons de le voir précédemment, le générateur d'interface (GI) produit l'interface homme/machine (IHM) associé à un instrument intelligent. Pour ce faire, il se base sur le fichier qui a été fourni par le compilateur CAP.

L'interface constitue la partie client de la relation entre l'instrument et l'utilisateur (modèle client/serveur). Elle permet ainsi à un utilisateur de piloter l'instrument par le biais de requêtes qui sont transmises sur les liens de communication.

La figure ci-dessous (Figure 2-8), donne un exemple d'interface.

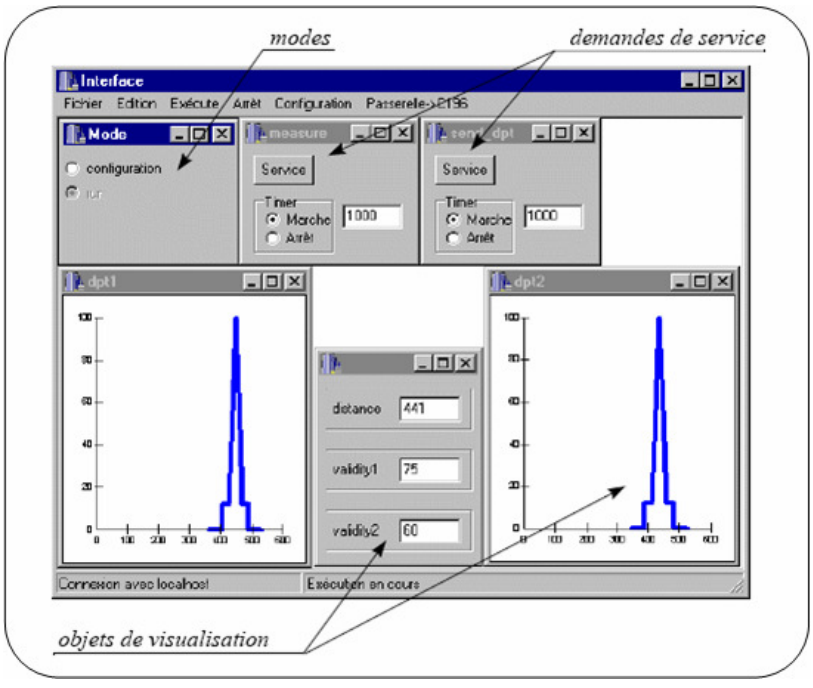



**Figure 2-8 : Exemple d'IHM [Tailland 00]**

Son implémentation a été réalisée en Java et est basée sur un principe de fonctionnement assez simple. En effet, le rôle de l'IHM est d'envoyer des requêtes à destination de l'instrument et de consommer les variables produites par ce dernier. L'IHM se comporte donc comme un producteur/consommateur de messages. Pour produire ou consommer ces messages, l'IHM va s'appuyer sur des objets graphiques prédéfinis. Le rôle du GI est donc d'instancier ces objets à partir des informations fournies par le compilateur. Les objets manipulés sont de quatre types :

- Un premier type d'objet, relatif aux modes, indique sous la forme d'une liste de boutons radio dans quel mode externe (ou mode d'utilisation) se trouve l'instrument. Cet objet permet à l'utilisateur d'effectuer des demandes de changement de mode en utilisant la souris. Seuls les modes accessibles à partir du mode courant peuvent être sélectionnés conformément aux modèles et à la configuration introduite. De ce point de vue, on peut considérer que l'utilisation est sûre. L'instrument ne peut ainsi jamais être placé dans une configuration non définie par l'utilisateur au travers du langage CAP.
- Un second type d'objet regroupe les messages relatifs aux demandes de services. Chaque service externe de l'instrument est une instance d'un objet permettant à l'utilisateur d'envoyer une requête de manière manuelle (en cliquant sur un bouton) ou automatique (la requête est émise périodiquement à une fréquence fixée par l'utilisateur). De manière analogue au point précédent, ne sont accessibles, à un instant donné, que les services valides dans le mode en cours.
- Un troisième type d'objet concerne les émissions de variables. Il permet à l'utilisateur de modifier la valeur d'une variable externe.
- Le dernier type d'objet permet de visualiser les variables externes produites par l'instrument. Différents types de visualisation sont disponibles en fonction de la nature de la variable. Ils permettent ainsi d'afficher des valeurs numériques, des chaînes de caractères, mais aussi des graphiques, des boutons, etc. Plusieurs variables peuvent être regroupées dans le but de constituer un groupe de réception, afin de construire des tableaux de bord plus ergonomiques.

## II.E. Le générateur d'instruments intelligents (GII)

Tout comme le générateur d'interface, le générateur d'instruments intelligents (GII) se base sur le fichier fourni par le compilateur CAP pour créer le code source d'un instrument intelligent.

Pour produire ce code source, il doit être capable de créer :

- les données concernant les services internes, les services externes, les modes, les événements, etc.
- un automate capable de gérer l'exécution de l'instrument intelligent (les changements de mode, le déclenchement de service externe, etc.).

En ce qui concerne les données, il suffit de créer les variables qui correspondent à chaque entité décrite dans le fichier intermédiaire. Ces données sont donc spécifiques à chaque instrument intelligent.

Pour l'automate, les choses sont différentes. En effet, il va être identique dans tous les instruments intelligents car ils doivent tous avoir le même comportement. L'algorithme de l'automate associé à l'instrument est un programme séquentiel. Son fonctionnement est décomposé en quatre étapes représentées sur la figure ci-dessous.

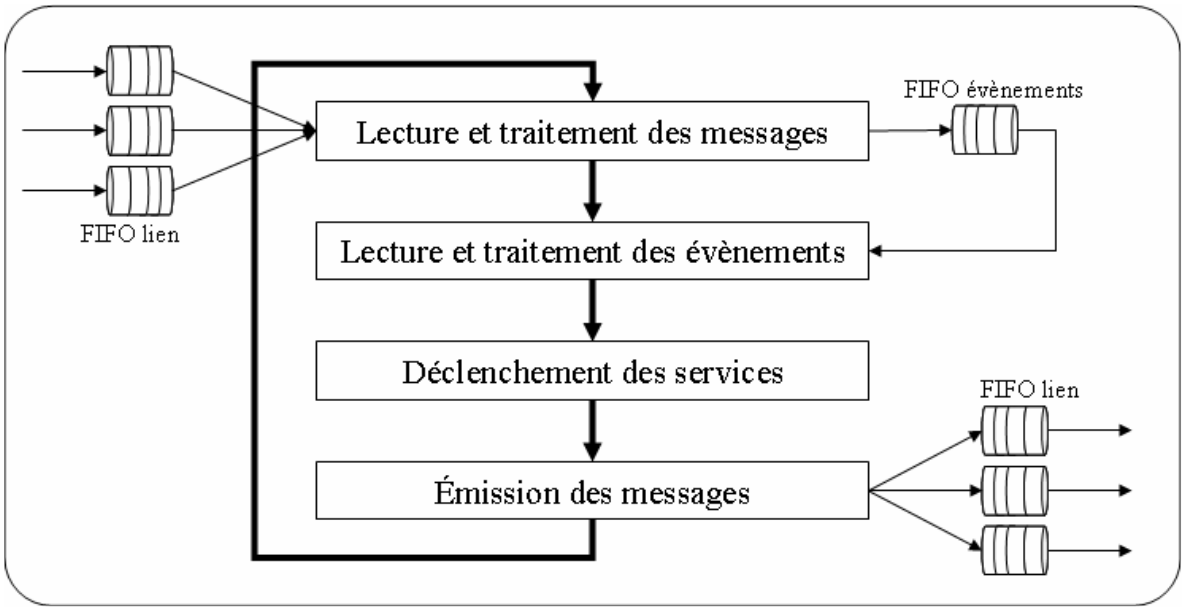



**Figure 2-9 : Fonctionnement de l'automate [Tailland 00]**

La première étape consiste à lire un message sur chaque lien de communication[2]. Seuls les messages destinés à l'instrument sont traités selon leur type. Ainsi dans le cas d'une variable, celle-ci est mise à jour et l'évènement qui lui est associé est activé. Si le message est celui d'un évènement, ce dernier est stocké dans une file d'attente de type FIFO pour être traité ultérieurement. Si le message reçu concerne une demande d'identification du mode courant alors celui-ci est exporté sur tous les liens. Enfin, un dernier cas traite les liens entre les variables externes de l'instrument. Pour plus de détails, voir [Tailland 00].

Globalement, le générateur d'instrument produit un code qui est très lié au type de matériel utilisé. Les processeurs utilisés étaient des Intel 80c196 et le code généré était donc un exécutable chargé en ROM. Le langage de programmation était le C++.

## II.F. Les avantages et les manques d'une telle approche

La création de CAPTool a été une première grande avancée pour la conception d'instruments intelligents basés sur le modèle présenté dans ce chapitre. En effet, de tels outils permettent de diminuer grandement les erreures de conceptions qui peuvent être commises.

L'avancée la plus significative a été la création d'un langage pivot : le langage CAP. Ce langage est un langage semi-formel qui simplifie grandement le travail du concepteur d'instruments car il reprend les termes du domaine de l'instrumentation intelligente. Ce langage CAP est ensuite directement utilisé pour la génération du code source de l'instrument.

La deuxième avancée a été la possibilité de vérifier dès la phase de conception des propriétés qui garantissent le bon prototypage des instruments. Ces vérifications sont directement réalisées au sein du compilateur CAP.

[2] Les liens de communication font référence au réseau qui propage l'information entre les différents instruments intelligents ou entre les instruments intelligents et un utilisateur.

Une telle approche a de nombreux avantages. Néanmoins, nous restons limités sur plusieurs aspects :

- Dans la pratique, la séparation entre le travail du programmeur et du concepteur d'instruments n'est pas très nette. En effet, la création du code des services internes se trouve directement à l'intérieur du fichier CAP créé par le concepteur d'instruments (voir section II.A). Les deux acteurs interviennent donc sur le même fichier, ce qui peut engendrer des erreurs.
- Les propriétés qui doivent être vérifiées sont directement implémentées dans l'outil CAPTool. Cela signifie que si l'on désire ajouter une nouvelle propriété ou en modifier une existante, on devra refaire le compilateur.
- Le langage CAP a été créé pour concevoir rapidement des instruments intelligents qui suivent le modèle présenté dans ce chapitre. Le problème est que si ce modèle a besoin d'évoluer pour des raisons diverses, le compilateur devra être profondément modifié.
- Nous avons vu à la fin de la section I.A.2.b que l'utilisateur ne peut pas toujours savoir si un service externe est terminé ou toujours en cours car aucun évènement externe de fin de service externe n'est délivré. En fait, il n'obtient cette information que si le service externe met à jour une variable externe à la fin de ce traitement. Cette information pourrait être importante dans certains cas ou un service externe doit impérativement être exécuté après un autre (comment savoir quand exécuter le deuxième service externe si l'on ne sait pas quand le premier se termine ?).

Ces limitations sont soit dues au modèle utilisé pour concevoir les instruments intelligents soit dues à la façon dont CAPTool a été implémenté.

En ce qui concerne les limitations du modèle, nous verrons comment le problème a été traité dans le chapitre 4. Pour remédier aux problèmes d'implémentations de CAPTool, nous nous sommes penchés sur un nouveau type de réalisation qui a vu le jour depuis peu : le développement centré architecture à base de composants. Dans le chapitre suivant, nous allons vous présenter brièvement cette nouvelle approche pour le développement de logiciels.

# *III. Conclusion*

Dans ce chapitre, le domaine de l'instrumentation intelligente a été présenté. Tout d'abord, le modèle sur lequel nos travaux sont basés a été présenté. Il est à la fois composé d'un modèle interne et d'un modèle externe. Utiliser un modèle en couche comme celui-là diminue les difficultés de conception en séparant les différentes tâches à réaliser. Le fait de séparer le travail du programmeur et celui du concepteur d'instruments permet d'adapter rapidement un produit aux besoins de l'utilisateur. En fait, le concepteur crée des algorithmes de base. Ces derniers sont ensuite assemblés par le développeur d'instrument.

Les outils qui forment CAPTool ont ensuite été présentés. Ces outils permettent de diminuer les erreures qui peuvent être commises grâce à des vérifications effectuées en phase de conception. De plus, grâce notemant à la création d'un langage spécifique, ils aident grandement à la conception des instruments intelligents.

Néanmoins, ce type d'outils reste limité sur plusieurs aspects et notamment sur leur faculté d'évolution et leur facilité de mise en œuvre. Dans le contexte actuel, ces deux facultés

sont primordiales pour une entreprise. En effet, le mot d'ordre actuel pour les entreprises est : réactivité. Il faut donc repenser ces outils dans l'optique de les améliorer.

Il est apparu que les limitations auxquelles nous étions confrontées étaient principalement dues à deux choses :

- aux limitations du modèle de conception des instruments intelligents,
- à la façon dont avait été implémentée CAPTool.

La suite du manuscrit est structurée de façon à améliorer à la fois le modèle des instruments intelligents mais aussi de CAPTool.

Dans le chapitre suivant, le domaine du développement centré architecture est présenté. Il sera démontré qu'il possède les qualités nécessaires pour développer des outils de types CAPTool, c'est-à-dire des outils destinés à des personnes qui ne sont pas forcément spécialistes en informatique mais qui pourtant ont des logiciels « sûrs » à développer. Ce chapitre contiendra donc un état de l'art sur les principaux langages de description d'architectures (ADL en anglais) qui existent afin de pouvoir faire ressortir celui qui conviendra le mieux à nos besoins. L'ADL choisi sera ensuite présenté plus en détail.

# Chapitre 3 : LA CONCEPTION CENTREE ARCHITECTURE

# Chapitre 3 : La conception centrée architecture

# La conception centrée architecture

La notion de conception centrée architecture est assez récente et constitue une suite logique dans l'évolution de l'ingénierie informatique. En effet, la réduction des coûts et des délais ont toujours été pour les industriels des objectifs majeurs et le domaine du logiciel n'échappe pas à cette règle. Ici, les industriels visent à permettre ces réductions au stade du développement et au stade de la maintenance, tout en sachant que les systèmes sont de plus en plus grands, complexes et distribués.

Depuis l'apparition de l'informatique, les techniques de développement ont évolué de façon considérable et ont entraîné avec elles les langages. Néanmoins, cette évolution a toujours pour but d'améliorer le travail d'équipe pour obtenir des logiciels de qualité. La meilleure manière d'arriver à ce résultat est de permettre aux acteurs :

- d'améliorer leur compréhension d'un système,
- de réutiliser le plus possible les acquis,
- d'éviter les erreurs ou tout du moins de les corriger le plus tôt possible.

La figure suivante (Figure 3-1) schématise l'évolution des concepts au cours des cinquante dernières années. Ce schéma [Garlan 03] traduit notamment l'évolution d'une programmation "linéaire" vers une programmation "modulaire".

Au début des années 90, différentes équipes de recherche [Shaw 90] [Perry&Wolf 92] [Garlan&Shaw 93] identifient un domaine émergeant : les architectures logicielles. Le terme « architecture » a été choisi pour se démarquer de la « conception ». Avec l'arrivée de ce terme, d'autres notions sont apparues comme celles d'abstraction, de style[3], etc.

Actuellement, de nombreux résultats sont attendus avec l'émergence du développement centré architecture comme discipline majeur. Au cours du développement d'un système sa compréhension diminue tandis que le coût dû aux erreurs augmente. Le but est de préserver la compréhension du système au cours du temps, de détecter et de corriger les erreurs le plus tôt possible. L'architecture doit être à la fois un cadre pour la satisfaction du cahier des charges, une base technique pour la conception et la base gestionnaire pour

[3] Nous donnerons une définition de ce terme dans la suite du manuscrit.

l'estimation des coûts et du processus de gestion. Elle doit être aussi une base performante pour la réutilisation et la base pour les analyses de dépendance et de cohérence.

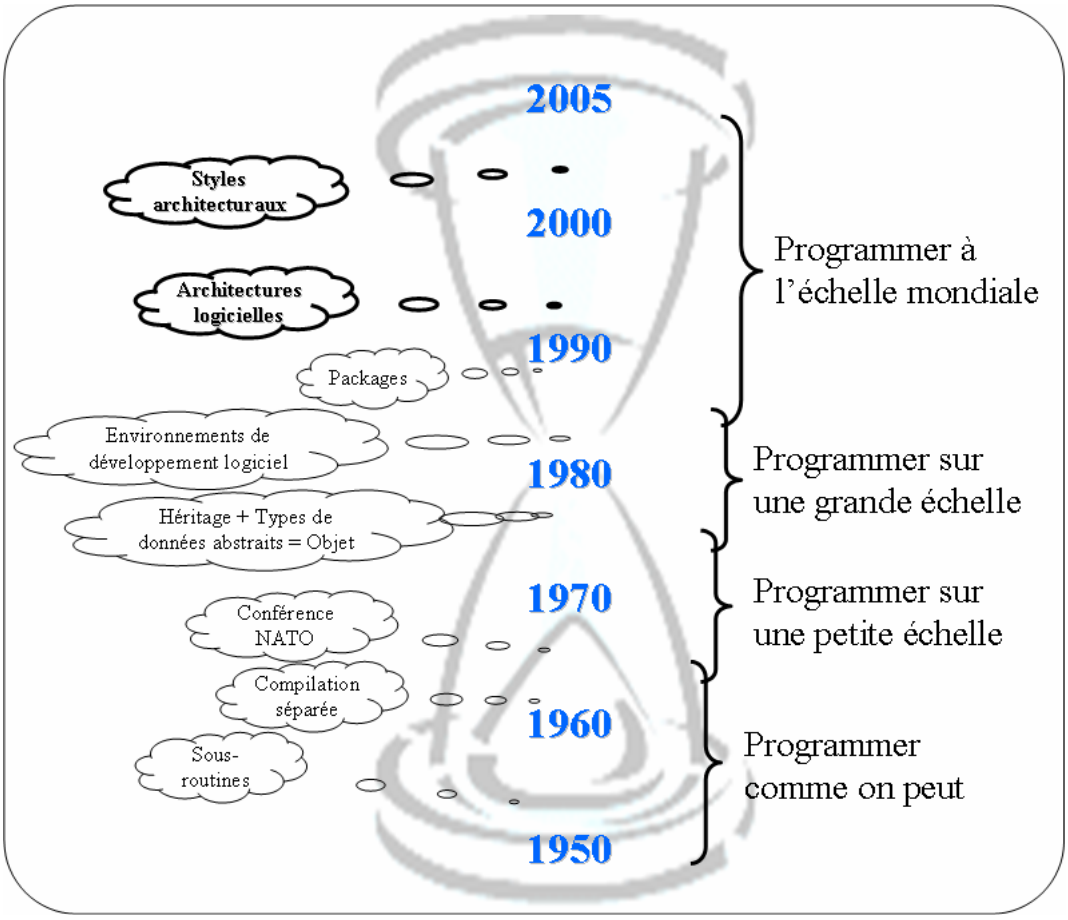



**Figure 3-1 - Evolution du développement de logiciels**

Dans ce chapitre, nous allons tout d'abord présenter la notion d'architecture logicielle (section I). Ensuite, nous nous attarderons sur la notion de style architectural qui sera une des notions de base de nos travaux (section II). Pour mettre en œuvre concrètement une approche centrée architecture pour la conception d'instruments intelligents, nous devons utiliser un Langage de Description d'Architecture (ADL en anglais). Afin de choisir le plus approprié, une analyse des besoins que nous avons sera faite (section III). A partir de là, un état de l'art des principaux ADL sera réalisé (section IV). Tout ceci va donc nous permettre de faire ressortir l'ADL le plus approprié à nos travaux.

# I. La notion d'architecture logicielle

Plusieurs définitions du terme « architectures logicielles » sont proposées dans la littérature [Boasson 95] [Bass et al. 99] [Garlan et al. 92] [Perry&Wolf 92]. Nous en proposons ici celle donnée par l'IEEE (IEEE Std 1471-2000) :

> *Une **architecture logicielle** est l'organisation fondamentale d'un système incarnée dans ses composants, leurs relations avec chacun des autres et avec l'environnement, et les principes guidant sa conception et son évolution*

Par conséquent, la description architecturale d'un système spécifie :

- sa **structure** : composants et interactions,
- son **comportement** : fonctionnalités et protocoles de communication, dynamisme, évolution,
- ses **propriétés globales** : propriétés fonctionnelles ou non fonctionnelles.

Les architectures de la plupart des systèmes logiciels ont longtemps été décrites de façon informelle (diagrammes où les composants logiciels sont des « boîtes » et les interactions des « lignes »). Cette description induit d'une part, des difficultés au niveau de leur interprétation et, d'autre part, des limitations [Abowd et al. 95]. Aussi, des langages de description d'architectures (LDA) ont été définis. Ils offrent de nombreux avantages par rapport aux approches semi-formel, comme par exemple la précision, la capacité à prouver des propriétés, et la possibilité d'analyser la structure architecturale. Ainsi spécifiées, les architectures logicielles jouent un rôle important dans au moins six aspects du développement logiciel [Garlan 00] :

1. **Compréhension** : les architectures logicielles rendent plus facile la compréhension du fonctionnement de systèmes complexes en les représentant à un haut niveau d'abstraction,
2. **Réutilisation** : les descriptions architecturales supportent la réutilisation à de multiples niveaux. Les travaux actuels, dans le domaine de la réutilisation, se concentrent généralement sur l'utilisation de bibliothèques de composants. La conception orientée architecture supporte, en plus, la réutilisation de composants complexes et des structures dans lesquelles ces composants peuvent être intégrés. Ceci est prouvé par de nombreux travaux existants dans les domaines des *Domain-Specific Software Architectures*, des architectures de référence, ou des patrons de conceptions [Mettala&Graham 92] [Buschmann et al. 96],
3. **Construction** : une description architecturale fournit un plan de développement en indiquant les composants principaux et les dépendances entre eux,
4. **Evolution** : l'architecture d'un système peut décrire la manière dont ce système est censé évoluer. La définition explicite des limites d'évolution d'un système permet de faciliter sa maintenance et d'estimer plus précisément les coûts des modifications. De plus, les descriptions architecturales distinguent les aspects fonctionnels des composants de la façon dont ces composants interagissent entre eux. Cette séparation permet de modifier facilement les mécanismes de connexion, ce qui favorise l'évolution en termes de performance, d'interopérabilité et de réutilisation,
5. **Analyse** : les descriptions architecturales fournissent des moyens d'analyse, tels que la vérification de la cohérence d'un système [Allen&Garlan 94] [Luckham et al. 95], la vérification de la conformité aux contraintes imposées par un style architectural [Abowd et al. 93], la vérification de la conformité à des attributs qualité [Clements et al. 95], l'analyse de dépendance [Stafford et al. 93], ainsi que des analyses spécifiques au style à partir des architectures construites [Coglianese&Szymanski 93] [Magee et al. 95] [Garlan et al. 94],
6. **Gestion** : l'expérience a montré que la définition précise d'une architecture logicielle est un facteur clé dans la réussite d'un processus de développement logiciel. L'évaluation d'une architecture mène à une meilleure compréhension des besoins, des stratégies d'implémentation, et des risques potentiels [Boehm et al. 94].

La figure ci-dessous (Figure 3-2), représente un processus de développement centré architecture ainsi que les acteurs : l'architecte d'application, l'ingénieur, ainsi qu'éventuellement l'analyste.

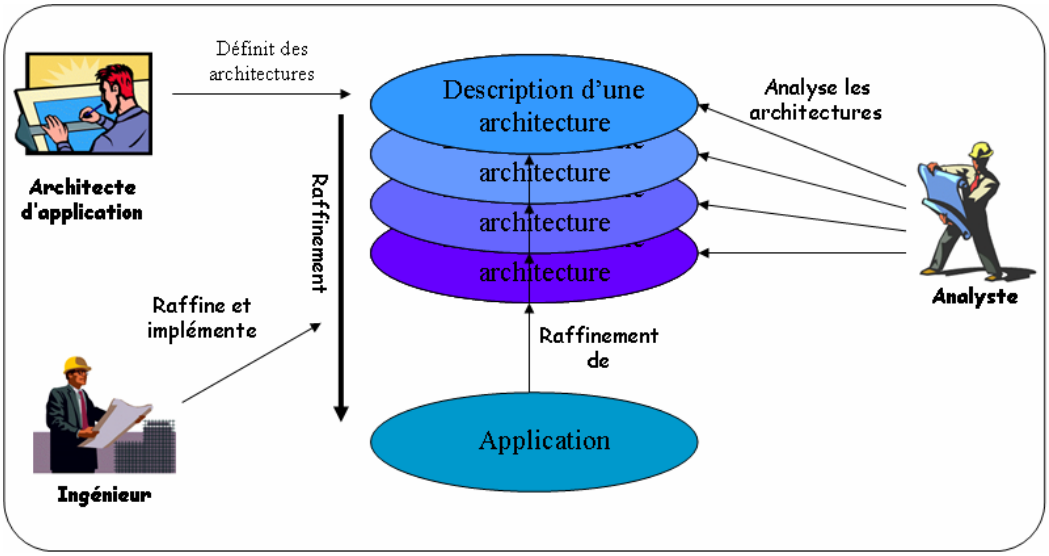



**Figure 3-2 : Processus de développement centré architecture**

L'architecte a pour rôle de définir l'architecture qui servira de base au développement de l'application. L'ingénieur raffine cette architecture de façon à s'approcher, petit à petit, de l'application finale. Pour cela, il implémente les composants ainsi que leurs interactions (les connecteurs) en respectant la structure et les propriétés définies par l'architecture de départ. A chaque étape du raffinement, l'analyste doit être en mesure de vérifier que l'architecture raffinée est conforme à l'architecture du niveau d'abstraction supérieur. Ce processus permet de garantir que l'application obtenue respecte les propriétés fonctionnelles, structurelles et comportementales définies par l'architecte en accord avec le client et les utilisateurs.

Le nombre de raffinements successif peut être très différent d'une application à l'autre. En effet, il dépend de la précision nécessaire de l'architecture pour pouvoir passer au code de l'application. Cette étape de raffinement est nécessaire pour les grosses applications qui ne peuvent pas être construites en une seule passe, mais on peut très facilement s'en passer pour des applications de plus petites tailles. Dans ce cas, l'application pourrait directement être implémentée à partir de la première architecture si celle-ci est assez précise.

Avec le développement centré architecture est apparu une nouvelle notion qui est celle du style architectural. Nous allons la présenter brièvement dans la partie suivante.

## II. Notion de style architectural

La définition d'un style architectural selon [Abowd et al. 93] est la suivante :

> *Un **style architectural** permet de caractériser une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques.*

De ce fait, un style architectural précise les propriétés et les contraintes qui fixent les règles et les limites de construction de l'architecture.

L'objectif des styles architecturaux est de simplifier la conception des logiciels et la réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système [Monroe et al. 97]. Un style architectural est moins contraignant et moins complet qu'une architecture spécifique. Il spécifie uniquement les contraintes les plus importantes, au niveau par exemple de la structure, du comportement, de l'utilisation des ressources des composants et des connecteurs [Abd-Allah 96].

En d'autres termes, un style architectural fournit [Garlan 95] :

- un vocabulaire pour concevoir les types spécifiques de composants utilisables,
- des règles de conception (des contraintes) pour spécifier les compositions d'éléments autorisées,
- une interprétation sémantique pour définir la signification des compositions d'éléments contraintes par les règles de conception,
- les analyses pouvant être appliquées sur les systèmes construits à partir de ce style.

D'une façon générale, les styles architecturaux permettent à un développeur de réutiliser l'expérience concentrée de tous les concepteurs qui ont précédemment fait face à des problèmes similaires [Klein&Kazman 99]. En outre, l'utilisation des styles architecturaux comporte des intérêts précis :

- elle favorise la réutilisation dès la conception du système [Monroe&Garlan 96],
- elle autorise une réutilisation significative de code,
- elle favorise la normalisation des familles d'architecture, ce qui facilite la compréhension de l'organisation d'un système,
- elle autorise l'utilisation d'analyses spécifiques au style concerné [Ciancarini&Mascolo 96].

Ces analyses spécifiques ou contraintes fixent des contraintes de construction au style créé. Ces contraintes sont de trois types :

- topologiques,
- comportementales,
- d'attributs.

Les contraintes ***topologiques*** précisent le type des éléments de construction, le nombre d'occurrences possibles au sein de l'architecture et les règles de configuration contraignant leurs interactions (par exemple, deux composants sont toujours liés par l'intermédiaire d'un connecteur).

Les contraintes ***comportementales*** définissent d'une part, le comportement des éléments de l'architecture et, d'autre part, le comportement globale de l'architecture (par exemple, il n'y a pas d'interblocage entre les différents composants).

Les contraintes *d'**attributs*** concernent les aspects non structurels et non fonctionnels d'une architecture. Les attributs apportent des informations complémentaires sur les éléments architecturaux en contraignant leur type, leur nom et leur plage de valeurs (par exemple, la durée de traitement réalisée par un composant ne doit pas dépasser 0.5 seconde). Elles peuvent également spécifier la manière dont les valeurs des attributs sont liées avec d'autres aspects architecturaux (topologie et comportement).

La figure suivante (Figure 3-3) présente un processus de développement général qui inclut la définition formelle et l'exploitation de styles architecturaux. Ce schéma reprend et étend le processus de développement architectural présenté auparavant.

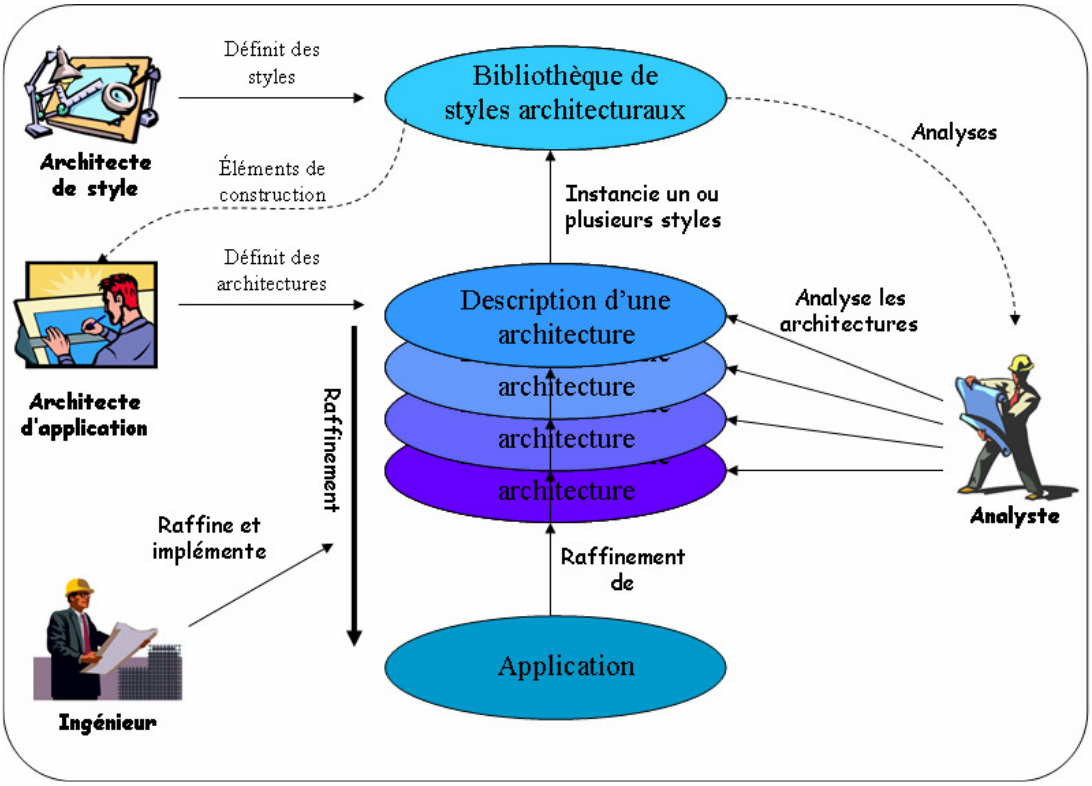



**Figure 3-3 : Processus de développement architectural orienté style**

La première étape de ce processus est la formalisation du style architectural ou des styles architecturaux. Au fur et à mesure des définitions de styles, une bibliothèque peut être définie. L'architecte d'application instancie alors le ou les styles qui correspondent à ses besoins spécifiques et les utilise comme éléments de construction pour définir une architecture de base. La suite du développement est identique au processus précédemment étudié : l'ingénieur raffine petit à petit l'architecture de base en architectures de plus en plus concrètes jusqu'à l'obtention de l'application finale. Il est toutefois à noter que l'utilisation d'un tel processus de développement permet à l'analyste d'appliquer aux différentes architectures des analyses définies par les styles utilisés. Les analyses disponibles ne sont donc plus uniquement des analyses génériques, applicables à toutes les architectures. Ces analyses peuvent aussi être des analyses spécifiques à un domaine d'application particulier, applicables uniquement aux architectures définies à l'aide des styles correspondant à ce domaine.

Les styles architecturaux ont longtemps été définis et utilisés comme des guides de conception informels, n'ayant pas de langages dédiés à leur exploitation. Les travaux menés, depuis lors, ont souligné l'intérêt de formaliser les styles [Abowd et al. 95] et de nombreux langages dédiés à cette tâche ont été développés.

La section suivante (section III) va répertorier les différents besoins que nous avons afin de pouvoir concevoir des instruments intelligents en utilisant une méthode de conception centrée architecture.

# III. Les critères de choix du langage à utiliser

Pour pouvoir répondre pleinement à notre problématique, l'ADL qui correspondra le mieux à nos besoins devra être choisi. Pour ce faire, nous devons, tout d'abord, clairement exprimer ces derniers. D'après l'étude du domaine qui a été réalisé dans le chapitre 2, on peut faire ressortir quatre points incontournables pour la réalisation de nos travaux :

- le pouvoir d'expression du langage,
- la notion de style architectural,
- la vérification de propriété,
- les environnements de développement.

Le pouvoir d'expression d'un langage est une des choses primordiales à étudier pour notre cas. En effet, nous avons pu voir dans le chapitre précédent que le modèle que nous avons choisi pour concevoir les instruments intelligents était très hiérarchisé. Nous avons besoin de pouvoir représenter différentes structures tel que le graphe des services internes, les services externes ou encore les modes. La possibilité de pouvoir structurer une application grâce à un ADL est donc une chose importante pour nous. Le deuxième critère primordial pour nous au niveau du pouvoir d'expression d'un ADL concernera sa capacité à exprimer le comportement d'une application. En effet, le logiciel qui doit gérer un instrument intelligent est principalement là pour gérer son comportement. Par exemple, nous avons vu qu'un service externe pouvait être vu comme un enchaînement de services internes. Un langage qui ne nous permettrait pas de représenter ceci nous serait totalement inutile.

Le fait de pouvoir représenter un instrument intelligent d'un point de vue structurel et comportemental est une chose importante. Mais le fait de pouvoir vérifier que cette représentation est correcte par rapport au modèle choisi est aussi important. Nous avons vus dans le chapitre précédent que de nombreuses règles structurelles devaient être respectées. Il faut donc que le langage nous offre la possibilité de spécifier l'ensemble de ces règles de construction. Néanmoins, cette seule possibilité n'est pas suffisante. En effet, nous verrons aussi que des propriétés comportementales et d'attributs seront, dans certains cas, utiles.

La grande réussite de CAPTool avait été de pouvoir fournir au concepteur un langage qui utilisait les différents termes du domaine de l'instrumentation intelligente. Il faut conserver cette avancée et si possible l'améliorer. Nous avons vu précédemment que cette fonctionnalité était offerte, dans une approche centrée architecture, par l'intermédiaire de la notion de style architectural. Il s'agit donc d'une fonctionnalité qui nous est essentielle. Ce n'est pas la seule raison pour laquelle cette notion de style doit être utilisée. En effet, nous avons vu que des règles de conception ou des contraintes pour spécifier les compositions d'éléments qui sont permises grâce à cette notion. Par conséquence, grâce à la notion de style, il va être possible, par exemple de spécifier les différentes contraintes de constructions qui ont été évoquées pour le graphe des services internes.

Enfin, afin de pouvoir appliquer nos travaux sur des exemples concrets, il faut que le langage qui va être choisi possède des outils de développement qui nous permettent d'utiliser les différentes fonctionnalités qu'il est censé avoir. En effet, si le langage nous permet, par exemple, de faire de la vérification de propriété comportementale, il faut que des outils aient été développés afin de pouvoir faire des vérifications sur des cas concrets.

Nous venons de lister les notions les plus importantes que doit reprendre l'ADL qui sera choisi pour réaliser nos travaux. Dans la section suivante (section IV), un état de l'art des principaux langages de description d'architecture va être fait.

# IV. Les langages de description d'architectures (ADL)

Pour pouvoir utiliser un développement centré architecture pour la conception de logiciels, il ne suffit pas d'avoir une méthode mais il faut aussi disposer d'outils. Parmi eux, les langages de descriptions d'architectures sont parmi les plus important car ils permettent de décrire formellement et sans ambiguïté les architectures logicielles.

Afin de choisir le plus adapté à la conception d'instruments intelligents, il est nécessaire de classer et de comparer les principaux ADLs existants. La classification présentée dans cette section est basée sur les travaux de [Leymonerie et al. 02].

Nous allons vous présenter une classification de différents ADL par rapport à trois critères :

- le contexte dans lequel il a été développé ;
- les concepts d'architecture et de style architectural qu'il exploite ;
- la façon dont il gère les mécanismes de styles tels que l'instanciation, l'héritage ou le raffinement.

Par rapport à ces critères, il faut noter que chaque ADL supporte au moins un style architectural qui, en fait, représente sa sémantique et ses mécanismes [Leymonerie et al. 02]. Ce style est plus ou moins générique selon le domaine de l'ADL, par exemple : META-H est un ADL spécifique aux architectures des systèmes multiprocesseurs temps réel pour l'aviation [Binns et al. 96] ; ACME [Garlan et al. 97] et π-SPACE [Chaudet&Oquendo 01] ne sont pas spécifiques à un domaine, leur style propre est plus générique. Afin d'augmenter le niveau de réutilisation et de s'adapter à n'importe quel domaine, quelques ADLs permettent aux utilisateurs de formaliser leurs propres styles.

Les ADLs permettent de spécifier d'autres informations au sein des formalisations de styles afin de pouvoir les exploiter. Ces informations peuvent être : des analyses d'architecture, des règles de traduction pour la génération automatique de code, une syntaxe spécifique au style, une visualisation des éléments architecturaux, des outils spécifiques ou de la documentation.

Les ADLs que nous allons étudier sont donc les suivants :

- **ACME** [Garlan et al. 97] : c'est un langage générique « d'interchange » qui permet d'utiliser en même temps plusieurs ADL. Il permet de décrire les structures architecturales et d'ajouter une sémantique à ces structures. Il décrit des composants, des connecteurs, des ports, des rôles et des configurations. ACME supporte aussi la définition des styles et permet d'assurer le respect des contraintes de conception à l'aide de ses outils. ARMANI [Monroe 98] est une extension d'ACME conçu pour modéliser les contraintes architecturales. Dynamic ACME [Wile 01] est une extension permettant de décrire la dynamicité des architectures ;
- **AESOP** [Garlan et al.. 94] : il permet de construire des architectures avec des environnements spécifiques générés à partir de description de style ;
- **AML** (Architectural Meta-Language) [Wile 99] : il s'agit d'un métalangage pour la spécification de la sémantique des ADLs. Il décrit une architecture comme un

ensemble d'éléments (*element*) liés par des relations (*relationship*) soigneusement décrites et contraintes ;

- **ArchWare ADL / ASL** [Oquendo 03a][Cimpan et al. 03] : ArchWare ADL fournit la structure de base et les constructions comportementales pour décrire les architectures logicielles dynamiques. C'est un langage formel de spécification conçu pour être exécutable et pour supporter l'analyse et le raffinement automatique des architectures. ArchWare ADL spécialise le π-calcul [Milner et al. 92] pour permettre la description de comportements dynamiques. L'extension ASL, construite à partir de ArchWare ADL, constitue le langage de création des styles ;
- **DARWIN** [Magee et al. 95] : il supporte l'analyse de systèmes de transmission de messages distribués ;
- **META-H** [Vestal 92] : il fournit un guide pour décrire, analyser, et implémenter les architectures de systèmes de contrôle temps réel embarqués dans le domaine de l'aviation ;
- **π-SPACE** [Chaudet&Oquendo 01] est un ADL pour les architectures dynamiques permettant la description de comportements architecturaux. σπ-SPACE [Leymonerie et al. 01] est son extension pour la description des styles architecturaux. π-SPACE et σπ-SPACE utilisent le π-calcul pour la description de comportements dynamiques ;
- **RAPIDE** [Luckham et al. 95][Rapide 97] : ce langage décrit les interfaces des composants et les communications entre celles-ci ainsi que les modèles événementiels. Il permet de simuler les architectures et propose des outils pour analyser les résultats de ses simulations. Il ne présente pas de mécanisme propre à la formalisation des styles ;
- **SADL** [Moriconi et al. 95][Moriconi&Riemenschneider 97] : il permet la description d'architectures spécifiques et d'architectures paramétrées. Il fournit en outre une base formelle pour le raffinement architectural ;
- **UNICON** [Shaw et al. 95] : c'est un ADL qui permet la construction de systèmes à partir de la description de leur architecture. Il possède un compilateur de haut niveau qui supporte l'utilisation de types de composants et de connecteurs hétérogènes. UNICON-2 [DeLine 96] est une extension pour la prise en charge des styles architecturaux. Il permet la génération de code pour une grande variété de styles ;
- **WRIGHT** [Abowd et al. 93][Allen&Garlan 97] : il s'agit d'un modèle formel pour les architectures logicielles, il est largement utilisé pour l'analyse des protocoles d'interaction entre les composants architecturaux. Cet ADL permet la définition des comportements au moyen d'une algèbre formelle, le CSP [Hoare 85]. Le CSP permet de décrire des comportements en terme de patrons d'évènements. En comparaison au π-calcul utilisé dans σπ-SPACE et ArchWare ADL, il ne permet pas la description de comportement dynamique. Les extensions de WRIGHT [Allen 97][Allen et al. 98] permettent d'une part, la définition des styles architecturaux d'une manière similaire à ARMANI, et d'autre part, la description (limitée) du dynamisme des architectures.

Bien entendu, cette liste d'ADLs n'est pas exhaustive. En effet, il existe d'autres ADLs mais leur contexte est plus éloigné de celui de cette thèse et nous avons donc prix le parti de ne pas les étudier ici.

Comme il a été dit précédemment, la section suivante va permettre de comparer les différents ADL par rapport à leur façon de formaliser les architectures.

## IV.A. La formalisation des architectures

Les deux tableaux ci-dessous (Tableau 3-1 et Tableau 3-2) présentent les différents ADLs qui ont été étudiés pour ces travaux. Nous les comparons sur leur capacité à définir une architecture. Pour ce faire, différents critères sont étudiés :

- capacité à représenter la structure d'une application,
- capacité à représenter le comportement d'une application,
- des éléments non fonctionnels (attributs) peuvent-ils être déclarés et utilisés,
- la composition d'éléments architecturaux est-elle possible.

D'autres critères moins importants dans notre cas sont aussi comparés. Ce sont les notions de :

- dynamicité : possibilité de modifier la structure et/ou le comportement d'une application en cours d'exécution,
- mobilité : possibilité de faire transiter des éléments architecturaux d'un endroit à un autre dans l'application.

Dans les tableaux ci-dessous, les cases hachurées signifient que la notion n'est pas supportée par le langage. Les cases blanches signifient, au contraire, que la notion est supportée. Une case blanche, pour la même notion, pour deux ADLs différents, ne signifie pas forcement qu'elle est traitée aussi bien dans les deux ADLs.

| ADLs / Caractériqtiques | σπ-SPACE | SADL | RAPIDE | UNICON-2 | WRIGHT |
|---|---|---|---|---|---|
| **Représentation de la structure d'une application** | | | | | |
| **Représentation du comportement d'une application** | *basé sur le $\pi$-calcul* | | *contraintes sur les composants* | | *basé sur CSP* |
| **Déclaration d'attributs** | | | | | |
| **Composition d'éléments architecturaux** | | | | | |
| **Dynamicité** | *tous les éléments sont instanciables dynamiquement* | | *permet la définition de règles décrivant des connections dynamiques* | | *Dynamique WRIGHT, permet de simuler la dynamicité par l'ajout d'évènements de contrôle dans les composants* |
| **Mobilité** | | | | | |

**Tableau 3-1 : Caractéristique des ADLs concernant la définition d'architectures**

| ADLs / Caractériqtiques | ACME / ARMANI | AESOP | AML | META-H | ArchWare ADL |
|---|---|---|---|---|---|
| **Représentation de la structure d'une application** | | | | | |
| **Représentation du comportement d'une application** | | | *Quantificateurs temporels : always, sometimes* | | *basé sur le $\pi$-calcul typé d'ordre supérieur* |
| **Déclaration d'attributs** | | | | | |
| **Composition d'éléments architecturaux** | | | | | |
| **Dynamicité** | | | | | |
| **Mobilité** | | | | | |

**Tableau 3-2 : Caractéristique des ADLs concernant la définition d'architectures (suite)**

ACME fournit une ontologie architecturale consistant en sept éléments de conception. Tout d'abord, il y a les briques de construction : les composants (***Component***) et les connecteurs (***Connector***). Ceux-ci sont interfacés par des ***Ports*** dans le cas d'un composant et des ***Rôles*** dans le cas d'un connecteur. Les composants et les connecteurs sont interconnectés par des attachements (***Attachments***) au sein d'un système (***System***). Ensuite, un mécanisme appelé ***Representation*** permet la décomposition hiérarchique d'un élément architectural en un sous-système. Associé à ce mécanisme un autre appelé ***Rep-map*** permet de faire correspondre la représentation interne d'un système avec l'interface externe d'un composant ou d'un connecteur. Ces sept classes d'éléments de conception sont suffisantes pour définir la structure d'une architecture comme un graphe hiérarchique de composants et de connecteurs. Par contre on peut remarquer qu'ACME ne pas capable de représenter un comportement.

AESOP est assez proche d'ACME au niveau des concepts et il reprend la même terminologie. Il ne gère pas non plus le comportement.

Dans AML, les architectures logicielles sont représentées comme un ensemble d'éléments (***element***) liés par des relations topologiques (***relationship***) et des contraintes. Le mot clé ***element*** permet en fait de déclarer des instances. La structure est définie par des assertions via le mot clé ***assume.*** AML permet une représentation très sommaire du comportement. Cela se limite au fait qu'il permet de définir des contraintes à travers des assertions temporelles utilisant les quantificateurs ***sometimes*** et ***always***.

META-H intègre un langage de description d'architectures un peu particulier car il permet de représenter aussi bien une architecture logicielle que matérielle. Une architecture logicielle se construit par instanciation de composants primitifs (appelés ***processus***) ou composés (appelés ***macros***) et par la définition des connexions entre eux. Un composant est défini par une interface contenant la déclaration des points d'accès typés (moniteurs partageables, ports événements). On remarque que cet ADL ne permet pas de représenter le comportement d'une architecture.

Dans ArchWare ADL, les éléments architecturaux sont définis en terme de comportements (***behaviour***). Un comportement est défini par un ensemble d'actions ordonnancées qui spécifie à la fois le traitement interne de l'élément architectural (actions internes) et les interactions avec son environnement (actions de communication). Un élément architectural communique avec les autres par une interface représentée par un ensemble de ***connexions***.

Les éléments architecturaux communiquent en transitant des données par ces connexions. Pour interagir, les éléments architecturaux sont mis en relation par un mécanisme de composition et un mécanisme de liaison, appelé ***unification*** (c'est une substitution au sens du π-calcul). Lorsque plusieurs éléments sont composés, ils peuvent interagir lorsque leurs connexions sont liées. Afin de réutiliser des définitions de comportement, ArchWare ADL fournit un mécanisme de réutilisation appelé ***abstraction***. L'abstraction permet d'encapsuler des définitions paramétrables de comportement. On obtient un comportement d'une abstraction par l'*application* de cette dernière, en fournissant éventuellement une liste de paramètres. Le comportement est fondé sur le π-calcul typé d'ordre supérieur étendu avec un sous-langage d'expression.

Dans σπ-SPACE, les architectures sont représentées par des configurations de ***composites***, de ***composants***, et de ***connecteurs***. Un composite est un élément composé et définissant une configuration. Les composants et les connecteurs sont composés de ports qui définissent l'interface de communication. Ils sont eux mêmes définis comme un ensemble de canaux (points d'interaction élémentaires), d'un comportement définissant les interactions de l'élément. Un comportement est défini sur les bases d'une description de processus en π-calcul, d'opérations (pour les composants) qui définissent les traitements internes.

Une ***Architecture*** décrite en SADL est composée de plusieurs entités. Tout d'abord, le ***component*** contient la partie computationnelle et est interfacé grâce à des ***ports***. Ensuite, la jonction entre ces différents components est gérée par des ***connectors***. Le connector est un objet typé. Enfin, une partie ***configuration*** permet de contraindre les liaisons entre les components et les connectors d'une architecture. Ceci est concrètement réalisé par des ***connections*** et des ***constraints***. SADL ne permet pas de décrire le comportement d'une architecture.

RAPIDE défini la brique de base comme *composant*. L'interface d'un composant possède des *constituants*. Ceux ci sont créés à partir de constructeurs permettant de définir des modèles de communication entre les composants. Le constructeur ***action*** spécifie une communication asynchrone et le constructeur ***functions*** une communication synchrone. Chacun de ces constructeurs peut être utilisé avec les mots clés ***extern*** ou ***public*** qui permettent de définir respectivement un port de sortie ou un port d'entrée. Les composants sont reliés entre eux par les *constituants* ***extern*** et ***public*** de leurs interfaces. De plus, une des particularités de RAPIDE est de permettre de décrire des contraintes sur le comportement d'un composant.

Une architecture UNICON2 est un composant composé d'autres composants et connecteurs interconnectés. Les constituants des composants et des connecteurs sont

- l'***interface***, pour le composant, ou le protocole (***protocol***), pour le connecteur, qui spécifie:
  - le type de l'élément,
  - ses points de communication (***player***(composant), ***role***(connecteur)),
  - ses attributs (***property***)
- l'***implémentation*** qui spécifie une configuration de composants et de connecteurs ou un pointer vers un document source. Ainsi on différencie respectivement deux types de composants, les composants composites et les composants primitifs.

Cet ADL n'intègre aucune notion de comportement.

Les concepts de WRIGHT sont extrêmement proches de ceux d'ACME. Il y a les composants, les connecteurs, les ports et les rôles. Un ensemble d'éléments interconnectés est appelé une ***configuration***. Cet ADL intègre la notion de comportement en se basant sur CSP.

Après ce bref tour d'horizon des ADL étudiés sur leur façon de formaliser les architectures, on se rend compte que de nombreuses disparités apparaissent. Celle qui nous intéresse le plus est celle concernant leur capacité à représenter le comportement d'une architecture. Parmi ces langages, certains sont seulement concernés par des aspects structuraux comme DARWIN ou ACME. D'autres ADLs sont fondés sur des algèbres de processus et supportent les aspects comportementaux: σπ-SPACE et ArchWare ADL sont fondés sur le π-calcul [Milner 89] et WRIGHT est fondé sur CSP. D'après la liste des besoins qui a été présentée précédemment, il va de soit que les ADLs qui ne prennent pas en compte le comportement ne conviendront pas.

Dans la section suivante (section IV.B), nous allons étudier les ADLs afin de voir comment la notion de style est gérée au sein de chacun d'entre eux. Bien entendu, étant donné que seuls les ADLs qui permettent une représentation du comportement seront étudiés puisque cette fonctionnalité nous est essentielle. Il reste donc :

- ArchWare ADL,
- σπ-SPACE,
- AML,
- Rapide,
- Wright.

## IV.B. Formalisation des styles architecturaux

Selon [Garlan 01], un style architectural définit typiquement un vocabulaire pour des types d'éléments de conception et des règles pour la composition des instances de types. Ils définissent un ensemble de règles de configuration, ou des contraintes topologiques, qui déterminent les compositions autorisées de ces éléments, et donnent l'interprétation sémantique de ces compositions. Un modèle définit également les analyses qui peuvent être exécutées sur des systèmes établis selon le style [Garlan 95].

Néanmoins, cette définition n'est pas adoptée pour tous les ADLs qui ont été étudiés. Les styles peuvent être définis comme :

- des **types génériques**,
- un **ensemble de classes** dans une approche orienté objet,
- un **constructeur**,
- un **ensemble de types**.

Le tableau ci-dessous (Tableau 3-3) présente les mécanismes de définition des styles propres à chacun des ADLs restant, ainsi que les contraintes pouvant être exprimées. Il est à noter que les ADLs Rapide et Wright ne figurent plus ici. Ils ont été enlevés non pas parce qu'ils ne permettent pas la définition d'un style mais car ils ne proposent pas de mécanismes particuliers permettant à un utilisateur de définir ses propres styles. Dans notre cas, on doit créer des styles qui seront propres au domaine de l'instrumentation intelligente.

| ADLs / Caractéristiques | AML | σπ-SPACE | ArchWare ADL |
|---|---|---|---|
| **Style** | *types génériques* | *types génériques* | *types génériques* |
| **Contraintes topologiques** | *cardinalités sur les liaisons entre les* | *cardinalités sur les liaisons entre les éléments* | *propriétés structurelles* |
| **Contraintes comportementales** | *quantifieurs temporels* | *propriétés comportementales* | *propriétés comportementales* |
| **Contraintes sur les attributs** | | | *propriétés sur des valeurs* |
| **Héritage** | *ajout de relations respectant les relations existantes* | *ajout de contraintes* | *un style peut-être étendu pour obtenir un sous-style (ajout de contraintes, d'éléments architecturaux, de constructeurs)* |
| **Dynamicité** | | *tous les éléments sont instanciables dynamiquement* | *tous les éléments sont instanciables dynamiquement, le nombre d'instanciations possibles pouvant être limité en utilisant des contraintes* |
| **Mobilité** | | | *les connections, ports, composants et connecteurs sont mobiles* |
| **Génération de code** | | *génération automatique de code pour l'architecture* | *mécanismes de raffinement, génération de code, hypercode* |
| **outils d'exploitation** | | | *environnement de développement, analyses, visualisation, …* |

**Tableau 3-3 : Caractéristiques des ADLs concernant la définition de styles**

AML, σπ-SPACE et ArchWare ADL proposent de définir un style comme un type générique. Par exemple, en AML, un élément architectural suivant un ***kind*** doit respecter les relations que ce dernier établit. En σπ-SPACE, on peut avoir des styles pour les composants, les connecteurs, les composites et les ports ; ils définissent des contraintes topologiques et comportementales régissant la structure interne de leurs instances. Les mécanismes de définition de contraintes sont différents selon leur type.

Il est intéressant de remarquer qu'un seul ADL est capable d'exprimer les trois types de contraintes possibles (topologiques, comportementales, d'attributs). Il s'agit d'ArchWare ADL (voir Tableau 3-3).

Cet ADL est le mieux placé pour nous permettre de mettre en place une conception de logiciel centrée architecture pour la conception d'instruments intelligents, encore faut-il qu'il possède des outils performants qui permettent de mettre en œuvre les différents concepts du langage. C'est ce que nous allons voir dans la section suivante (section IV.C).

## IV.C. Outils et environnements architecturaux

Etant donné que la conception d'architectures et de styles architecturaux est devenue une discipline importante de l'ingénierie logicielle, il a été nécessaire de développer des outils et des environnements permettant la description, l'analyse et l'exploitation d'architectures. On retrouve plusieurs types d'outils architecturaux. Les principaux sont des :

- éditeurs et visualisateurs graphiques et textuels ;
- outils d'analyse statique ;
- outils d'analyse dynamique, utilisés pour l'exécution d'architectures (simulation et supervision) ;
- outils d'implémentation (interfaces ou générateurs de code) ;
- outils permettant l'évolution d'architectures ;

- outils de génération de documentation ;
- outils de « traduction » de descriptions architecturales entre différents ADLs.

La plupart des langages décrits dans les sections précédentes possèdent des outils ou des environnements de développement permettant de définir et d'exploiter des descriptions architecturales. Ceci est le cas d'ArchWare ADL.

Le langage ArchWare ADL possède un environnement de développement complet incluant notamment :

- un compilateur et un moteur d'exécution d'architectures ;
- un éditeur graphique (basé sur UML) et un éditeur textuel permettant la définition et la manipulation d'architectures ;
- un animateur graphique ;
- des outils de vérification de propriétés (statiques et dynamiques) ;
- un raffineur et un générateur de code ;
- un outil (customizer) permettant de vérifier que les architectures définies à partir de styles architecturaux spécifiques, via l'environnement ArchWare, respectent bien les contraintes spécifiées.

Tous ces outils permettent d'utiliser les différents concepts d'ArchWare ADL de façon concrète.

## IV.D. Conclusion

Les différentes comparaisons qui ont été faites permettent de conclure que ArchWare ADL est le langage le plus adapté pour la conception d'instruments intelligents.

Tout d'abord, celui-ci possède les mécanismes appropriés pour décrire efficacement les styles architecturaux. Ces derniers permettent non seulement une bonne réutilisation du code mais, chose importante dans notre cas, une grande souplesse dans la création de nouveaux langages. En ce qui concerne le pouvoir d'expression du langage, ArchWare ADL est un des plus puissant puisqu'en plus du fait de pouvoir représenter la structure d'une architecture, il est aussi capable d'en décrire le comportement. Ceci est dû au fait qu'ArchWare ADL est basé sur le $\pi$-calcul. Concernant la gestion des contraintes, ArchWare ADL est le seul qui, en plus d'être efficace dans la spécification des contraintes topologiques et d'attributs, permet la spécification de contraintes comportementales. Comme il a été précisé dans le paragraphe précédent, ArchWare ADL possède en plus un environnement de développement complet. Enfin, un élément non négligeable qui joue en faveur d'ArchWare ADL est que le laboratoire LISTIC possède de nombreuses personnes ayant elles même travaillées à l'élaboration des outils supportant cet ADL. Ceci permet donc d'avoir toutes les informations nécessaires rapidement.

Dans la section suivante (section V), nous allons présenter plus en détail la famille de langages ainsi que certains outils qu'ArchWare fournis pour le développement centré architecture.

# *V. Le projet ArchWare*

Les sections précédentes ont fait ressortir que nos travaux vont devoir s'appuyer sur les langages et les outils du projet européen IST ArchWare *–Architecting Evolvable Software–*

référencé sous le numéro IST-2001-32360 [ArchWare 02]. L'objectif du projet ArchWare est de répondre à une demande toujours croissante des systèmes logiciels qui peuvent évoluer au cours de leur vie. Pour atteindre cet objectif, un ensemble intégré de langages et d'outils orientés architecture ont été créés durant ce projet. ArchWare est composé [Oquendo et al. 04] :

- de langages formels (avec des notations textuelles et graphiques) pour modéliser des architectures dynamiques (incluant l'expression de leur structure, de leur comportement, et de leurs propriétés sémantiques d'un point de vue composant-connecteur), ainsi que pour exprimer leurs analyses et raffinements ;
- d'un environnement de développement personnalisable pour l'ingénierie logicielle centrée architecture, incluant des processus et outils servant de support à la description d'architectures, à leur analyse, à leur raffinement, à la génération de code, ainsi qu'à leur évolution.

La figure ci-dessous (Figure 3-4), donne une vision de l'ensemble des outils fournis dans le projet.

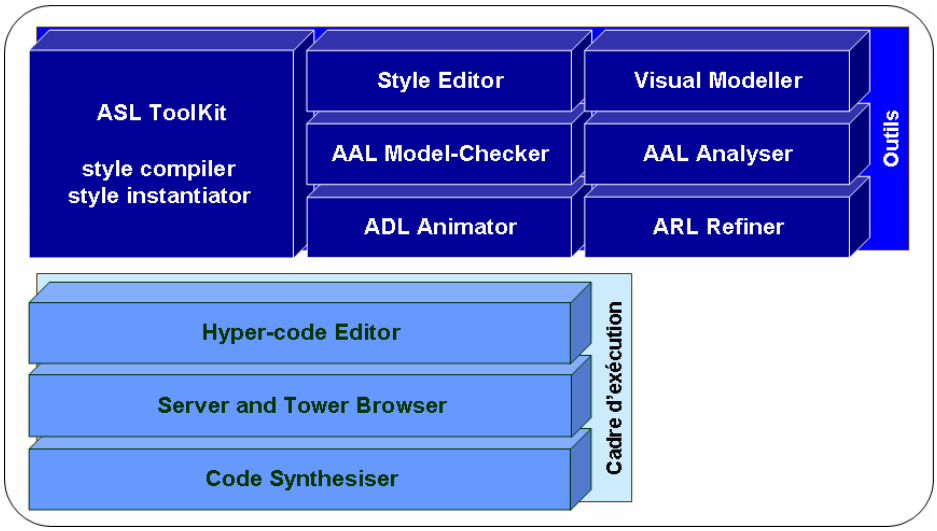



**Figure 3-4 : L'environnement ArchWare**

Le ***cadre d'exécution ArchWare*** inclut un moteur d'exécution d'architectures basé sur des processus de développement évolutifs (Hyper-code Editor), un processus de raffinement de description d'architecture (Refiner et Code Synthesiser) et des mécanismes supportant l'interopérabilité des outils de l'environnement (Server and Tower Browser).

Les ***outils centrés architecture ArchWare*** fournissent des supports pour :

- la définition d'architectures,
- la définition des styles architecturaux (de nouveaux styles peuvent être définis en spécialisant des styles existants, une hiérarchie de styles peut être fournie),
- la validation des architectures (respect des besoins propres à un domaine),
- la vérification des propriétés fonctionnelles et extra-fonctionnelles des architectures,
- le raffinement des descriptions d'architecture depuis un niveau abstrait jusqu'à un niveau concret,
- la génération de code des systèmes dans différents langages de programmation (cette génération de code utilise des règles explicites).

Avant de présenter chacun de ces outils, nous présentons succinctement les différents langages développés dans le cadre de ce projet. Nous développerons un peu plus le langage de description de style car ce sera le langage pivot de nos travaux.

## V.A. Les langages proposés par l'environnement ArchWare

Le projet européen ArchWare fournit un ensemble de langages centrés architecture :

- l'***ArchWare Architecture Description Language*** (π-ADL), langage pour la description d'architectures dynamiques (évolutives),
- l'***ArchWare Architecture Analysis Language*** (AAL), un langage pour la description de propriétés architecturales,
- l'***ArchWare Architecture Refinement Language*** (ARL), un langage pour la description de raffinements d'architectures,
- l'***ArchWare Architecture Style Language*** (ASL) qui permet de construire des styles formalisés.

La figure ci-dessous (Figure 3-5) présente la manière dont sont interconnectés les différents langages et sur quelles bases ils reposent.

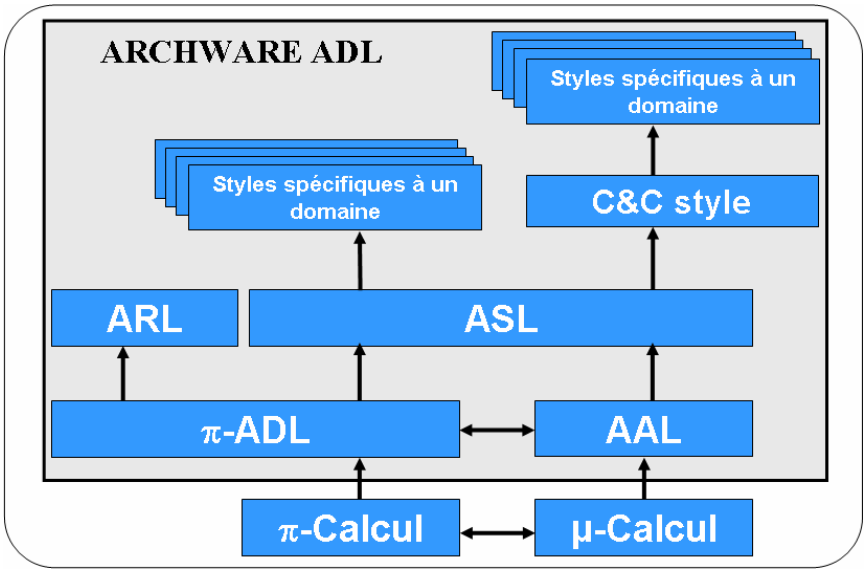



**Figure 3-5 : Les différents langages du projet ArchWare et leurs fondations**

Dans notre approche, nous nous appuyons sur ces différents langages et notamment sur l'ASL, le π-ADL, l'AAL et l'ARL. Dans les sections suivantes, une brève présentation des ces quatre langages va être faite. Nous ne vous présenterons pas leur syntaxe détaillée mais plutôt leurs différents concepts sous-jacents.

### V.A.1. Le langage π-ADL

Le langage π-ADL [Oquendo et al. 02] [Cimpan et al. 02] est un langage formel conçu pour supporter la spécification exécutable d'architectures logicielles dynamiques et évolutives. Il est fondé sur le π-calcul [Milner 99]. π-ADL est défini comme une extension

du π-calcul typé d'ordre supérieur[4] pour un domaine spécifique : c'est une extension *bien formée* pour définir un calcul d'éléments architecturaux mobiles et communicants. Ce langage permet donc de formaliser la structure et le comportement au sein d'une même description. Ce langage est ce que l'on appelle le langage noyau du projet ArchWare.

En π-ADL, une ***architecture*** est un ensemble d'éléments, appelés éléments architecturaux, qui sont reliés par des liens de communication. Ces éléments sont définis en terme de comportement (***behaviour***, ***abstraction***). Ce dernier est défini par un ensemble d'actions ordonnancées qui spécifie le traitement interne de l'élément (actions internes) et les interactions avec son environnement (actions de communication).
Un élément architectural communique avec les autres par une interface caractérisée par un ensemble de connexions (***connections***) qui permet de faire transiter des données. Un mécanisme de composition et un mécanisme de liaison, appelé ***unification*** (c'est une substitution au sens du π-calcul) permettent la mise en relation des éléments architecturaux. Ces éléments peuvent interagir lorsqu'ils sont composés et liés.

Un élément architectural peut être défini comme une composition d'autres éléments (**composite**). En d'autres termes, un comportement peut être défini comme un ensemble d'autres comportements interconnectés.

π-ADL permet la description ***d'architectures dynamiques***. En effet, les éléments architecturaux peuvent, d'une part, être composés ou décomposés *à la volée*[5] et, d'autre part, des éléments architecturaux peuvent être créés et liés dynamiquement. Enfin, des éléments architecturaux peuvent transiter comme des données à travers les connexions. Cette notion d'architecture dynamique ne sera pas utilisée dans nos travaux.

### V.A.2. Le langage AAL

AAL [Alloui et al. 02] est un langage formel pour exprimer les propriétés des architectures logicielles évolutives modélisées en π-ADL. Il est défini comme une extension du μ-calcul [Kozen 83] pour exprimer à la fois des propriétés structurelles et comportementales. En effet, le μ-calcul étant limité à l'expression de propriétés comportementales, il est nécessaire de l'étendre pour exprimer les propriétés structurelles. Aussi, l'AAL s'appuie sur la logique des prédicats pour exprimer ces propriétés. Il fournit donc un support à la vérification de propriétés architecturales, cette vérification est implémentée par *theorem proving* et *model checking*.

En AAL, les propriétés sont définies comme des formules prédicatives. Ce langage fournit des prédicats prédéfinis et des mécanismes (opérateurs et quantificateurs) pour construire de nouveaux prédicats.

### V.A.3. Le langage ARL

L'ARL [Oquendo 03b] est un langage formel basé sur le π-calcul, dédié au raffinement d'architectures logicielles avec d'une part, la prise en compte du raffinement des données, des comportements, des connexions et des structures et, d'autre part, la préservation des propriétés architecturales à travers le processus de raffinement. Le noyau du langage est un

[4] Dans cette version du π-calcul, les variables sont typées et des processus peuvent transiter via des canaux de communication.
[5] En cours d'exécution.

ensemble d'opérations supportant des transformations architecturales. ARL fournit donc un ensemble d'opérations appelées *actions de raffinement* pour la transformation d'architectures.

Le raffinement d'une architecture abstraite vers une architecture concrète en vue de son implémentation se fait en plusieurs étapes successives [Megzari 04]. Chaque étape implique l'application d'une action de raffinement qui fournit une solution architecturale correcte. Dans ARL, les actions de raffinement sont exprimées par des *pré-conditions*, des *transformations* et des *post-conditions*.

Les pré-conditions sont des conditions qui doivent être satisfaites dans une architecture avant l'application d'une action de raffinement. Les post-conditions doivent, quant à elle, être satisfaites suite à l'application d'une action de raffinement. Une transformation exprimée par une action de raffinement montre comment une architecture satisfaisant les pré-conditions peut être transformée en une architecture satisfaisant les post-conditions.

### V.A.4. Le Langage ASL

Cette section présente le langage ASL [Leymonerie 04]. Nous allons présenter un peu plus en détail ce langage car il s'agit du langage de base sur lequel s'appuient nos travaux.

Nous avons vu dans le chapitre 2 (section IV.B) qu'un style architectural permettait de caractériser une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques. Dans le cadre du projet ArchWare, on peut préciser cette définition par la suivante :

> *Un **style architectural** est un ensemble de contraintes définissant une famille d'architectures et un support à la conception des architectures de cette famille en terme de description et d'analyse.*

Le langage ASL se base sur deux langages issus du projet ArchWare (voir Figure 3-5). Le premier est π-ADL. Comme il a été vu précédemment, il s'agit du langage de description d'architecture. Le second est AAL qui est le langage de description de propriétés architecturales. Ces deux langages n'ont pas été intégrés tel quel dans ASL mais ont été étendus. Un autre aspect dont nous n'avons pas encore parlé ici gravite autour de la notion de style dans ArchWare. Il s'agit de la notion de vocabulaire spécifique à un domaine.

Ainsi, ASL a été construit en suivant trois axes et permet donc de :

- définir des contraintes : ASL permet, au travers l'expression de différentes contraintes, de délimiter un espace de conception qui caractérise la famille d'architectures.
- décrire des architectures : dans une optique de réutilisation et de compréhension, ASL permet non seulement de décrire une architecture, mais il fournit aussi les mécanismes pour réutiliser des concepts et pour fournir une syntaxe adaptée.
- définir des analyses spécifiques aux architectures suivant un style donné.

Notons ici qu'il ne faut pas confondre les notions de contraintes et d'analyses. Dans ASL, une contrainte qu'elle soit structurelle, comportementale ou d'attribut est là pour définir un style. L'ensemble des contraintes définit ainsi un espace de conception qui englobe toutes les architectures conformes au style. Ceci n'est pas le cas des analyses qui elles sont des

traitements sur les architectures qui permettent d'en déterminer certaines caractéristiques. Dans nos travaux, seule la notion de contrainte sera utilisée.

### *V.A.4.a. Définition d'un style en ASL*

La formalisation d'un style en ASL est structurée par trois entités :

- les contraintes (qui définissent ce que sont les styles),
- les constructeurs (qui fournissent un support à la description architecturale),
- les analyses (cette entité ne sera pas développée car elle n'est pas utilisée dans nos travaux).

De plus, afin de favoriser la réutilisation des styles et la compréhension de leur relation, ASL met en place deux mécanismes :

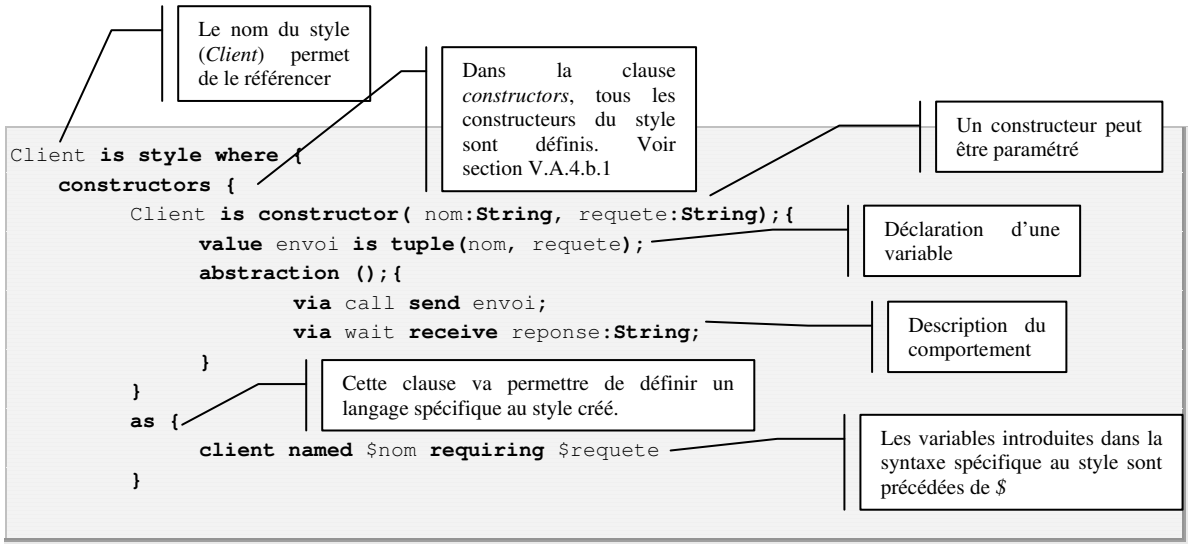
- l'héritage,
- l'agrégation.

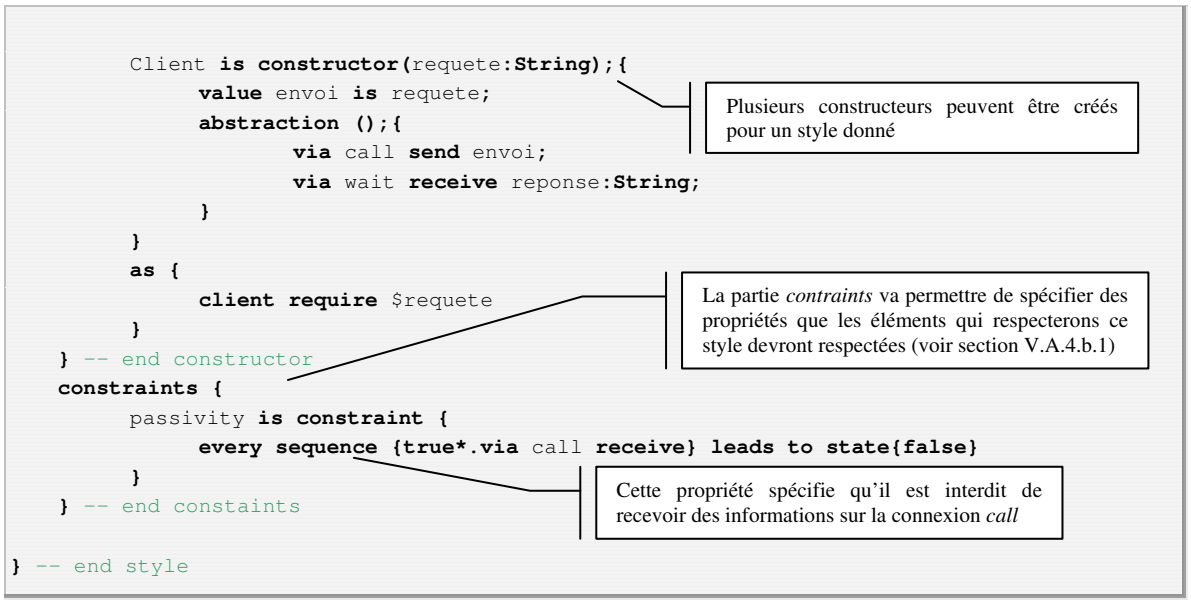
L'héritage est un mécanisme de spécialisation. Il permet de décrire un nouveau style tout en conservant les caractéristiques d'un autre style. L'agrégation est un mécanisme de composition. Il permet de structurer un style par plusieurs autres styles. Par exemple, un style qui défini les architectures Client-Serveur peut-être constitué d'un style Client et d'un style Serveur. Le style Client définit les caractéristiques propres à un aspect particulier du Client-Serveur.

### *V.A.4.b. Description d'un style en ASL*

La syntaxe d'ASL ne sera pas présentée ici, néanmoins, quelques points précis vont être abordés pour une meilleure compréhension des styles créés lors de ces travaux. Pour présenter ces quelques points, nous allons nous baser sur un style Client. Nous développons ici aussi bien les contraintes que le constructeur d'un style car, dans nos travaux, ce sont deux notions qui ont une grande importance. Le fonctionnement du client qui est décrit ci-dessous est très simple :

- Envoyer une requête (à un serveur quelconque). Cette requête contient deux information : un nom et la requête à proprement dite qui correspond à ce nom.
- Attendre la réponse du serveur contacté.
- Une fois ces deux opérations effectuées, le client s'arrête.






Dans les sous-sections suivantes, quelques précisions vont être données sur des notions particulières : les constructeurs (section V.A.4.b.1), les contraintes (section V.A.4.b.1), l'héritage (section V.A.4.b.3) et enfin l'agrégation (section V.A.4.b.4).

#### V.A.4.b.1 Les contraintes

Les contraintes de style caractérisent une famille d'architecture. En fait, il s'agit d'une assertion définissant des caractéristiques architecturales pouvant être évaluées sur une architecture. Elles définissent les limites de l'espace de définition des architectures d'un domaine. Les contraintes sont le cœur du style.

Nous rappelons qu'il existe différents types de contraintes :

- les contraintes *topologiques* (par exemple, une architecture ne contient pas de cycle),
- les contraintes *comportementales* (par exemple, il n'y a pas d'interblocage),
- les contraintes d'*attribut* (par exemple, la valeur d'une variable doit toujours être comprise entre 1 et 10).

La formalisation des contraintes est nécessaire pour vérifier qu'une architecture satisfait un style. Cette vérification peut être réalisée lorsqu'un architecte veut vérifier qu'une architecture particulière respecte les contraintes du style. Les contraintes définissent ce qu'est le style. En fait, un architecte peut définir plusieurs architectures différentes et contrôler si elles vérifient les contraintes et donc le style.

Pour pouvoir définir une contrainte, le langage ASL étend AAL. Ainsi, certains prédicats ont été rajoutés. Pour plus de détails sur la construction des contraintes en ASL, voir [Leymonerie 04].

#### V.A.4.b.2 Les constructeurs

Les constructeurs fournissent le support à la description architecturale. Ils permettent à la fois de fournir des éléments, et des mécanismes architecturaux.

Le concept de constructeur fournit un mécanisme formel pour décrire un support à la conception architecturale, et plus particulièrement à la description des architectures. Parmi les intérêts des constructeurs il y a:

- la génération d'architectures à partir d'un style : à ce titre, on peut faire le rapprochement entre un constructeur de style et un constructeur de classe dans le langage objet.
- la réutilisation de code : un constructeur permet de définir des valeurs architecturales et des mécanismes architecturaux qui peuvent être réutilisés.
- la définition d'une syntaxe spécifique (*mixfix*) : un constructeur permet de définir une syntaxe plus proche des concepts d'un domaine particulier. Cela favorise la lecture et la compréhension d'une définition architecturale. Comme il a été vu précédemment, un mixfix est défini dans la clause *as*.

Ce dernier point est très intéressant dans le cadre de nos travaux. En effet, du fait que nos styles s'adressent à un public de non informaticien la plupart du temps (les concepteurs d'instruments), on ne peut pas se permettre de leur fournir un nouveau langage qui n'aurait rien à voir, au niveau de sa syntaxe, avec ce qu'ils manipulent tous les jours, c'est-à-dire, le vocabulaire du domaine de l'instrumentation intelligente. De plus, ce langage spécifique peut être modifié à volonté sans avoir à modifier autre chose, par conséquent, non seulement on peut créer un langage spécifique au domaine, mais encore mieux, on peut créer un langage spécifique à chaque concepteur d'instruments sans pour autant avoir à créer des compilateurs spécifiques etc.

#### V.A.4.b.3 L'héritage

Un mécanisme d'héritage (*extending*) offre la possibilité de construire un nouveau style à partir de la définition d'un style parent. Ce mécanisme permet la réutilisation d'un style existant et la définition de différents niveaux d'abstraction (généralement, du plus abstrait au plus concret).

Imaginons que nous désirions construire un style de client spécifique qui hériterait du style client précédent. Son code sera le suivant :

```
Client2 is style extending Client where {
    [...]
 }
```

Le style *Client2* hérite du style *Client*

De façon visuelle (Figure 3-6 ci-dessous), si on représente l'ensemble des architectures suivant le style Client et celui suivant le style Client2, on voit que le fait d'hériter d'un style revient à lui ajouter d'autres contraintes. Les contraintes du style parent seront toujours respectées. Schématiquement, à aucun moment nous ne pourrons avoir une partie de l'ensemble des architectures qui suivent le style Client2 en dehors de celui représentant les architectures qui suivent le style Client.

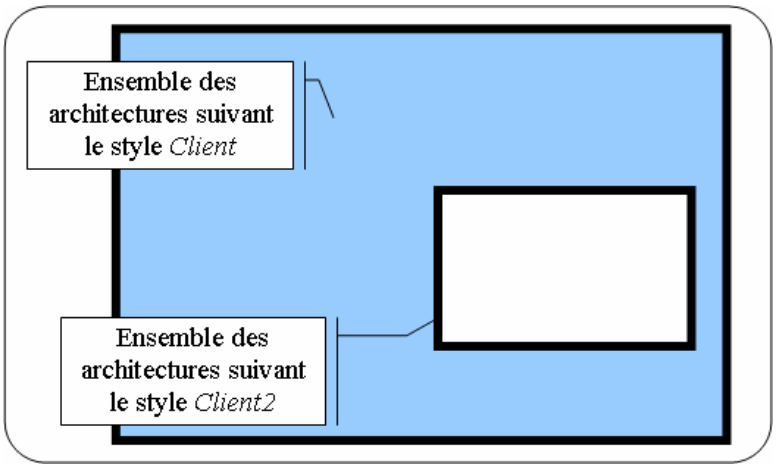



**Figure 3-6 : représentation ensembliste de l'héritage**

Il est à noté que dans le cadre du projet, quelques styles spécifiques ont été implémentés et peuvent servir de base pour la définition d'autres styles plus spécifiques. C'est notamment le cas du style C&C (Composant et Connecteur) qui figure dans la plus haut (Figure 3-5 section V.A). Les architectures de type composant-connecteur sont bien connues dans le domaine des architectures logicielles. Ces architectures sont souvent considérées comme la base de toute construction architecturale. En effet, il est généralement accepté que les architectures sont faites de composants (éléments de traitement) et de connecteurs (éléments de communication). Pour construire un style, il est donc possible de partir directement du langage ASL ou alors, par exemple d'un style existant comme C&C. Dans le deuxième cas, le mécanisme d'héritage est mis en place. Dans notre cas nous écrirons directement nos styles à partir du langage ASL pour des raisons que nous expliciterons plus tard.

#### V.A.4.b.4 L'agrégation

Le mécanisme d'agrégation permet de structurer un style en terme d'autres styles dits agrégats. Les styles agrégats encapsulent des constructeurs, des contraintes et des analyses qui alimentent le style contenant. En d'autres termes un style est composé d'éléments, chacun d'entre eux pouvant être défini à partir d'autres styles.

Le mécanisme d'agrégation apporte une meilleure réutilisation des styles déjà créés. De plus, il apporte une meilleure lisibilité et une meilleure compréhension du style contenant. Le mécanisme d'agrégation a été introduit pour promouvoir la réutilisation. En effet, un même style peut servir à la définition de différents autres styles. On a ainsi la possibilité de construire une bibliothèque de styles propres à un domaine particulier.

La structure d'un style en termes d'agrégats peut être construite parallèlement à celle des architectures qu'il représente. Un style qui représente les architectures Client-Serveur peut-être structuré avec un style Client et un style Serveur. Un style agrégat donne un support pour des sous partis des architectures.

Les styles agrégats qui composent un style sont définis dans une clause *styles* qui se trouve au même niveau hiérarchique que les clauses *constraints* ou *constructors.* Comme le montre l'exemple suivant, ils peuvent être définis par une *description de style* (comme *Serveur*) ou par *référence* à un style déjà défini en dehors de la description (comme *Client*).

Ainsi la définition de *Serveur* est donnée explicitement (Serveur **is style where{…}**) tandis que celle de Client3 est celle d'un autre style appelé *Client* (Client3 **is** Client).

```
ClientServeur is style where {
          styles {
                Serveur is style where{…};
                Client3 is Client
          }
          [...]
}
```

On peut noter que lors d'un héritage, un sous-style hérite des agrégats de son parent. Il peut aussi définir ses propres agrégats.

### *V.A.4.c. Utilisation des styles*

Une fois qu'un style a été conçu, il faut pouvoir l'utiliser, c'est-à-dire l'instancier. C'est là un des rôles du ou des constructeurs qui ont été définis au sein du style. Dans nos travaux, tous les styles qui devront être utilisés par un concepteur d'instruments intelligent auront un mixfix de défini dans la clause *as*. De ce fait, pour créer, par exemple, deux instances du style Client défini plus haut, on pourra écrire le code suivant :

```
value MonClient1 is client named "jerome" require "quel est mon age";
value MonClient2 is client require "quel est ma taille"
```

Suivant la syntaxe, le premier ou le deuxième constructeur sera utilisé. Par contre, dans les deux cas, les mêmes contraintes seront à vérifier.

Dans ces sections, nous venons de présenter les différents langages développés au sein du projet ArchWare. Dans la section suivante (section V.B), nous parlerons brièvement du cadre d'exécution d'ArchWare puis nous terminerons par la présentation des outils qui ont été implémentés afin de pouvoir mettre en œuvre les différents concepts introduits dans les langages (section V.C).

## V.B. Le cadre d'exécution ArchWare

Le cadre d'exécution fourni par le projet ArchWare s'appuie sur un outil *Server*, un navigateur appelé *Tower Browser* et une syntaxe concrète d'ArchWare ADL hyper codée.

L'outil *Server* et le navigateur *Tower Browser* sont les éléments du cadre d'exécution qui permettent d'intégrer les différents outils logiciels définis dans le cadre de ce projet. Le *Server* est une machine virtuelle évoluée au sein de laquelle toutes les descriptions d'architectures, de styles, de propriétés sont stockées. Ce serveur dispose également de fonctionnalités telles que la capacité de compiler et d'exécuter des programmes écrits en π-ADL. Le *Tower Browser* permet de gérer l'organisation du *Server*. Au sein de cet outil logiciel, l'organisation de l'espace de stockage est définie de façon arborescente, chaque nœud référence un espace particulier. Cet outil permet d'une part, de définir les arborescences et, d'autre part, de naviguer entre les différents nœuds. De plus, il permet, grâce à des mécanismes d'interaction, d'appeler les différents outils pour les appliquer sur un espace de stockage particulier.

Nous ne détaillerons pas plus les différents outils qui composent le cadre d'exécution d'ArchWare car nous ne les utiliserons pas dans nos travaux.

## V.C. Les outils de l'environnement ArchWare

Nous présentons les outils de l'environnement en nous appuyant sur la logique de l'ingénierie logicielle centrée architecture qui peut être représentée par le schéma suivant que nous avions vu la section II (Figure 3-3) :

Les différents outils sont :

- Le *Visual Modeller* et le *Style Editor* (section V.C.1),
- Le *Customiser* (section V.C.2),
- L'*Animator* (section V.C.3),
- Le *Model-Checker* (section V.C.4),
- L'*Analyzer* (section V.C.5),
- Le *Refiner* (section V.C.6),
- Le *Code Synthesizer* (section V.C.7).

### V.C.1. Les outils Visual modeller et Style editor

Le *visual modeller* permet la définition des styles et des architectures sous forme graphique à partir de profils UML, il s'appuie sur Objecteering. Le *style editor* est un simple éditeur de texte permettant la définition des styles sous forme textuelle.

### V.C.2. L'outil ASL Toolkit

L'outil *ASL Toolkit* [Leymonerie 04] permet la personnalisation d'un environnement en s'appuyant sur la définition de styles architecturaux. Il est implémenté en Java et en XSB Prolog. Cet outil s'appuie sur deux modules : le module « *style compiler* » et le module « *style instantiator* ». Le premier module compile des définitions de styles écrites en ASL, tandis que le second génère des instances du style, c'est à dire des architectures conformes au style (en π-ADL).

Dans le cadre de cette thèse, cet outil sera utilisé pour transformer le langage de style en π-ADL (voir chapitre 6 section II).

### V.C.3. L'outil Animator

L'outil *Animator* [Pourraz et al. 05] anime des descriptions d'architectures écrites en π-ADL, il est implémenté en Java et en XSB Prolog. Cet outil permet de valider le comportement d'une architecture et, par conséquent, les styles sur lesquels sont basés cette architecture. Cet outil permet de vérifier certaines propriétés (propriétés comportementales).

Dans le cadre de cette thèse, nous utilisons cet outil pour valider le comportement de nos architectures (voir chapitre 6 section IV.C).

### V.C.4. L'outil Model-Checker

L'outil *Model-Checker* [Bergamini et al. 04] se base sur des traces d'exécution des descriptions π-ADL pour vérifier les propriétés comportementales décrites en AAL. Ces traces sont générées par le *Server*. Le Model-Checker est implémenté en SYNTAX, LOTOS NT [ISO 88][Garavel et al. 02] , et en C. Le résultat de la vérification correspondant à la validité d'une propriété par rapport à la trace d'exécution utilisée, il est exprimé sous forme booléenne.

Cet outil n'est pas suffisant pour réaliser toutes les vérifications formelles. En effet, il ne vérifie que le comportement des architectures.

### V.C.5. L'outil Analyzer

L'outil *Analyzer* [Azaiez&Oquendo 05] vérifie des propriétés d'architectures exprimées en AAL, il est implémenté en Java et XSB Prolog. Il s'appuie sur les démonstrations logiques pour réaliser des vérifications de propriétés exprimées en AAL.

Dans le cadre de notre cas d'étude, nous utilisons cet outil pour vérifier les propriétés de nos architectures (voir chapitre 6 sections II et III).

### V.C.6. L'outil Refiner

L'outil *Refiner* [Megzari&Oquendo 04] se base sur le langage ARL pour construire (par raffinement) une architecture, il est implémenté sous MAUDE [Clavel et al. 03]. Il s'appuie sur les actions de raffinement pour générer, de façon formelle, une nouvelle architecture.

Dans le cadre de cette thèse, cet outil ne sera utilisé directement pour implémenter les mécanismes de raffinement. Un outils qui intègre et étend le *Refiner* sera utilisé (voir chapitre 6 section III).

### V.C.7. L'outil Code Synthesizer

L'outil *Code Synthesizer* [Balasubramaniam et al. 04] s'appuie sur des règles de transformation pour générer la description d'une architecture dans un langage cible (exécutable). Concrètement, cet outils permet de transformer du code écrit en π-ADL en n'importe quel langage cible pour peu que des règles de transformation soient écrites.

## *VI. Conclusion*

Ce chapitre nous a permis de faire ressortir les différents besoins que nous avions au niveau de la conception centrée architecture afin de l'utiliser pour la conception d'instruments intelligents. Après un état de l'art des principaux ADL existant, il est ressorti que celui qui était le plus adapté à nos travaux était ArchWare ADL.

Nous avons pu voir que le projet ArchWare avait pour but de répondre à la demande de plus en plus importante des systèmes logiciels à s'adapter aux différents changements durant leur cycle de vie. Pour ce faire un ensemble de langages centrés architecture et un ensemble de logiciels ont été créés. Ces langages et leurs outils associés permettent donc de développer des applications en utilisant les concepts de l'ingénierie centrée architecture.

Pour notre cas plus particulièrement, le projet ArchWare fournis à la fois des langages et des outils qui permettent de créer des styles architecturaux. Les propriétés qui contraignent ces styles peuvent à la fois être structurelles, comportementales et portées sur des attributs de l'architecture. De plus, la couche style d'ArchWare va permettre de construire un langage simple et qui reprend la sémantique du domaine pour concevoir des instruments intelligents. Ce langage, en plus de ces avantages, sera un langage formel ce qui signifie que de nombreuses vérifications pourrons être entreprises.

Dans le chapitre suivant, nous allons commencer à présenter les travaux réalisés au sein de cette thèse et notamment l'évolution du modèle des instruments intelligents pour palier aux manques que nous avions repérés dans le chapitre 2.

# Chapitre 4 : VERS UN NOUVEAU MODELE D'INSTRUMENTS INTELLIGENTS

# Chapitre 4 : Vers un nouveau modèle d'instruments intelligents

# Vers un nouveau modèle d'instruments intelligents

Dans le chapitre 2, nous avons abordé le domaine de l'instrumentation intelligente. Ceci nous a permis de choisir le modèle de conception de la partie logicielle d'un instrument intelligent avec lequel nos travaux ont débutés. Ce modèle avait toutefois de nombreux manques qui étaient principalement dû à la façon dont il avait été implémenté jusqu'à présent.

Dans le chapitre 3, un état de l'art sur les architectures logicielles a permis de mettre en avant les qualités d'ArchWare ADL. Cet ADL et ces outils associés vont nous permettre de régler la majorité des problèmes rencontrés dans le chapitre 2 avec le modèle mais pas tous. En effet, le modèle de conception sur lequel nous somme partis ne permet pas de concevoir n'importe quel type d'instrument intelligent.

Dans ce chapitre, les limitations du modèle actuel vont donc être présentées (section I). Ensuite, nous vous en proposerons un nouveau qui répond aux manques du premier (section II).

## *I. Le défaut du modèle actuel*

### I.A. Exemple

Le problème qui nous intéresse ici se situe au niveau de la couche externe d'un instrument intelligent et plus particulièrement au niveau de la définition des services externes. En effet, dans le modèle utilisé dans CAPTool, un service externe est formé d'un ensemble de services internes atteignables.

La figure ci-dessous (Figure 4-1), donne un exemple de service externe nommé *Mesure1* et composé de trois services internes. La représentation qui est utilisée dans cette figure est, en fait, une concaténation des représentations qui ont été utilisées dans les figures 2.2 et 2.4 du chapitre 2. En effet, les événements représentés habituellement dans le graphe de

dépendances sont ici directement intégrés dans la figure pour les services internes qui nous intéresse.

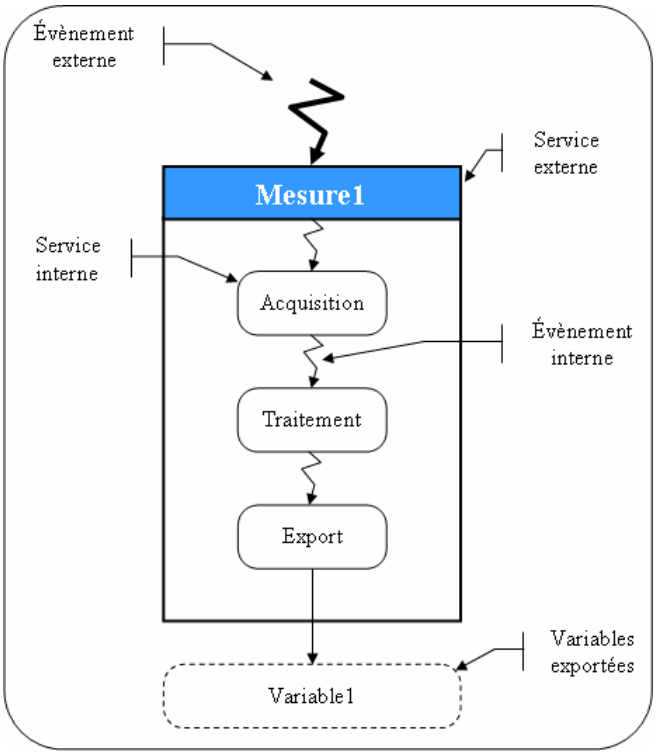



**Figure 4-1 : Service externe "Mesure1"**

Dans cet exemple, le service externe *Mesure1* exécute tout d'abord le service interne *Acquisition*. Ensuite, le service *Traitement* est déclenché. Enfin, c'est au tour du service *Export* d'être activé. Ce dernier a pour but d'envoyer la valeur mesurée sur le réseau.

Une partie du code CAP nécessaire à la conception de ce service externe est le suivant (toutes les informations qui ne nous intéressent pas ici ont délibérément été enlevées) :

```
[...]
iService Acquisition on ievent = START {
      [...]
}
iService Traitement on end(Acquisition) {
      [...]
}
iService Export on end(Traitement) {
      [...]
}
Service Mesure1 on [...] in [...] {
      uses Acquisition, Traitement, Export;
}
```

Le service externe *Mesure1* tel qu'il est décrit ici est donc tout à fait concevable. Maintenant, supposons que, pour une raison quelconque, un traitement supplémentaire doit être effectué, de temps à autre, sur la variable avant qu'elle ne soit exportée.

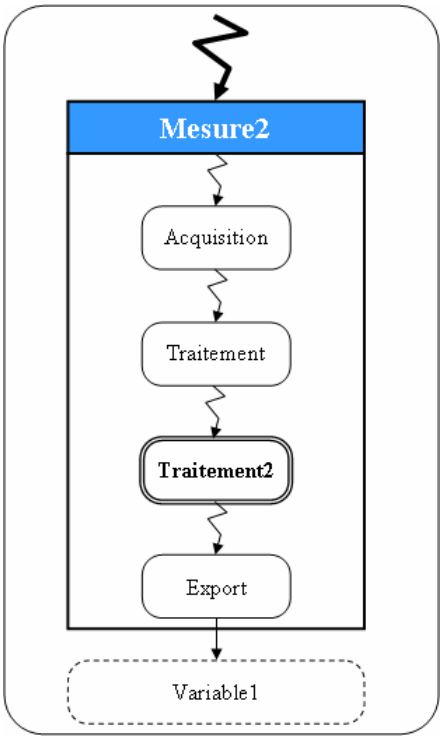



**Figure 4-2 : Service externe Mesure2**

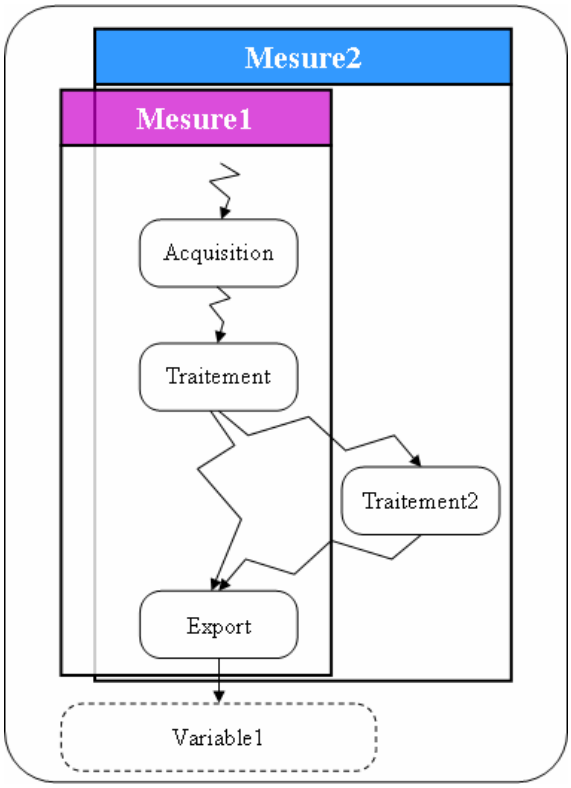



**Figure 4-3 : Mesure1 et Mesrure2**

Un autre service interne, nommer *Mesure2*, doit alors être intercalé entre *Traitement* et *Export* (voir Figure 4-2 ci-dessus).

Le problème qui ressort est que ce service externe *Mesure2* tel que l'on veut qu'il fonctionne, ne peut être conçu. La figure ci-dessus (Figure 4-3), représente les deux services externes précédent de façon superposée.

Le service interne *Export* se déclanche maintenant sur deux évènements internes différents (fin de *Traitement* et fin de *Traitement2*). De ce fait, si le service externe *Mesure2* est exécuté, une fois le service interne *Traitement* terminé, les deux services internes *Traitement2* et *Export* vont se déclencher. De plus, une fois le service interne *Traitement2* terminé à son tour, un nouvel *Export* va être réalisé. Or ce n'est pas le comportement qui était attendu. Il fallait que l'*Export* se fasse seulement une fois le *Traitement2* réalisé mais pas après le simple *Traitement*.

Ceci pose deux problèmes. Tout d'abord, l'*Export* est réalisé deux fois au lieu d'une, mais plus grave, la première valeur qui est exportée est fausse par rapport à ce que doit délivrer le service externe *Mesure2* !

A partir du problème rencontré dans cet exemple, nous allons pouvoir généraliser le phénomène dans la partie suivante.

## I.B. Généralisation du problème

Afin de généraliser, posons le problème suivant (voir Figure 4-4 ci-dessous).

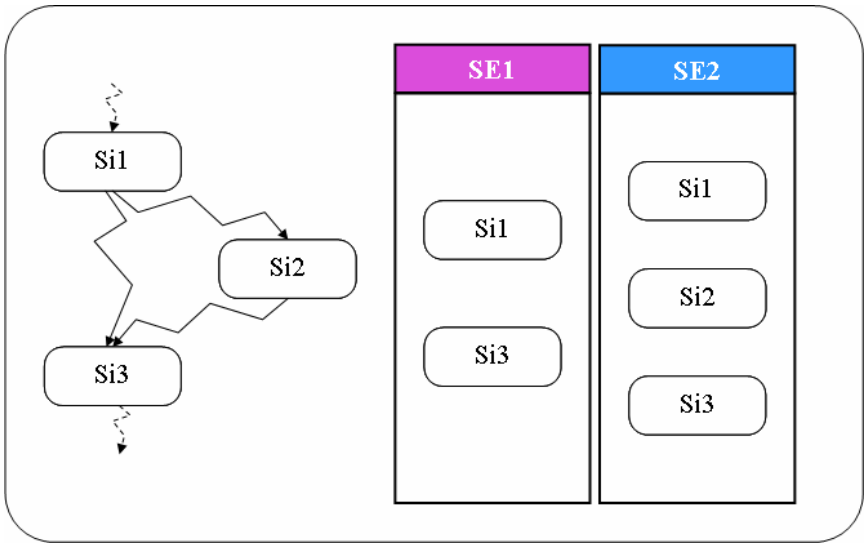



**Figure 4-4 : Configuration problématique**

Pour faire le parallèle avec la figure 4.3, SE1 et SE2 sont représentés aussi dans la figure ci-dessous (Figure 4-5) de façon superposée.

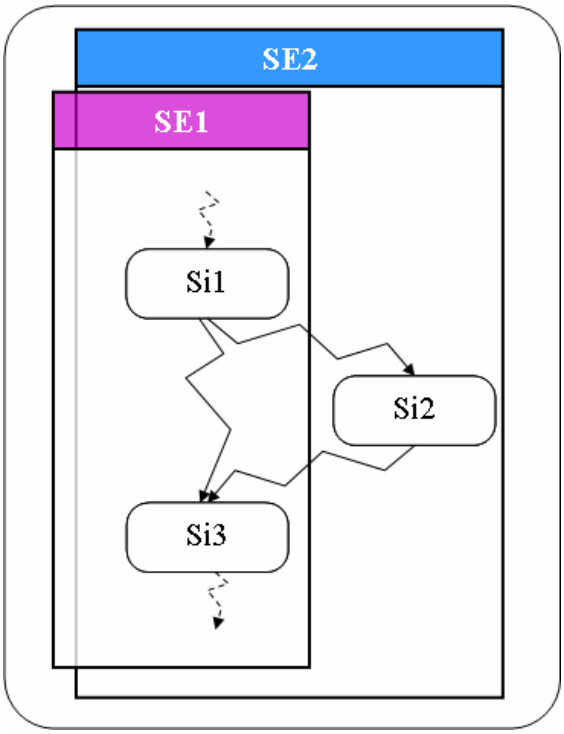



**Figure 4-5 : SE1 et SE2 superposés**

Cette figure montre bien que l'on se trouve dans le même cas de figure que celui de la section précédente. En effet, le service externe *SE2* intercale un service interne (*Si2*) entre deux autres services internes (*Si1* et *Si3*) qui eux sont exécutés directement l'un après l'autre dans *SE1*. Le code CAP associé à cet exemple est présenté ci-dessous. Comme précédemment, seules les informations nécessaires à l'explication de cet exemple ont été conservées.

```
[...]
iService Si1 on end([...]) {
      [...]
}
iService Si2 on end(Si1) {
      [...]
}
iService Si3 on end(Si1) or end(Si2) {
      [...]
}
Service SE1 on [...] in [...] {
      uses Si1, Si3;
}
Service SE2 on [...] in [...] {
      uses Si1, Si2, Si3;
}
[...]
```

*SE2* a été créé dans le but d'exécuter *Si1* puis *Si2* et enfin *Si3*. Or un autre service externe *SE1* devra exécuter *Si1* puis *Si3*. Cette configuration implique que le service interne *Si3* soit déclenché soit à la fin de *Si1* soit à la fin de *Si2*.

Par conséquent, définir *SE2* devient impossible. En effet, une fois l'exécution de *Si1* terminée, il y aura le déclenchement simultané de *Si2* et de *Si3*, ce qui n'est pas le comportement désiré.

Le problème qui est rencontré au niveau du modèle est qu'un service externe est défini seulement comme un ensemble de services internes. A aucun moment, le graphe de dépendances des services internes ne rentre en compte dans sa création. Pour remédier à ce problème ce modèle va être modifié.

# II. Vers un nouveau modèle d'instruments intelligents

## II.A. Le nouveau modèle Client/Serveur

Baser directement les services externes sur la couche des services internes n'est plus suffisant. Il faut donc rajouté une nouvelle couche au modèle client/serveur (Figure 4-6 ci-dessous).

La nouvelle couche qui a été intégrée se nomme *Graphe des services internes*. Elle a pour but de matérialiser le graphe de dépendances complet de l'instrument intelligent. Désormais, les services externes sont liés à cette nouvelle couche et non plus directement aux services internes.

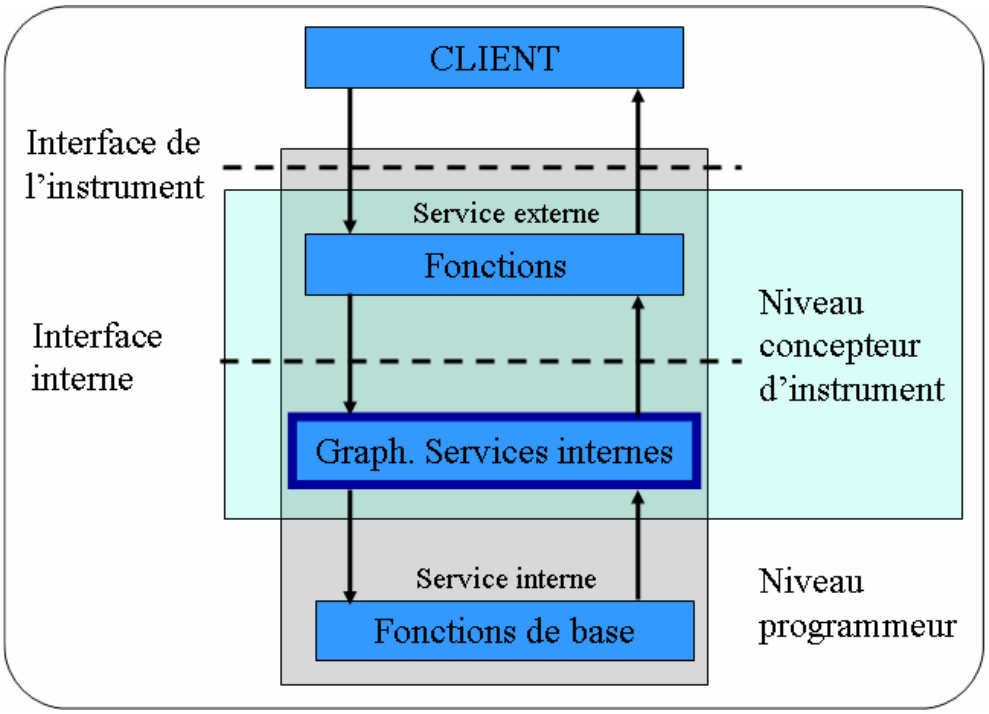



**Figure 4-6 : Le nouveau modèle des instruments intelligents**

Grâce à ce modèle, un service externe est maintenant défini de la façon suivante :

> *Un **service externe** est un sous ensemble non vide du graphe des services internes.*

Cette nouvelle couche est bien entendue conçue par le concepteur d'instruments puisque pour pouvoir créer un tel graphe, il faut connaître les fonctionnalités de l'instrument.

## II.B. Le graphe des services internes

### II.B.1. Sa construction

Cette nouvelle couche doit permettre de modéliser les services externes comme des sous graphes du graphe des services internes. Des modifications doivent être faites par rapport au graphe de dépendances que nous avions dans le modèle précédent afin que l'information que l'on peut en tirée soit plus précise.

Etant donné que les transitions entre les services internes doivent maintenant être explicitées dans les services externes, il faut que l'information soit facilement identifiable. En effet, si l'on prend l'exemple de la figure ci-dessous (Figure 4-7), *Si3* s'exécute-t-il après *Si1* OU *Si2*, ou *Si3* s'exécute-t-il après *Si1* ET *Si2* ?

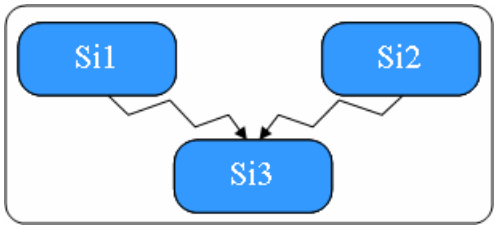



**Figure 4-7 : ET ou bien OU ?**

Il est impossible ici de faire la différence entre un *ET* et un *OU* en regardant simplement la représentation graphique du graphe de dépendances. La seule façon de savoir est de consulter le code CAP de l'instrument (dans la définition du service interne). Or, ce code CAP doit être créé, en partie, à partir de ce graphe de dépendances.

Pour pouvoir différencier ces deux cas nous avons choisi d'expliciter le *ET* en le représentant sous forme d'un nœud au niveau du graphe de dépendances. Le *OU* restera inchangé (Figure 4-8).

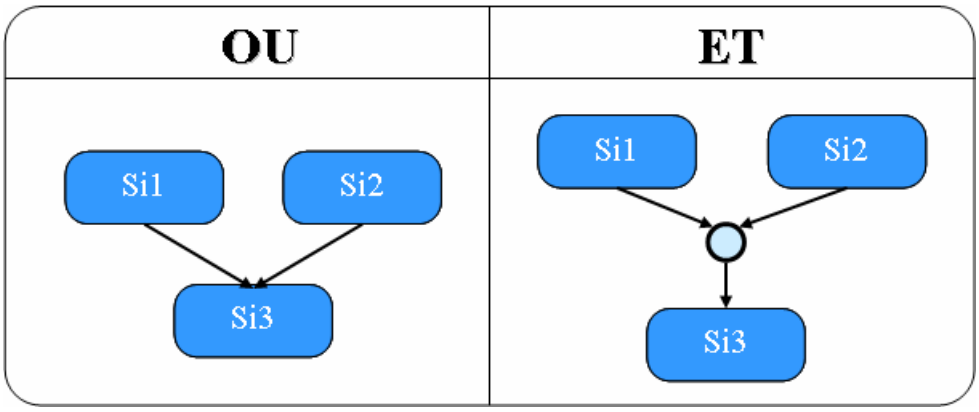



**Figure 4-8 : Nouvelle notation**

Cette nouvelle notation pour le *ET* nous oblige à ne plus voir les flèches qui relient les services internes comme des évènements internes mais comme des relations de causalité. Ce *ET* est appelé un *Nœud Rendez-vous* (ou *Nœud Rdv*) car sa fonction est d'attendre que les services internes exécutés en amont soient tous terminés avant de déclencher le/les service(s) interne(s) à exécuter en aval.

Par la suite, nous nommerons ces relations de causalité des connexions. De plus, nous avons vu précédemment que les services internes pouvaient avoir besoin de consommer des variables pour s'exécuter. C'est pourquoi, ces connexions peuvent aussi être vues comme un flux de données entre les services.

La figure ci-dessous (Figure 4-9) présente un exemple plus complet de graphe des services internes.

Ce type de graphe est composé des briques de base qui ont été créées par le concepteur logiciel : les services internes (▭). Divers éléments ont été rajoutés afin de pouvoir enchaîner ces derniers. Tout d'abord, les différents services internes peuvent être reliés par des flèches (→) qui permettront de définir la façon dont ils s'enchaînent. Ensuite, afin de pouvoir synchroniser différentes opérations, des *nœuds Rdv* (○) ont été rajoutés.

La dernière chose qui a évoluée est le fait d'introduire ce qui a été appelé les *nœuds d'entrée/sortie* (▭). Ces nœuds ont pour fonction de spécifier où se situent les points interaction avec ce graphe. En fait, les *nœuds d'entrée* correspondent aux endroits ou vont pouvoir être réceptionnés des évènements externes et les *nœuds de sortie* sont la pour exporter des valeurs qui pourraient servir à un autre instrument ou à une IHM. Ces nœuds de sortie seront utiles aussi pour signifier qu'un service externe est terminé. Dans ce cas un simple évènement de fin de service sera délivré. Ceci répond au manque du modèle précédent sur la fin des services externes.

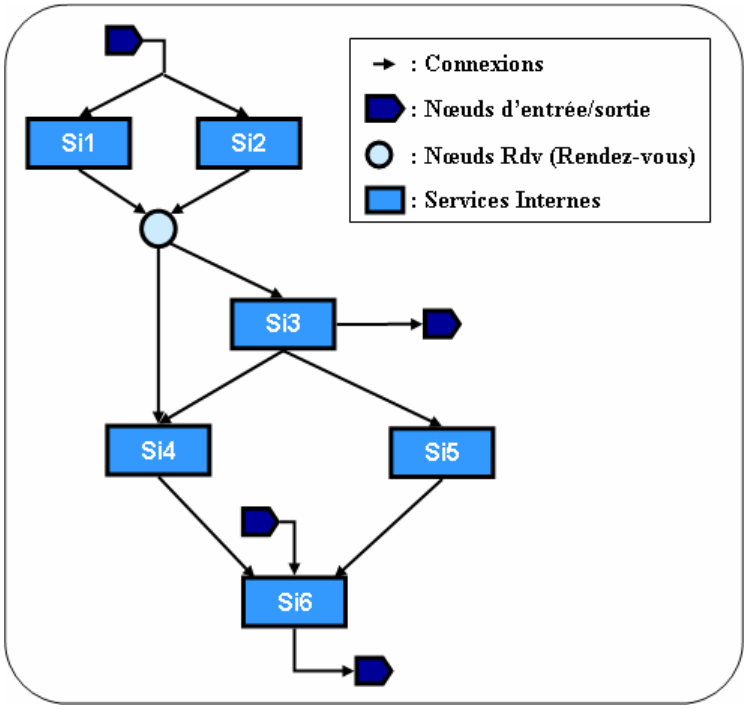



**Figure 4-9 : Exemple de graphe des services internes**

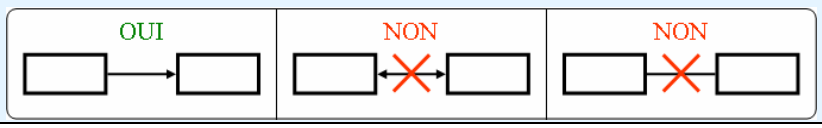
De façon plus formelle, on peut définir le graphe des services internes (GSI) de la façon suivante :

> *Soit* GSI *un* ***graphe des services interne****,* $IS$ *l'ensemble des services internes,* $\mathcal{RDV}$ *l'ensemble des nœuds rendez-vous,* $I$ *l'ensemble des nœuds d'entrée et* $O$ *l'ensemble des nœuds de sortie.*
>
> $$GSI = (S, A)$$
>
> Avec $S = \{IS \cup \mathcal{RDV} \cup O \cup I\}$ = *ensemble des nœuds* et avec A ensemble des arcs ;
>
> $A \subset (S \times S)$.

Etant donné que le modèle de conception des instruments intelligents vient d'évoluer, les propriétés à vérifier doivent elles aussi changer.

## II.B.2. Propriétés

Modéliser un graphe des services internes de la sorte ne protège pas le concepteur d'instruments d'une mauvaise conception. Donc, de façon similaire au chapitre 2, pour construire un graphe des services externes de façon correcte, plusieurs règles doivent être respectées. Elles vont être listées ci-dessous. Il est à noter que les propriétés suivantes sont exprimées à l'aide du vocabulaire de la théorie des graphes.

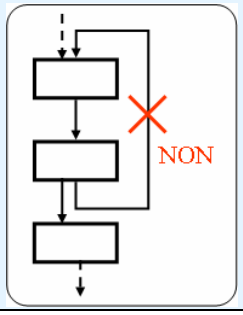
> **Propriété 4.1**
> *Le graphe des services internes est un graphe orienté.*
>
> 
> 

Cette première propriété permet de respecter le fait que les services internes doivent être enchaînés pour constituer un service externe. Etant donné que le graphe des services internes est maintenant à la base de la création des services externes, il faut que ce graphe soit orienté. Cette propriété était déjà présente pour le graphe de dépendances du modèle précédent.

**Propriété 4.2**
*Le graphe des services internes est un graphe connexe.*

Cette deuxième propriété permet de dire que l'on n'a pas de services internes isolés (qu'on ne peut pas atteindre) dans l'instrument intelligent.

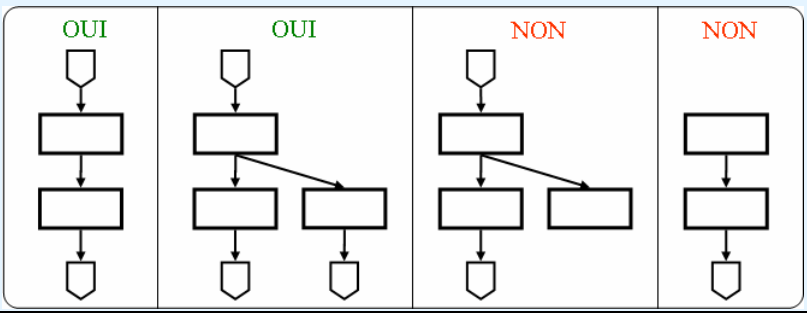
**Propriété 4.3**
*Il faut vérifier qu'il n'y a pas de rebouclage. Autrement dit le graphe des services internes ne doit pas comporter de circuit.*




Cette troisième propriété garantie le fait qu'un service externe ne tourne pas indéfiniment. En effet, le modèle ne disposant pas de nœud *d'aiguillage* (ce qui équivaudrait à un « si » en programmation), nous ne sortirions jamais d'une configuration qui aurait une boucle. Si un service externe a besoin de tourner en permanence, il faudra gérer cela au niveau de l'évènement externe qui lui est associé. En effet, il faudra faire en sorte que l'évènement de fin de service externe déclenche l'évènement de début du même service externe.

**Propriété 4.4**
*Le graphe des services internes commence toujours par au moins un nœud d'entrée et fini toujours par au moins un nœud de sortie. Autrement dit :*

*-Le degré entrant d'un nœud d'entrée est 0.*
*-Le degré sortant d'un nœud de sortie est 0.*
*-Les autres éléments (Services internes, nœuds Rdv) ont tous un degré entrant et un degré sortant supérieur à 0.*



Enfin, cette dernière propriété nous contraint sur plusieurs points :

- Le premier point oblige à concevoir un graphe des services internes avec au moins un nœud d'entrée et au moins un nœud de sortie. Ces contraintes sont obligatoires car sans elles, les services externes ne pourraient pas être exécutés ni indiquer à quel moment ils se terminent.
- Le deuxième point que nous devons respecter grâce à cette propriété est que le graphe des services internes ne laisse pas la possibilité de concevoir des services externes que l'on ne pourrait soit pas déclencher soit qui ne délivreraient pas d'évènements externes de fin.

Cette nouvelle couche au modèle étant la nouvelle base de création des services externes, ces derniers ont aussi évolués.

## II.C. Les Services Externes

Comme nous l'avons dit précédemment, les services externes sont des sous graphe du graphe des services internes. Dans la figure ci-dessous (Figure 4-10), trois exemples de services externes qui peuvent être créés dans le graphe des services internes de la section précédente sont présentés.

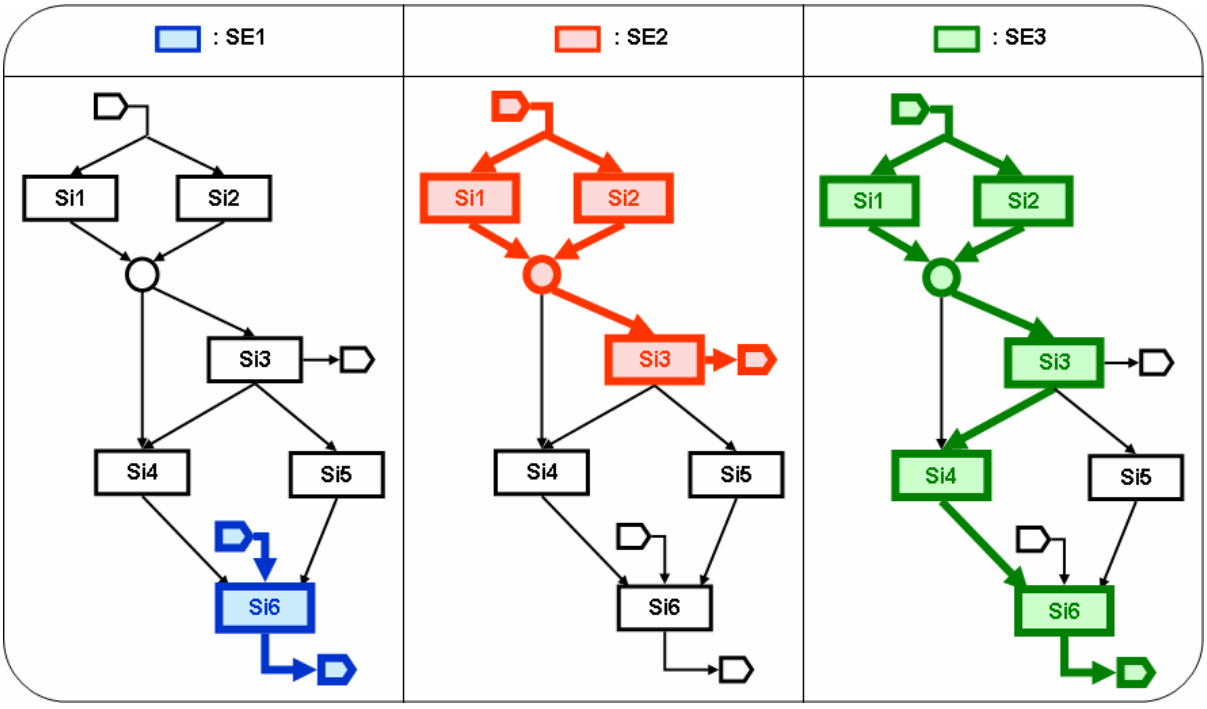



**Figure 4-10 : Exemple de Service Externe et de Mode**

Dans cet exemple, les trois services externes sont:

- *SE1* (▭)
- *SE2* (▭)
- *SE3* (▭)

Ces trois services externes sont des exemples. Beaucoup d'autres peuvent être créés sur un tel graphe des services internes. Le tout étant de respecter des règles de construction qui font que les services externes créés sont corrects.

Tout d'abord, étant donné que nous nous basons sur le graphe des services internes, nous allons voir si les propriétés de ce dernier sont conservées :

- Propriété 4.1 : cette propriété est conservée par construction et n'est donc plus à vérifier.
- Propriété 4.2 : cette propriété doit être vérifiée car tous les services internes d'un service externe doivent être atteignables. Par conséquent :

**Propriété 4.5**
*Un service externe est représenté par un graphe connexe.*

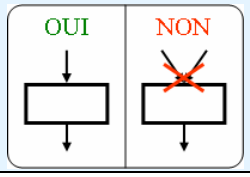
- Propriété 4.3 : cette propriété est conservée par construction puisqu'un service externe est un sous graphe du graphe des services internes.
- Propriété 4.4 : Cette propriété doit être conservée et vérifiée pour deux raisons. La première est que pour pouvoir être exécuté, un service externe doit commencer par un nœud d'entrée. La deuxième raison, est que dorénavant, tout service externe doit signaler lorsqu'il se termine. Il faut donc qu'il ait au moins un nœud de sortie.

**Propriété 4.6**
*Un service externe commence toujours par au moins un nœud d'entrée et fini toujours par au moins un nœud de sortie. Autrement dit :*
*-Le degré entrant d'un nœud d'entrée est 0.*
*-Le degré sortant d'un nœud de sortie est 0.*
*-Les autres éléments (services internes, nœuds Rdv) ont tous un degré entrant et un degré sortant supérieur à 0.*

Les règles précédentes sont nécessaires mais pas suffisantes. Voici les différentes règles que nous devons ajouter :

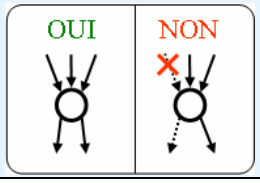
- Au sein d'un service externe, si un même service interne peut être exécuté de deux façons différentes, plusieurs problèmes se posent. Tout d'abord, il y a celui du partage de ressources. En effet, une ressource physique étant souvent derrière un service interne, il est impossible que se dernier soit exécuté plusieurs fois en parallèle. Le deuxième problème est un problème de donnée. En effet, si deux données différentes doivent être traitées par un même service interne, comment savoir si la première donnée qui ressortira de ce service interne sera celle issue du traitement de la première donnée entrée ou de la deuxième. Par conséquent, ce cas est interdit par la propriété suivante :

**Propriété 4.7**
*Un service interne ne peut être atteint que d'une seule manière dans un service externe. Autrement dit :*
*-Le degré entrant d'un service interne est 1.*



- Pour ne pas avoir de blocage du service externe, étant donné que nous avons rajouté un nouvel élément qui est le nœud Rdv, cette dernière propriété doit être vérifiée :

**Propriété 4.8**
*Lorsque n connexions se rejoignent sur un noeud Rdv, ces n connexions doivent faire partie du service externe. Autrement dit :*
*-Si le degré entrant d'un nœud Rdv est n dans le graphe des services interne, il doit aussi être de n dans les services externes dans lequel il se trouve.*




Une fois les services externes définis par l'utilisateur, ces derniers sont comme toujours, organisés dans des modes.

## II.D. Les Modes

Le dernier travail que doit réaliser le concepteur d'instruments est de classer les différents services externes dans des modes.

La définition que nous utilisons pour les modes est toujours celle du chapitre 2 :

*Un mode est un sous-ensemble de l'ensemble des services externes de l'instrument intelligent. Un mode comprend au moins un service externe.*

Nous partons du principe que lorsque l'on se trouve dans un mode donné, tous les services externes accessibles peuvent être exécutés en parallèle. En partant de là, si l'on ne respecte pas certaines règles de construction pour ces modes, des problèmes vont survenir.

Partons des services externes présentés dans la section précédente (SE1, SE2, SE3). Si on les regroupe au sein d'un même mode et que l'on superpose leur graphe des services internes, on obtient ceci (Figure 4-11 ci-dessous).

Imaginons que, dans ce cas de figure, les trois services externes soient simultanément actifs. Nous aurions ici un problème au niveau de Si6. Nous aurions a nouveau le même problème que pour les services externes (résolu par la propriété 4.7).

Par conséquent, la seule propriété que l'on a à vérifier pour un mode est la suivante :

**Propriété 4.9**
*Un service interne ne peut être atteint que d'une seule manière dans un mode. Autrement dit :*
*-Le degré entrant d'un service interne au sein d'un mode est de 1.*

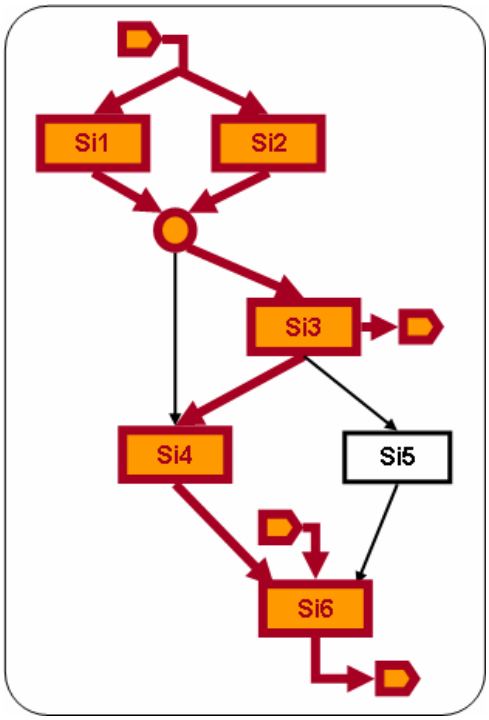



**Figure 4-11 : Superposition de SE1, SE2 et SE3**

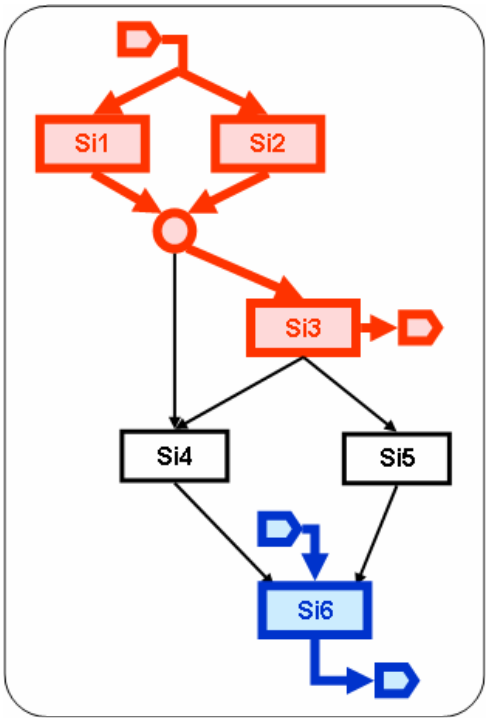



**Figure 4-12 : Service SE1 et SE2**

Ceci nous montre donc que l'on ne peut pas mettre SE1, SE2 et SE3 au sein du même mode. Par contre, on peut faire les regroupements suivants :

- SE1 et SE2 dans un mode (Figure 4-12) et SE3 dans un autre mode,
- SE2 et SE3 dans un mode et SE1 dans un autre,
- SE1 dans un mode, SE2 dans un autre et SE3 dans un troisième.

On peut remarquer que dans ce modèle, lorsque deux services externes, qui partagent les mêmes ressources, sont exécutés en même temps (par exemple SE2 et SE3), les informations produites par les services internes sont partagées entre les deux services externes.

## *III. Conclusion*

Ce chapitre avait pour but d'étudier le modèle présenté dans le chapitre 2 afin de faire ressortir les éventuels problèmes. Un nouveau modèle d'instruments intelligents qui construit les services externes sur une nouvelle couche qui se nomme graphe des services internes a été construit pour éliminer les problèmes rencontrés. Les services internes ne sont plus vus ici comme un ensemble de services internes mais comme un sous graphe du graphe des services internes. Le modèle ayant changé, les propriétés qui devaient être vérifiées lors de la conception d'un instrument intelligent ont évoluées elles aussi.

Ce nouveau modèle permet dorénavant que concevoir des instruments intelligents que l'on n'aurait pas pu concevoir avec le modèle précédent. Dans le chapitre suivant, comme les problèmes du modèle sont résolus, l'approche centrée architecture va être utilisée afin de concevoir des instruments intelligents qui suivent ce modèle.

# Chapitre 5 : APPROCHE CENTREE ARCHITECTURE POUR L'INSTRUMENTATION INTELLIGENTE

# Chapitre 5 : Approche centrée architecture pour l'instrumentation intelligente

# Approche centrée architecture pour l'instrumentation intelligente

Dans le chapitre précédent, le nouveau modèle de conception de la partie logicielle d'un instrument intelligent a été présenté. Ce modèle a pour base le graphe des services internes sur lequel on peut construire les services externes et les modes.

Le but de ce chapitre est de, tout d'abord, présenter comment les différents concepts de l'approche centrée architecture ont été utilisés dans le domaine de la conception d'instruments intelligents (section I). Ensuite, nous rentrerons plus dans le détail en présentant ce qui a été fait pour pouvoir concevoir les graphes des services internes (section II). Nous ferons de même pour la conception des services externes et des modes (section III). Grâce à ces différentes parties, un concepteur d'instruments sera capable de concevoir complètement un instrument intelligent. Cependant, nous verrons dans le chapitre suivant qu'un concepteur d'instruments sera capable de simuler le comportement d'un instrument intelligent directement à partir de son code ADL. Pour ce faire, toute la gestion du changement de service externe et du mode doit être implémentée en ADL. Les concepts de cette implémentation en ADL du comportement d'un instrument intelligent seront présentés dans la section IV.

## *I. Du modèle des instruments intelligents aux concepts architecturaux*

Comme il a été vu dans le chapitre 4, pour pouvoir spécifier complètement un instrument intelligent il faut décrire le graphe des services internes, les services externes et les modes. Nous avons vus aussi que ces trois types d'informations étaient caractérisés de façon différente :

- un graphe des services internes est représenté par un graphe orienté,
- un service externe est représenté par un sous graphe du graphe des services internes,

- un mode est représenté par un ensemble de services externes.

Du fait de leurs différentes caractéristiques, ces trois informations ne vont pas être construites de façon similaire dans le domaine des architectures logicielles. Dans la section I.A, nous verrons comment seront caractérisés les graphes des services internes. Dans la section I.B la façon dont nous avons construit les graphes des services externes et les modes sera abordée.

## I.A. Représentation des graphes des services internes

Il a été vu que la structure d'un instrument intelligent repose entièrement sur son graphe des services internes. Il est largement reconnu qu'une telle architecture peut être définie comme une configuration de composants et de connecteurs [Perry&Wolf 92] [Bass et al. 99]. Les termes composants et connecteurs peuvent être définis comme ceci :

> *Un **composant** correspond a une unité de traitement ou de stockage, il est le constituant de base d'une architecture. Un composant dispose de **ports** pour s'interfacer avec son environnement.*

> *Un **connecteur** formalise les protocoles d'échange entre les différents composants. Un connecteur dispose également de ports.*

Pour pouvoir organiser ces différents composants et connecteurs et pour pouvoir les faire communiquer, le mécanisme de configuration est utilisé.

> *Une **configuration** précise l'organisation des composants et des connecteurs en identifiant les **connexions (canaux)** entre leurs ports.*

Ces éléments architecturaux (composants et connecteurs) peuvent être de deux types :

- **Atomiques** : c'est à dire que ce sont des éléments simples (qui ne sont pas composés d'autres éléments),
- **Composites** : c'est à dire que ces éléments sont eux mêmes composés d'autres éléments (qui peuvent être eux aussi atomiques ou composites).

D'un point de vu structurel, un composant ou un connecteur est vu comme étant une boite noire accessible uniquement par l'intermédiaire de ses ports. Ces derniers sont pourvus de connexions. La figure ci-dessous (Figure 5-1), vous donne un exemple d'élément architectural (composant ou connecteur).

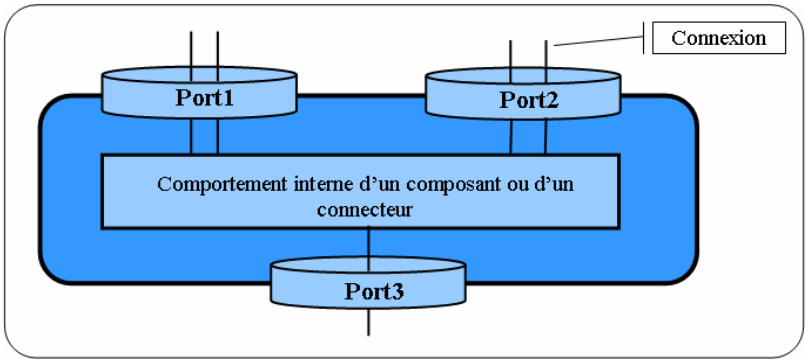



**Figure 5-1 : Exemple d'élément architectural**

Cet exemple montre un élément qui est composé de trois ports. Chaque port comporte un certain nombre de connexions qui permettent d'interagir avec l'extérieur (l'environnement).

Comme nous l'avons dit précédemment, la structure d'un instrument intelligent est principalement basée sur son graphe des services internes. Par conséquence, nous allons voir par la suite comment ont été identifiés les composants et les connecteurs au sein de cette architecture. La figure ci-dessous (Figure 5-2), redonne l'exemple de graphe des services internes que nous avions présenté dans le chapitre 4. Ce graphe va nous permettre de faire le parallèle entre les concepts de l'instrumentation intelligente et ceux des architectures logicielles de type composant-connecteur.

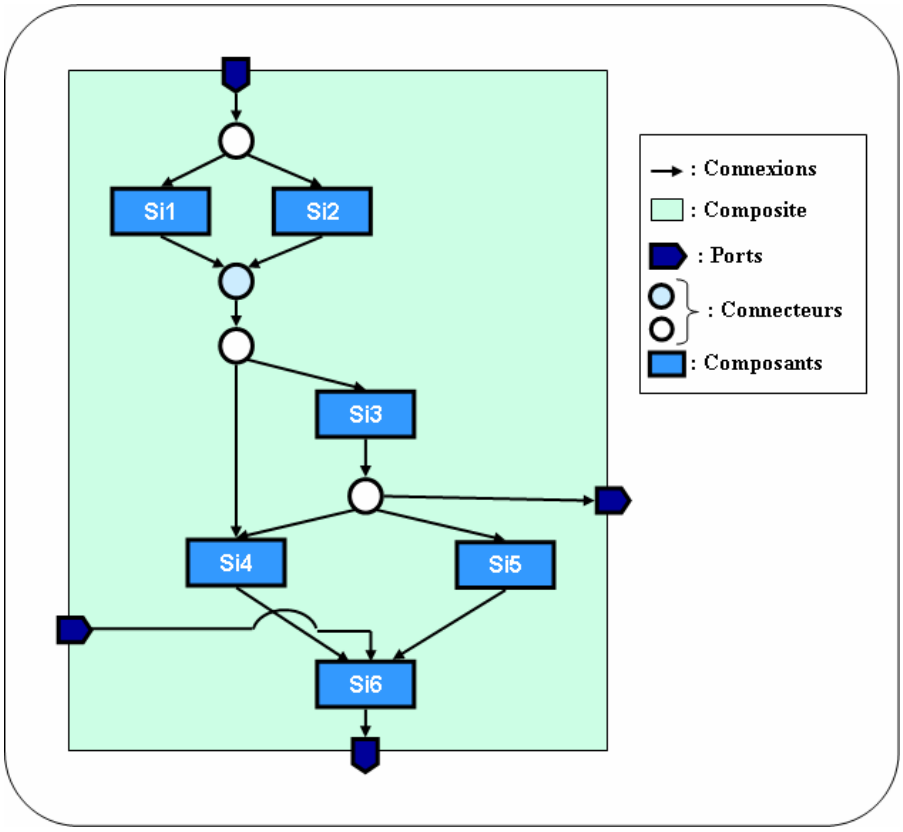



**Figure 5-2 : Graphe de services internes en vue composant-connecteur**

Rappelons qu'un composant est une entité qui donne une fonctionnalité de base d'un système. Les services internes sont représentés comme des composants (▭) car ce sont eux qui donnent ses fonctionnalités à un instrument intelligent. Rappelons aussi que les connecteurs sont des composants spéciaux dédiés à la communication. C'est pourquoi, les nœuds Rdv (○) sont représentés comme des connecteurs. En effet, ils ne servent qu'à synchroniser les données. Ce type de connecteur attend de recevoir toutes ces entrées et envoie en sortie une concaténation de ses entrées.

Le passage du domaine de l'instrumentation intelligente à celui des architectures logicielles implique la création d'un nouveau connecteur appelé connecteur *Multicast* (○). Ces éléments Multicast sont eux aussi des connecteurs car ils ne font que redistribuer ce qu'ils reçoivent en entrée à plusieurs services internes. Ce sont les bases même d'ArchWare ADL qui nous ont obligées à créer un tel connecteur. En effet, ArchWare ADL est basé sur le π-calcul. Or, le π-calcul permet notamment d'émettre une variable sur

un canal donné. Si cette variable est envoyée, une seule entité sera capable de la recevoir (sur le canal en question). Si l'on veut gérer l'émission d'une même variable vers plusieurs entités, il faut auparavant dupliquer l'information. C'est le but du connecteur Multicast.

Enfin, si l'on considère ce graphe des services internes comme étant un composant composite (▭), on en déduit que les événements d'entrée ou de sortie (▰) peuvent être assimilés à des ports. En effet, leur rôle consiste uniquement pouvoir faire communiquer cette couche interne avec son environnement.

Comme il a été vu dans le chapitre 3 section V.A, un style composant-connecteur (C&C) générique a été implémenté au sein du projet ArchWare. Toutefois, nous n'utiliserons pas ce style car il est beaucoup trop complexe pour ce que nous voulons faire. En effet, le style C&C d'ArchWare intègre des mécanismes de gestion de la dynamicité que nous n'avons pas abordé ici et qui sont totalement inutiles pour nos travaux. Nous verrons plus tard qu'un outil ArchWare sera utilisé pour transformer le langage ASL en π-ADL. Or, partir du style C&C nous fournirait des fichiers inutilement complexe pour ce que nous désirons faire.

Nous venons de voir comment sont représentés les éléments propres au domaine de l'instrumentation intelligente dans une sémantique composant-connecteur. Dans les sections suivantes, nous allons rentrer en détail dans chaque élément architectural et voir le style architectural qui a été créé pour chacun.

## I.B. Représentation des services externes et des modes

Comme il a été précisé plus haut, les services externes sont des sous parties du graphe des services internes et les modes sont des regroupements de services externes. Ces deux types d'entités n'apportent donc pas de fonctionnalités supplémentaires (au niveau des services de base) par rapport à un graphe des services internes. En effet, tous les éléments qui appartiennent à un service externe ont déjà été créés au sein du graphe des services interne et il en est de même pour les éléments qui composent un mode (ensemble de services externes donc indirectement ensemble de services internes, de nœuds Rdv, de nœuds Multicast et d'évènements d'entrée et sortie). En fait, la seule chose que nous voulons faire au niveau de la création des services externes et des modes avec ArchWare ADL est de pouvoir vérifier leurs propriétés respectives. En partant de là, nous n'allons pas recréer des styles pour les services externes et les modes car nous ne ferions que dupliquer l'information qui se situe déjà dans le graphe des services internes.

Dans le chapitre 3 qui présente entre autre ArchWare, nous avons parlé d'un langage nommé ARL (Architecture Refinement Language). Ce langage est, comme il a été précisé, dédié au raffinement d'architectures. Dans la section suivante (section I.B.1), la notion de raffinement telle qu'elle est intégrée dans ArchWare va être présentée. Enfin, dans la section I.B.2, nous présenterons comment nous utilisons ce mécanisme de raffinement pour définir les services externes et les modes.

### I.B.1. La notion de raffinement architectural

En règle générale, une architecture logicielle est la description abstraite d'un système logiciel. Le raffinement d'une architecture logicielle permet d'obtenir une description plus « concrète » du système, c'est-à-dire une architecture de plus bas niveau pouvant être

proche d'une implémentation [Oquendo 04]. Le raffinement d'architectures logicielles constitue une base de développement des systèmes logiciels corrects. Pour cela, le raffinement d'une architecture logicielle doit garantir que l'architecture logicielle concrète est valide au regard de l'architecture logicielle initiale. Ce raffinement conduit à définir de nouveaux composants, des nouvelles inter-connexions, etc. Ceci se fait au cours du processus de développement centré architecture où sont opérées des transformations en plusieurs étapes successives :

- la réalisation d'un modèle d'architecture indépendant de toute plateforme d'implémentation, appelé modèle abstrait,
- le raffinement de ce modèle par étapes successives jusqu'à obtention d'un modèle concret,
- le choix d'une plateforme de mise en oeuvre et la génération du modèle spécifique correspondant.

En effet, les systèmes complexes ne peuvent être architecturés en une seule fois. Ils doivent donc être raffinés progressivement depuis la construction d'un modèle abstrait jusqu'à l'obtention d'un degré de détail souhaité. Nous pouvons alors distinguer deux directions pour une transformation : *verticale* et *horizontale*.

Le raffinement vertical concerne certaines parties de la spécification et est décidé par des choix liés à la plateforme d'implémentation. Il change le niveau d'abstraction sans pour autant remettre en cause la spécification initiale. Ce type de raffinement est typiquement celui utilisé dans le processus centré architecture présenté dans la figure 2.13 du chapitre 2. Il permet à chaque raffinement spécifique d'être de plus en plus précis dans la description du système. Le raffinement horizontal quant à lui, souvent appelé décomposition, n'engendre pas de changement de niveau d'abstraction mais qui ajoute de nouvelles parties à la spécification. Dans notre cas, le second type de raffinement (horizontal) sera utilisé, cependant, au lieu de rajouter des éléments à chaque raffinement, nous en enlèverons. La section suivante (section I.B.2) présente plus en détail notre approche.

## I.B.2. Notre utilisation de la notion de raffinement

Il existe plusieurs formes de raffinement possible sur un même élément architectural :

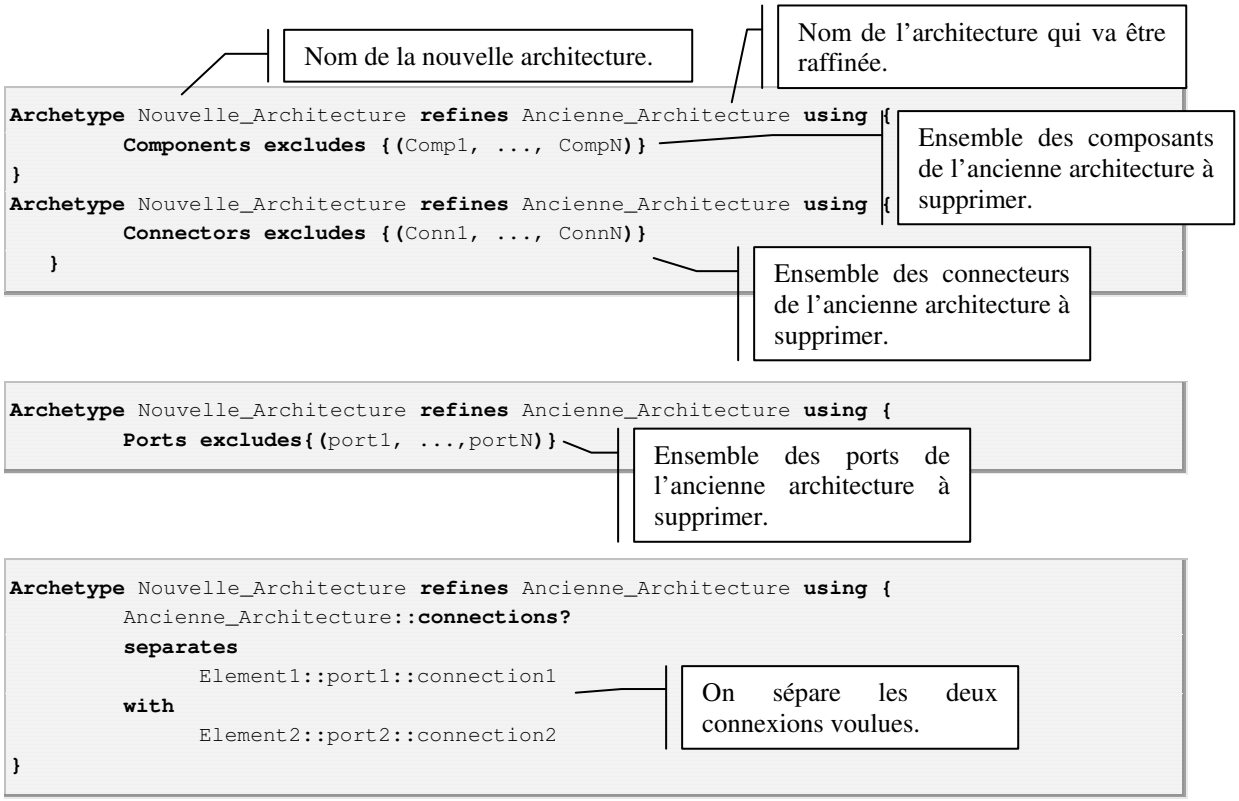
- le raffinement du comportement,
- le raffinement de l'interface,
- le raffinement de la structure,
- le raffinement des données.

La définition des services externes et des modes, dans notre modèle, fait que le raffinement qui va être utilisé est celui qui concerne la structure. En effet, les services externes et les modes peuvent être considérés comme des sous parties du graphe des services internes (voir chapitre 4). Rien ne doit être modifié au niveau du comportement des différents éléments, de leur interface ou des données qu'ils contiennent.

On dit que des *actions de raffinement* vont être effectuées pour pouvoir définir des services externes à partir d'un graphe des services internes. Ils sont au nombre de quatre :

- la suppression de composant,
- la suppression de connecteur,
- la suppression de port,
- le détachement de deux connexions.

Contrairement à l'ASL, nous n'avons pas encore présenté la syntaxe d'ARL. C'est pourquoi, celle des différentes actions de raffinement ARL que nous allons utiliser est présentée ci-dessous [Megzari 04] [Oquendo 03b] :




Si l'on désire poursuivre notre objectif de simplification du travail du concepteur d'instruments, il faut parvenir à lui fournir un langage spécifique au domaine de l'instrumentation intelligente. Nous verrons dans la section III comment cela a été réalisé.

## II. Styles du graphe des services internes et des éléments qui le compose

Dans cette partie, nous allons présenter les différents styles architecturaux qui ont été définis pour représenter un instrument intelligent dans le domaine des architectures logicielles grâce à la couche style d'ArchWare ADL (ASL). Tout d'abord, les différents styles architecturaux des éléments qui composent la couche interne seront présentés (section II.A). Les styles créés pour cette couche interne sont :

- le style des événements d'entrée (ports),
- le style des évènements de sortie (ports),
- le style des services internes (composants),
- le style des nœuds Rdv (connecteurs),
- le style des nœuds Multicast (connecteurs).

Ensuite, le style représentant la couche interne (le graphe des services internes) sera présenté (section II.B). Enfin, nous expliquerons comment va être représentée la couche externe d'un instrument intelligent, c'est à dire les services externes et les modes (section III).

Au niveau de leur constructeur, tous les styles qui vont suivre vont être plus ou moins construits de la même façon. En effet, pour chacun d'eux, nous trouverons :

- une partie déclaration,
- une partie description du comportement.

La partie déclaration peut être divisée en deux elle aussi :

- une première partie contient la déclaration des différents éléments qui sont nécessaires à l'élément qui sera construit à partir de ce style (connexions, variables, ports éventuels, etc.),
- une deuxième partie est la déclaration d'abstraction (voir chapitre 3 section V.A.1) :
    - Une abstraction récursive nommée *actif* : cette abstraction a pour fonction de gérer l'état (actif ou non) de l'élément qui va instancier le style. En fait, nous avons pu voir dans le chapitre 2 que l'exécution du code source d'un instrument était gérée par un d'automate. Ce dernier a pour but de récupérer les informations venant du réseau afin de faire évoluer le comportement de l'instrument intelligent (changement de mode, exécution d'un service externe, etc.). Cet automate est écrit en ADL pour pouvoir simuler l'exécution de l'instrument (voir section IV). Cette abstraction à pour but de gérer la valeur d'une variable booléenne qui se nomme *bActiverElement*. Cette variable va permettre d'activer ou non le comportement de l'élément auquel elle est associée. Par exemple, si un service interne donné n'appartient pas au service externe qui est en train d'être exécuté alors, cette variable sera passée à *false*. De cette manière, ce service interne sera inactif. Toute cette gestion du comportement est gérée par une autre abstraction dont le principe est expliqué dans la suite de ce chapitre.
    - Une abstraction récursive nommée *protocole* : cette abstraction contient le comportement à proprement dit de l'élément à représenter.

La partie description de comportement est en faite constituée par une composition (*compose*) des différents comportements des éléments déclarés précédemment. En fait, cette partie a pour but de composer toutes les abstractions qui ont été utilisées dans la construction du style. Elle composera au minimum les deux abstractions *actif* et *protocole*. Cette composition a pour but de lancer l'exécution du comportement contenu dans les abstractions qu'elle contient. C'est pourquoi, lorsque le style décrit contient aussi des ports (qui instancient les styles *SInPort* ou *SOutPort*), la composition exécute aussi leur comportement.

## II.A. Styles des éléments du graphe des services internes

### II.A.1. Les styles de ports : événements d'entrée ou de sortie

Comme il a été vu précédemment, un événement d'entrée ou de sortie est considéré comme un port du composant composite représentant le graphe des services internes. La figure ci-dessous (Figure 5-3), présente l'architecture d'un évènement d'entrée ou de sortie.

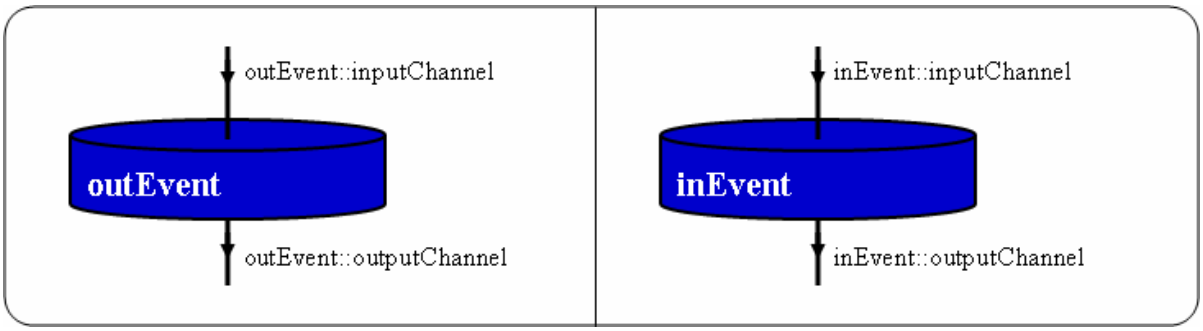



**Figure 5-3 : Structure des évènements d'entrée et de sortie**

Un événement d'entrée ou de sortie n'est, en fait, qu'un port muni d'un canal qui va servir d'interface avec le monde extérieur. D'un point de vue structurel, tous les évènements d'entrée ou de sortie sont construis de la même façon. La seule chose qui les différencie est le type de la donnée qui va circuler dans le canal.

Voici ci-dessous le style des événements d'entrée tel qu'il est décrit en ArchWare ASL.

```
SInEvent is style where {
   constructors
   {
      SInEvent is constructor(InOutType:Type);
      {
            value bActiverElement is location(false);
            value activeElement is free connection(Boolean);
            value inputChannel is free connection(InOutType);
            value outputChannel is free connection(InOutType);
            recursive value actif is abstraction();
            {
                  via activeElement receive bVal:Boolean;
                  bActiverElement := bVal;
                  actif()
            };
            recursive value protocole is abstraction();
            {
                  if ('bActiverElement) do {
                        via inputChannel receive data:InOutType;
                        via outputChannel send data
                  };
                  protocole()
            };
            compose{
                  protocole()
                  and
                  actif()
            }
      }
      as
      {
            input event with $InOutType type
      }
   }
```

Chaque événement d'entrée est paramétré par le type de donnée qu'il véhicule

Déclaration des connexions ayant pour type celui passé en paramètre.

Cet évènement d'entrée ne fait qu'envoyer des informations de type *InOutType()*

La partie *mixfix* se trouve après le mot clef *as*. C'est grâce à cette instruction que le langage spécifique au domaine est créé

```
   Constraints
   {
      to connections apply {
            forall (c1| c1.name=inputchannel and every sequence {true*.via c2 send any}
                                                leads to state {false})
      },
      to connections apply {
            forall (c2| c2.name=outputchannel and every sequence {true*.via c2 receive any}
                                                leads to state {false})
      },
      to connections apply {
            forall (c3| c3.name=activeElement and every sequence {true*.via c3 receive any}
                                                leads to state {false})
      },
      to connections apply {
            exist([1]c4| c4.mane=inputchannel) and exist([1]c5| c5.mane=outputchannel)
            and exist([1]c6| c6.mane=activeElement))
      }
   }
}
```

Après le constructeur de l'événement d'entrée, on spécifie la ou les propriétés qu'il doit respecter. Ici, la première propriété signifie que l'on ne peut que recevoir par la connexion nommée *InputChannel*. La deuxième est similaire et signifie que l'on ne peut qu'envoyer des données par la connexion nommée *outputchannel*. La troisième contraint la connexion *activeElement*.

Cette dernière propriété signifie qu'il n'existe que 3 connexions. L'une d'elle se nomme *inputchannel*, l'autre *outputchannel* et la dernière *activeElement*

L'utilisation du constructeur garantie que les contraintes sont respectées. Néanmoins, on pourrait avoir des ports déjà construits que l'on voudrait utiliser et donc vérifier. Des modifications pourraient aussi être envisagées pour le style ci-dessus. Dans ce cas, les propriétés devront a nouveau être vérifiées.

Dans le code ASL, une instruction importante est utilisée. Il s'agit de l'instruction *free*. Cette instruction peut être apparentée à l'instruction *public* de la programmation orientée objet. Cette instruction permet de spécifier si une connexion est visible depuis l'extérieur de l'abstraction à laquelle elle appartient. La figure suivante (Figure 5-4) détaille les différentes connexions d'un évènement d'entrée (seul la connexion de transport de données a été représentée. En effet, celle qui concerne la gestion du comportement est totalement transparente à l'utilisateur).

Nous verrons ce que représentent les unifications dans la section II.D. Les évènements d'entrée sont déclarés à l'intérieur de l'abstraction qui contient le code du graphe des services internes de l'instrument intelligent. Par conséquent, toutes les connexions qui appartiennent à l'événement d'entrée sont accessibles au niveau du graphe. Par contre, comme l'on désir faire en sorte que la connexion d'entrée de l'événement soit aussi accessible depuis l'extérieur du graphe, l'instruction *free* doit être utilisée.

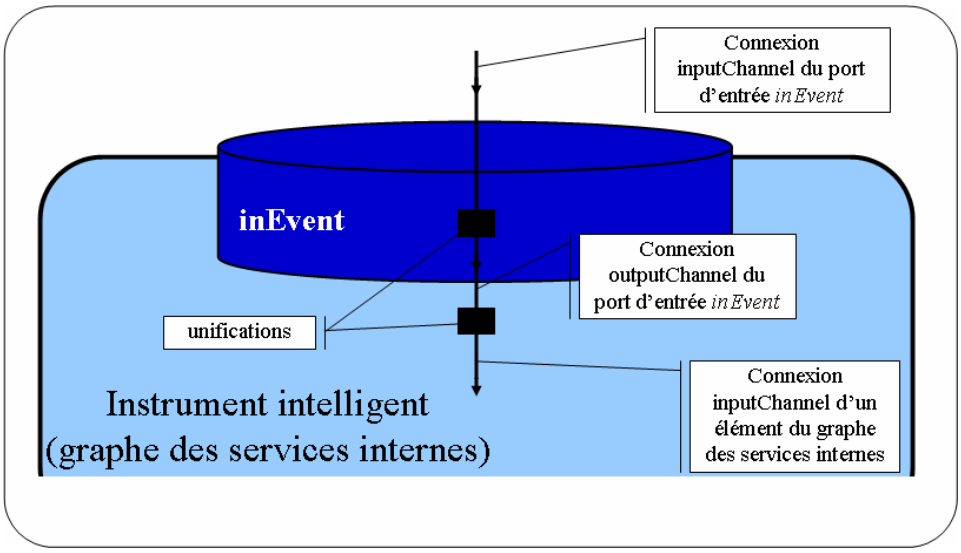



**Figure 5-4 : Détail des connections d'entrées d'un élément**

Le style *SOutEvent* est décrit de manière analogue. En fait, au niveau de la construction, un éventement d'entrée et un évènement de sortie sont deux éléments pratiquement similaires. La seule chose qui les différencie est le fait de mettre la connexion entrante *free* pour les évènements d'entrée alors que pour les évènements sortants, il s'agit de la connexion sortante (voir annexe 1).

Pour le concepteur d'instruments, la partie importante du code ASL est celle contenant la construction du langage qu'il va utiliser. Ceci est géré par la clause a*s* du code ASL. Cette définition de style est donc totalement transparente à ses yeux. La seule chose qui l'intéresse est de savoir comment l'utiliser. Ici, par exemple, si un concepteur d'instruments désir créer un événement de sortie qui transporte des booléens, il devra écrire seulement ceci :

```
value MonEvenementEntrant is output event with Boolean type;
```

*MonEvenementEntrant* est le nom de l'élément. On peut voir que l'on a mis *Boolean* comme type de sortie.

## II.A.2. Les styles des ports des composants et des connecteurs

Il a été vu précédemment que les éléments architecturaux étaient interfacés par l'intermédiaire de leurs ports (voir Figure 5-1). Or, il s'avère que ces différents ports sont construits de manière identique pour plusieurs éléments architecturaux (composant ou connecteurs). Par conséquent, deux styles spécifiques ont été créés afin de faciliter l'écriture de ces derniers. Ils correspondent aux ports d'entrée et aux ports de sortie.

D'un point de vue structurelle, les ports d'entrée et les ports de sortie ressemblent respectivement aux évènements d'entrée et de sortie. Si l'on veut faire le parallèle avec la figure du paragraphe précédent (Figure 5-3), on peut assimiler l'élément *inEvent* à un port d'entrée, l'élément *outEvent* à un port de sortie et le graphe de service interne pourra être remplacé par un composant ou un connecteur.

Le code suivant est celui du style du port d'entrée (*SInPort*)

```
SInPort is style where {
   constructors
   {
         SInPort is constructor(InOutType:Type);
         {
               value bActiverElement is location(false);
               value activeElement is free connection(Boolean);
               value inputChannel is free connection(InOutType);
               value outputChannel is connection(InOutType);

               recursive value actif is abstraction();
               {
                     via activeElement receive bVal:Boolean;
                     bActiverElement := bVal;
                     actif()
               };
               recursive value protocole is abstraction();
               {
                     if ('bActiverElement) do{
                           via inputChannel receive data:InOutType;
                           via outputChannel send data
                     };
                     protocole()
               };
               compose{
                     protocole()
                     and
                     actif()
               }
         }
   }
   Constraints
   {
      to connections apply {
            forall (c1| c1.name=inputchanel and every sequence {true*.via c2 send any}
                                                leads to state {false})
      },
      to connections apply {
            forall (c2| c2.name=outputchanel and every sequence {true*.via c2 receive any}
                                                leads to state {false})
      },
      to connections apply {
            forall (c3| c3.name= activeInPort and every sequence {true*.via c3 receive any}
                                                leads to state {false})
      },
      to connections apply {
            exist([1]c4| c4.mane=inputchanel) and exist([1]c5| c5.mane=outputchanel)
            and exist([1]c6| c6.mane=activeInPort))
      }
   }
}
```

Le constructeur du port d'entrée est aussi paramétré. En effet, tous les ports d'entrée sont identiques mise à part le fait qu'ils ne transportent pas forcément le même type de donnée

L'instruction *free* signifie que la connexion est accessible depuis l'extérieur du port.

Un port d'entrée ne fait que récupérer les valeurs en entrée afin de les redistribuées sur son canal de sortie.

On remarque que les propriétés des événements d'entrée sont valables pour les ports de sortie (mise à par pour le nom de la connexion d'activation).

On remarque que le code ASL du port d'entrée est identique à celui de l'événement d'entrée. Ceci est tout à fait normal puisque leur rôle est identique, simplement ils agissent à des endroits différents. Le port d'entrée sert d'interface aux éléments du graphe des services internes alors que l'événement d'entée sert d'interface au graphe des services internes lui même. Il aurait été possible de créer un seul et unique style pour les deux types

d'interfaces, cependant, dans un souci de clarté et de faciliter d'évolution du modèle, il a été décidé de séparés ces deux concepts.

Le code du style du port de sortie (*SOutPort*) est donné en annexe (Annexe 1) car il est construit de façon identique.

Il est intéressant de remarquer que ces deux styles n'ont pas de partie mixfix. Ceci est tout à fait normal car ils ne sont pas destinés à être utilisés par le concepteur d'instruments. En effet, ils vont simplement servir au sein d'autres styles (composant ou connecteur).

### II.A.3. Le style composant : style service interne

Tous les services internes sont des composants et suivent le même style. D'un point de vue structurel, un service interne est composé d'un port d'entrée (*inPort*), d'un port de sortie (*outPort*) ayant chacun un canal (*inputChannel*, *outputChannel*). Ils sont aussi composés d'un autre composant qui gère toute la partie fonctionnelle du service interne : l'opération. La figure ci-dessous (Figure 5-5), vous présente cette structure.

Les seules différences entre deux services internes portent sur le type des données des différents canaux et sur l'opération à effectuer (le service interne). C'est pourquoi, si l'on regarde le code du style service interne ci-dessous, on remarque que son comportement est paramètré par le type des données que l'on aura sur les canaux d'entrée et de sortie (*inType*, *outType*) et par l'opération qu'il aura à effectuer (*Operation_Java*).

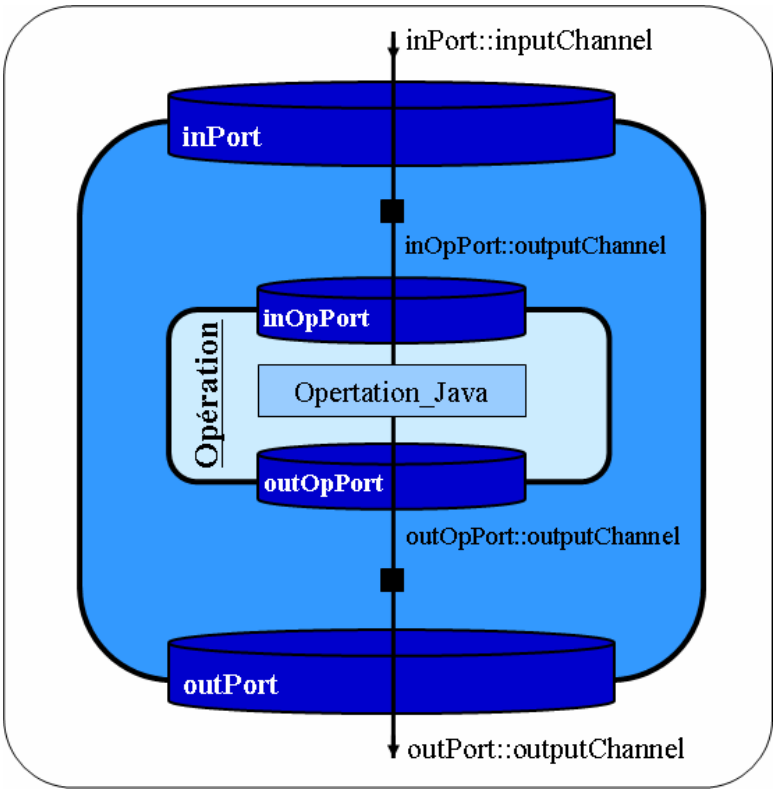



**Figure 5-5 : Structure des services internes**

Actuellement, on aurait put mettre directement l'opération à l'intérieur du service interne sans l'encapsuler dans un autre composant. Néanmoins, il a été fait de la sorte car dans un

futur proche, le modèle des instruments intelligents pourrait être modifié afin de pouvoir intégrer plusieurs opérations au sein d'un même composant service interne.

```
SCompServiceInterne is style where {
   constructors
   {
         SCompServiceInterne is constructor(inType:Type, outType:Type, op_Java:String);
         {
               value bActiverElement is location(false);
               value outPort is SOutPort(outType);
               value inPort is SInPort(inType);
               value operation is SOperation(inType, outType, op_Java);
               value activeElement is free connection(Boolean);
               value outputChannel is connection(outType);
               value inputChannel is connection(inType);
               recursive value actif is abstraction();
               {
                     via activeElement receive bVal:Boolean;
                     bActiverElement := bVal;
                     via operation::activeOperation send bVal;
                     via inPort::activeInPort send bVal;
                     via outPort::activeOutPort send bVal;
                     actif()
               };

               recursive value protocole is abstraction();
               {
                     if ('bActiverElement) do{
                           via inputChannel receive var:inType;
                           via outputChannel send outType()
                     }
                     protocole()
               };
               compose
               {
                     actif()
                     and
                     inPort()
                     and
                     outPort()
                     and
                     operation()
                     and
                     protocole()
               }
               where
               {
                     inport::outputChannel unifies inputChannel
                     outport::inputChannel unifies outputChannel
               }
         }
         as
         {
               internal service with $inType input type $outType output type
                                                   operating with $operation_Java
         }
   }
}
```

On retrouve ici les paramètres qui différencient chaque service interne

- Le composant *operation* est le composant qui contient concrètement l'opération effectuée par le service interne. (voir ci-dessous)
- On voit aussi que l'on déclare les ports d'entrée et de sortie qui sont de type *SInPort* et *SOutPort*.

Partie *mixfix* définissant le langage spécifique au domaine.

Dans ce code, nous pourrons remarquer que le type de la variable *op_Java* est *string*. En fait, il s'agit d'une chaîne de caractère qui contient le nom de la fonction java à effectuer. Ce fichier sera intégré lors de la création du code source de l'instrument intelligent.
Le style du composant opération est présenté ci-dessous :

```
SOperation is style where {
   constructors
   {
         SOperation is constructor(inType:Type, outType:Type, op_Java:String);
         {
               value bActiverElement is location(false);
               value outOpPort is SOutPort(outType);
               value inOpPort is SInPort(inType);
               value activeOperation is connection(Boolean);
               value outputChannel is connection(outType);
               value inputChannel is connection(inType);
               value nomOpJava is location(op_Java);

               recursive value actif is abstraction();
               {
                     via activeOperation receive bVal:Boolean;
                     bActiverElement := bVal;
                     via inOpPort::activeInPort send bVal;
                     via outOpPort::activeOutPort send bVal;
                     actif()
               };



               recursive value protocole is abstraction();
               {
                     if ('bActiverElement) do{
                           via inputChannel receive var:inType;
                           unobservable;
                           via outputChannel send
                     };
                     protocole()
               };
               compose
               {
                     actif()
                     and
                     outOpPort()
                     and
                     inOpPort()
                     and
                     protocole()
               }
               where
               {
                     inport::outputChannel unifies inputChannel;
                     outport::inputChannel unifies outputChannel
               }
         }
}}
```

Variable de stockage du nom de l'opération java à effectuer.

Le protocole du style opération consiste juste à récupérer une variable sur sa connexion d'entrée et à l'émettre sur son canal de sortie.

On peut voir à la fin du style du service interne qu'une partie mixfix (a*s*) est présente. Si nous voulons créer un service interne qui doit effectuer une acquisition de pression, qui a comme type de canal d'entrée un booléen, comme type de canal de sortie un flottant et

comme opération Java à exécuter « SE1.java », le concepteur d'instruments n'aura qu'à écrire :

```
value SE1 is internal service with Boolean input type
                                   Real output type
                                   operating with "SE1.java"
```

## II.A.4. Les styles des connecteurs

En ce qui concerne les connecteurs (connecteur Rdv, connecteur Multicast), le principe est à peu près le même. Dans cette section, nous allons détailler la construction de leurs styles.

### *II.A.4.a. Le connecteur Multicast*

Comme nous l'avons dit précédemment, un connecteur Multicast, d'un point de vue structurel, peut être représenté comme sur la figure ci-dessous (Figure 5-6).

Rappelons que le but d'un connecteur Multicast est de dupliquer l'information reçue sur son canal d'entrée sur ses canaux de sorties. Les seuls aspects qui peuvent différencier deux connecteurs Multicast sont le type de la donnée qui doit être transmise et le nombre de canaux de sortie. On en déduit donc que tous les canaux d'entrée et de sortie de ces connecteurs sont du même type.

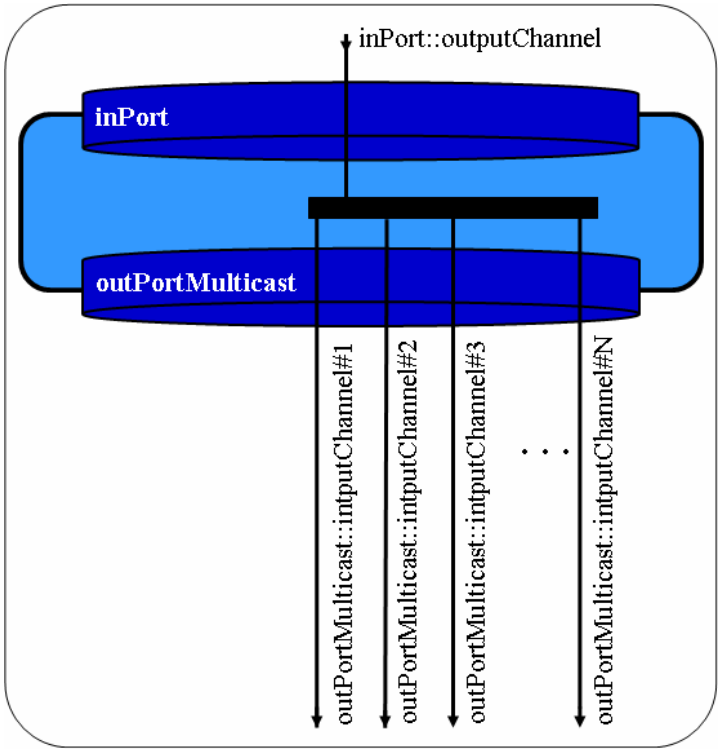



**Figure 5-6 : Structure des connecteurs Multicast**

Voici, ci-dessous, le code du style connecteur Multicast. La première chose à remarquer est le fait que le style connecteur Multicast contient un autre style qui se nomme *SOutPortMulticast*. Le mécanisme d'agrégation est donc utilisé ici. En effet, contrairement au port de sortie d'autres composants ou connecteurs (II.A.2), le port de sortie d'un

connecteur Multicast est spécifique. On peut remarquer que sa structure est différente : il contient un nombre indéfini de canaux. Etant donné que ce style de port ne sera utilisé qu'au sein de ce connecteur, il va de soit qu'il doit être incorporé à l'intérieur de ce dernier. Dans le cas où il serait utile ailleurs, plus tard, il suffirait de la sortir du style *SConnectorMulticast* pour pouvoir l'utiliser.

La clause style permet l'utilisation du mécanisme d'agrégation

Le constructeur du port de sortie du connecteur Multicast est paramétré par le type de la donnée transportée et par le nombre de canaux qu'il possède.

Ici, la variable *outputChannel* est une séquence de connexions.

Le comportement consiste, comme tous nos ports, à recopier les canaux d'entrée sur les canaux de sortie.

La partie *Constructors* du style *SConnectorMulticast* commence à partir d'ici.

Les paramètres du constructeur du connecteur Multicast sont identiques à ceux de son port de sortie

```
SConnectorMulticast is style where {
   styles
   {
         SOutPortMulticast is style where
         {
               constructors
               {
                     SOutPortMulticast is constructor(inOutType:Type, nbchannels:Integer);
                     {
                           value bActiverElement is location(false);
                           value activeOutPortMulticast is connection(Boolean);
                           value inputChannel is  sequence using nbChannels values
                                                      connection(inOutType);
                           value outputChannel is  free sequence using nbChannels values
                                                      connection(inOutType);
                           recursive value actif is abstraction();
                           {
                                 via activeOutPortMulticast receive bVal:Boolean;
                                 bActiverElement := bVal;
                                 actif()
                           };
                           recursive value protocole is abstraction();
                           {
                                 if ('bActiverElement) do{
                                       iterate sequence (#1..nbChannels) by i:Natural do
                                       {
                                             via inputChannel::i receive data:inOutType();
                                             via outputChannel::i send data
                                       }
                                 };
                                 protocole()
                           };

                           compose{
                                 protocole()
                                 and
                                 actif()
                           }
                     }
               }
         }
   }
   constructors
   {
         SConnectorMulticast is constructor(inOutType:Type, nbChannels:Integer);
         {
               value bActiverElement is location(false);
               value outPort is SOutPortMulticast(inOutType, nbChannels);
               value inPort is SInPort(inOutType);
               value activeElement is free connection(Boolean);
               value inputChannel is connection(inOutType);
               value outputChannel is  sequence using nbChannels values
                                                      connection(inOutType);
```

```
            recursive value actif is abstraction();
            {
                  via activeElement receive bVal:Boolean;
                  bActiverElement := bVal;
                  via inPort::activeInPort send bVal;
                  via outPort::activeOutPortMulticast send bVal;
                  actif()
            };
            recursive value protocole is abstraction();
            {
                  if ('bActiverElement) do{
                        via inputChannel receive data:inOutType;
                        iterate sequence (#1..nbChannels) by i:Natural do
                        {
                              via outputChannel::i send data
                        }
                  };
                  protocole()
            };
            compose
            {
                  actif()
                  and
                  inPort()
                  and
                  outPort()
                  and
                  protocole()
            }
            where
            {
                  inport::outputChannel unifies inputChannel;
                  iterate sequence (#1..nbChannels) by j:Natural do
                  {
                        outport::inputChannel::j unifies outputChannel::j
                  }
            }
      }

      as
      {
            multicast connector with $inOutType type and  $nbChannels output channels
      }
   }
}
```

Le comportement consiste à remplir un par un tous les champs de la séquence avec la valeur reçue sur le canal de sortie.

Partie *mixfix* définissant le langage spécifique au domaine.

Si le concepteur d'instruments voulait créer un connecteur Multicast nommé *Mult1* qui dupliquerait l'information de type *booléen* reçue en entrée sur *trois* canaux de sortie, il n'aurait qu'à écrire :

```
value Mult1 is multicast connector with Boolean type and 3 output channels
```

On peut voir ici que, tout comme les autres ports de composants ou de connecteurs, l'instanciation du connecteur de sortie du connecteur Multicast est transparente à l'utilisateur.

### *II.A.4.b. Le connecteur rendez-vous (Rdv)*

Comme pour les autres éléments, tous les connecteurs Rdv, d'un point de vue structurel, peuvent être représentés comme sur la figure ci-dessous (Figure 5-7).

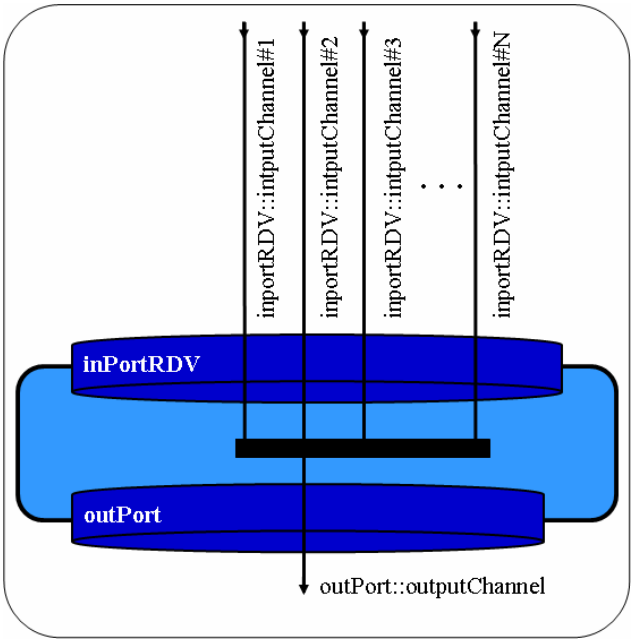



**Figure 5-7 : Structure des connecteurs RDV**

Rappelons que le but d'un connecteur Rdv est de récupérer chaque information de ses connections d'entrée, de les rassemblées au sein d'un même ensemble et de transmettre ce dernier sur son port de sortie.

Les seuls aspects qui peuvent différentier deux connecteurs Rdv sont le type de la donnée qui arrive sur chaque canal d'entrée, le nombre de ces canaux d'entrée et le type de la donnée du port de sortie.

Au niveau de ce connecteur, un problème majeur est apparu. En effet, il s'est avéré impossible de le représenter avec le langage ASL. Le problème qui se pose est que l'on ne peut pas représenter un élément qui a, à la fois, un nombre de connexion illimité et un type de donnée différent pour chaque connexion. On remarque ici que l'on devrait avoir au sein du constructeur du style nœud Rdv des paramètres qui ressembleraient à cela.

Ce paramètre contient le nombre de canaux d'entrée que doit avoir le connecteur Rdv.

Le type de chaque canal peut être différent. On a donc ici un nombre de paramètre qui n'est pas fixe.

```
SConnectorRDV is constructor(nbChannels:Natural, type1:Type, type2:Type, ..., typeN:Type);
```

La solution précédente est impossible à mettre en oeuvre car un constructeur ne peut pas avoir un nombre de paramètre non fixe. On aurait put imaginer aussi faire ceci :

Ce paramètre nous permet de régler le problème précédent en fixant le nombre de paramètre à 2 quelque soit le nombre de canaux.

```
SConnectorRDV is constructor(nbChannels:Natural, seqT:sequence[Type]);
```

Cette solution n'est pas permise car, comme il a été dit précédemment, le type de chaque canal d'entrée peut être différent. Or, au sein d'une même séquence on ne peut regrouper que des objets de même type.

La dernière solution envisagée avait été celle du *tuple*. Le tuple permet de regrouper des objets de type différents au niveau d'un même ensemble. Malheureusement, au niveau de la construction d'un tuple, il est impossible d'avoir un nombre indéterminé de type. En effet, dès sa déclaration, un tuple doit avoir une taille bien déterminée. Ceci ne nous convient pas puisque, pour nous justement, cette taille est un paramètre. Pour pouvoir arriver à construire un connecteur Rdv générique, il faut donc un objet qui regroupe les avantages de la séquence et ceux du tuple. Cet objet n'existant pas dans le langage ASL, une autre option a été implémenté. En effet, il a été décider de créer plusieurs styles pour le connecteur Rdv :

- SConnectorRDV2,
- SConnectorRDV3,
- SConnectorRDV4,
- etc.

Dans cette solution, le nombre de canaux d'entrée n'est plus un paramètre. En effet, un élément suivant le style SConnectorRDV2 aura deux connections d'entrées, un autre suivant le style SConnectorRDV3 aura 3 connections d'entrée, etc. Cette solution n'est bien entendue pas parfaite puisque nous sommes obligés de dupliquer du code ASL et que cela rajoute des types de connecteurs. Cependant, il est rare que dans un instrument intelligent, un connecteur Rdv de plus de cinq entrées soit nécessaire. Dans l'hypothèse où cela serait nécessaire, il faudrait bien entendu demander sa création à quelqu'un qui connaît le langage ASL. Chaque nouveau connecteur qui sera créé viendra alors enrichir la bibliothèque de connecteur existants ce qui comblerai rapidement les besoins.

A titre d'exemple, le code du connecteur *SConnectorRDV3* vous est présenté ci-dessous. Il est à noté qu'au même titre que le connecteur Multicast présenté précédemment, le connecteur Rdv possède un port spécial : son port d'entrée. Par conséquence, le mécanisme d'agrégation est aussi utilisé ici.

Les paramètres du port d'entrée du connecteur Rdv sont les types de chaque canal d'entrée

```
SConnectorRDV3 is style where {
   styles {
         SInPortConnectorRDV3 is style where {
               constructors
               {
                     SInPortConnectorRDV3 is constructor(type1:Type, type2:Type, type3:Type);
                     {
                           value bActiverElement is location(false);
                           value activeInPortConnectorRDV is connection(Boolean);
                           value inputChannel1 is free connection(type1);
                           value inputChannel2 is free connection(type2);
                           value inputChannel3 is free connection(type3);
```

Le port d'un tel connecteur à donc 3 canaux d'entrée et trois de sortie. Chacune de ces connexions est d'un type particulier.

```
                        value outputChannel1 is connection(type1);
                        value outputChannel2 is connection(type2);
                        value outputChannel3 is connection(type3);
                        recursive value actif is abstraction();
                        {
                              via activeInPortConnectorRDV receive bVal:Boolean;
                              bActiverElement := bVal;
                              actif()
                        };
                        recursive value protocole is abstraction();
                        {
                              if ('bActiverElement) do{
                                    via inputChannel1 receive Data1:type1;
                                    via inputChannel2 receive Data2:type2;
                                    via inputChannel3 receive Data3:type3;
                                    via outputChannel1 send Data1:type1;
                                    via outputChannel2 send Data2:type2;
                                    via outputChannel3 send Data3:type3
                              };
                              protocole()
                        };
                        compose{
                              protocole()
                              and
                              actif()
                        }
                  }
            }
      }
}
constructors
{
      SConnectorRDV3 is constructor(type1:Type, type2:Type, type3:Type);
      {
            value bActiverElement is location(false);
            value inPort is SInPortConnectorRDV3(type1, type2, type3);
            value outPort is SOutPort(tuple(type1, type2, type3));
            value activeElement is connection(Boolean);
            value outputChannel is free connection(tuple(type1, type2, type3));
            value inputChannel1 is connection(type1);
            value inputChannel2 is connection(type2);
            value inputChannel3 is connection(type3);
            recursive value actif is abstraction();
            {
                  via activeElement receive bVal:Boolean;
                  bActiverElement := bVal;
                  via inPort::activeInPortConnectorRDV send bVal;
                  via outPort::activeOutPort send bVal;
                  actif()
            }
            recursive value protocole is abstraction();
            {
                  if ('bActiverElement) do{
                        via inputChannel1 receive Data1:type1;
                        via inputChannel2 receive Data2:type2;
                        via inputChannel3 receive Data3:type3;
                        via outputChannel send tuple(Data1, Data2, Data3)
                  };
                  protocole()
            };
            compose
```

Le comportement d'un tel port consiste à récupérer la valeur de ses 3 canaux d'entrée et de la transmettre sur son canal de sortie correspondant.

Le travail d'un tel connecteur est de récupérer les valeurs venant de son port d'entrée et de renvoyer le tuple de ces valeurs en sortie

```
                {
                        actif()
                        and
                        inPort()
                        and
                        outPort()
                        and
                        protocole()
                }
                where
                {
                        inport::outputChannel1 unifies inputChannel1;
                        inport::outputChannel2 unifies inputChannel2;
                        inport::outputChannel3 unifies inputChannel3;
                        outport::inputChannel unifies outputChannel
                }
        }
        as
        {
                rdv connector 3 with $type1, $type2, $type3 input types
        }
   }
}
```

En partant du code ci-dessus, on peut voir que créer des connecteurs Rdv avec plus ou moins de canaux d'entrée n'est pas très difficile. Pour finir, si le concepteur d'instruments voulait créer un connecteur Rdv nommé *Rdv1* qui aurait *trois* canaux d'entrée ayant pour type respectif booléen, string et réel, il n'aurait qu'à écrire :

```
value Rdv1 is rdv connector 3 with Boolean, String, Real input types
```

## II.B. Style du graphe des services internes

Dans cette partie, nous allons présenter le style qui va permettre au concepteur d'instruments de décrire le graphe des services internes. Comme il a été vu précédemment, ce type de graphe est formé des divers éléments qui ont été présentés ci-dessus (les composants service interne et les connecteurs Rdv et Multicast), c'est pourquoi ce style va être construit d'une façon particulière. En effet, comme vous pouvez le remarquer dans le code de la page suivante, les paramètres du constructeur du style de la couche interne ont, pour la majorité un type particulier :

- *Sequence(meta_constituents)*
- *Expression[Behaviour]*

Un paramètre de type Expression[Behaviour] est la définition d'un comportement et non pas une instance de comportement qui s'exécute. Pour le type Sequence(meta_constituents) le principe est le même sauf que l'on déclare toujours le même type de donnée. Ces types particuliers ne peuvent être utilisés qu'en paramètre d'un constructeur. (voir section II.D). Nous ne présentons pas tout le code ici, mais nous allons simplement mettre l'accent sur deux points importants (voir ci-dessous).

Mise à part la variable *Num*, les trois autres paramètres du constructeur sont de type particulier. On retrouve dans ce constructeur tout ce qui sert à construire un graphe des services internes : les ports (évènements d'entrée et de sortie, les constituants (composants et connecteurs) et les attachement au niveau des connexions (lier la sortie d'un élément à l'entrée d'un autre : construction du graphe).

```
SGraphServiceInterne is style where {
   constructors
   {
         SGraphServiceInterne is constructor(Num:Integer, seqPorts:sequence(meta_port),
                                             seqConstituents:sequence(meta_constituents),
                                             seqAttach:Expression[Behaviour]);
         {
               value InstrumentNumber is location(Num);
               seqPorts;
               seqConstituents;
               iterate seqPorts by i:Natural do
               {
                     new seqPorts::i::name
               };
               iterate seqConstituents by i:Natural do
               {
                     new seqConstituents::i::name
               };
               seqAttach
         }
         as
         {
               internal service graph with{
                     Num($Num)
                     Interfaces{$seqPorts}
                     Constituents{$seqConstituents}
                     Links{$seqAttach}
               }
         }
   }
   constraints
   {
         -- Définition des contraintes d'un graphe des services interne en AAL
         -- voir section suivante (section II.C)
   }
}
```

La valeur du paramètre *Num* est stockée dans la variable qui contient le numéro de l'instrument sur le réseau.

Utilisation des autres paramètres du constructeur.

Dans la partie mixfix, on retrouve les différents paramètres à renseigner.

La partie *constraints* va nous permettre de définir les propriétés qui doivent être respectés par la couche graphe des services internes. En effet, nous avions vu dans le chapitre 5 section II.B.2 que la création d'un graphe des services externes était soumise à différentes règles de constructions. Celles-ci ont été transposées dans le langage de création de propriétés d'ArchWare ADL qui est AAL.

## II.C. Les propriétés d'un graphe des services internes

Les propriétés d'un graphe des services internes telles qu'elles ont été définies dans le chapitre 4 section II.B.2 sont au nombre de quatre.

La première propriété à vérifier est intrinsèque à la définition d'un graphe des services internes que l'on trouve dans le chapitre 4 section II.B.1. En effet, un graphe des services internes ne doit être composé que d'éléments de type évènements d'entrée, évènements de sortie, de services internes, de nœuds Rdv et de nœuds Multicast. Ceci se traduit par la propriété suivante :

**Propriété 5.1**
*Un graphe des services internes ne doit être composé que d'éléments de type évènements d'entrée (SInEvent), évènements de sortie (SOutEvent), connecteurs Rdv à deux, trois, quatre ou cinq entrées (SConnectorRDV2, SConnectorRDV3, SConnectorRDV4, SConnectorRDV5), connecteurs Multicast (SConnectorMulticast) et composants services internes (SCompServiceInterne).*

Voici, ci-dessous, la transformation de cette première propriété en AAL. On peut remarquer que cette propriété va être appelée à évoluée au cours du temps, notamment si l'on a besoin de connecteurs Rdv avec plus de cinq entrées. Cela est le cas de toutes les autre propriétés qui mentionnent le port d'entrée du connecteur Rdv ou le connecteur Rdv lui même.

```
forall(x | ((x in style "SOutEvent")
            or (x in style "SInEvent")
            or (x in style "SConnectorRDV2")
            or (x in style "SConnectorRDV3")
            or (x in style "SConnectorRDV4")
            or (x in style "SConnectorRDV5")
            or (x in style "SConnectorMulticast")
            or (x in style "SCompServiceInterne"));
```

Cette propriété vérifie un à un tous les éléments d'un graphe des service interne et vérifie qu'ils instancient bien un des type autorisé

La seconde propriété qui va devoir être vérifiée découle du fait que maintenant, d'un point de vue structurel, notre style de composant et nos styles de connecteurs sont tous munis d'entrées et de sorties. C'était déjà le cas dans le graphe des services internes tel qu'on le représentait dans le chapitre 4 mais tout était moins formel. Cette construction nous oblige à faire attention à de nouvelles erreurs possibles.

***Propriété 5.2***
*On ne branche pas la connexion de sortie d'un composant ou d'un connecteur sur celle d'un autre composant ou connecteur.*

Ce qui correspond au code AAL suivant :

On passe en revue tous les ports d'entrée et on vérifie que leurs canaux ne sont pas connectés à des canaux d'autres ports de sortie. Il faut penser qu'il existe plusieurs types de ports de sortie.

```
ForAll((O1 in style SOutPort or O1 in style SOutPortMulticast or O1 in style SInEvent),
       (O2 in style SOutPort or O2 in style SOutPortMulticast or O1 in style SInEvent)).
ForAll C1, C2 : Channel .
((C1 isin Channels(O1)) and (C2 isin Channels(O2))) implies not connect(C1,C2);
```

Le mot clef *connect* permet de vérifier que deux connections sont liées.

La troisième propriété est identique à la précédente mais concerne les ports d'entrée.

***Propriété 5.3***
*On ne branche pas l'entrée d'un composant ou d'un connecteur sur l'entrée d'un autre composant ou connecteur.*

Voici sa traduction en AAL :

```
ForAll (( I1 in style SInPort or I1 in style SinPortRdv2 or I1 in style SinPortRdv3
          or I1 in style SinPortRdv4 or I1 in style SinPortRdv5
          or I1 in style SOutEvent),
          I2 in style SInPort or I2 in style SinPortRdv2 or I2 in style SinPortRdv3
          or I2 in style SinPortRdv4 or I2 in style SinPortRdv5
          or I2 in style SOutEvent)) .
ForAll C1, C2 : Channel .
(C1 isin Channels(I1)) and (C2 isin Channels(I2)) implies not connect(C1,C2);
```

Passons maintenant à la transcription des propriétés du chapitre 4 :

- La propriété 4.1 dit que le graphe des services internes est un graphe orienté. Cette propriété est directement vérifiée ici par construction. En effet, comme il a été dit précédemment, des canaux d'entrée et de sortie ont été définis pour chaque composant ou connecteur du graphe des services internes. De ce fait, le graphe créé est automatiquement orienté.
- La propriété 4.2 précise qu'un graphe des services internes est un graphe connexe. Afin de pouvoir vérifier cette propriété, il faut vérifier que tous les éléments du graphe des services internes soient atteignables. De plus, d'un point de vue structurel, à la vue de la construction de chaque élément du graphe, il est bon de vérifier que tous les canaux qui sont déclarés sont utilisés. Cela évite, par exemple, de déclarer un connecteur Rdv à quatre entrées alors que seulement trois seraient nécessaires. Les propriétés vérifiées sont donc les suivantes :

> ***Propriété 5.4***
> *Tous les canaux d'entrée sont connectés à au moins un canal de sortie.*

> ***Propriété 5.5***
> *Tous les canaux de sortie sont connectés à au moins un canal d'entrée.*

> ***Propriété 5.6***
> *Tous les éléments sont atteignables.*

Ceci ce traduit en AAL de la façon suivante :

```
-- Tous les canaux d'entrée sont connectés à au moins un canal de sortie :
ForAll (O in style SOutPort or O in style SOutPortBroadcast or O in style SInEvent) .
Exists C1, C2 : Channel .
Exists (I in style SInPort or I in style SinPortRdv2 or I in style SinPortRdv3
          or I in style SinPortRdv4 or I in style SinPortRdv5
          or I in style SOutEvent) .
(C1 isin Channels(O)) and (C2 isin Channels(I)) and connect(C1,C2);

-- Tous les canaux de sorties sont connectés à au moins un canal d'entrée :
ForAll (I in style SInPort or I in style SinPortRdv2 or I in style SinPortRdv3
          or I in style SinPortRdv4 or I in style SinPortRdv5
          or I in style SOutEvent) .
Exists C1, C2 : Channel .
Exists (O in style SOutPort or O in style SOutPortMulticast or O in style SInEvent) .
(C1 isin Channels(O)) and (C2 isin Channels(I)) and connect(C1,C2);
```

```
-- Vérifier que tous les éléments sont atteignables :
ForAll (I in style SInPort or I in style SinPortRdv2 or I in style SinPortRdv3
      or I in style SinPortRdv4 or I in style SinPortRdv5
      or I in style SOutEvent) .
Exist C:Channel .
C isin Channels(I) .
Some sequence{
      True*.via I receive.True*
}leads to state{True};
```

Cette propriété est définie de façon comportementale. On vérifie que tous les ports d'entrée déclarés au sein d'un graphe des services internes ont la possibilité de recevoir un jour une donnée.

- La propriété 4.3 précise qu'il ne doit pas y avoir de rebouclage au sein d'un graphe des services internes. On peut la formuler de la même manière ici.

> ***Propriété 5.7***
> *Il n'y a pas de rebouclage au sein d'un graphe des services internes.*

Cette propriété se formule de la façon suivante en AAL :

```
ForAll I in style SInPort or I in style SinPortRdv2 or I in style SinPortRdv3
       or I in style SinPortRdv4 or I in style SinPortRdv5 .
Exist C1:Channel .
C1 isin Channels(I1) .
Every sequence{
      True*.via I1 receive.True*.via I1 receive
}leads to state{False};
```

Cette propriété est vérifiée tout simplement en regardant si l'on ne passe pas plusieurs fois par la même connexion d'entrée. On remarque que les évènements de sortie ne sont pas concernés par cette recherche car il nous font sortir du graphe (il est donc impossible de reboucler après un tel port).

- Enfin, la propriété 4.4 précise qu'un graphe des services internes doit obligatoirement commencer par un évènement d'entrée et finir par un événement de sortie. Ceci est vérifié par construction pour peut que l'on ait de tels éléments. La propriété est donc la suivante.

> ***Propriété 5.8***
> *Il existe au moins un connecteur d'entrée et un connecteur de sortie au sein d'un graphe des services internes.*

Ceci se traduit en AAL par le code suivant :

```
      Exists IE in style SInEvent and exists OE in style SOutEvent;
```

Toutes ces propriétés permettent au concepteur d'instruments d'être sûr qu'il a bien construit un graphe des services internes tel qu'il a été définit dans le modèle. Les différents styles construits jusqu'ici vont donc permettre au concepteur d'instruments de construire des graphes des services internes corrects. Dans la section suivante, l'utilisation de ces différents styles va être présentée.

## II.D. Utilisation des styles de la couche interne

Afin de vous présenter un exemple d'utilisation des styles de la couche graphe des services internes, nous allons prendre celui qui nous suit depuis le début (Figure 5-8, ci-dessous).

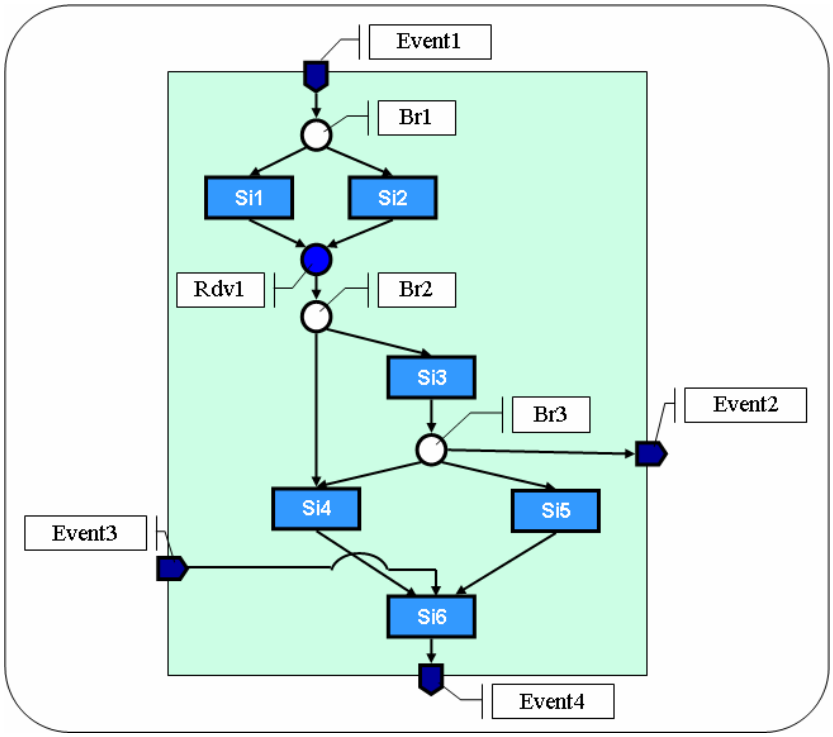



**Figure 5-8 : Exemple de graphe des services internes**

Dans cette figure, les différents éléments ont été nommés pour une plus grande compréhension du code qui va suivre. En ce qui concerne le type des canaux, ils ont été choisis de façon arbitraire. Néanmoins, la concordance des types dans le graphe a été respectée. Au niveau du nom des classes java de chaque service interne, il a été décidé dans cet exemple de le laisser identique à celui du service interne. Voici donc, ci-dessous, ce qu'aurait à écrire un concepteur d'instruments s'il avait à concevoir un tel graphe des services internes.

```
Value InternalServiceExample is internal service graph with {
   Num{1}
   Interfaces{
         value Event1 is input event with Boolean type;
         value Event2 is output event with tuple(Real,Real) type;
         value Event3 is input event with Real type;
         value Event4 is output event with Real type
   }

   Constituents{
         value Si1 is internal service with Boolean input type
                                            Real output type
                                            operating with "Si1.java";
         value Si2 is internal service with Boolean input type
                                            Real output type
                                            operating with "Si2.java";
         value Si3 is internal service with tuple(Real,Real) input type
                                            tuple(Real,Real) output type
```

Déclaration des évènements d'entrée et de sortie.

Déclaration des différents services externes, nœuds Multicast et nœuds Rdv

```
                                        operating with "Si3.java";
        value Si4 is internal service with tuple(Real,Real) input type
                                        Real output type
                                        operating with "Si4.java";
        value Si5 is internal service with tuple(Real,Real) input type
                                        Real output type
                                        operating with "Si5.java";
        value Si6 is internal service with Real input type
                                        Real output type
                                        operating with "Si6.java";
        value Mult1 is multicast connector with Boolean type and 2 output channels;
        value Mult2 is multicast connector with tuple(Real,Real) type and 2
                                                        output channels;
        value Mult3 is multicast connector with tuple(Real,Real) type and 3
                                                        output channels;
        value Rdv1 is rdv connector 2 with Real, Real input types
   }
   Links{
        Event1::outputChannel unifies Mult1::inPort::inputChannel;
        Mult1::outPort::outputChannel::1 unifies Si1::inPort::inputChannel;
        Mult1::outPort::outputChannel::2 unifies Si2::inPort::inputChannel;
        Si1::outPort::outputChannel unifies Rdv1::inPort::inputChannel::1;
        Si2::outPort::outputChannel unifies Rdv1::inPort::inputChannel::2;
        Rdv1::outPort::outputChannel unifies Mult2::inPort::inputChannel;
        Mult2::outPort::outputChannel::1 unifies Si4::inPort::inputChannel;
        Mult2::outPort::outputChannel::2 unifies Si3::inPort::inputChannel;
        Si3::outPort::outputChannel unifies Mult3::inPort::inputChannel;
        Mult3::outPort::outputChannel::1 unifies Si4::inPort::inputChannel;
        Mult3::outPort::outputChannel::2 unifies Si5::inPort::inputChannel;
        Mult3::outPort::outputChannel::3 unifies Event2::inputChannel;
        Si4::outPort::outputChannel unifies Si6::inPort::inputChannel;
        Si5::outPort::outputChannel unifies Si6::inPort::inputChannel;
        Event3::outputChannel unifies Si6::inPort::inputChannel;
        Si6::outPort::outputChannel unifies Event4::inputChannel
   }
}
```

Construction du graphe par unification des connexions

Tout d'abord, on remarque que, bien entendu, le langage spécifique au domaine est utilisé. On retrouve ainsi les quatre parties qui composent le style d'un graphe des services internes :

- Num : qui va contenir le numéro de l'instrument sur le réseau,
- Interfaces : qui va contenir les différents évènements d'entrée et de sortie,
- Constituents : qui va contenir les composants et les connecteurs d'un tel graphe,
- Links : qui va contenir le code qui va permettre de lier les différents éléments entre eux afin de construire un graphe.

Les trois dernières parties sont les plus intéressantes. Tout d'abord, étant donné que dans la partie *Interfaces* on déclare les évènements d'entrée et de sortie, le langage spécifique au domaine créé pour chacun de ces éléments est utilisé. Il en est de même pour la partie *Constituents* ou le langage spécifique créé pour les composants et les connecteurs est aussi utilisé. La dernière partie, *Links*, est plus spécifique. En fait, cette dernière partie permet de relier entre eux les différents éléments qui ont été créés dans les autres sections. Il s'agit en fait de la partie qui va concrètement créer la structure du graphe des services internes. Pour ce faire, il suffit de relier les connexions des différents éléments qui composent le graphe de manière adéquate. Afin de relier ces connexions entre elles, le mot clef *unifies* est utilisé. Cette instruction ne fait plus, en fait, qu'une seule connexion à partir de deux

autres. De plus, chaque connexion qui doit être unifiée à une autre doit pouvoir être identifiée. Pour ce faire, ArchWare ADL a mis en place un mécanisme qui pourrait s'apparenter à celui que l'on retrouve dans la programmation orientée objet en accèdent par exemple à une variable à l'intérieur d'une classe grâce au « . » :

```
NomClasse.NomVariable
```

Dans le cas d'ArchWare ADL, le mécanisme est le même. La seule chose qui diffère est que l'on utilise « :: » à la place du « . ». Par conséquent, étant donné que les différentes connexions sont situées à l'intérieur des ports et que ces derniers sont situés dans les différents composants ou connecteurs, on accède à une connexion comme ceci :

```
NomElement::NomPort::NomConnexion
```

Il y a toutefois un cas particulier à cette syntaxe qui est celui des évènements d'entrée et de sortie. En effet, étant donné que ces derniers sont équivalents à des ports, et que, donc, les connexions sont directement accessibles à travers eux, il suffit d'écrire :

```
NomEvenement::NomConnexion
```

La dernière chose à remarquer est que pour accéder à une connexion particulière pour les connecteurs Rdv ou les connecteurs Multicast, il suffit de rajouter le numéro de la connexion que l'on désir utiliser. Par exemple, la première connexion sera identifiée de la manière suivante :

```
NomConnexion::1
```

On peut voir par cet exemple que créer le graphe des services internes d'un instrument intelligent est une tâche assez intuitive pour une personne qui n'est pas informaticienne mais spécialiste du domaine de l'instrumentation intelligente car le langage utilisé reprend les termes de ce dernier. De plus, toutes les propriétés que doit respecter un tel graphe étant implémentées, le concepteur d'instruments ne peut pas créer de graphes des services internes qui ne soient pas corrects. Le seul point qui peu paraître un peu plus délicat à une personne qui n'est pas informaticienne est la dernière partie concernant la liaison entre les connexions.

Nous venons de voir comment a été construite la couche graphe des services internes. En effet, nous vous avons présenté comment étaient construits les divers éléments architecturaux de cette couche (d'un point de vue structurel et comportemental) et comment nous avions pu créer le langage spécifique au domaine. Enfin, nous avons montré comment utiliser ce langage pour créer la couche interne d'un instrument donné. Nous allons maintenant présenter comment sont créés les services externes et les modes.

## *III. Les services externes et les modes*

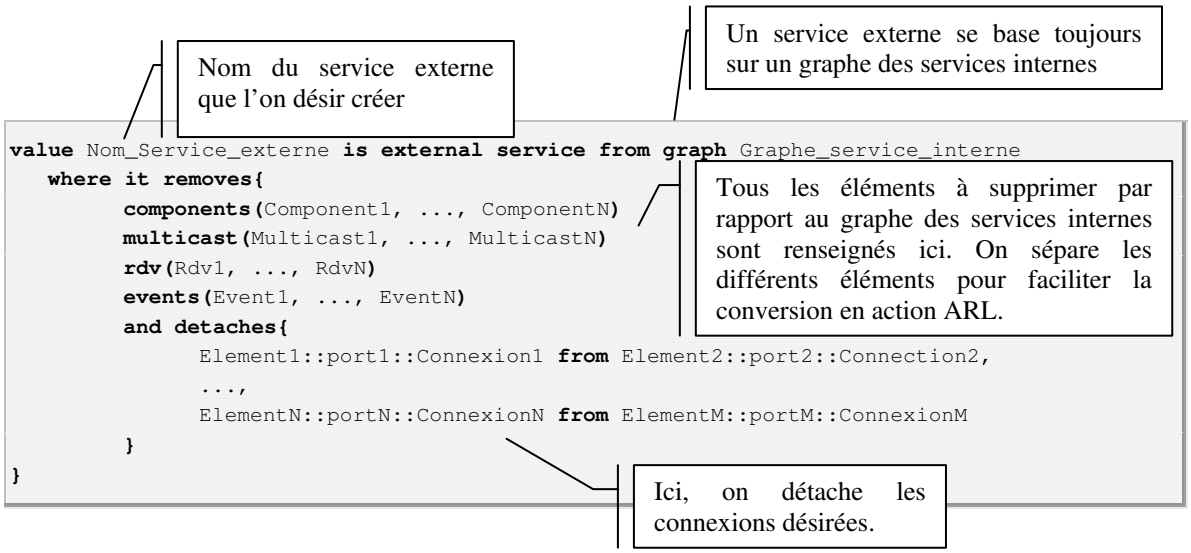
Il a été vu dans la section I.B que les services externes et les modes étaient obtenus en raffinant la structure du graphe des services internes. Les actions de raffinement qui doivent être utilisées sont écrites en ARL. Or, notre politique est de fournir aux concepteurs d'instruments des langages spécifiques au domaine de l'instrumentation intelligente. Dans cette optique, un autre langage a été créé, avec une syntaxe spécifique et qui se base sur ARL. L'approche centrée architecture ne permet pas la création d'un langage spécifique pour spécifier des règles de raffinement, un outil spécifique à dû être conçu (voir chapitre 6 section III.B). Le problème est donc que, si un concepteur d'instruments trouve que le langage spécifique ne lui convient pas, il faudra créer un

nouveau compilateur pour la nouvelle syntaxe. Ce travail devra bien entendu être fait par un programmeur. Cependant, en utilisant par exemple un langage de programmation comme JavaCC, ce type d'outils peut être réalisé facilement et rapidement.

La section suivante présente les deux langages qui ont été créés (section III.A). Ensuite, les différentes propriétés qui doivent être vérifiées pour chaque service externe et pour chaque mode seront transposées en AAL (section III.B). Enfin, un petit exemple d'utilisation des langages sera présenté (section III.C).

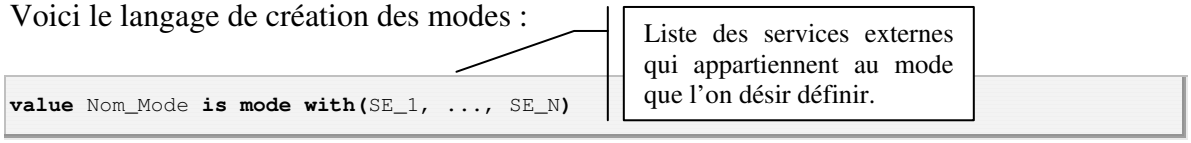
## III.A. Langages de création des services externes et des modes

Ci-dessous ce trouve la syntaxe du langage spécifique qui a été créé pour remplacer les actions de raffinement ARL. Tout d'abord, voici le langage de construction d'un service externe :




On retrouve dans cette syntaxe qu'un service externe est une sous partie d'un graphe des services internes particulier auquel on peu supprimer différents éléments et séparer des connexions. Bien entendu, certaines clauses ne sont pas obligatoires. Par exemple, si aucun détachement de connexion spécifique ne doit être fait, la clause *and detach* pourra être enlevée. Il en est de même si l'on ne désir pas enlever tel ou tel type d'élément. Il faut préciser aussi que lorsque l'on supprime un élément, les connexions dont il dispose sont aussi supprimées. Par conséquence, la partie qui permet de détacher des connexions n'est la que pour permettre de détacher des connexions d'éléments

Voici le langage de création des modes :




## III.B. Les propriétés à vérifier

Il a été vu dans le chapitre 4 que la construction des services externes et des modes sont eux aussi soumis à différentes règles. Comme il a été fait précédemment pour le graphe des services internes, nous allons les présentées dans cette section. Tout d'abord, les propriétés des services externes serons listées (section III.B.1) puis ce sera au tour des propriétés des modes (section III.B.2).

### III.B.1. Propriétés des services externes

Nous avons vu que les propriétés à vérifier pour qu'un service externe soit construit de façon correcte sont au nombre de quatre. Nous allons donc les lister une à une et les formuler de façon à pouvoir les écrire en AAL :

- La première propriété qui doit être vérifiée est la propriété 4.5 qui impose qu'un graphe des services externes doit être un graphe connexe. Cette propriété est vérifiée au niveau du graphe des services internes par la propriété 5.6. Deux autres propriété (5.4 et 5.5) doivent aussi être vérifiées. La propriété 5.4 concerne le fait que toutes les connexions d'entrée d'un élément doivent être reliées à au moins une connexion de sortie. Cette propriété doit toujours être vérifiée et est complétée par la propriété 5.12 ci-dessous. Par contre, la propriété 5.5 n'est plus vraie pour un service externe. En effet, les connecteurs Multicast doivent pouvoir avoir des connexions non utilisées (voir le service externe SE2 de la figure 4-9 du chapitre 4). Les propriétés à vérifiées ici sont donc les suivantes :

> ***Propriété 5.9***
> *Tous les canaux d'entrée des éléments d'un service externe sont connectés à au moins un canal de sortie.*

> ***Propriété 5.10***
> *Tous les éléments d'un service externe sont atteignables.*

La représentation AAL de ces trois propriétés est identique à celle qui concerne le graphe des services internes.

- La seconde propriété à vérifier est la propriété 4.6. Cette propriété est elle aussi identique à une propriété à vérifier pour la construction des graphes des services internes.

> ***Propriété 5.11***
> *Il existe au moins un connecteur d'entrée et un connecteur de sortie au sein d'un service externe.*

La représentation AAL est aussi identique à celle de la propriété sur les graphes des services internes.

- La propriété suivante à vérifier est la propriété 4.7. Cette propriété interdit de pouvoir avoir deux connexions unifiées à une troisième.

> ***Propriété 5.12***
> *Une connexion d'entrée d'un élément d'un service externe ne peut être unifiée qu'à une seule connexion de sortie d'un autre élément de ce même service externe.*

La représentation AAL de cette propriété est donc la suivante :

```
ForAll (I in style SInPort or I in style SinPortRdv2 or I in style SinPortRdv3
          or I in style SinPortRdv4 or I in style SinPortRdv5
          or I in style SOutEvent) .
```

```
Exists C1, C2, C3 : Channel .
Exists (O1 in style SOutPort or O1 in style SOutPortMulticast or O1 in style SInEvent) .
Exists (O2 in style SOutPort or O2 in style SOutPortMulticast or O2 in style SInEvent) .
(C1 isin Channels(O1)) and (C2 isin Channels(I)) and
(C3 isin Channels(O3)) and connect(C1,C2) .
not connect(C3,C2)
```

- La dernière propriété à vérifier pour les services externes est la propriété 4.8. Cette propriété est dorénavant vérifiée grâce aux propriétés précédentes (notamment grâce aux propriétés 5.9 et 5.11.

Le respect de ces différentes propriétés garanties donc au concepteur d'instruments la création de services externes corrects. Dans la section suivante, les propriétés que doivent respectées les mode vont être à leur tour exposées.

### III.B.2. Propriétés des modes

Une seule propriété est à vérifiée lorsque l'on désir créer un mode. Il s'agit de la propriété 4.9. Cette propriété est formulé de façon identique à la propriété 5.12 concernant les services externes et est donc la suivante :

> ***Propriété 5.13***
> *Une connexion d'entrée d'un élément d'un mode ne peut être unifiée qu'à une seule connexion de sortie d'un autre élément de ce même mode.*

Sa formulation AAL est identique à celle de la propriété 5.12. Si cette propriété est vérifiée, le concepteur d'instruments est sur d'avoir conçu des modes de fonctionnement corrects.

Nous venons de voir dans ces différentes sections comment se construisait ce dont allait se servir les concepteurs d'instrument pour pouvoir créer des instruments intelligents qui respectent le modèle qui est utilisé. Dans la section suivante, un bref exemple d'utilisation va être présenté.

## III.C. Utilisation des langages créés

Afin d'illustrer l'utilisation des langages de création des services externes et des modes, l'instrument fictif qui nous a servi d'exemple dans le chapitre 4 et dont le graphe de services internes à été construit dans la section II.D précédente va être utilisé. La figure ci-dessous (Figure 5-9) présente les trois services externes qui doivent être créés.

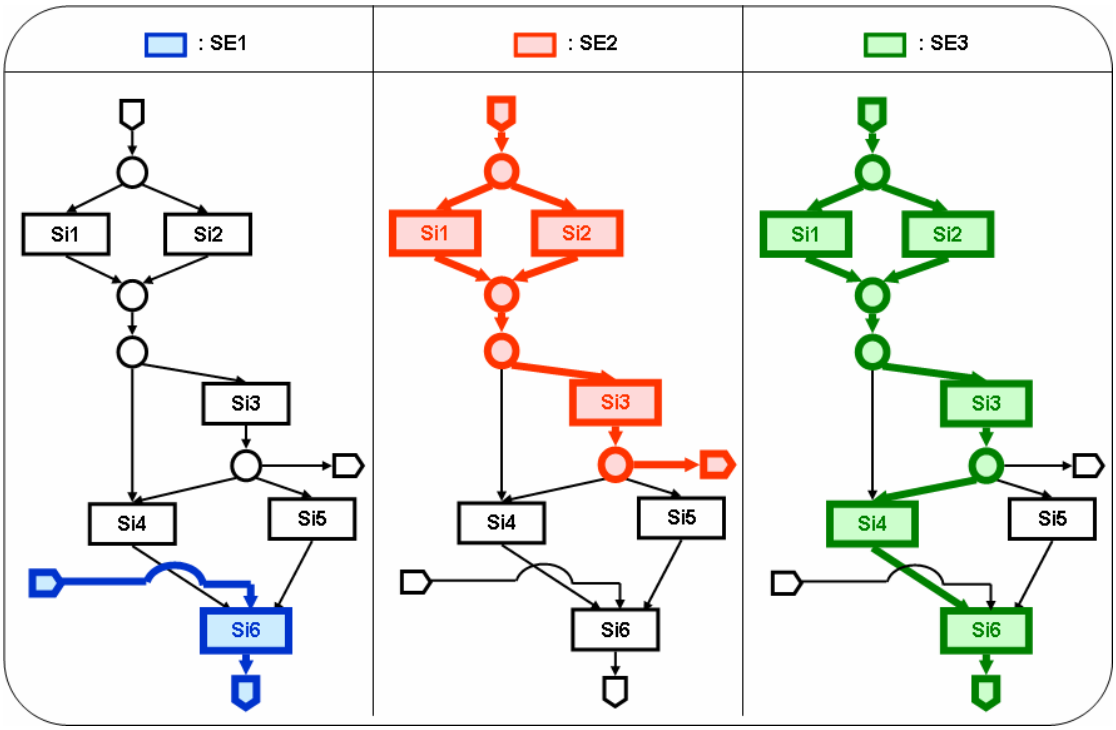



**Figure 5-9 : Services externes à concevoir**

Ces trois services externes sont corrects car ils respectent toutes les propriétés des services externes. Le code de création de ces services externes est le suivant :

```
value SE1 is external service from graph InternalServiceExample where is remove{
   components(Si1, Si2, Si3, Si4, Si5)
   multicast(Mult1, Mult2, Mult3)
   rdv(Rdv1)
   events(Event1, Event2)
};
value SE2 is external service from graph InternalServiceExample where is remove{
   components(Si4, Si5, Si6)
   events(Event3, Event4)
};
value SE3 is external service from graph InternalServiceExample where is remove{
   components(Si5)
   events(Event2, Event3)
   and detach{
         Mult2::outPort::outputChannel::2 from Si4::inPort::inputChannel
   }
}
```

Dans le chapitre 4, nous avions put voir que l'on avait plusieurs possibilités de création de mode pour ces services externes. Nous n'allons présenter seulement celle ou l'on place SE1 et SE2 dans un même mode (Mode1) et SE3 dans un autre (Mode2). Voici donc ce que devrait écrire un concepteur d'instruments :

```
value Mode1 is mode with(SE1,SE2) ;
value Mode2 is mode with(SE3)
```

# IV. Le comportement d'un instrument intelligent

Une abstraction particulière est créée pour chaque instrument afin de pouvoir simuler son comportement à l'aide d'un outil créé pendant le projet ArchWare : l'Animator. Cette abstraction est l'automate qui va gérer le comportement global des instruments intelligents. Le principe de cette simulation est de pouvoir simuler l'exécution et le changement des services externes mais aussi le changement de mode. En effet, un instrument intelligent créé par le concepteur d'instruments a beau être correct du point de vue du modèle, il n'en demeure pas moins possible que ce dernier ne face tout de même pas ce que le concepteur d'instruments pensait. Afin de vérifier ceci, il faut vérifier que les services externes créés et les modes créés sont les bons. Au travers de cette vérification, on peut aussi voir plus globalement si le graphe des services internes est correct. Le comportement qui va être simulé ne va bien entendu pas intégrer les opérations Java qui auront été créés par le programmeur puisque la partie matérielle de l'instrument intelligent n'est pas présente lors de cette simulation.

Comme il a été précisé plus haut, la première chose à faire pour pouvoir simuler le comportement du changement de service externe et de mode est de pouvoir gérer l'activation des différents éléments du graphe des services internes qui les composent. Pour ce faire, notre mécanisme est basé sur la gestion de variables booléennes. En effet, on peut remarquer que tous les styles créés au long de ce chapitre contiennent une abstraction spéciale nommée *Actif*. Le but de cette abstraction est de gérer l'activation ou la non activation de l'élément à laquelle elle appartient. Pour ce faire cette abstraction *Actif* met à *True* ou à *False* une variable booléenne nommée *bActiverElement* qui est déclarée pour chaque élément. L'objectif de cette variable est de permettre ou non l'exécution du comportement qui se trouve dans l'abstraction *Protocole* (qui est l'abstraction qui contient le comportement de chaque élément).

```
recursive value protocole is abstraction();
{
   if ('bActiverElement) do{
         -- comportement de l'élément
   };
   Protocole();
}
```

Pour ce faire, une condition est donnée à l'exécution du comportement de chaque élément (voir ci-dessus). Cette activation est donc simplement conditionnée par la valeur de la variable *bActiverElement*.

Pour pouvoir mettre à jour la variable *bActiverElement*, chaque élément (composant ou connecteur) possède une connexion particulière dédiée nommée *activeElement*. Cette connexion est une connexion d'entrée. Dès qu'une valeur est envoyée à cette connexion, elle est transmise à tous les éléments qui composent le composant ou le connecteur (ports d'entrée, ports de sortie, opération). Cette valeur est ensuite affectée à la variable *bActiverElement* de chaque élément concerné. Si l'on regarde le code de chaque élément, on remarque que cette connexion *activeElement* n'est pas accessible à travers les ports des composants ou des connecteurs mais directement à travers le corps de ces derniers (voir Figure 5-10 ci-dessous).

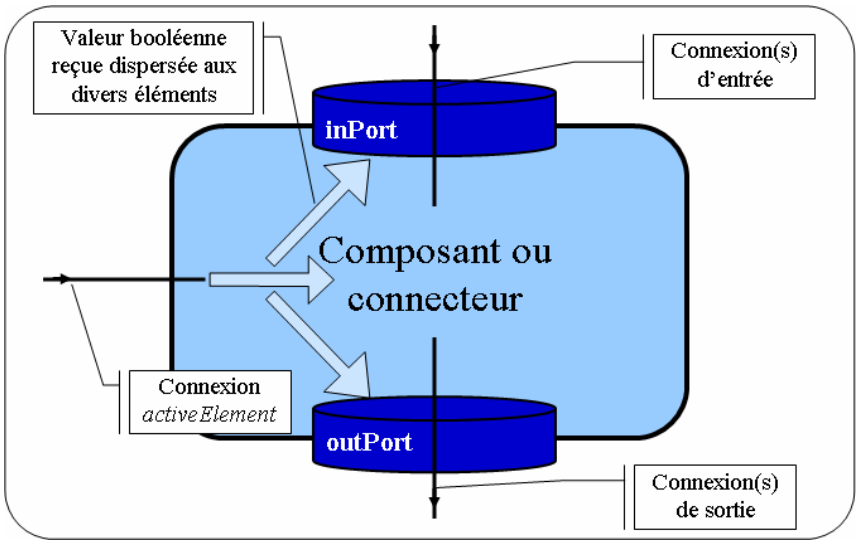



**Figure 5-10 : Dispersion de la valeur booléenne *bActiverElement***

Le choix qui a été fait de ne pas mélanger la connexion *bActiverElement* aux autres à tout simplement été motivé par une volonté de clarté. C'est-à-dire que l'on a voulu séparer les connexions qui appartenaient réellement au graphe des services internes des autres.

Grâce à ce mécanisme, on est maintenant capable de choisir quel élément d'un graphe des services internes doit être activé ou non en fonction du service externe qui est déclenché. Maintenant, le changement de mode et de service externe va être abordé. Pour ce faire, une gestion du comportement par des variables booléennes a aussi été adoptée. En fait, chaque service externe a deux variables booléennes qui lui sont associées :

- Une du type *b_SE_Activable_NomServiceExterne* : cette variable va déterminer si un service externe est activable ou non. Cela revient à savoir si l'on est dans le bon mode ou pas.
- Une autre du type *b_SE_Actif_NomServiceExterne* : cette variable permet de définir l'activation ou non d'un service externe.

Ces différentes variables sont positionnées par différentes abstractions (voir le code ASL des différents styles présentés précédemment) :

- Une abstraction *setMode* qui positionne les variables booléennes de type *b_SE_Activable_NomServiceExterne* en fonction du mode d'exécution qui a été choisi.
- Une abstraction *setService* qui positionne les variables booléennes de type *b_SE_Actif_NomServiceExterne* en fonction du ou des service(s) externe(s) qui doit/doivent être exécuté(s).
- Une abstraction *reset_ServiceExterne* qui comme son nom l'indique arrête tous les services externes (par exemple en cas de changement de mode).

Une autre abstraction a été créée. Il s'agit d'une abstraction qui permet de mettre à jour la variable *bActiverElement* de chaque composant ou connecteur présent dans le graphe des services internes et qui permet aussi de détacher ou de rattacher certaines connexions en fonction du service externe qui désire être exécuté. Cette abstraction se nomme *gestion_entree_element*. Grâce à cette dernière abstraction, l'exécution des services externes est maintenant possible.

Une dernière abstraction doit être créée pour lancer le mécanisme. Actuellement, un problème se pose. En effet, l'outil Animator qui devra interpréter tout ceci n'est qu'un prototype qui a été créé pour les besoins du projet ArchWare. Or, les besoins que nous avons ici sont bien supérieurs car il faudrait que l'on soit capable d'interagir avec cet outil pour lui dire quel mode on veut activer, quel service(s) externe(s) on souhaiterait exécuter, etc. Ce genre de fonctionnalité ne fait pas partie des spécifications initiales car elles sont très spécifiques au domaine de l'instrumentation intelligente. Cette dernière abstraction n'est donc actuellement pas spécifiée.

Toutes ces abstractions ont des mécanismes généraux qui sont identiques pour tous les instruments intelligents. Mais chacune d'entre elle sera tout de même différente d'un instrument intelligent à l'autre en fonction du graphe des services interne, de services externes et des modes créés. Cela jouera sur les différentes variables booléennes. Par conséquent, cette abstraction, qui gère le comportement global d'un instrument intelligent, sera générée par un outil spécifique (voir chapitre 6 section IV.B).

# V. Conclusion

Le but de ce chapitre était de montrer comment les concepts de la conception centrée architecture pouvaient être appliqués à la conception des instruments intelligents. La première notion qui a été transposée à ce domaine est la notion de graphe des services internes. De par sa structure et sa fonction au sein d'un instrument intelligent, il a été décidé de construire cette couche en utilisant le concept de style et plus particulièrement le style Composant-Connecteur. Cette notion de style architectural a de multiples avantages dans la conception d'instruments intelligents qui sont principalement la construction d'un langage spécifique au domaine et la possibilité d'exprimer différentes contraintes qui seront vérifiées en phase de conception.

Ensuite, le problème de la représentation des services externes et des modes s'est posé. La solution proposée est fondée sur le raffinement architectural. Contrairement au principe de style, aucun mécanisme de création de langage spécifique n'existe dans cette notion et dans son implémentation au sein d'ArchWare ADL. En effet, le langage de raffinement est utilisé normalement pour faire évoluer une architecture afin de la rendre plus concrète ce qui fait que ce type de langage est déstiné à des spécialistes de la conception centrée architecture qui n'ont pas besoins de langage spécifique. C'est pourtant ce que nous devons fournir pour que le concepteur d'instruments n'ait pas de difficulté lors de la conception. Ce langage a donc été créé par nos propres moyens et un outil spécifique a été conçu pour passer de ce langage à ARL. Ce nouveau langage est aussi un langage formel puisqu'il se base sur ARL.

Une fois toute la structure d'un instrument intelligent représentée, le comportement de ce dernier restait à être modélisé et ceci afin de pouvoir simuler son comportement. Simuler le comportement d'un instrument intelligent revient à simuler le changement de mode, le changement de service externe et l'enchaînement des différents services internes au sein d'un service externe. Pour parvenir à simuler tout ceci, un mécanisme de gestion de variable booléenne a été mis en place directement au niveau des styles et une abstraction spéciale, spécifique à chaque instrument, est générée automatiquement pour chaque simulation.

La suite de ce manuscrit va présenter le nouveau processus de conception des instruments intelligents. En effet, dorénavant, les outils ArchWare vont devoir être intégrés afin de pouvoir utiliser les différentes notions utilisées.

# Chapitre 6 : PROCESSUS DE CONCEPTION

# Chapitre 6 : Processus de conception

# Processus de conception

Dans ce chapitre, nous allons présenter en détail le nouveau processus de conception centré architecture adapté pour la conception des instruments intelligents. Pour ce faire, des outils fournis par le projet ArchWare ont été intégrés et combinés avec d'autres outils spécialement conçus pour ce nouveau processus.

Dans la suite de ce chapitre, nous allons tout d'abord présenter une vue d'ensemble du processus tel qu'il est perçu par le concepteur d'instruments intelligents (section I). Ensuite, nous entrerons dans le détail et chaque partie sera expliquée (section II à V).

## *I. Le processus de conception vu par le concepteur d'instruments.*

Il ne faut pas perdre de vue qu'un des objectifs de ces travaux est de faciliter le travail du concepteur d'instruments. De ce fait, le processus de conception tel qu'il est perçu par ce dernier doit être simple.

Nous avons vu dans le chapitre 3 qu'un processus de conception centré architecture classique consiste à raffiner plusieurs fois une architecture de haut niveau jusqu'à l'obtention du code source de l'application. ArchWare supporte bien entendu ce type de conception. Néanmoins, afin de respecter l'objectif précédemment cité, le problème a été abordé différemment. En effet, nous avons vu dans le chapitre précédent que les architectures qui sont créées par le concepteur d'instruments sont déjà très précises puisqu'elles sont issues de styles qui sont eux mêmes déjà très détaillés. De ce fait, un processus de conception par raffinements successifs n'est pas nécessaire. De plus, ce genre de processus de conception s'adresse plutôt à des personnes qui connaissent bien le développement centré architecture puisque généralement de nombreuses modifications doivent être faites directement au niveau du code ASL ou π-ADL.

Le processus de conception tel qu'il est vu par le concepteur d'instruments intelligent est donc le suivant (Figure 6-1) :

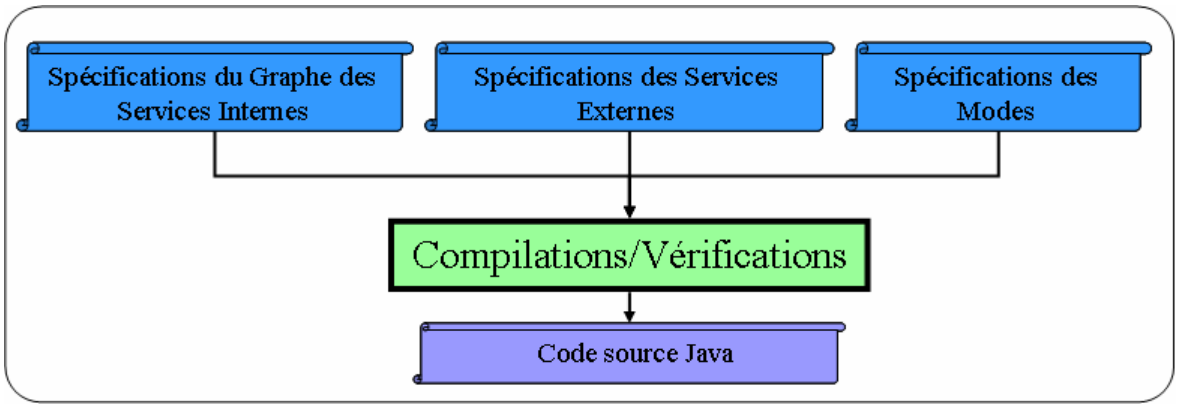



**Figure 6-1 : Processus de conception du point de vue concepteur d'instruments**

Ce processus de conception ne concerne, bien entendu, que le concepteur d'instruments intelligents. C'est pourquoi, il faut partir du principe qu'avant de l'initier, il faut que les services internes aient été définis par le programmeur.

Le concepteur d'instruments va devoir fournir trois fichiers pour définir un instrument intelligent. Il y aura un fichier qui contiendra la spécification du graphe des services internes (GSI) de l'instrument qui est en cours de conception, ensuite, le deuxième fichier contiendra la spécification de ses services externes et enfin, le dernier fichier contiendra la spécification de ses différents modes de fonctionnement. Bien entendu, les spécifications de ces différents éléments vont se faire à l'aide des styles et des règles de raffinement qui ont été présentés dans le chapitre précédent. Une fois ces trois fichiers créés, le concepteur d'instruments n'a plus qu'à lancer le processus de compilation et de vérification qui générera, si tout est correct, le code java complet de l'instrument intelligent. Si le concepteur d'instruments fait une erreur dans l'un des fichiers, il en sera alors averti et il devra la corriger afin de pouvoir générer le code java.

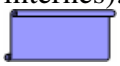
La partie « Compilation/Vérification » n'est pas aussi simple que ce que l'on peut imaginer en voyant le processus de conception du point de vue du concepteur d'instruments. En effet, toutes les vérifications qui sont faites dans une architecture se font sur du code π-ADL. Etant donné que les trois fichiers fournis par le concepteur d'instruments sont écrit dans un langage spécifique au domaine (grâce à la couche style et à la réécriture des règles de raffinement), il faut donc tout d'abord transformer les fichiers en π-ADL avant de pouvoir vérifier quoi que ce soit. C'est seulement à partir de cet instant que les différentes propriétés présentées dans le chapitre 5 vont pouvoir être vérifiées. Une fois le code source de l'instrument intelligent généré en π-ADL, une simulation de son comportement pourra être faite. Enfin, la génération du code java sera effectuée.

Concrètement, l'étape *Compilation/Vérification* de la figure précédente peut être décomposée en quatre étapes :

1. génération du code source du graphe des services internes en π-ADL et vérification des propriétés qui lui sont associées,
2. génération du code source des services externes et des modes en π-ADL puis vérification des propriétés de chacun,
3. génération du code de l'instrument intelligent complet (hors services internes java) en π-ADL et validation du comportement par animation,
4. génération du code source java de l'instrument intelligent.

Ces différentes étapes permettent de garantir au concepteur d'instruments que le logiciel qu'il est en train de créer est bien celui qui correspond à l'instrument intelligent qu'il désir concevoir. Cela signifie, tout d'abord, que le concepteur d'instruments respecte toutes les règles de construction inhérentes au modèle des instruments intelligents que nous utilisons et ensuite, que l'instrument intelligent créé a le comportement désiré et non celui d'un autre instrument.

Nous allons détailler ces quatre phases dans les sections suivantes dans l'ordre où elles sont présentées ci-dessus. Dans les figures des sections suivantes, nous avons respecté les codes suivants :

-  : Fichiers initiaux fournis par le concepteur d'instruments intelligents (spécifications du Graphe des Services Internes, spécification des services externes ou spécification des modes) ou par le programmeur (code java des services internes).
-  : Fichiers générés en cours de processus.
-  : Outils ayant été développés dans le cadre du projet ArchWare.
-  : Outils spécifiques au processus de développement des instruments intelligents. Ces outils ont été créés dans le cadre de cette thèse
- ( xxx ) : langage dans lequel est écrit le fichier.

Dans les sections suivantes, dès qu'un outil spécifique est inséré dans le processus, une brève présentation de celui-ci sera faite.

## II. Compilation et vérification du graphe des services internes

Le fichier qui contient la spécification du graphe des services internes (GSI) est donc le premier à être compilé. La figure ci-dessous (Figure 6-2) présente le processus.

Tout d'abord, nous avons vu que pour créer le fichier contenant les spécifications du graphe des services internes, nous utilisons le langage spécifique qui a été créé grâce à la couche style d'ArchWare ADL (chapitre 5 section II). Ce fichier est fourni à l'outil ArchWare nommé ASL Toolkit (chapitre 4 section V.C.2). Cet outil transforme la description du graphe des services internes, précédemment faite, en langage de base d'ArchWare ADL (π-ADL). Une fois ce fichier créé, il est automatiquement transmis à l'outil de vérification de propriétés qui est l'Analyzer (chapitre 4 section V.C.5). Ce dernier est capable de vérifier que l'architecture en π-ADL qui lui est fournie respecte bien les règles de construction d'un graphe des services internes. Pour ce faire, il doit s'assurer que toutes les propriétés fournies sont vérifiées. Si cela est le cas, on peut passer à la seconde phase du processus de conception. Sinon, un message d'erreur est transmis au concepteur qui doit corriger.

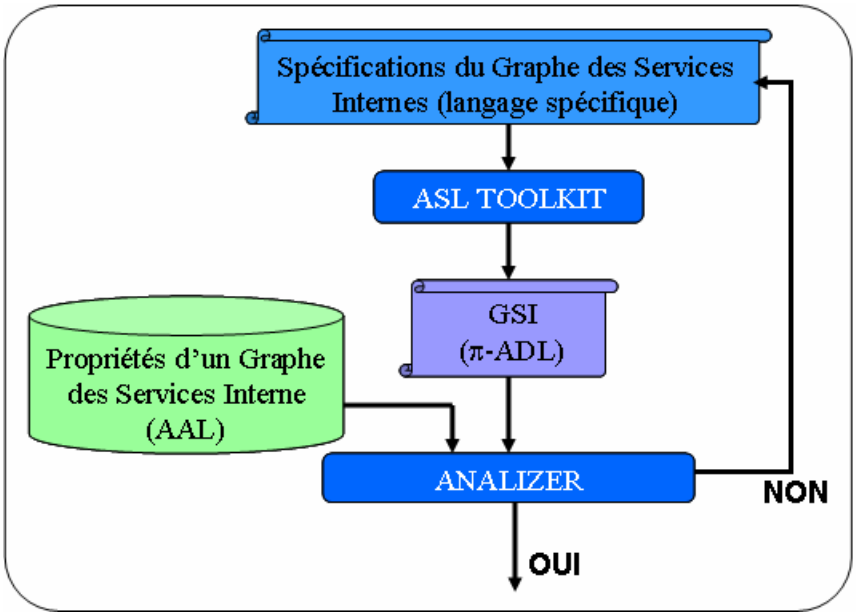



**Figure 6-2 : Processus de compilation et de vérification du fichier contenant la spécification du graphe des services internes**

Cette première phase permet donc de vérifier que le fichier graphe des services internes fournis par le concepteur est correct à la fois d'un point de vue syntaxique mais aussi d'un point de vue sémantique (c'est à dire que toutes les propriétés du modèle spécifique au graphe des services internes sont respectées). A partir de là, nous allons pouvoir vérifier si les fichiers contenant, d'une part, la définition des services externes et d'autre part, la définition des modes sont eux aussi corrects (voir section suivante).

# *III. Compilation et vérification des fichiers contenant les services externes et les modes*

## III.A. Le processus

La seconde étape de ce processus de conception concerne donc la compilation des deux fichiers restants :

- le fichier contenant les spécifications des différents services externes,
- le fichier contenant les spécifications des modes

La figure ci-dessous (Figure 6-3) détaille le processus de compilation et de vérification de ces deux fichiers.

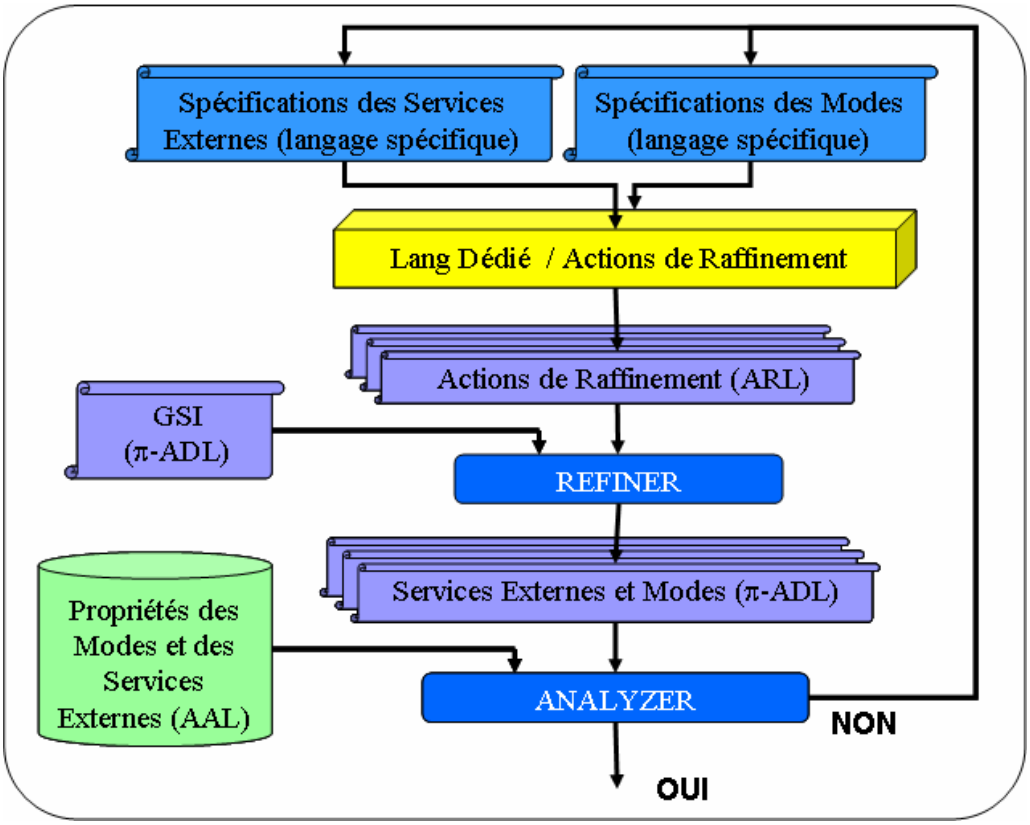



**Figure 6-3 : Compilation et vérification des services externes et des modes**

Tout d'abord, les deux fichiers sont transmis dans un outil spécifique qui a été créé durant ces travaux et qui transforme le langage dédié de spécification des services externes et des modes (chapitre 5 section III) en actions de raffinement écrites en ARL. On se retrouve avec une règle de raffinement par service externe et par mode. Ces règles de raffinement sont utilisées par l'outil Refiner (chapitre 4 section V.C.6) conjointement au ficher du graphe des services internes créé dans l'étape précédente par l'ASL Toolkit. Le Refiner fournit donc un fichier par service externe et par mode. Le Refiner est directement capable de transformer les fichiers en π-ADL. Chaque fichier contient la description de l'architecture d'un service externe ou d'un mode en π-ADL. Chaque fichier est ensuite transmis à l'Analyzer afin de vérifier si l'architecture décrite (d'un service externe ou d'un mode) respecte bien les propriétés adéquates. Si une erreur est détectée, le processus de compilation s'arrête et un message est délivré au concepteur d'instruments.

## III.B. L'outil spécifique

L'outil qui a été créé ici a donc pour but de transformer les deux fichiers de spécification des modes et de spécification de services externes en actions de raffinement ARL. Cet outil a été créé à l'aide d'un outil appelé JavaCC. JavaCC est un outil capable de lire une spécification grammaticale et de la convertir en un programme Java capable de reconnaître les correspondances de cette grammaire. Autrement dit, JavaCC est un générateur de compilateur basé sur la technologie Java.

Le principe est de créer un fichier (extension « .jj ») qui contient la grammaire du fichier à analyser. Dans notre cas, cette grammaire est celle de nos langages de création de services externes et de modes. Ce fichier contient aussi toutes les instructions java nécessaires à la

transformation de ces deux fichiers de spécification de services externes et de modes en fichiers contenant les différentes actions de raffinement ARL correspondantes.

Après compilation du fichier « .jj », un programme java est généré. Dans notre cas, notre programme se nomme *Convertisseur*. Ce programme est donc un compilateur. Pour lancer ce dernier, la ligne de commande suivante sera utilisée :

```
java Convertisseur nom_instrument
```

Le paramètre *nom_instrument* représente le nom de l'instrument qui veut être défini. En fait, une arborescence spécifique a été créée pour nos outils. Dans un répertoire spécifique nommé *SourceADL* se trouve nos fichiers de spécification des services externes et des modes. Ils seront nommés de la façon suivante :

```
nom_instrument.SExterne
nom_instrument.Mode
```

Le résultat de l'exécution de cette ligne de commande sera la création des fichiers contenant les actions de raffinement ARL. Ces fichiers ont le format suivant :

```
SE_NomServiceExterne.arl
Mode_NomMode.arl
```

Il y aura donc un fichier arl par service externe défini dans le *nom_instrument.SExterne* et un fichier arl par mode défini dans le fichier *nom_instrument.Mode*. Ces fichiers seront ensuite directement utilisés par le « Refiner ». Les fichiers générés par l'outil sont placés dans un répertoire *Instrument/nom_instrument/ARL.*

Voici ci-dessous un exemple de transformation du langage spécifique en actions de raffinement ARL. En fait, le fichier qui contient le code spécifique suivant génèrera trois fichiers ARL (il s'agit de la définition des services externes de l'exemple des chapitre précédents) :

```
value SE1 is external service from graph InternalServiceExample where is
removing {
   components{Si1, Si2, Si3, Si4, Si5}
   multicast{Mult1, Mult2, Mult3}
   rdv{Rdv1}
   events{Event1, Event2}
};
value SE2 is external service from graph InternalServiceExample where is
removing {
   components{Si4, Si5, Si6}
   events{Event3, Event4}
};
value SE3 is external service from graph InternalServiceExample where is
removing {
   components{Si5}
   events{Event2, Event3}
   and detach{
        Mult2::outPort::outputChannel::2 from Si3::inPort::inputChannel
   }
}
```

Un fichier pour SE1 nommé *SE_SE1.ARL* :

```
archetype SE1 refines InternalServiceExample using {
      Components excludes {(Si1, Si2, Si3, Si4, Si5)}
      And
      Connectors excludes {(Mult1, Mult2, Mult3, Rdv1)}
      And
      Ports excludes {(Event1, Event2)}
}
```

Un fichier pour SE2 nommé *SE_SE2.ARL* :

```
archetype SE2 refines InternalServiceExample using {
      Components excludes {(Si4, Si5, Si6)}
      And
      Ports excludes {(Event3, Event4)}
}
```

Et un fichier pour SE3 nommé *SE_SE3.ARL* :

```
archetype 'SE3 refines 'InternalServiceExample using {
      Components excludes {('Si5)}
      And
      Ports excludes {('Event2, 'Event3)}
      And
      InternalServiceExample::connections?
      separates {
          Mult2::outPort::outputChannel::2 with Si3::inPort::outputChannel
      }
}
```

## III.C. Conclusion

Ces deux phases de compilation et de vérification (section II et section III) nous ont donc permis de vérifier que les trois fichiers fournis par le concepteur d'instruments sont corrects. Le concepteur d'instruments est, à partir de ce moment, sûr d'avoir décrit **un** instrument intelligent de façon correcte, c'est à dire qui respecte toutes les contraintes des différents styles. Néanmoins, il n'est encore pas certain d'avoir conçu **l'**instrument désiré. L'étape suivante va permettre de le vérifier.

# *IV. Génération de l'instrument intelligent en π-ADL et vérification du comportement par animation*

## IV.A. Le processus

Dans cette étape du processus de conception d'un instrument intelligent, nous allons donc construire le code complet de l'instrument intelligent en π-ADL. La figure ci-dessous (Figure 6-4) présente le processus de génération de l'instrument intelligent en π-ADL.

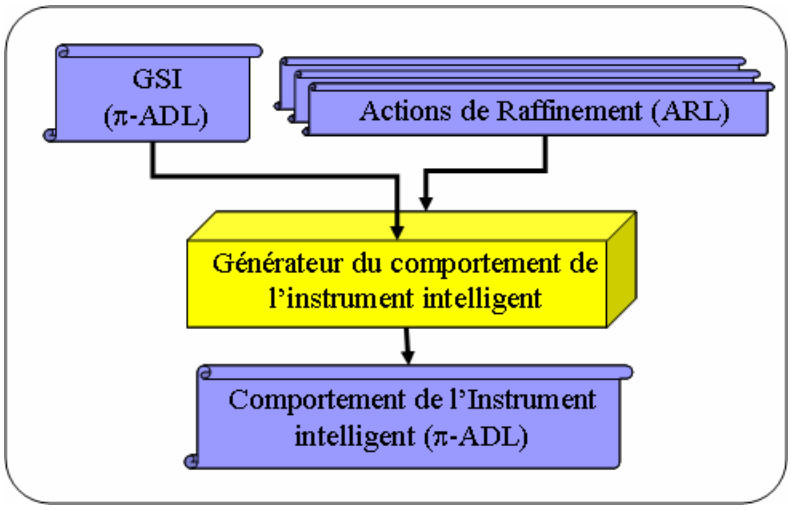



**Figure 6-4 : Génération de l'instrument intelligent en π-ADL**

Lorsque l'on parle du comportement d'un instrument intelligent à ce niveau, il s'agit en fait de décrire tout ce qui va permettre de gérer l'exécution et le changement des modes et des services externes[6]. Pour ce faire, nous devons déjà utiliser le fichier GSI car c'est ce dernier qui contient tout ce qu'il faut pour pouvoir exécuter un service externe (le graphe des services internes). Nous aurions pu utiliser le fichier fournis par le concepteur d'instruments mais l'outil aurait dû être modifié en cas de changements dans le langage de spécification du graphe des services interne ou des éléments qui le compose.

Tous les instruments intelligents gèrent le changement de service externe et de mode de façon identique, cependant, le code qui va permettre de gérer le comportement d'un instrument en particulier dépend de ses modes et de ses services externes. Pour pouvoir générer le comportement d'un instrument intelligent au niveau de ses changements de modes et de service externes, il faut donc utiliser en plus les fichiers qui contiennent la spécification des services externes et des modes. Plutôt que d'utiliser les fichiers qui sont fournis par le concepteur d'instruments, nous nous servirons de ceux qui ont été créés précédemment et qui contiennent les actions de raffinement ARL. Tout comme pour le graphe des services internes, nous aurions pu utiliser les fichiers de spécification des services externes et des modes fournis par le concepteur d'instruments mais cela nous aurait obligé à modifier cet outil si le langage dédié utilisé évoluait.

## IV.B. L'outil spécifique

Cet outil spécifique a aussi été créé avec JavaCC. Pour ce faire, un autre générateur de compilateur ( jj) a été créé. L'outil généré à partir de ce dernier a été nommé *GenerateurComp*. Il s'agit aussi d'un compilateur qui se lance en ligne de commande grâce à l'instruction suivante :

```
java GenerateurComp nom_instrument
```

Comme pour l'outil de création d'actions ARL, *nom_instrument* représente le nom de l'instrument qui veut être défini. Ce compilateur va chercher les fichiers dans le répertoire qui a été créé précédemment et qui contient les différentes actions de raffinement ARL de

[6] Au sein des services externes, l'enchaînement des différents services internes a déjà été défini avec le graphe des services internes.

chaque service externe et de chaque mode. En ce qui concerne le fichier du graphe des services internes, il s'agit toujours de celui nommé *nom_instrument.adl*.

Cet outil génère un nouveau fichier nommé *comportement_nom_instrument.adl* qui contiendra tout ce dont l'Animator aura besoin pour pouvoir simuler le comportement de l'instrument intelligent conçu.

## IV.C. Animation de l'instrument intelligent

Une fois le code du comportement de l'instrument intelligent généré, il va pouvoir être possible de le simuler grâce à l'Animator d'ArchWare ADL. La figure suivante (Figure 6-5) présente ce que l'on doit fournir à l'outil Animator afin de pouvoir simuler le comportement de l'instrument intelligent.

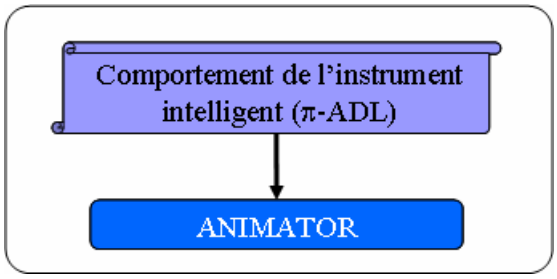



**Figure 6-5 : Vérification du comportement avec l'Animator**

Pour pouvoir simuler l'exécution d'un instrument intelligent, l'Animator a seulement besoin du fichier précédemment créé.

Les fonctions Java sont pour la plupart en relation avec la partie matérielle d'un instrument intelligent. Or, dans cette étape, seule une simulation de l'instrument est effectuée et aucun matériel n'est à disposition. De plus, on part du principe que ces éléments ont déjà été vérifiés auparavant et que donc, ils n'engendrent pas de mauvais fonctionnements. Par conséquent, le code java des services internes n'est pas nécessaire.

En utilisant une version de l'Animator qui intégrerait les fonctionnalitées de gestion de modes et de services externes, le concepteur d'instruments serait capable de simuler des évènements externes qui vont permettre d'exécuter des services externes ou de changer de mode. Il va ainsi pouvoir vérifier si son graphe de changement de mode est correct ou encore si ses services externes sont bien construits. Suite à cette étape, le concepteur d'instruments a donc pu vérifier que l'instrument qu'il venait de concevoir correspondait bien à celui qu'il désirait concevoir au départ. Dans ces conditions, rien ne l'empêche maintenant de générer le code Java de l'instrument intelligent qui devra être chargé dans son EPROM.

# V. Génération du code source Java de l'instrument intelligent

## V.A. Le processus

La dernière étape de la conception va consister à fournir le code source de l'instrument intelligent. Pour ce faire, plusieurs informations sont nécessaires (voir Figure 6-6 ci-dessous).

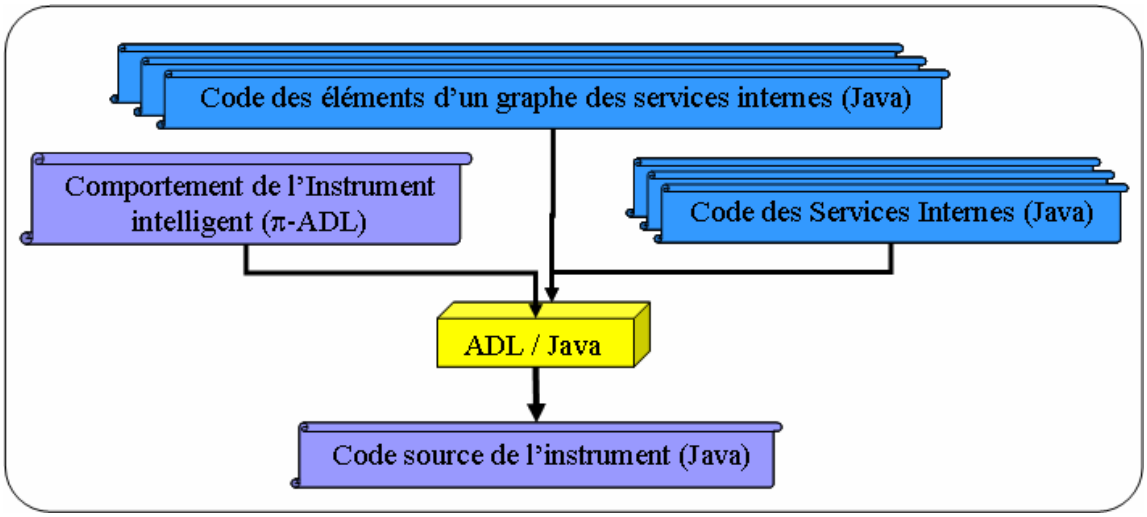



**Figure 6-6 : Génération du code source de l'instrument intelligent**

Pour pouvoir fournir ce code Java, nous avons besoin du fichier contenant le comportement de l'instrument précédemment créé. Néanmoins, ce fichier est suffisant pour une simulation mais pas pour une génération complète du code Java de l'instrument intelligent. En effet, à ce niveau, il faut obligatoirement que le code Java des services internes soit intégré. De plus, tous les éléments qui composent la structure du logiciel sont créés à partir de styles. Lors du passage en Java, le même principe est utilisé. En effet, des classes Java ont été créées pour chaque élément. Ces différentes classes seront présentées dans la section V.B ci-dessous.

Le code source Java fournit par l'outil est donc composé d'une seule et unique classe. Cependant, ce dernier se charge aussi de copier les classes contenant le code des services internes et des éléments d'un graphe des services internes dans le répertoire approprié (voir section V.D).

## V.B. Le code Java des éléments d'un graphe des services internes

Pour éviter d'avoir à créer le code complet d'un instrument intelligent à chaque compilation, le principe de patrons de conceptions a été utilisé. Concrètement, à chaque style créé correspond une classe java qui a les mêmes caractéristiques comportementales. Ce type de construction a beaucoup d'avantages dont le principal est la plus grande facilité d'évolution du code. En effet, si un style évolue, la seule chose qu'il faudra faire est de faire évoluer sa classe Java en conséquence. Si tout le code était généré avec notre outil spécifique, une modification dans un style engendrerait une modification de celui-ci, ce qui implique une nouvelle version de l'environnement.

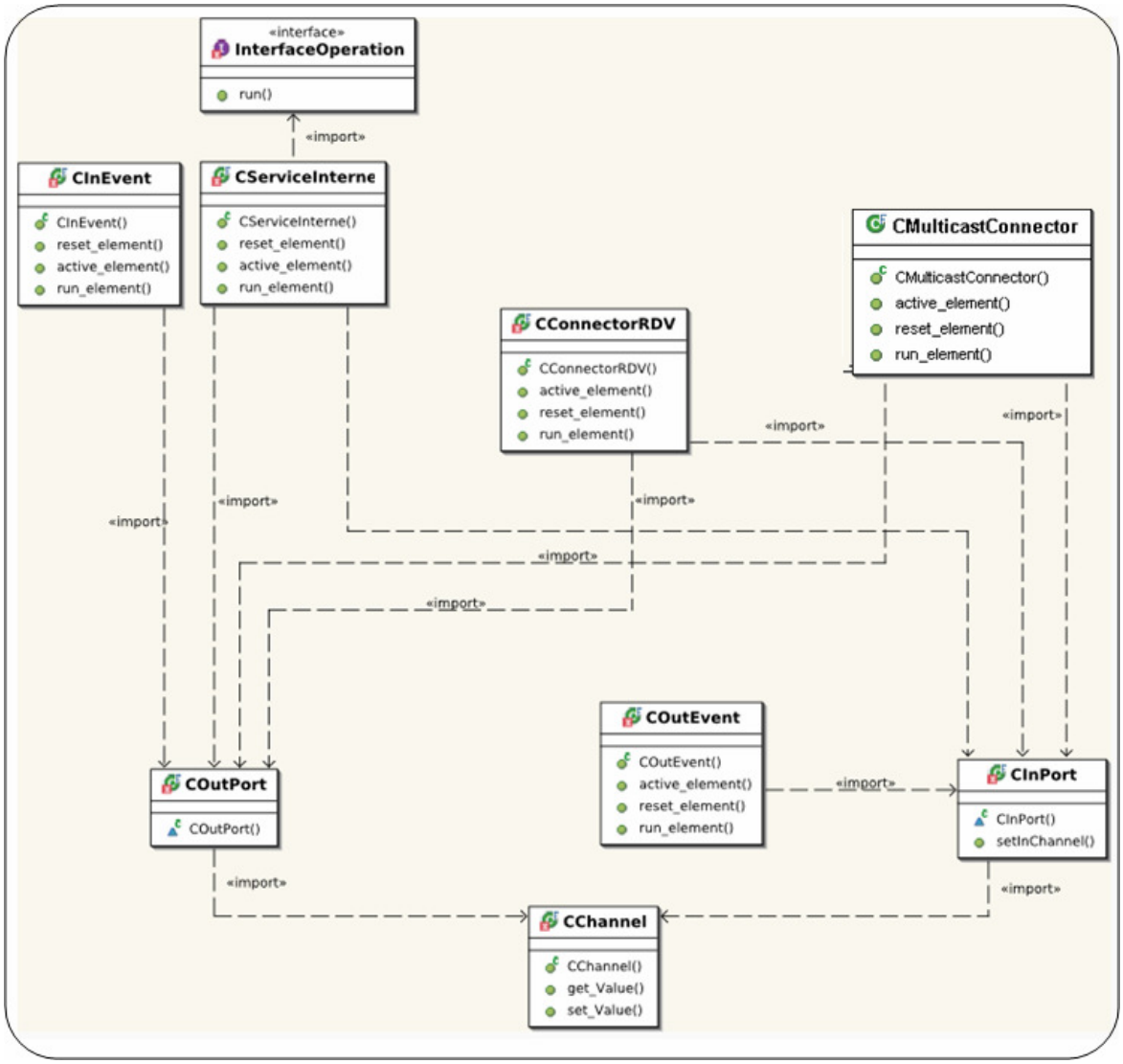
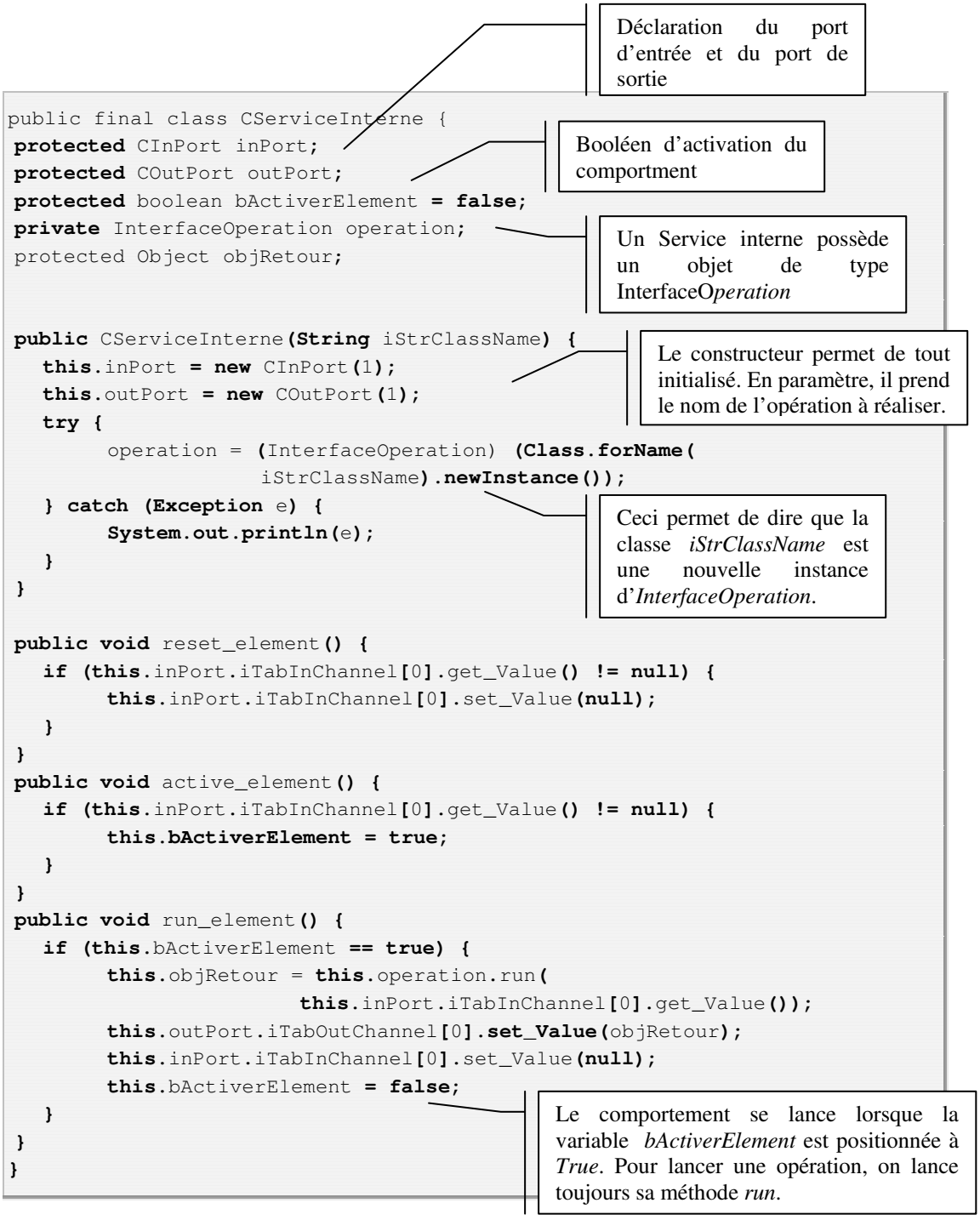



**Figure 6-7 : Diagramme de classe**

Le seul élément architectural qui n'a pas une classe par style est le connecteur Rdv car il a suffit de paramétrer la classe avec le nombre de canaux de sortie. Il a été fait en sorte que tous les éléments aient deux ports et que ces derniers possèdent les connections. La figure ci-dessus (Figure 6-7) représente le diagramme de classe qui montre comment sont construits les différents éléments qui composent le graphe des services internes.

Le même principe d'activation ou de désactivation d'élément a été conservé pour une plus grande lisibilité. En effet, chaque élément possède un booléen nommé *bActiverElement* qui permet d'activer ou non l'exécution de son comportement. Cette variable est gérée par la fonction *active_element()*. Une autre méthode appelée *reset_element()* est aussi présente dans tous les éléments qui composent un graphe des services interne. Cette procédure permet de remettre la valeur du ou des canaux d'entrée à la valeur *null*. Ceci est nécessaire lors de tout changement de mode ou de service externe. Enfin, une dernière méthode spécifique nommée *run_element()* permet de lancer l'exécution du comportement de l'élément.

L'exemple du code de l'élément service interne est présenté ci-dessous.




```
public final class CServiceInterne {
protected CInPort inPort;
protected COutPort outPort;
protected boolean bActiverElement = false;
private InterfaceOperation operation;
protected Object objRetour;


public CServiceInterne(String iStrClassName) {
   this.inPort = new CInPort(1);
   this.outPort = new COutPort(1);
   try {
        operation = (InterfaceOperation) (Class.forName(
                    iStrClassName).newInstance());
   } catch (Exception e) {
        System.out.println(e);
   }
}

public void reset_element() {
   if (this.inPort.iTabInChannel[0].get_Value() != null) {
        this.inPort.iTabInChannel[0].set_Value(null);
   }
}
public void active_element() {
   if (this.inPort.iTabInChannel[0].get_Value() != null) {
        this.bActiverElement = true;
   }
}
public void run_element() {
   if (this.bActiverElement == true) {
        this.objRetour = this.operation.run(
                        this.inPort.iTabInChannel[0].get_Value());
        this.outPort.iTabOutChannel[0].set_Value(objRetour);
        this.inPort.iTabInChannel[0].set_Value(null);
        this.bActiverElement = false;
   }
}
}
```

Le code source Java des autres éléments est présenté dans l'annexe 2.

## V.C. La classe principale de gestion du comportement

Cette classe a été construite de la même façon que le fichier qui contient les abstractions de gestion du comportement d'un instrument intelligent en ADL. C'est à dire qu'elle contient toutes les fonctions qui vont permettre de gérer l'activation ou non des différents booléen *bActiverElement* et pour changer de service externe et de mode. Mais ce n'est pas tout. En effet, cette classe permet aussi de gérer les communications entre l'instrument intelligent et le réseau. De ce fait, elle implémente une classe *SWTPListener* qui contient toutes les

méthodes pour la communication. Cette classe étend aussi la classe *Thread* afin que son comportement soit récursif.

Ci-dessous se trouve le squelette du code utilisé. Les méthodes *synchronized* sont déclarées ainsi car elles sont utilisées par la méthode *run* du thread. Un exemple complet est présenté dans le chapitre suivant.

```
  public final class InternalServiceExample extends Thread implements
SWTPListener{
  // Déclaration des divers éléments de l'instrument intelligent
  ...
  // Booléens rendant possible l'activation des SE
  ...
  // Booléens activant les SE
  ...
  public void link(){
        ...
  }

  Public synchronized void setMode(String iStrMode) {
        ...
  }

  public synchronized void reset_ServiceExtern() {
        ...
  }

  public synchronized void setService(String iStrSE, boolean on) {
        ...
  }

  public synchronized void gestion_sortie_element() {
        ...
  }

  public synchronized void gestion_entree_element() {
        ...
  }

  public synchronized void lancer_element(){
        ...
  }
```

La méthode *link* permet de lier les différents éléments par leurs connexions.

La méthode *setMode* permet de changer de mode. Comme son homologue en ADL, elle manipule des booléens. Elle est rendue obligatoire par l'interface *SWTPListener*. Elle est accessible par le réseau.

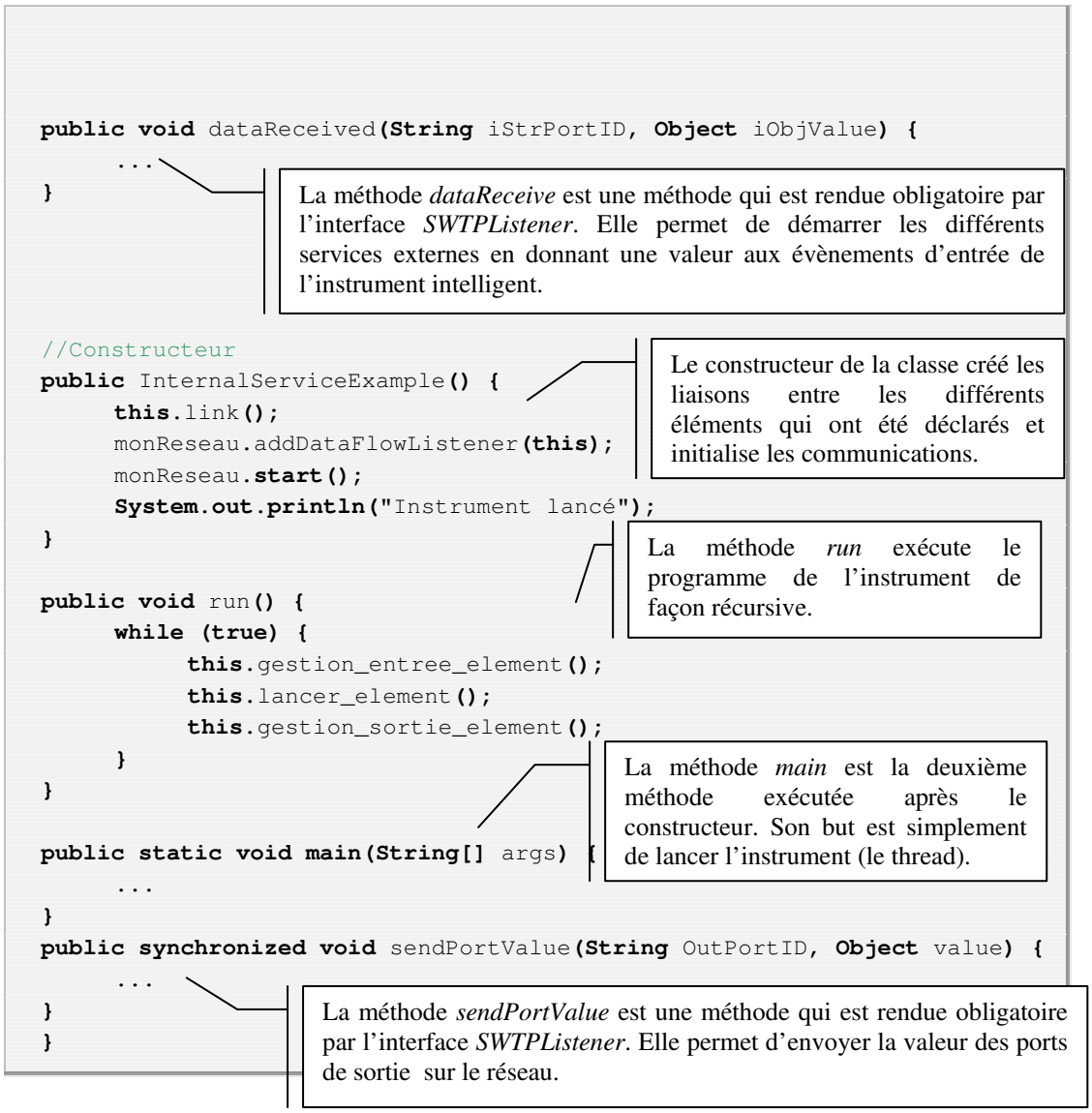
La méthode *reset_ServiceExtern* permet de désactiver tous les services externes.

La méthode *setService* active ou désactive un service externe donné suivant la valeur du booléen passé en paramètre. Elle est rendue obligatoire par l'interface *SWTPListener*. Elle est accessible par le réseau.

La méthode *gestion_sortie_element* permet d'envoyer les valeurs des évènements de sortie sur le réseau par l'intermédiaire de la méthode *sendPortValue* qui se trouve plus bas.

La méthode *gestion_entree_element* permet d'activer ou non un élément en fonction du mode et du service externe qui est demandé. Pour ce faire elle utilise les méthodes *active_element* et *reset_element* de tous les éléments de l'instrument intelligent.

Lance la méthode *run_element* de tous les éléments de l'instrument intelligent.




Le squelette est donc identique à tous les instruments intelligents. Par contre, le corps des méthodes évolue en fonction du graphe des services internes, des services externes et des modes de l'instrument qui est conçu. Ce fichier est donc géré par un outil spécifique qui va être présenté dans la section suivante (section V.D).

## V.D. L'outil spécifique

L'outil créé pour générer le code source java est toujours basé sur la même technologie que les précédents, c'est-à-dire JavaCC. Il se nomme *GenerateurJava*. Il s'agit aussi d'un compilateur qui se lance en ligne de commande grâce à l'instruction suivante :

```
java GenerateurJava nom_instrument
```

*nom_instrument* représente le nom de l'instrument qui veut être défini. Ce compilateur va chercher les fichiers dans différents répertoires. Dans le répertoire *ComposantsConnecteurs* on trouvera les classes java des composants, des connecteurs et des différents ports. Ces classes sont identiques à tous les instruments intelligents. Dans le répertoire *Operations* se trouve les différentes opérations (classes) créées par le

programmeur. Ces opérations sont toutes conçues de la même façon. Ce sont des classes qui implémentent toutes au moins une méthode *run*. Voici un exemple ci-dessous pour une opération nommée *Op_AcquirePressure1*.

```
class Op_AcquirePressure1 implements InterfaceOperation{
     public Object run(Object valIn){
          ...
     }
     ...
}
```

La méthode *run* peut ensuite faire appel à d'autres fonctions suivant comment le programmeur a construit la classe. Pour éviter certaines erreurs, la class implémente toujours d'une interface nommée *InterfaceOperation*.

Le ficher généré par l'outil est un fichier nommé *nom_instrument.java* qui sera placé dans un répertoire *Instrument/nom_instrument/JAVA*. Dans ce répertoire seront aussi placé toutes les classes des services internes et des éléments du graphe des services internes. Ce répertoire n'aura plus qu'à être télécharger sur l'instrument intelligent.

# VI. Conclusion

Ce chapitre nous a permis de présenter en détail le nouveau processus de conception que nous avons dû élaborer afin de pouvoir concrètement utiliser un développement centré architecture dans la conception des instruments intelligents. Ce qu'il est important de retenir est le fait que le processus est très simple du point de vue du concepteur d'instruments intelligents et assure la construction correcte des instruments. En effet, seuls trois fichiers sont nécessaires à la création d'un instrument intelligent :

- le fichier contenant le graphe des services internes,
- le fichier contenant la spécification des services externes,
- le fichier contenant la spécification des modes.

Théoriquement, une fois ces trois fichiers fournis au processus, le concepteur d'instruments intelligent est averti s'il y a des erreurs ou non. S'il n'y en a aucune, il n'a plus qu'à visualiser le comportement de ses services externes et de ses modes grâce à l'outil ArchWare Animator. Une fois ceci terminé, le code source Java de l'instrument est créé et peut être directement téléchargé dans la mémoire de celui-ci.

Néanmoins, ceci n'est pas encore aussi simple. En effet, pour que ce soit possible, il reste à créer une application qui se situerait au dessus de tout ce processus et qui serait chargée d'enchaîner l'exécution des différents outils. Par exemple, chaque mode et chaque service externe doit être vérifié un par un. Pour ce faire, l'outil Analyzer doit être appelé plusieurs fois. Le nombre de fois correspond au nombre de services externes et de modes que contient l'instrument intelligent qui est conçu.

Même si cet outil n'a pas été créé, nous avons pu voir que le processus avait de nombreux avantages par rapport à ce que faisait un outil comme CAPTool. En effet, le modèle peut évoluer fortement sans pour autant que cela ait un effet sur le processus. Tant que le modèle d'un instrument intelligent est fait de telle sorte que l'on ait une couche de base qui

représente la structure globale et d'autres couches qui sont des sous parties de la couche de base, le processus n'a pas à évoluer.

De plus, les outils eux mêmes n'ont pratiquement pas à évoluer. En effet, le seul outil qui peut être à modifier est celui qui transforme le langage spécifique en action de raffinement ARL. Ceci est le cas si des changements au niveau du langage spécifique se font sentir. Les autres sont fait de telle sorte qu'ils utilisent les fichiers qui sont écrit dans des langages qui ne bougent pas : le π-ADL et l'ARL.

Dans le chapitre suivant, afin de valider nos travaux, un exemple complet de conception de la partie logicielle d'un instrument intelligent va être présenté.

# Chapitre 7 : VALIDATION

# Chapitre 7 : Validation

# Validation

Le but de ce chapitre est de valider ce qui a été présenté dans les chapitres précédents au travers d'un exemple. L'instrument intelligent qui a été choisi est un gant numérique. Ce gant numérique permet de commander un petit robot en lui passant différents ordres de base (*avance*, *recule*, *tourne à gauche*, etc.).

Comme il a été dit dans le chapitre 3, le projet ArchWare est un projet de recherche dont le but premier était de fournir un ensemble intégré de langages et d'outils orientés architecture. Etant un projet de recherche, la majorité des efforts se sont concentrés sur la spécification des langages. Les outils, quand à eux, ont fait l'objet de spécifications très complètes mais étaient avant tout créés afin de montrer la faisabilité et l'intérêt d'une approche centrée architecture. Ce ne sont que des prototypes. A ce titre, les fonctionnalités qui ont été implémentées se résument souvent à celles qui étaient nécessaires pour mener à bien les cas d'étude du projet. La majeure partie des outils ArchWare était, de plus, en phase de développement pendant le déroulement des travaux proposés dans cette thèse. Tout ceci fait que tout le travail de conception du nouveau processus s'est fait sur les spécifications des outils et non sur les outils eux même. En ce qui concerne nos travaux, les styles créés utilisent beaucoup de notions qui n'ont pas été implémentées dans les prototypes des outils de support au langage ASL, ce qui fait que la validation de l'approche ne peut se faire automatiquement d'un bout à l'autre sans aucune intervention par rapport au processus prévu.

Dans un premier temps, l'application va être présentée (section I). Ensuite, la modélisation du gant sera effectuée (section II), suivie de sa transcription en langage dédié (section III). Enfin, le processus de conception sera parcouru étape par étape pour l'application choisie (section IV).

## *I. L'application*

L'instrument intelligent qui sera développé ici est le gant numérique. L'application qui contient ce gant est celle se trouvant dans [Allevard 05]. Son but est de contrôler un petit robot par l'intermédiaire du gant numérique.

### I.A. Description du gant

Le gant numérique utilisé dans ce chapitre est le *Cyberglove* [Kramer 96] (voir Figure 7-1). C'est un gant dans lequel sont intégrés plusieurs capteurs de flexion se retrouvant au

niveau des différentes articulations dont on veut mesurer l'angle (voir Figure 7-2). Ce gant a concrètement été utilisé au laboratoire LISTIC notamment dans la thèse de [Allevard 05] qui avait deux objectifs :

- Décrire la configuration de la main à l'aide de termes du langage naturel. Par configuration nous entendons la forme que prend la main, indépendamment de sa position et de son orientation dans l'espace. Cette description est qualifiée de lexicale ou de symbolique.
- Extraire des informations à partir de cette description lexicale de la configuration de la main, plus précisément reconnaître des signes statiques, des signes proportionnels et des commandes pour le contrôle d'un robot mobile (voir section I.C).


**Figure 7-1 : Le Cyberglove [Allevard 05]**

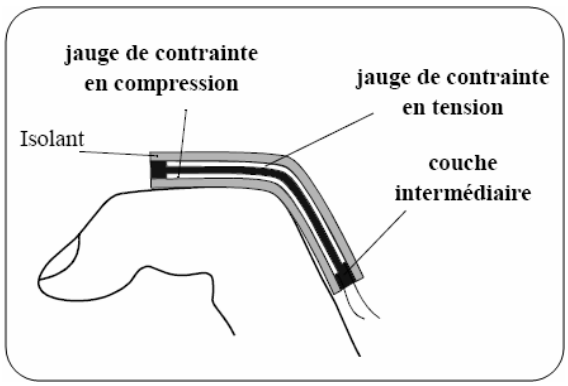



**Figure 7-2 : Capteur de flexion utilisant deux jauges de contraintes utilisés dans le Cyberglove [Allevard 05]**

Il existe plusieurs types de gant Cyberglove. Celui qui est utilisé possède 18 capteurs de flexion disposés sur toute la main mais seulement 15 sont utilisés (Figure 7-3 ci-dessous).

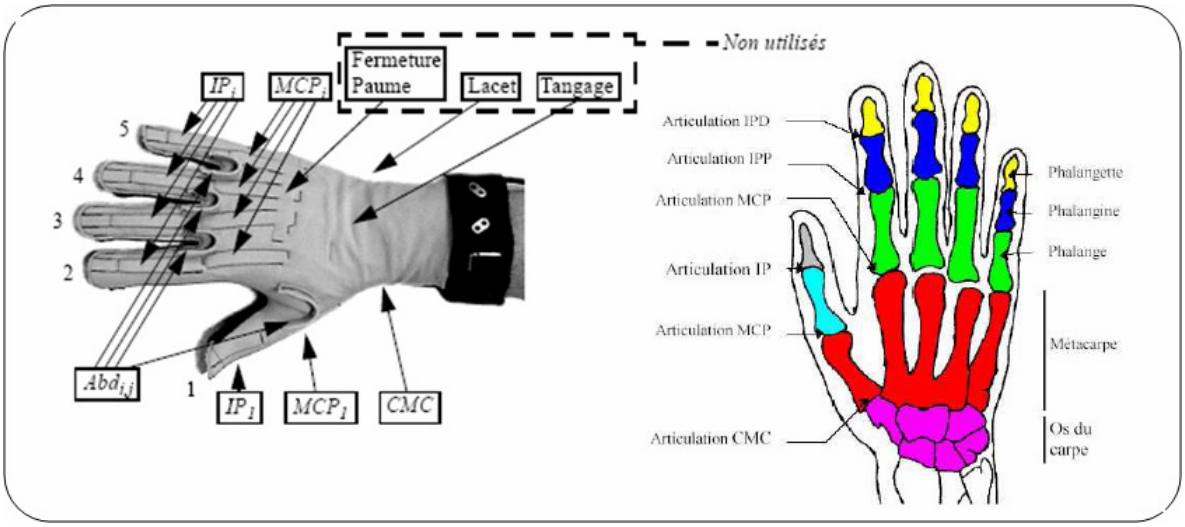



**Figure 7-3 : Les dix-huit capteurs du Cyberglove, la numérotation des doigts et les articulations de la main [Allevard 05].**

L'écartement de chaque paire de doigts est mesuré par un capteur d'abduction/adduction appelé *ABD*. On a donc les capteurs suivants :

- ABD_pouce_index,
- ABD_index_majeur,

- ABD_majeur_annulaire,
- ABD_annulaire_auriculaire.

La flexion d'un grand doigt, est mesurée par les deux capteurs de flexion *IP* et *MCP*. On a donc :

- IP_index et MCP_index,
- IP_majeur et MCP_majeur,
- IP_annulaire et MCP_ annulaire,
- IP_ auriculaire et MCP_ auriculaire.

Le pouce est traité de façon particulière par rapport aux autres doigts. On va bien entendu vouloir savoir s'il est fléchi ou non avec un capteur *IP*, mais on va aussi vouloir connaître son orientation. Cette dernière est mesurée par trois capteurs :

- CMC_pouce,
- MCP_pouce,
- ABD_pouce.

Tous ces capteurs vont donc servir à connaître la position dans laquelle se trouve la main (poing serré, index pointé, etc.). La façon dont toutes ces données sont traitées pour pouvoir identifier le geste n'est pas l'objet de cette thèse, par conséquent, si le lecteur désir de plus amples informations sur le sujet, celles-ci sont disponibles dans [Allevard 05].

## I.B. Le robot

Le robot qui sera commandé par le gant est le robot de *LEGO Mindstorm* (voir Figure 7-4 ci-dessous).

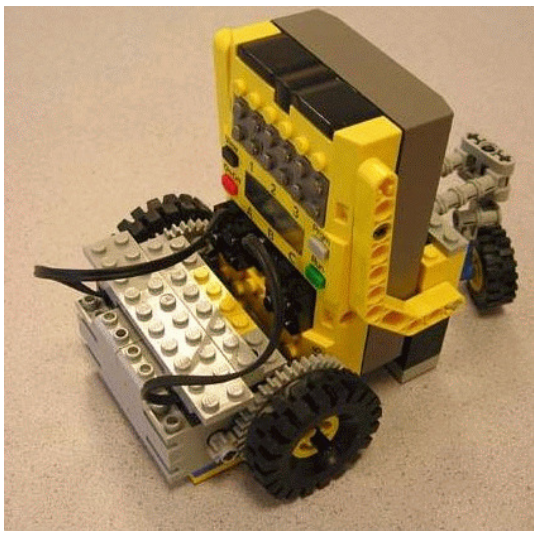


**Figure 7-4 : Le robot de LEGO Mindstorm [Allevard 05]**

Le LEGO Mindstorm est constitué d'un ensemble de pièces LEGO et de l'unité *RCX* (unité de commande du robot disposant d'un afficheur à cristaux liquides). La brique *RCX* est un système programmable autonome bâti autour d'un micro-contrôleur *Hitachi H8/3932*. La brique *RCX* peut être utilisée pour contrôler des actionneurs (moteurs, générateur de sons) et lire des entrées en provenance de capteurs (capteur de lumière, détecteurs de contact). D'autres capteurs et actionneurs peuvent être ajoutés (capteurs de pression, de température,

de rotation, etc.). La brique *RCX* possède également un petit écran LCD, utile pour afficher de l'information, et un émetteur - récepteur infrarouge, utile pour télécharger des programmes et communiquer avec les autres *RCX*. L'unité *RCX* est conçue pour pouvoir facilement être attachée à d'autres briques LEGO.

## I.C. Les différents gestes de commande du robot

[Allevard 05] a défini deux types de signes possibles :

- des signes statiques : posture particulière de la main.
- des signes proportionnels : gestes ayant une durée temporelle (d'autres paramètres tels que la vitesse d'exécution d'un geste, son amplitude, sont pris en compte).

Dix actions possibles ont été définies pour ce robot. Chaque action *A* est associée à un signe statique $S_A$. Cinq actions lui permettent de se déplacer (*Avance*, *Recule*, *Gauche*, *Droite*, *Stop*). Deux actions permettent des rotations rapides du robot sur lui-même, deux autres actions permettent l'émission de sons par le robot et enfin une action correspond à une mise en marche/arrêt du robot. Les dix signes choisis pour ces actions sont représentés ci-dessous (Figure 7-5).

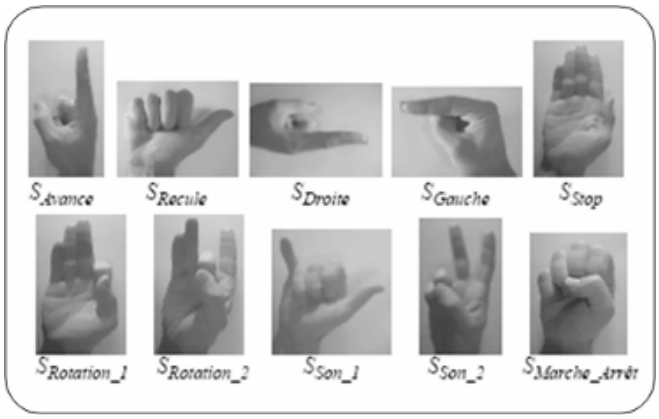



**Figure 7-5 : Les dix signes statiques correspondant à des commandes du robot mobile [Allevard 05]**

Comme le montre la figure ci-dessous (Figure 7-6), un autre « mode de fonctionnement » pour le signe $S_{avance}$ a été envisagé. En effet, il est possible de commander la vitesse avec laquelle le robot va avancer. Celle-ci est déterminée en fonction de la position de l'index (plus ou moins plié). En fait, plus l'index est tendu, plus le robot avance vite.

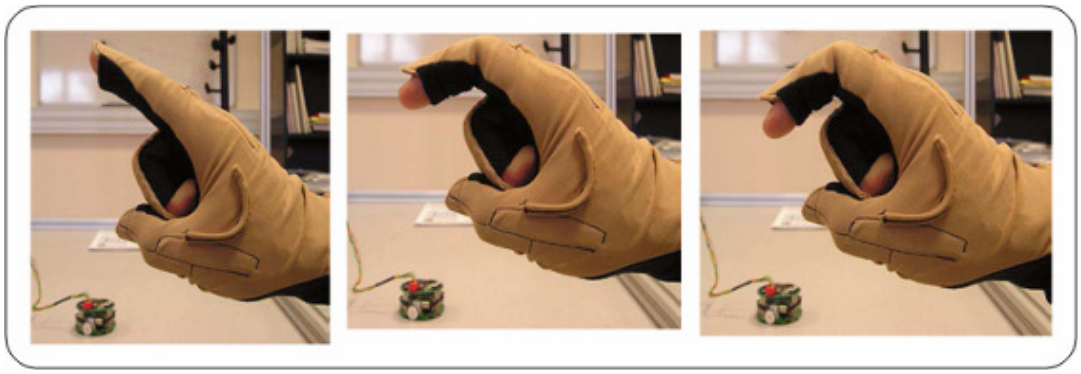


**Figure 7-6 : Différentes amplitudes pour la posture du signe $S_{avance}$ en proportionnel [Allevard 05]**

## II. Modélisation du gant numérique

La modélisation du gant est faite ici en se basant sur le nouveau modèle de conception des instruments intelligents proposé dans cette thèse. Pour pouvoir concevoir un tel instrument intelligent, il faut que l'on fournisse son graphe des services internes, ses services externes et ses modes.

Deux modes sont créés :

- Mode_statique,
- Mode_proportionnel.

Ces deux modes correspondent aux deux types de geste que l'on peut avoir. Dans chacun de ces modes, les dix gestes décris précédemment pourront être utilisés. La seule chose qui changera entre ces deux modes est le signe *Avance*. En effet, dans le mode dynamique, en plus de la reconnaissance du geste, l'amplitude de la posture sera analysée afin de faire avancer le robot plus ou moins vite.

Deux services externes seront créés :

- Statique,
- Proportionnel.

Ces deux services externes appartiendront respectivement au mode statique et proportionnel.

En ce qui concerne le graphe des services internes du gant, pour des raisons de clarté, il a été décidé d'utiliser uniquement l'information venant de deux doigts : le pouce et l'index. En effet, la partie du graphe qui concerne l'index est identique à celle des autres doigts. Ceci nous oblige à réduire le nombre de gestes qui sont reconnus comme des commandes du robot. Le graphe des services internes du gant numérique restreint au pouce et à l'index est le suivant :

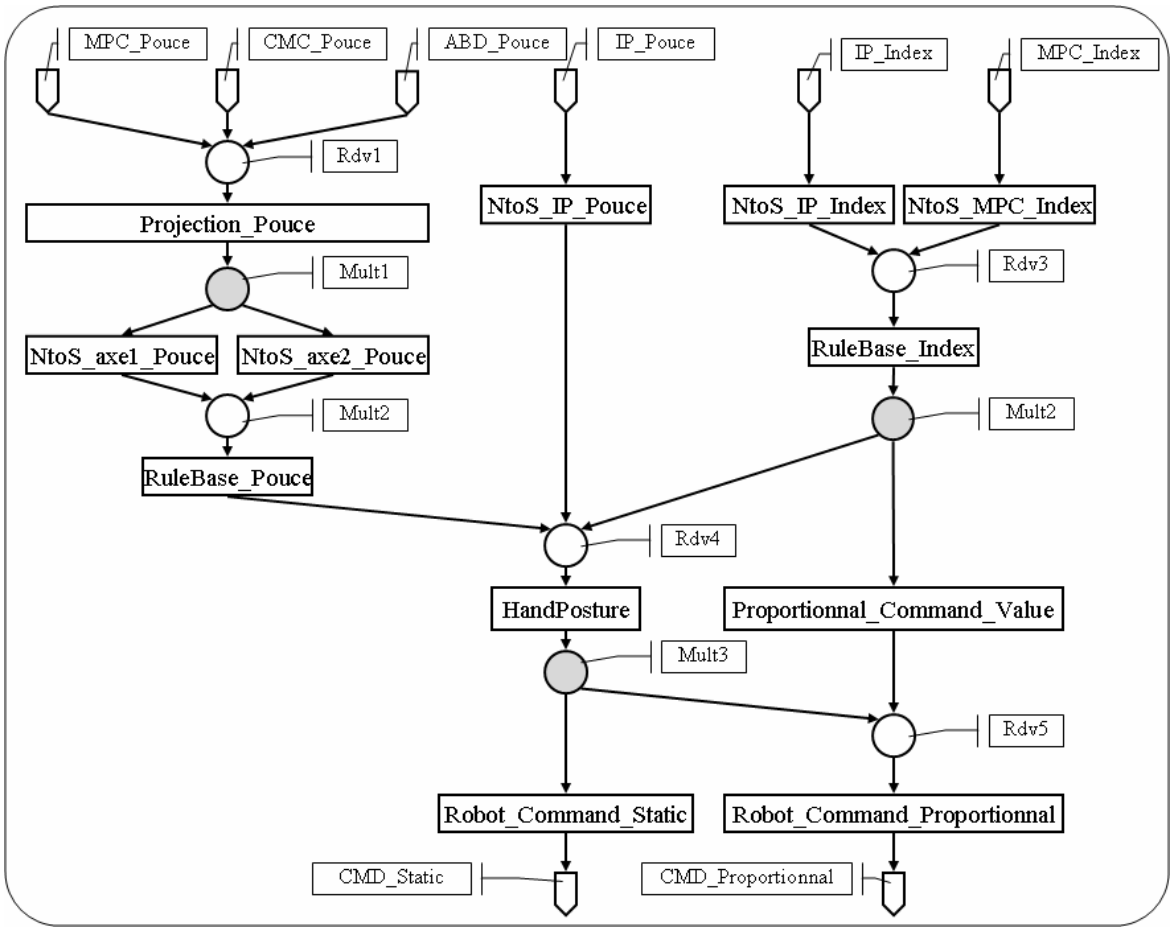



**Figure 7-7 : Graphe des services internes du gant pour le pouce et l'index**

Ce graphe est composé de :

- 5 ports d'entrée reliés chacun à un capteur situé sur le gant :
  - **MPC_Pouce**
  - **CMC_Pouce**
  - **ABD_Pouce**
  - **IP_Pouce**
- 2 ports de sortie qui permettent de dire qu'une commande a été envoyée au robot :
  - **CMD_Static**
  - **CMD_Proportionnal**
- 12 services internes qui vont permettre de transformer le geste effectué par la main en commande interprétable par le robot :
  - **Projection_Pouce :** *Fait une combinaison linéaire des 3 capteurs d'entrée afin d'obtenir 2 valeurs (composantes discriminantes) plus facilement interprétables par les services internes suivants.*
  - **NtoS_axe1_Pouce :** *Transforme une donnée discriminante numérique en donnée symbolique interprétable par une base de règle.*
  - **NtoS_axe2_Pouce :** *Transforme la deuxième donnée discriminante numérique en donnée symbolique interprétable par une base de règle.*
  - **RuleBase_Pouce :** *base de règle qui détermine la position du pouce en fonction des données symboliques reçues.*
  - **NtoS_IP_Pouce :** *transforme la donnée numérique du capteur* IP_Pouce *en une donnée symbolique interprétable par une base de règles.*

  - **NtoS_IP_Index :** *transforme la donnée numérique du capteur* IP_Index *en une donnée symbolique interprétable par une base de règles.*
  - **NtoS_MPC_Index :** *transforme la donnée numérique du capteur MPC_Index en une donnée symbolique interprétable par une base de règle.*
  - **RuleBaseIndex :** *base de règles qui détermine la position de l'index en fonction des données symboliques reçues.*
  - **HandPosture :** *base de règle qui détermine le geste de la main en fonction de la position des différents doigts.*
  - **Proportionnal_Command_Value :** *détermine la valeur numérique de la commande à envoyer au robot en fonction de la position de l'index.*
  - **Robot_Command_Static :** *détermine la commande à envoyer au robot en fonction du geste fait par la main.*
  - **Robot_Command_Proportionnal :** *détermine la commande à envoyer au robot en fonction du geste fait par la main et de la valeur calculée par le service interne* Proportionnal_Command_Value.
- 5 connecteurs Rdv :
  - **Rdv1**
  - **Rdv2**
  - **Rdv3**
  - **Rdv4**
  - **Rdv5**
- 3 connecteurs Multicast :
  - **Mult1**
  - **Mult2**
  - **Mult3**

Deux services externes sont créés sur ce graphe des services internes comme dit précédemment (voir Figure 7-8 ci-dessous).

Comme leur nom l'indique, le service *Statique* récupère la valeur de tous les capteurs afin d'envoyer une commande statique au robot alors que le service proportionnel récupère aussi la valeur de tous les capteurs mais envoie une commande proportionnelle.

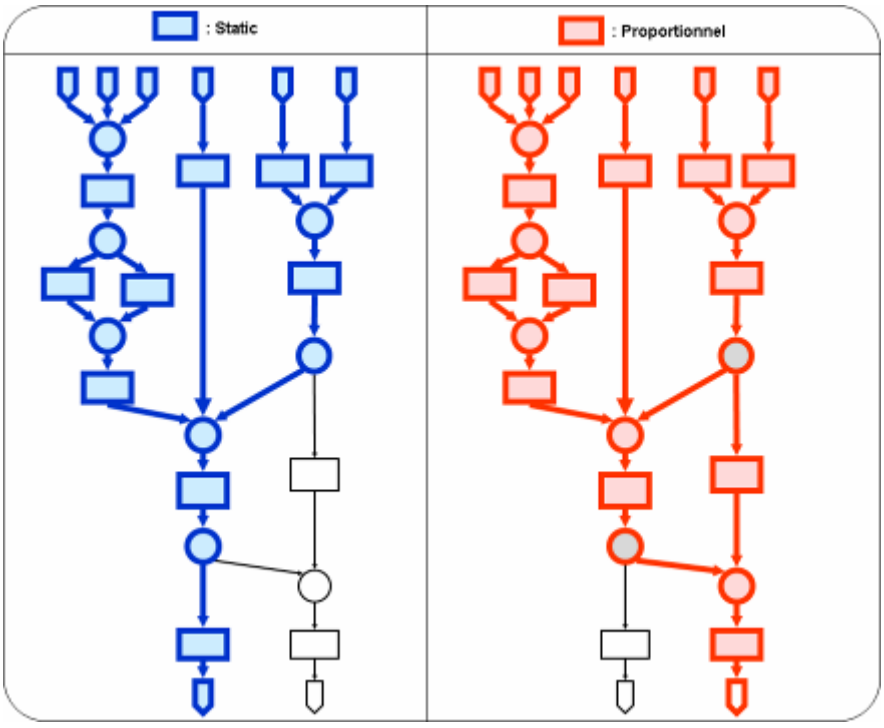



**Figure 7-8 : Les deux services externes**

# III. Création des fichiers pour la conception du gant

Trois fichiers sont à fournir pour la conception du gant :

- la spécification du graphe des services internes,
- la spécification des services externes,
- la spécification des modes.

Les trois fichiers sont présentés ci-dessous.

## III.A. Le fichier de spécification du graphe des services internes

```
value Gant is internal service graph with {
  Num{1}
  Interfaces{
    value MPC_Pouce is input event with Real type;
    value CMC_Pouce is input event with Real type;
    value ABD_Pouce is input event with Real type;
    value IP_Pouce is input event with Real type;
    value MPC_Index is input event with Real type;
    value IP_Index is input event with Real type;
    value CMD_Static is output event with String type;
    value CMD_Proportionnal is output event with String type
  }
  Constituents{
    value Projection_Pouce is internal service with tuple(Real, Real, Real) input type
                tuple(Real, Real) output type operating with "Projection_pouce.java";
    value NtoS_Axe1_Pouce is internal service with tuple(Real, Real) input type
                String output type operating with "NtoS_axe1_Pouce.java";
    value NtoS_Axe2_Pouce is internal service with tuple(Real, Real) input type
                String output type operating with "NtoS_axe2_Pouce.java";
    value RuleBase_Pouce is internal service with String input type
                String output type operating with "RuleBase_Pouce.java";
```

```
    value NtoS_IP_Pouce is internal service with Real input type
                String output type operating with "NtoS_IP_Pouce.java";
    value NtoS_IP_Index is internal service with Real input type
                String output type operating with "NtoS_IP_Index.java";
    value NtoS_MPC_Index is internal service with Real input type
                String output type operating with "NtoS_MPC_Index.java";
    value RuleBase_Index is internal service with tuple(String, String) input type
                String output type operating with "RuleBase_Index.java";
    value StaticPosture is internal service with tuple(String, String) input type
                String output type operating with "StaticPosture.java";
    value Proportionnal_Command_Value is internal service with String input type
                Real output type operating with "Proportionnal_Command_Value.java";
    value Robot_Command_Static is internal service with String input type
                String output type operating with "Robot_Command_Static.java";
    value Robot_Command_Proportionnal is internal service with tuple(String, Real) input type
                String output type operating with "Robot_Command_Proportionnal.java";

    value Mult1 is multicast connector with tuple(Real, Real) type and 2 output channels;
    value Mult2 is multicast connector with String type and 2 output channels;
    value Mult3 is multicast connector with String type and 2 output channels;

    value Rdv1 is rdv connector 3 with Real, Real, Real input types;
    value Rdv2 is rdv connector 2 with String, String input types;
    value Rdv3 is rdv connector 2 with String, String input types;
    value Rdv4 is rdv connector 3 with String, String, String input types;
    value Rdv5 is rdv connector 2 with String, Real input types
  }
  Links{
    MPC_Pouce::outputChannel unifies Rdv1::inPort::inputChannel::1;
    CMC_Pouce::outputChannel unifies Rdv1::inPort::inputChannel::2;
    ABD_Pouce::outputChannel unifies Rdv1::inPort::inputChannel::3;
    IP_Pouce::outputChannel unifies NtoS_IP_Pouce::inPort::inputChannel;
    Rdv1::outPort::outputChannel unifies Projection_Pouce::inPort::inputChannel;
    Projection_Pouce::outPort::outputChannel unifies Mult1::inPort::inputChannel;
    Mult1::outPort::outputChannel::1 unifies NtoS_Axe1_Pouce::inPort::inputChannel;
    Mult1::outPort::outputChannel::2 unifies NtoS_Axe2_Pouce::inPort::inputChannel;
    NtoS_Axe1_Pouce::outPort::outputChannel unifies Rdv2::inPort::inputChannel::1;
    NtoS_Axe2_Pouce::outPort::outputChannel unifies Rdv2::inPort::inputChannel::2;
    Rdv2::outPort::outputChannel unifies RuleBase_Pouce::inPort::inputChannel;
    RuleBase_Pouce::outPort::outputChannel unifies Rdv4::inPort::inputChannel::1;
    NtoS_IP_Pouce::outPort::outputChannel unifies Rdv4::inPort::inputChannel::2;
    IP_Index::outputChannel unifies NtoS_IP_Index::inPort::inputChannel;
    MPC_Index::outputChannel unifies NtoS_MPC_Index::inPort::inputChannel;
    NtoS_IP_Index::outPort::outputChannel unifies Rdv3::inPort::inputChannel::1;
    NtoS_MPC_Index::outPort::outputChannel unifies Rdv3::inPort::inputChannel::2;
    Rdv3::outPort::outputChannel unifies Mult2::inPort::inputChannel;
    Mult2::outPort::outputChannel::1 unifies Rdv4::inPort::inputChannel::3;
    Mult2::outPort::outputChannel::2 unifies
                                      Proportionnal_Command_Value::inPort::inputChannel;
    Rdv4::outPort::outputChannel unifies StaticPosture::inPort::inputChannel;
    StaticPosture::outPort::outputChannel unifies Mult3::inPort::inputChannel;
    Mult3::outPort::outputChannel::1 unifies Robot_Command_Static::inPort::inputChannel;
    Mult3::outPort::outputChannel::2 unifies Rdv5::inPort::inputChannel::1;
    Proportionnal_Command_Value::outPort::outputChannel
                                            unifies Rdv5::inPort::inputChannel::2;
    Rdv5::outPort::outputChannel unifies Robot_Command_Proportionnal::inPort::inputChannel;
    Robot_Command_Static::outPort::outputChannel unifies CMD_Static::inputChannel;
    Robot_Command_Proportionnal::outPort::outputChannel
                                            unifies CMD_Proportionnal::inputChannel
  }
}
```

## III.B. Le fichier de spécification des services externes

```
value Static is external service from graph Gant where is removing {
    components{Proportionnal_Command_Value, Robot_Command_Proportionnal}
    rdv{Rdv5}
    events{CMD_Proportionnal}
};
value Proportionnal is external service from graph Gant where is removing {
    components{Robot_Command_Static}
    events{CMD_Static}
}
```

## III.C. Le fichier de spécification des modes

```
value Static_Mode is mode with {Static};
value Proportionnal_Mode is mode with {Proportionnal}
```

# *IV. Le processus de conception*

Les différentes étapes du processus de conception présenté dans le chapitre précédent vont être reprises.

## IV.A. Compilation et vérification du graphe des services internes

Pour mémoire, voici ci-dessous la figure (Figure 7-9) qui représente le processus détaillé de cette étape.

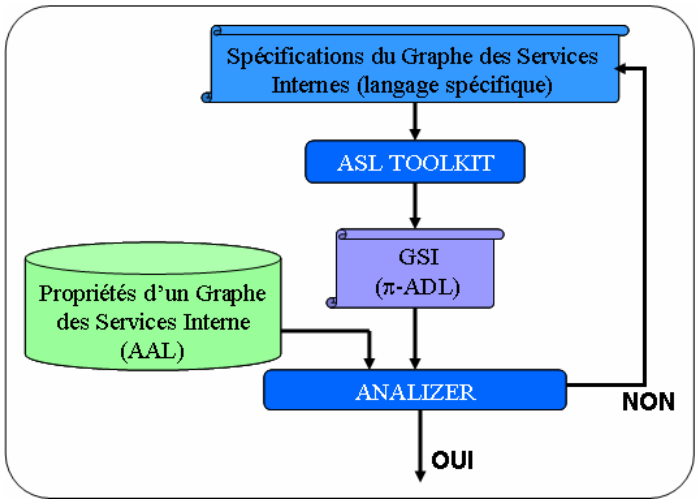



**Figure 7-9 : Rappel du processus de compilation et de vérification du fichier contenant la spécification du graphe des services internes**

La première étape consiste donc à transformer grâce à l'outil ArchWare *ASL Toolkit* le fichier de spécification du graphe des services internes en π-ADL.

L'outil ASL Toolkit ne peut, dans son état actuel de développement, générer ce fichier. Il doit donc être généré à la main. Cela est dû au fait que cet outil n'implémente pas encore la totalité de la grammaire du langage ASL. En fait, la principale notion qui reste à implémenter est celle qui permet de passer des types en paramètre. Cette notion a été

introduite dans le langage ASL suite aux travaux de la présente thèse et *l'ASL Toolkit* n'intègre pas cette nouvelle fonctionnalité du langage. Le fichier qui est créé se trouve dans l'annexe 3.

Une fois le fichier π-ADL créé, il faut qu'il soit vérifié par *l'Analizer*. Pour ce faire, les propriétés qui doivent être vérifiées ont été rassemblées dans un fichier nommé « *prop_graphe.aal* ». Le graphe des services internes se trouve dans un fichier nommé « *gant.adl* ». L'*Analyser* ne permet pas de vérifier les propriétés souhaitées sur notre architecture. En effet, le fichier qui doit être vérifié est beaucoup trop complexe (imbrication de nombreuses abstractions).

Cependant, à titre d'exemple, si l'outil *Analyser* marchait dans notre cas, la vérification se lancerait en tapant la ligne suivante (Figure 7-10) :

```
Checker('gand.adl', 'prop_graph.aal').
```

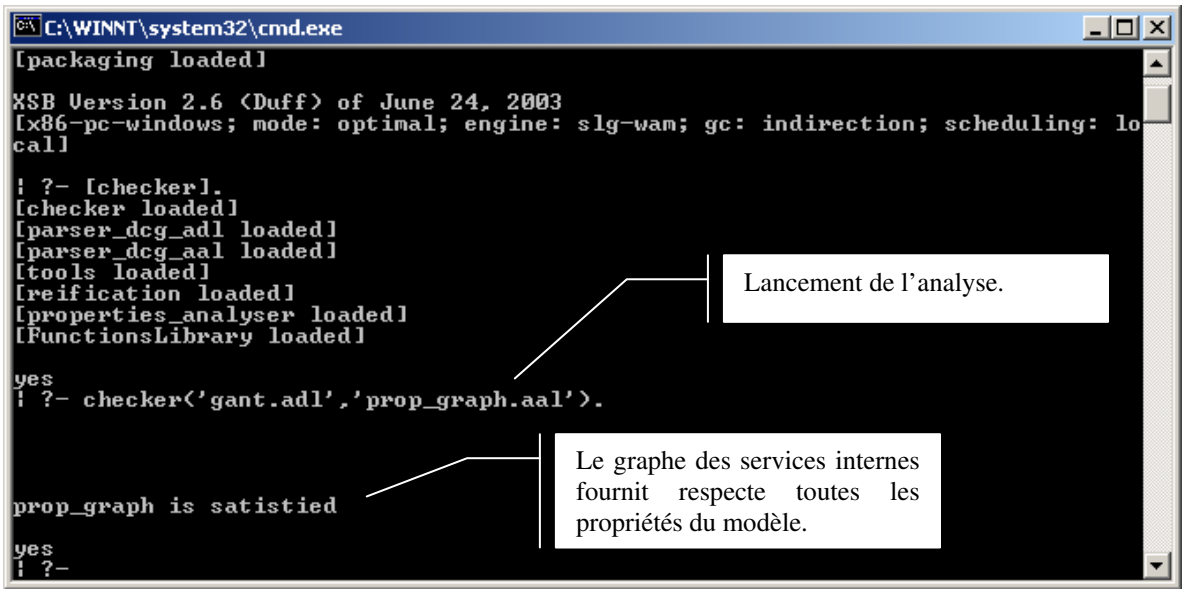



**Figure 7-10 : Vérification du graphe par l'*Analyzer***

*L'Analyser* effectuerait, tout d'abord, une vérification syntaxique du fichier *Gant.adl* afin de produire une représentation interne intermédiaire dont il se servirait pour vérifier les différentes propriétés. En fin d'exécution, *l'Analyzer* nous informerait simplement si les propriétés sont vérifiées (en écrivant *YES*) ou non (en écrivant *NO*).

A titre d'exemple, une erreur avait été introduite dans la conception du gant. Elle serait détectée lors de l'analyse et une réponse négative serait renvoyée (voir Figure 7-11 ci-dessous).

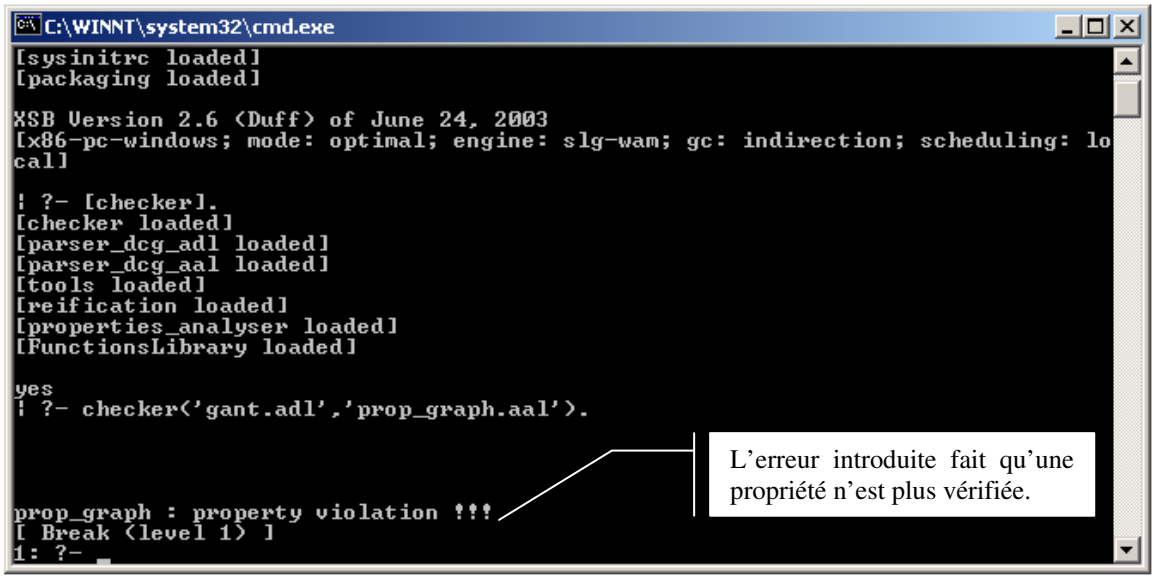



**Figure 7-11 : Réponse négative de l'Analyser lors de l'introduction de l'erreur**

## IV.B. Compilation et vérification des fichiers de spécification des services externes et des modes

Pour mémoire, la figure ci-dessous (Figure 7-12) reprend le détail du processus.

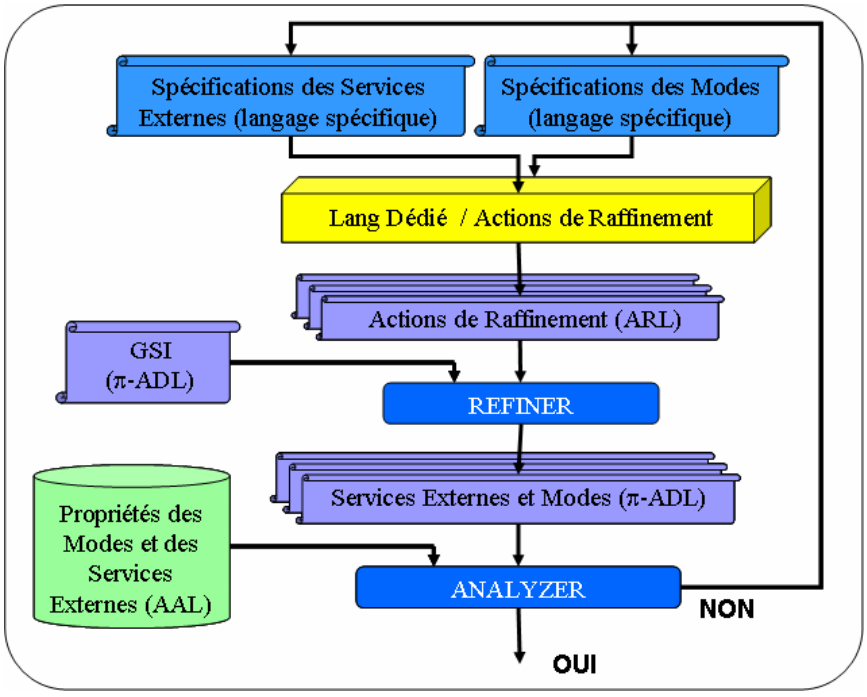



**Figure 7-12 : Rappel du processus de compilation et vérification des services externes et des modes**

La première étape qui consiste à transformer les deux fichiers de spécifications en action de raffinement ARL est effectuée par un outil spécifique qui se nomme *Convertisseur*. La figure ci-dessous (Figure 7-13) montre l'utilisation de cet outil dans notre exemple :

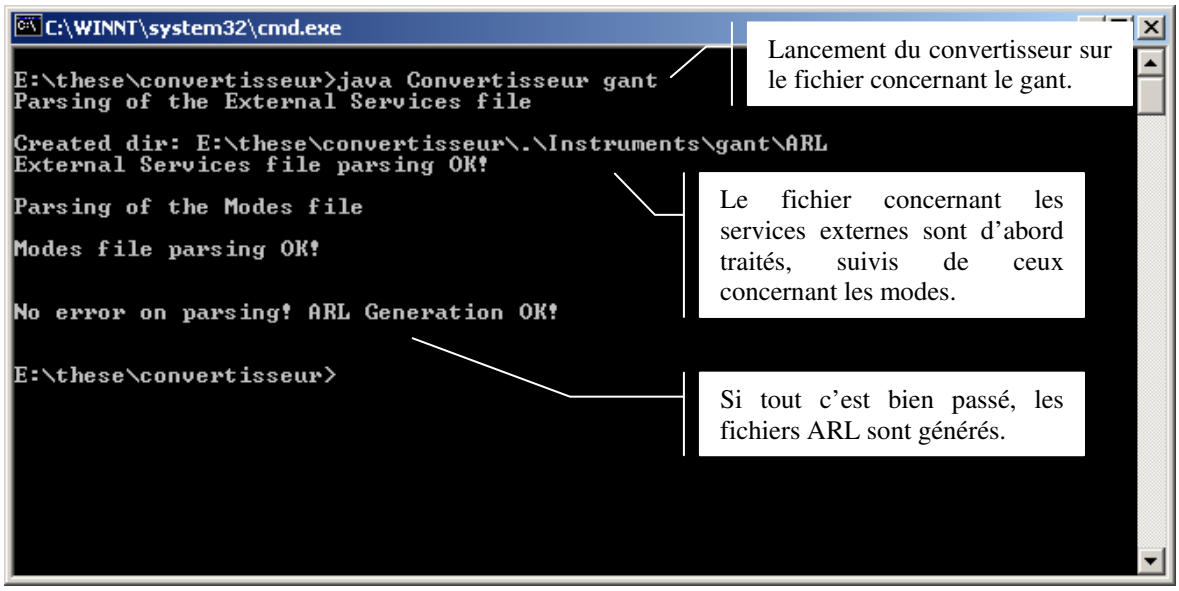



**Figure 7-13 : Exécution de l'outils spécifique *Convertisseur***

Les quatre fichiers générés par le C*onvertisseur* (un par mode et un par service externe) se nomment :

- Mode_Proportionnal_Mode.arl
- Mode_Static_Mode.arl
- SE_Proportionnal.arl
- SE_Static.arl

Les quatre fichiers générés se trouvent dans l'annexe 3.

L'outil qui est utilisé pour gérer la partie raffinement de notre processus de conception se nomme gMDEnv [Manset et al. 05]. Il intègre et étend l'outil Refiner créé pendant le projet ArchWare. Ci-dessous, une copie d'écran est présentée (Figure 7-14).

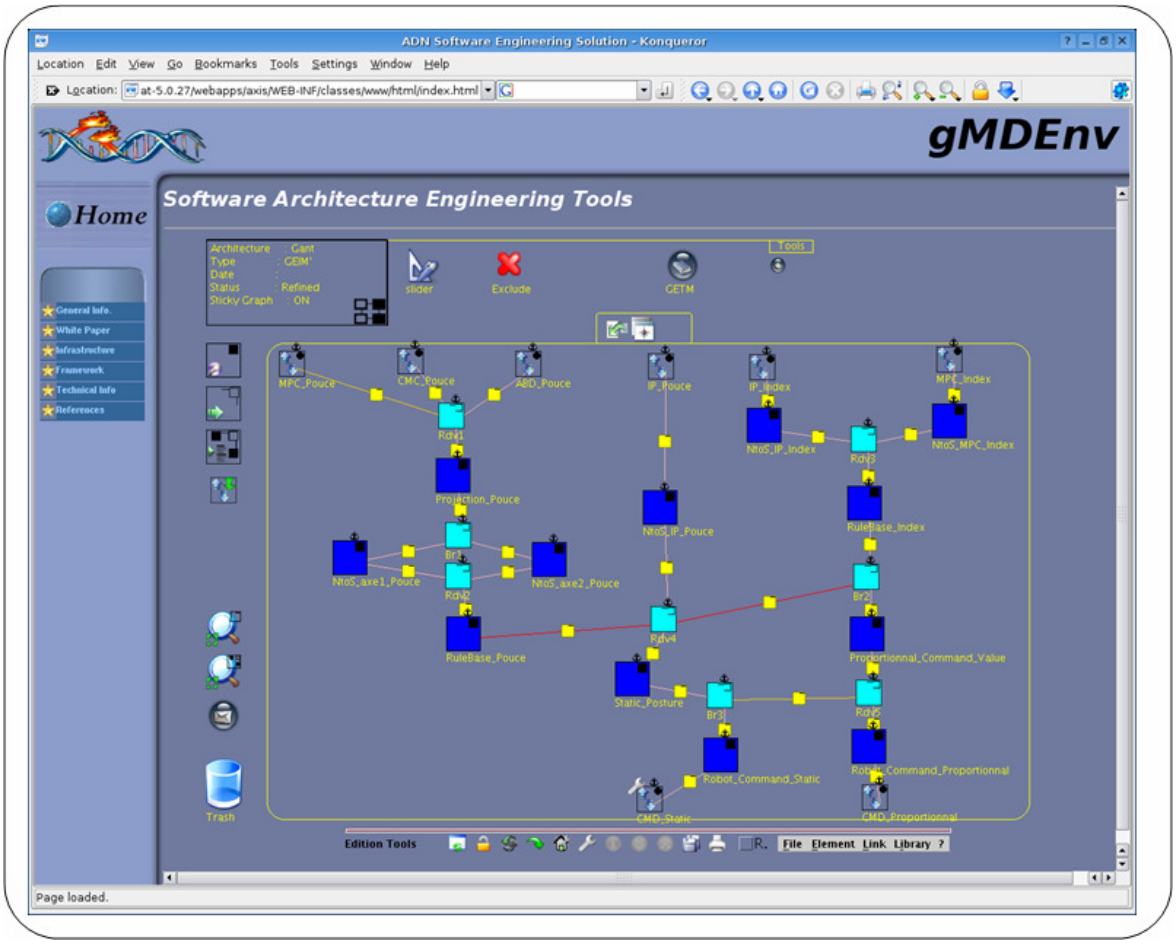



**Figure 7-14 : Vue d'ensemble de l'application gMDEnv**

L'application gMDEnv, bien qu'étant destinée à fonctionner uniquement avec le langage ARL peut interpréter le langage π-ADL à condition que l'on rajoute un mapping de concepts lui permettant d'interpréter chaque abstraction en tant qu'entité de type composant ou connecteur. Une fois ce mapping ajouté, la spécification du graphe des services internes du gant peut être visualisé (Figure 7-15 ci-dessous).

Après application des actions de raffinement qui concernent le service externe *Proportionnal*, on se retrouve avec l'architecture suivante (Figure 7-16) sur laquelle il faudra vérifier les propriétés des services mais aussi des modes puisque dans cet exemple, chaque mode ne contient qu'un seul service externe.

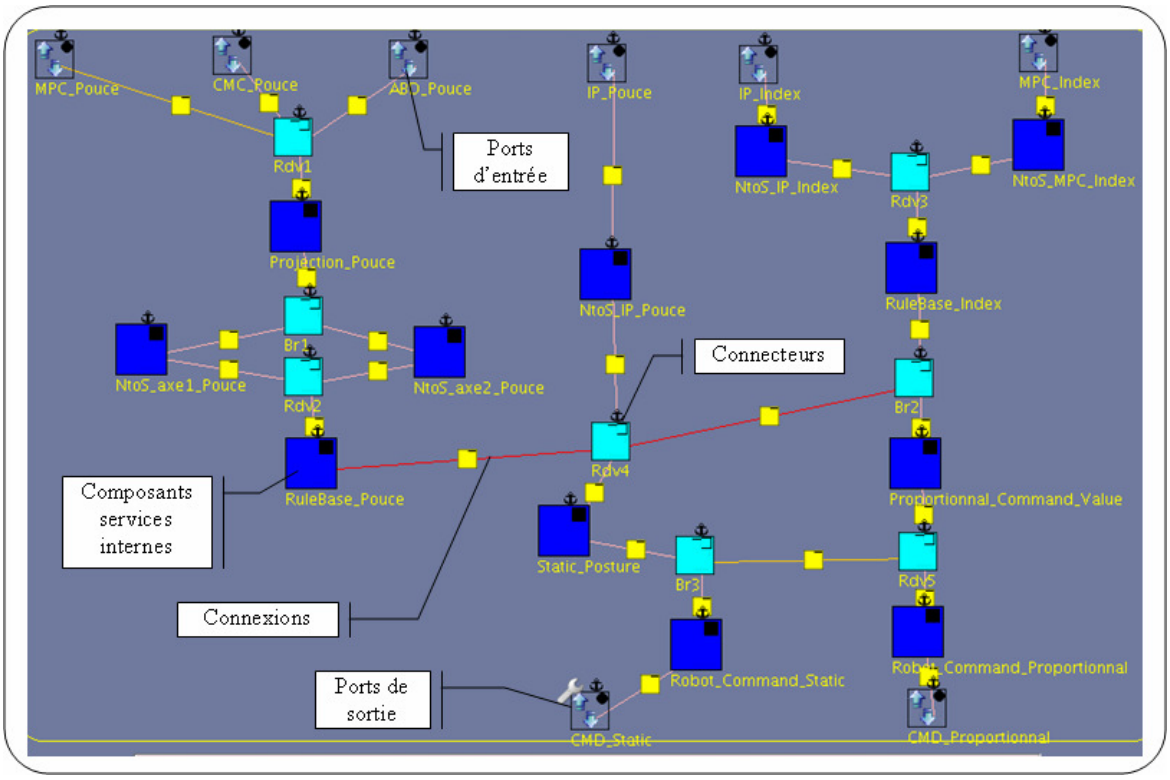



**Figure 7-15 : Zoom sur le graphe des services internes du gant**

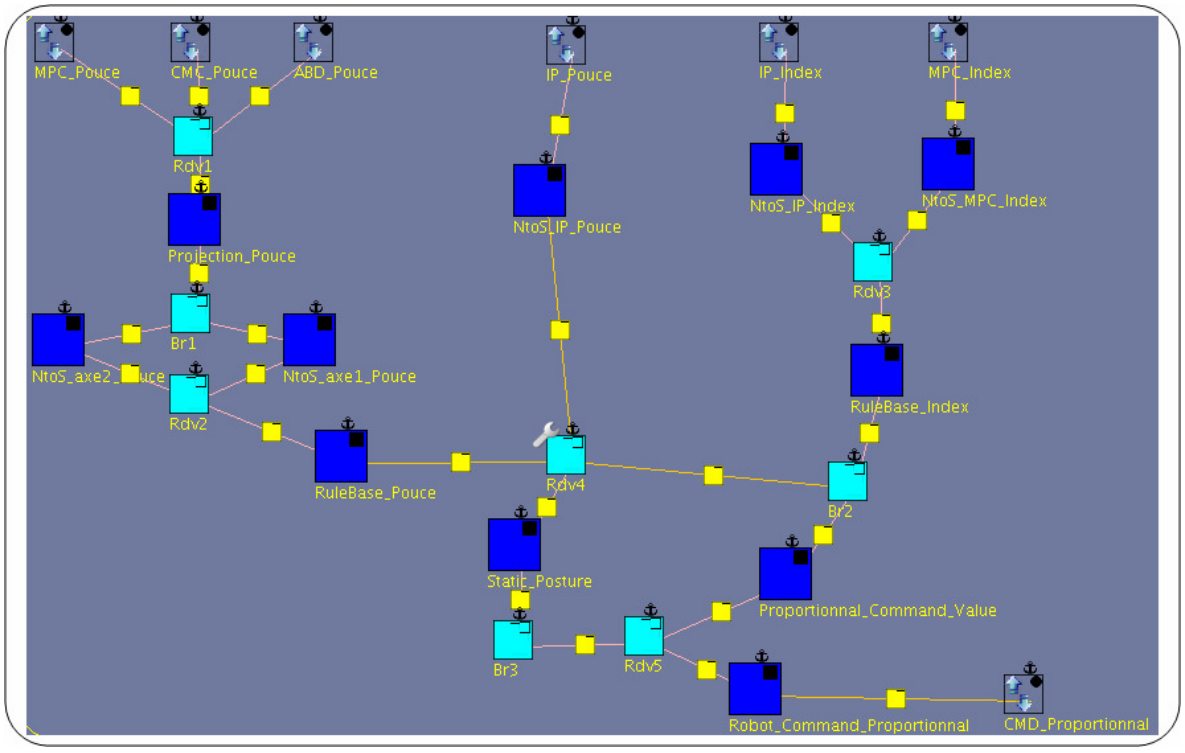



**Figure 7-16 : Architecture du service externe *Proportionnal* et du mode *Proportionnal_Mode***

Actuellement, pour pouvoir faire la même chose avec le service externe *Static*, il faut auparavant recharger l'architecture du service externe complet puis appliquer les actions de raffinements qui correspondent au service externe *Static* après. On obtient l'architecture suivante (Figure 7-17) qui sera, elle aussi celle du mode *Static_Mode* puisqu'il ne contient qu'un seul service externe.

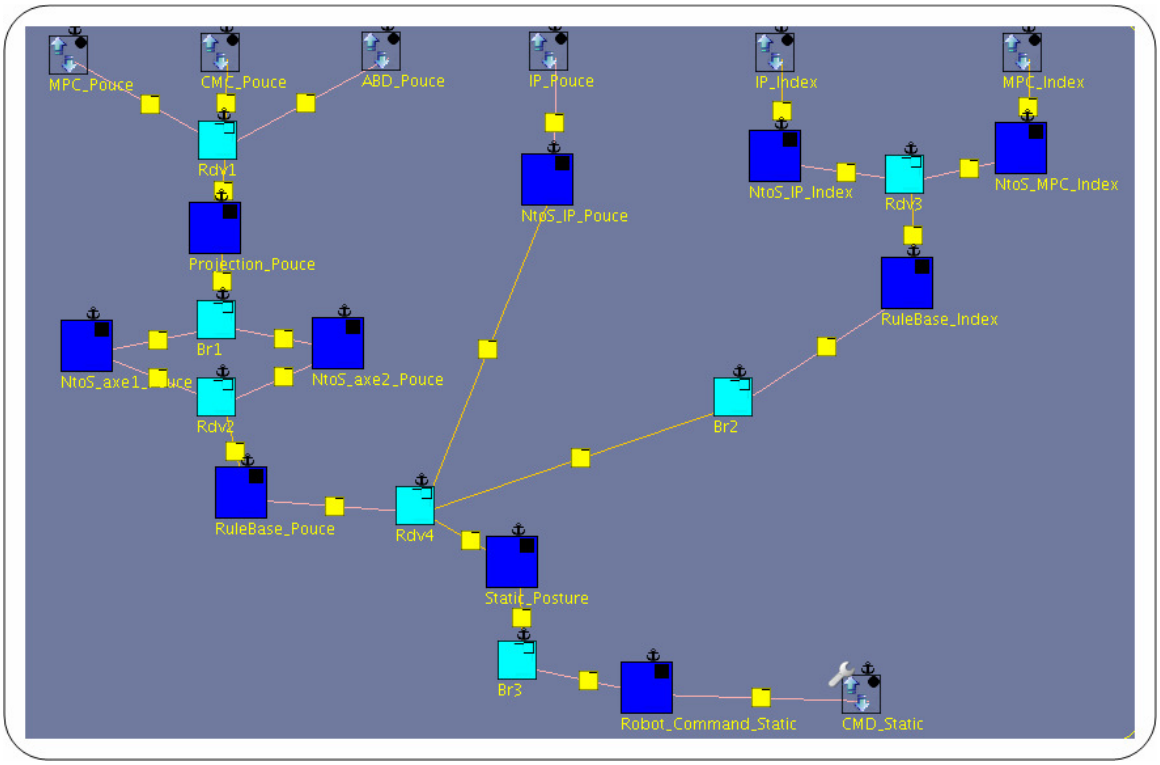



**Figure 7-17 : Architecture du service externe *Static* et du mode *Static_Mode***

Comme il a été dit plus haut, la dernière étape est celle qui concerne la vérification des propriétés que doivent respectées chaque service externe et chaque mode. Le fonctionnement devrait être identique à la vérification des propriétés sur le graphe des services internes.

## IV.C. Génération de l'instrument intelligent en π-ADL et vérification du comportement par animation

Rappelons que cette étape va permettre de générer complètement (sans les opérations en java) le comportement des instruments intelligents afin de pouvoir simuler des exécutions des services externes, des changements de modes et de services externes (voir le rappel du processus Figure 7-18 et Figure 7-19). Le sous-ensemble des fonctionnalités implémentées dans *l'Animator* ne supporte pas cette simulation. De ce fait, l'outil *GenerateurComp* génère le fichier π-ADL sans la dernière abstraction.

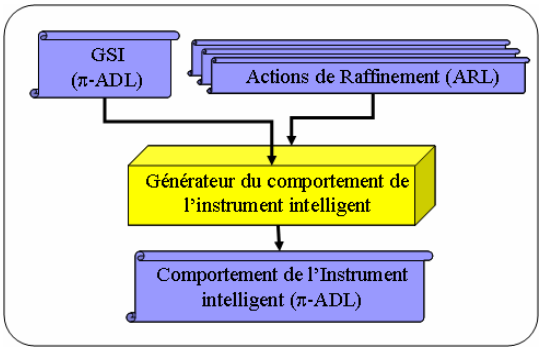



**Figure 7-18 : Rappel du processus de génération de l'instrument intelligent en π-ADL**

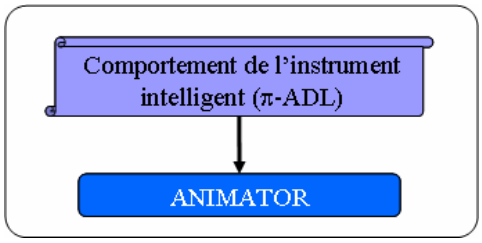



**Figure 7-19 : Rappel du processus de vérification du comportement avec l'Animator**

Afin de pouvoir se rendre compte de ce qu'est capable de représenter *l'Animator*, un fichier, très sommaire, a été créé de façon manuelle afin de pouvoir visualiser le graphe des services internes du gant dans l'outil (voir Figure 7-20 ci-dessous).

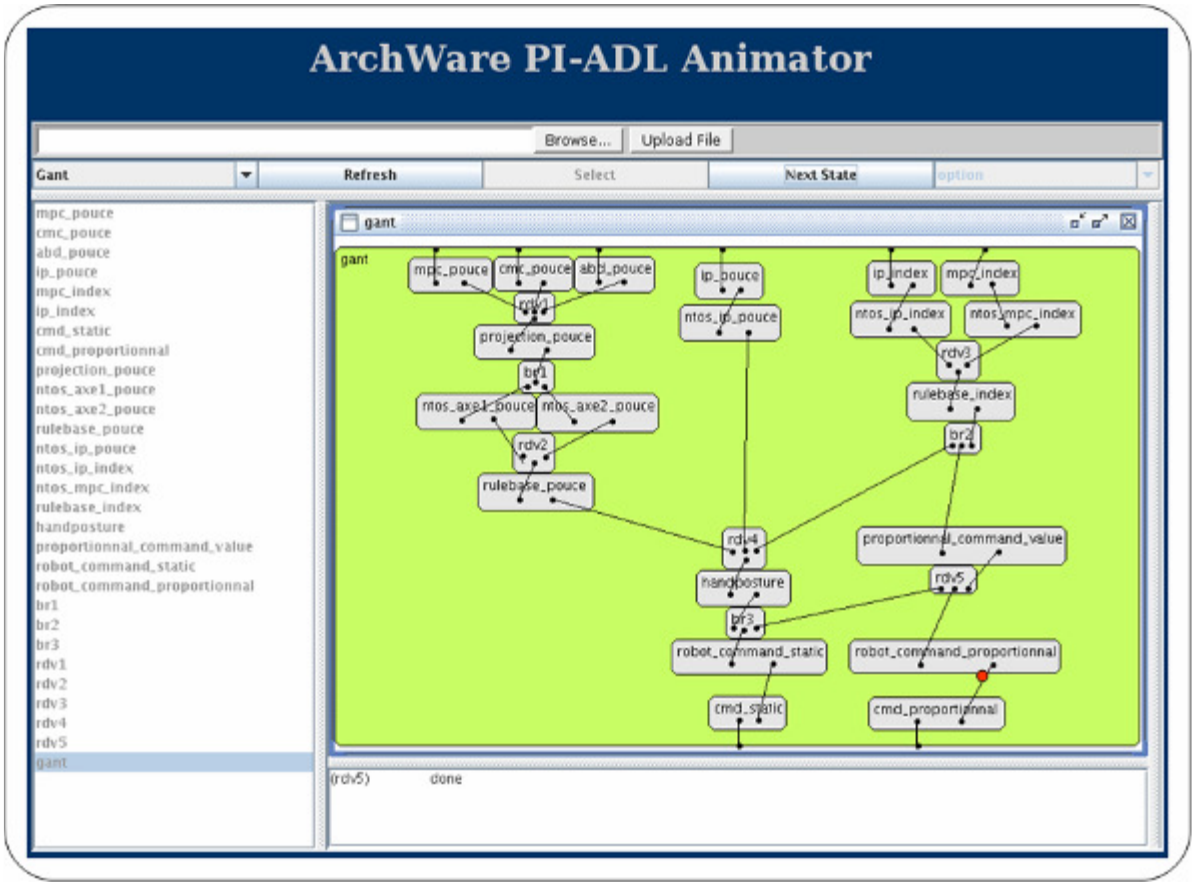



**Figure 7-20 : Le graphe des services internes du gant dans l'outil *Animator***

L'Animator permet de visualiser les transferts de données entre les différents services internes grâce à différentes animations (par exemple, le point rouge sur la figure ci-dessus se déplace sur la connexion du service interne *robot_command_proportionnal* au port de sortie *cmd_proportionnal*).

## IV.D. Génération du code source Java de l'instrument intelligent

La dernière étape du processus consiste à fournir le code java de l'instrument intelligent. Pour rappel, la figure ci-dessous (Figure 7-21) redonne le processus.

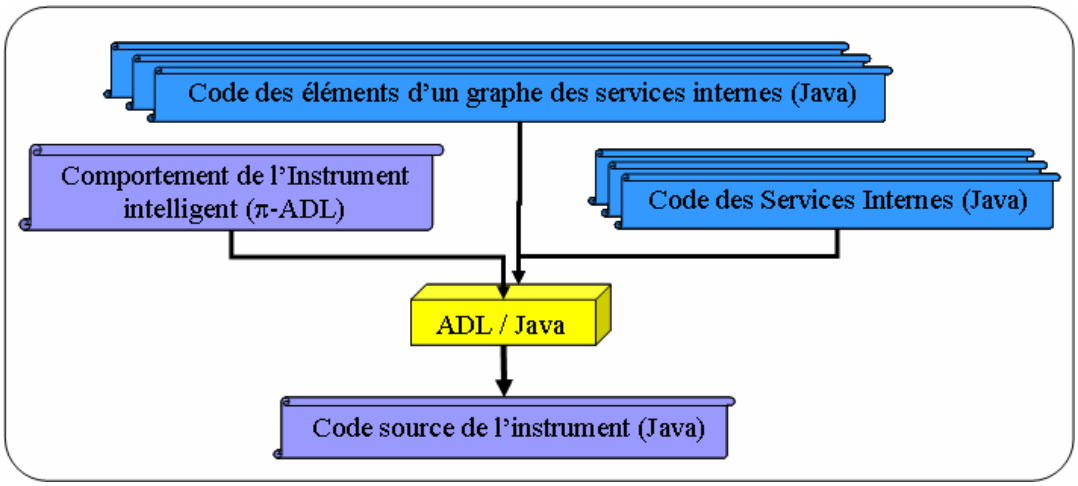



**Figure 7-21 : Rappel du processus de génération du code source de l'instrument intelligent**

Ceci est le processus théorique car comme il a été déjà dit, en réalité, ce n'est pas le fichier de comportement de l'instrument intelligent que l'on utilise mais les trois fichiers fournis par le concepteur d'instruments intelligents.

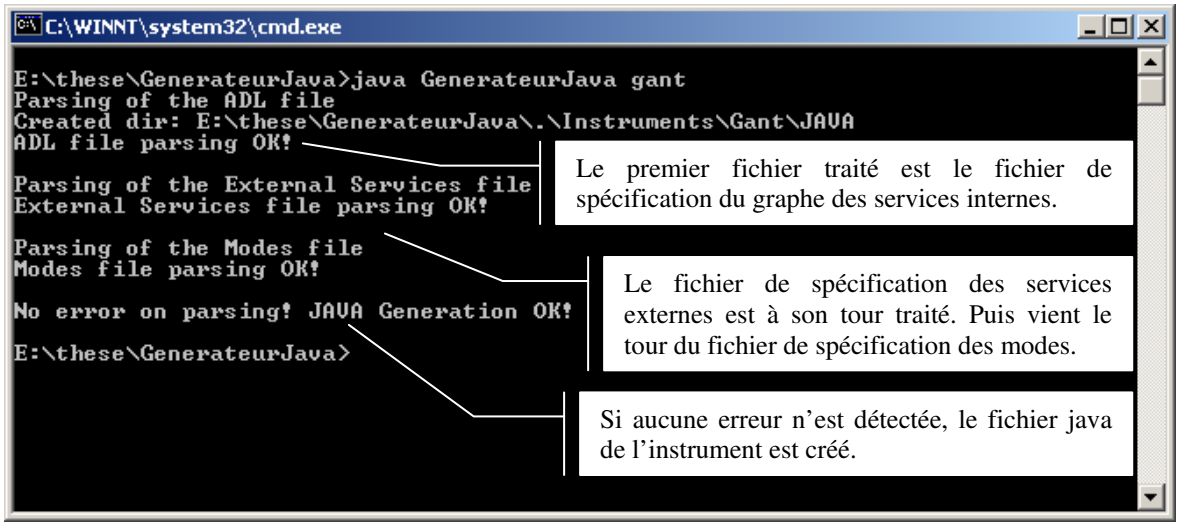
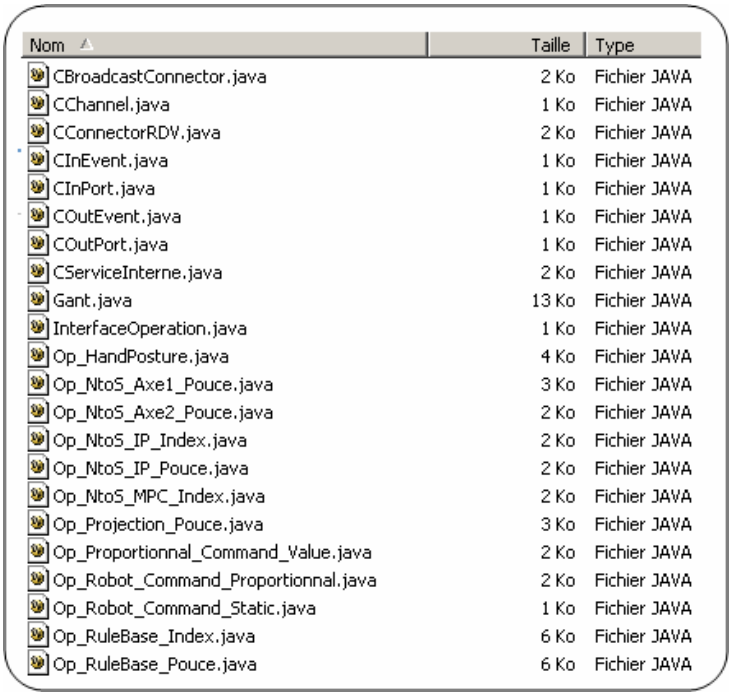



**Figure 7-22 : Exécution de l'outil spécifique GenerateurJava**

L'outil GenerateurJava vérifie avant tout que tous les services internes qui composent le graphe des services internes sont bien présents. C'est seulement à partir de ce moment qu'il vérifie les fichiers fournis. Comme il a été dit dans le chapitre 6, cet outil créé un répertoire dans lequel il va stocker le fichier créé, le code java des éléments qui composent le graphe des services internes et le code java des services internes. Dans le cas du gant, la figure ci-dessous (Figure 7-23) montre ces fichiers.



| Nom | Taille | Type |
|---|---|---|
| CBroadcastConnector.java | 2 Ko | Fichier JAVA |
| CChannel.java | 1 Ko | Fichier JAVA |
| CConnectorRDV.java | 2 Ko | Fichier JAVA |
| CInEvent.java | 1 Ko | Fichier JAVA |
| CInPort.java | 1 Ko | Fichier JAVA |
| COutEvent.java | 1 Ko | Fichier JAVA |
| COutPort.java | 1 Ko | Fichier JAVA |
| CServiceInterne.java | 2 Ko | Fichier JAVA |
| Gant.java | 13 Ko | Fichier JAVA |
| InterfaceOperation.java | 1 Ko | Fichier JAVA |
| Op_HandPosture.java | 4 Ko | Fichier JAVA |
| Op_NtoS_Axe1_Pouce.java | 3 Ko | Fichier JAVA |
| Op_NtoS_Axe2_Pouce.java | 2 Ko | Fichier JAVA |
| Op_NtoS_IP_Index.java | 2 Ko | Fichier JAVA |
| Op_NtoS_IP_Pouce.java | 2 Ko | Fichier JAVA |
| Op_NtoS_MPC_Index.java | 2 Ko | Fichier JAVA |
| Op_Projection_Pouce.java | 3 Ko | Fichier JAVA |
| Op_Proportionnal_Command_Value.java | 2 Ko | Fichier JAVA |
| Op_Robot_Command_Proportionnal.java | 2 Ko | Fichier JAVA |
| Op_Robot_Command_Static.java | 1 Ko | Fichier JAVA |
| Op_RuleBase_Index.java | 6 Ko | Fichier JAVA |
| Op_RuleBase_Pouce.java | 6 Ko | Fichier JAVA |

**Figure 7-23 : Répertoire contenant les fichiers java du gant**

Il peut y avoir toutefois besoin de rajouter manuellement quelques fichiers dans ce dossier dans le cas où les opérations des services internes font appel à d'autres bibliothèques java.

# V. Conclusion

Le but de ce chapitre était donc de valider notre approche pour la conception des instruments intelligents qui suivent le modèle que nous avons spécifié (lui même fondé sur les spécifications des outils ArchWare). Les outils ArchWare, n'ayant implémenté que des sous-ensembles des fonctionnalités spécifiées, ne permettent pas à l'heure actuelle une réalisation complète de notre spécification.

Néanmoins, aux vues de ce qui a été fait, on peut dire que, doté des outils qui implémenteraient complètement les spécifications, l'approche permettrait de concevoir aisément des instruments intelligents. De plus, toutes les vérifications nécessaires pourraient être vérifiées en phase de conception. Enfin, grâce au processus spécifié, des modifications pourront être faites au niveau du modèle sans que les outils qui le compose soient modifiés (sauf pour la transformation du langage spécifique basé sur ARL). Dans notre solution, les outils ArchWare servent à transformer le langage spécifique en π-ADL et à vérifier tout ce que le concepteur d'instrument fournis. Le code Java lui même est généré par un outil spécifique nommé GenerateurJava. Cet outil marche actuellement très bien. Pour qu'il soit pleinement conforme à ce qui avait été prévu au départ, il suffira de modifier la partie *parser* qui, au lieu de prendre en entrée les trois fichiers fournis par le concepteur d'instrument, prendra le fichier qui contiendra le comportement complet de l'instrument intelligent.

# Chapitre 8 : CONCLUSION

# Conclusion

Un concepteur d'instruments connaît, à l'heure actuelle, de nombreuses difficultés pour concevoir un instrument intelligent :

- la conception d'un instrument intelligent demande des compétences en informatique, en électronique, en physique et en mécanique. Or peu de personnes possèdent de telles compétences,
- les instruments intelligents sont fréquemment utilisés dans des environnements critiques. Le logiciel conçu doit alors répondre à des contraintes de sûreté de fonctionnement sévères.
- la configuration économique actuelle oblige les industriels à être innovant, à produire vite et à des coûts réduits. Or, dans le cas des instruments intelligents, ces différents objectifs sont difficiles à atteindre.

Comme il a été présenté dans le chapitre 1, ces difficultés peuvent être fortement réduites en effectuant des améliorations à différents niveaux de la conception des instruments intelligents :

1. Créer d'un modèle générique pour la partie logiciel des instruments intelligents. Cela signifie que ce modèle doit être valable pour n'importe quel type d'instrument intelligent.
2. Vérifier la conformité du logiciel créé par rapport au modèle le plus tôt possible dans le processus de développement. C'est-à-dire que l'on doit être capable d'effectuer, en phase de conception, toutes les vérifications nécessaires au respect du modèle de conception.
3. Avoir un modèle générique qui puisse évoluer facilement en fonction de l'apparition de nouveaux besoins dans le domaine ou de nouvelles technologies.
4. Avoir un processus de conception qui permette un passage aisé entre le modèle générique et le code source du logiciel à créer. L'idéal étant une génération automatique du code source depuis le modèle.
5. Le processus de conception ne doit pas subir de modifications importantes en cas d'évolution du modèle. Ceci concerne le processus en lui-même mais aussi les différents outils qui le compose.

La thèse [Tailland 00] a été réalisée avec pour objectif de créer un modèle et des outils qui facilitent le travail des concepteurs d'instruments intelligents. Ce modèle devait permettre de modéliser n'importe quel type d'instruments intelligents et les outils proposés, regroupés sous le nom de CAPTool, devaient garantir la sûreté et la fiabilité tant au niveau

de la conception que du fonctionnement. Néanmoins, ce type d'outils reste limité sur plusieurs aspects et notamment sur leur faculté d'évolution et leur facilité de mise en œuvre. De plus, on a pu rapidement s'apercevoir que l'on ne pouvait pas concevoir tous les types d'instruments intelligents avec le modèle créé.

Par conséquent, [Tailland 00] répond partiellement au point 1 précédent puisque le modèle n'est pas complet. Il répond complètement au point 2 puisque toutes les propriétés de son modèle sont vérifiées en phase de conception. Il répond au point 3 par l'utilisation d'un modèle en couche ou chacune est basée sur la théorie des graphes. Il répond au point 4 car une génération automatique du code est possible dans son approche. Par contre le point 5 n'est pas du tout pris en compte. En effet, si le modèle évolue, toute l'implémentation des outils qui se trouvent dans le processus doit être refaite.

Les travaux présentés dans ce manuscrit se positionnent dans la suite de la thèse de [Tailland 00] et ont pour but de repenser à la fois le modèle des instruments intelligents et les outils existants car il est apparu que les limitations auxquelles nous étions confrontées étaient à la fois liées au modèle de conception qui avait été proposé et à la façon dont avait été implémenté CAPTool.

# I. Bilan et remarques

Le travail réalisé pour cette thèse a donc porté sur deux points :

- Amélioration du modèle de conception des instruments intelligents afin de pouvoir modéliser tous les types d'instruments.
- Amélioration du processus de conception et des outils qui le compose afin que le tout soit évolutif.

***Modification du modèle existant :***

En ce qui concerne la modification du modèle existant, une nouvelle couche répondant aux manques repérés précédemment a été rajoutée : la couche graphe des services internes. Cette couche fait que les services externes ne sont plus traités comme des sous ensembles de l'ensemble des services internes mais comme des sous graphes du graphe des services internes. Grâce à cette nouvelle modélisation, une certaine catégorie d'instruments intelligents qui ne pouvait pas être conçue jusqu'alors l'est désormais (voir chapitre 4 section I). Un autre avantage de cette couche est que l'on peut dès lors énoncer des propriétés sur la structure et sur le comportement global des instruments intelligents.

***Le processus de conception et les outils qui le compose :***

La partie qui concerne l'élaboration d'un nouveau processus de conception et des outils qui le compose a été pensée en utilisant les principes et outils du domaine de la conception centrée architecture. Ce domaine est en plein essor dans l'ingénierie des logiciels. En fait, l'architecture décrit la structure et le comportement du système dans son ensemble, et repose généralement sur la description de composants (unités de calcul localisées et indépendantes), de connecteurs (unités d'interaction entre les composants) et de configuration (ensembles de composants combinés via des connecteurs). D'autre part, ce développement nous offre la possibilité de créer nos propres langages, formels, spécifiques au domaine. Le fait que ces langages soient formels autorise des vérifications accrues en phase de conception. Pour mettre en œuvre concrètement cette approche, le langage de description d'architecture nommé ArchWare ADL a été choisi car il est apparu comme le plus approprié à notre problématique.

La couche graphe des services internes est modélisée grâce au langage spécifique à la notion de style architectural : ASL. Les services externes et les modes sont eux construis par raffinement du graphe des services internes avec le langage ARL. Toutes les propriétés du modèle ont aussi été implémentées dans le langage ASL afin de pouvoir les vérifier en phase de conception. L'apport de cette nouvelle méthode de conception est immense car les langages utilisés, bien qu'ils utilisent une syntaxe appartenant au domaine, sont formels. Ceci permet de faire des vérifications poussées dès la phase de conception. De plus, le langage ASL est construit afin de pouvoir modifier le langage spécifique sans avoir à effectuer aucune autre modification. Il est, en plus, possible d'ajouter ou de supprimer des propriétés très facilement en cas d'évolution du modèle. La partie qui est plus difficile à modifier est celle qui gère le langage spécifique placé au dessus du langage ARL car cette fonctionnalité ne fait pas partie de l'ADL lui même.

Un nouveau processus de conception basé en majorité sur les outils issus du projet ArchWare a donc été conçu. Il comporte quelques outils spécifiques et notamment un chargé de la génération automatique du code source Java qui sera directement embarqué dans les instruments intelligents. Le fait d'utiliser en majorité des outils ArchWare diminue grandement (voir annule) les différents besoins de modification du processus qui pourraient survenir en cas d'évolution du modèle.

Les outils ArchWare, à la base, n'ont pas été créés pour être utilisés dans la conception des instruments intelligents. Ce qui peut apparaître comme un désavantage au début s'est très vite transformé en un atout. En effet, du point de vue des instruments intelligents, ArchWare a apporté le formalisme qu'il manquait jusqu'alors pour que la conception soit sûre. Mais les apports se sont faits dans les deux sens. Les langages ArchWare ont bénéficié de nos travaux (surtout le langage ASL) pour s'étoffer, s'améliorer. Par exemple, on s'est aperçu que le langage ASL était trop rigide au départ pour pouvoir concevoir des éléments génériques qui étaient structurés de façon identique mais qui transportaient des données de types différents. Le langage ASL a donc été amélioré en ajoutant la possibilité de passer des types en paramètre à un style. Le seul désavantage de ces améliorations est que l'implémentation des outils n'a pas suivi par manque de temps. Par conséquent, le processus qui a été créé pour la conception des instruments intelligents ne peut être utilisé tel quel actuellement.

***Remarque :***
Il est intéressant de remarquer que depuis quelques années, le fait de vouloir vérifier des logiciels en phase de conception est une approche qui commence à se démocratiser et nos travaux en sont la preuve. Le fait d'utiliser une telle approche n'est pas quelque chose de nouveau, cependant, jusqu'à présent, ce type de conception était réservé à des entreprises ou à des domaines qui avaient de gros moyens financiers et pour qui la moindre erreur dans la conception d'un logiciel entraînait immédiatement une perte financière énorme (ex : le domaine de l'aérospatiale). De plus, ce type d'approche demandait jusqu'alors beaucoup plus de temps qu'une conception classique et n'était pas à la portée de tous car les langages utilisés (par exemple le langage B) sont des langages complexes.

Les langages de description d'architectures commencent à mettre à portée de tous ce type d'approche car les efforts qui doivent être fournis en phase de conception ont fortement diminués. En effet, grâce à l'intégration de mécanismes tel que la création de langages

formels spécifiques, les notions d'héritage et d'agrégation, plusieurs points ont été nettement améliorés :

- les langages ne sont plus réservés à des spécialistes,
- les possibilités de réutilisation ont été accrues,
- le temps de conception a nettement diminué.

Tout ceci va sans doute nous emmener à une démocratisation de ce type de conception.

# II. Perspectives

Les perspectives de ces travaux sont diverses et peuvent être explorées en trois temps :

- court terme,
- moyen terme,
- long terme.

***Court terme :***

Une des perspectives à court terme de ces travaux est de pouvoir réaliser une validation complète de l'approche. Pour ce faire, la plus grande partie du travail va résider dans le fait de terminer les outils ArchWare afin qu'ils implémentent toutes les fonctionnalités prévues dans leurs spécifications.

Une deuxième perspective de recherche serait l'intégration de nos travaux dans l'application gMDEnv [Manset et al. 05]. Nous avons vu que cette application vise à implémenter les concepts relatifs à l'ingénierie des models (IDM) dans le cadre de la génération automatique d'applications Grid. Il est envisageable de l'étendre pour faire en sorte qu'elle puisse générer le code source des instruments intelligents. En effet, cette application, basée à l'origine sur le paradigme architecture orientée services, délivre un ensemble de fonctionnalités, dédiées aux systèmes Grid, similaires aux besoins rencontrés dans le domaine des instruments intelligents. De part sa conception, il est possible d'étendre le champ applicatif de cet environnement de développement par l'ajout de modèles propres à un domaine spécifique. Ainsi, dans le cadre de la conception d'instruments intelligents, il serait possible d'intégrer le modèle présenté dans nos travaux. L'application fournirait alors au concepteur d'instruments intelligents une interface graphique lui permettant de concevoir visuellement ses instruments et de générer automatiquement le code source. Il ne faut pas perdre de vue que le type de conception itilisé alors serait toujours une conception centrée architecture.

***Moyen terme :***

La perspective que l'on peut envisager à moyen terme est l'extension de notre approche à la conception d'une application complète, qui comporterait plusieurs instruments intelligents. Sachant que ces derniers peuvent interagir, c'est à dire qu'un instrument peut demander l'exécution d'un service à un autre, on peut imaginer pouvoir construire le graphe des services internes de l'application complète, qui serait en fait l'association de tous les graphes des services internes qui composent l'application. A partir de ce moment, d'autres questions de recherchent pourrais faire leur apparitions :

- Est ce qu'il y a des propriétés spécifiques aux applications composées d'instruments intelligents ?

- Les propriétés qui sont vraies pour le graphe des service internes des instruments intelligents sont elles toujours vraies pour le graphe d'une application totale ? (par exemple, le rebouclage au sein du graphe complet de l'application)

***Long terme :***

A plus long terme, une perspective importante sera la prise en compte d'une donnée importante dans le monde de l'instrumentation intelligente et qui manque actuellement au niveau du modèle : la notion de temps d'exécution des services. En effet, dans certaines applications, le temps d'exécution d'un service est une donnée importante car elle peut déterminer par exemple l'exactitude d'une mesure ou encore si un matériel est en panne ou non (un service interne qui doit durer en moyenne cents millisecondes et qui n'est toujours pas terminé au bout d'une seconde présente selon toute vraisemblance une anomalie). Il serait donc intéressant de pouvoir vérifier, en phase de conception, si des contraintes temporelles peuvent être vérifiées. Il pourrait y avoir deux méthodes possibles pour envisager ces vérifications. La première serait l'intégration dans ArchWare de cette notion temporelle et la deuxième serait de construire notre modèle pour faire en sorte de la simuler.

# REFERENCES BIBLIOGRAPHIQUES

**[Abd-Allah 96]**
A. Abd-Allah, *"Composing Heterogeneous Software Architecture",* Thèse. Center for Software Engineering, University of Southern California, 1996.

**[Abowd et al. 93]**
G. Abowd, R. Allen, D. Garlan., "*Using Style to Give Meaning to Software Architecture",* SIGSOFT'93:Foundation Software Eng., ACM, New York, 1993.

**[Abowd et al. 95]**
G. Abowd, R. Allen, D. Garlan, "*Formalizing Style to Understand Descriptions of Software Architecture",* ACM transaction on Software Engineering and Methodology, vol. 4, pp. 319-364, octobre 1995.

**[Allen&Garlan 94]**
R. Allen, D. Garlan, *"Formalizing Architectural Connection",* Proceedings of 16th International Conference Software Engineering, IEEE Computer Soc. Press, Los Alamitos, Californie, pp. 71-80, 1994.

**[Allen 97]**
R. Allen, *"A Formal Approach to Software Architecture"*, PhD thesis, Carnegie Mellon University, 1997.

**[Allen&Garlan 97]**
R. Allen, D. Garlan, *"A Formal Basis for Architectural Connection"*, ACM Transactions on Software Engineering and Methodology, juillet 1997.

**[Allen et al. 98]**
R. Allen, R. Douence, D. Garlan, *"Specifying and Analysing Dynamic Software Architectures"*, Proceedings of 1998 Conference on Fundamental Approaches to Software Engineering, Lisbonne, Portugal, mars 1998.

**[Allevard 05]**
Allevard T., *"Représentation symbolique de la configuration de la main - application à la reconnaissance de signes et au contrôle d'un robot mobile"*, Thèse de doctorat, Université de Savoie, juillet 2005

**[Alloui et al. 02]**
I. Alloui, H. Garavel, R. Mateescu, F. Oquendo, *"The ArchWare Architecture Analysis Language"*, ARCHWARE European RTD Project IST-2001-32360. Deliverable D3.1b, décembre 2002.

**[André 96]**
P. André, *"Méthodes formelles et à objets pour le développement du logiciel : Études et propositions"*, Thèse de Doctorat, Institut de Formation Supérieure en Informatique et Communication, Université de Rennes I, 7 juillet 1995.

**[ArchWare 02]**
ArchWare, *"Architecting Evolvable Software",* ARCHWARE European RTD Project IST-2001-32360, www.arch-ware.org. 2002-2005.

**[Pourraz et al. 05]**
F. Pourraz, H. Verjus, S. Azaiez, F. Oquendo, C. Zavattari, *"Extended ArchWare Architecture Animator – Release 2.1",* ARCHWARE European RTD Project IST-2001-32360. Deliverable D2.2d, avril 2005.

**[Azaiez&Oquendo 05]**
S. Azaiez, F. Oquendo, *"Final ArchWare Architecture Analysis Tool by Theorem Proving",* ARCHWARE European RTD Project IST-2001-32360. Deliverable D1.2b, December 2005.

**[Balasubramaniam et al. 04]**
D. Balasubramaniam, R. Morrison, G. Kirby, K. Mickan, S. Norcross, *"ArchWare Code Synthesiser"*, ARCHWARE European RTD Project IST-2001-32360. Deliverable D6.4a, mars 2004.

**[Bayart&Staroswiecki 92]**
M. Bayart, M. Staroswiecki, *"Smart Actuators : Generic Functional Architecture, Service and Cost Analysis"*, IEEE SICICI'92, Singapour International Conference on Intelligent Control and Instrumentation, Singapore, 17-21 février 1992, pp. 642-646.

**[Bayart&Staroswiecki 93]**
M. Bayart, M. Staroswiecki, *"A Generic Functional Model of Smart Instrument for Distributed Architectures"*, Intelligent Instrumentation for remote and on site measurement, Bruxelles, Belgique, 12-13 mai 1993.

**[Bayart& Staroswiecki 94]**
M. Bayart, M. Staroswiecki, *"Intelligent Components and Instruments for Control Applications"*, SICICA'94, 2nd IFAC Symposium on Intelligent Components and Instruments for Control Applications, Budapest, Hongrie, 8-10 juin 1994.

**[Bayart 94]**
M. Bayart, *"Instrumentation intelligente - Systèmes automatisés de production à intelligence distribuée"*, Habilitation à Diriger des Recherches, Université de Sciences et Technologies de Lille, 21 décembre 1994.

**[Bass et al. 99]**
L. Bass, P. Clements, R. Kazman, *"Software Architecture in Practice"*. Addison Wesley, ISBN 0-201-19930-0, 1999.

**[Beaudoin&Favennec 93]**
F. Beaudoin, J.M. Favennec *"Les capteurs intelligents : le concept et les enjeux"*, Revue Générale de l'Électricité, Mars 1993, No 3, pp. 1-8.

**[Benoit et al. 01]**
Benoit E., Foulloy L., Tailland J., *"InOMs model: a Service Based Approach to Intelligent Instruments Design"*, 5th World Conf. on Systemics, Cybernetics and Informatics (SCI 2001), Vol. XVI, Orlando, USA, July 2001, pp. 160-164.

**[Benkhellat 95]**
M. L. Benkhellat, *"Formalisation et vérification de l'interopérabilité dans les systèmes de communication"*, Thèse de Doctorat, Institut National Polytechnique de Lorraine, Nancy, 4 janvier 1995.

**[Bergamini et al. 04]**
D. Bergamini, D. Champelovier, N. Descoubes, H. Garavel, R. Mateescu, W. Serwe, *"ArchWare Architecture Analysis Tool by Model-Checking"*, ARCHWARE European RTD Project IST-2001-32360. Deliverable D3.6b, juin 2004.

**[Binns et al. 96]**
P. Binns, M. Engelhart, M. Jackson, S. Vestal, *"Domain-Specific Software Architectures for Guidance, Navigation, and Control",* International Journal of Software Engineering and Knowledge Engineering, 1996

**[Boasson 95]**
M. Boasson, *"The Artistry of Software Architecture"*. Guest editor's introduction, IEEE Software, 1995.

**[Boehm et al. 94]**
B. Boehm, P. Bose, E. Horowitz et M. J. Lee, *"Software Requirements Negotiation and Renegociation Aids: A Theory-W Based Spiral Approach"*. Proceedings of the 17th International Conference on Software Engineering, 1994.

**[Bouras 97]**
A. Bouras, *"Contribution à la conception d'architectures réparties : modèles génériques et interopérabilité d'instruments intelligents"*, Thèse de Doctorat, Université des Sciences et Technologies de Lille, Juillet 1997.

**[Buschmann et al. 96]**
F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad et M. Stal, « *Pattern Oriented Software Architecture: A System of Patterns* ». John Wiley & Sons, 1996.

**[Calvez 90]**
J. P. Calvez, *"Spécification et conception des systèmes - une méthodologie"*, Édition Masson, Paris, 1990.

**[Chaudet&Oquendo 01]**
C. Chaudet, F. Oquendo, *"π-SPACE: Modeling Evolvable Distributed Software Architectures"*, Proceedings of International Conference PDPTA'01, Las Vegas, juin 2001.

**[CIAME 87]**
CIAME, *"Livre blanc : Les capteurs intelligents - Réflexions des utilisateurs"*, CIAME AFCET, 1987.

**[Ciancarini&Mascolo 96]**
P. Ciancarini, C. Mascolo *"Analysing and Refining an Architectural Style"*. Proceedings of 10th International Conference of Z Users (ZUM'97), 1996.

**[Cimpan et al. 02]**
S. Cimpan, F. Oquendo, D. Balasubramaniam, G. Kirby, R. Morrison, *"The ArchWare ADL: Definition of the Textual Concrete Syntax"*, ArchWare European RTD Project IST-2001-32360, Deliverable D1.2b, décembre 2002.

**[Cimpan et al. 03]**
S. Cimpan, F. Leymonerie, F. Oquendo, *"The ArchWare Foundation Styles Library"*, Report R1.3-1, ArchWare European RTD Project, IST-2001-32360, juin 2003.

**[Clavel et al. 03]**
M. Clavel, F. Durán, S.Eker, P. Lincoln, N.Martí-Oliet, José Meseguer and C.Talcott. *"The Maude 2.0 System"*. In Proc. Rewriting Techniques and Applications, 2003, Springer-Verlag LNCS 2706, 76-87, June 2003.

**[Clements et al. 95]**
P. Clements, L. Bass, R. Kazman et G. Abowd, *"Predicting Software Quality by Architecture-Level Evaluation"*. Proceedings of the Fifth International Conference on Software Quality, Austin, Texas, octobre 1995.

**[Coglianese&Szymanski 93]**
L. Coglianese, R. Szymanski, *"DSSA-ADAGE: An Environment for Architecture-based Avionics Development"*. Proceedings of AGARD'93, mai 1993.

**[Dasarathy 85]**
B. Dasarathy, *"Timing constraints of real-time systems : constructs for expressing them, methods for validating them"*, IEEE Transactions on Software Engeenering, Vol. 11, No 1, 1985, pp 80-86.

**[DeLine 96]**
R. DeLine, *"Towards User-Defined Elements Types and Architectural Styles"*, Proceedings of the Second International Software Architecture Workshop, pp. 47-49, San Francisco, 1996.

**[Garavel et al. 02]**
H. Garavel, F. Lang, and R. Mateescu, *"Compiler Construction using LOTOS NT"*. In Proceedings of the 11th International Conference on Compiler Construction CC'2002 (Grenoble, France), Lecture Notes in Computer Science vol. 2304, pp. 9-13, avril 2002.

**[Garlan et al. 92]**
D. Garlan, M. Shaw, C. Okasaki, C. Scott, and R. Swonger. "*Experiences with a Course on Architectures for Software Systems",* Proceedings of the 6th SEI Conference on Software Engineering Education, 1992.

**[Garlan&Shaw 93]**
D. Garlan et M. Shaw. *"Introduction to Software Architecture",* Advances in Software Engineering and Knowledge Engineering. World Scientific Publishing Company, 1993.

**[Garlan et al. 94]**
D. Garlan, R. Allen et J. Ockerbloom. *"Exploiting Style in Architectural Design Environments",* SIGSOFT'94, ACM Press, New York, décembre 1994.

**[Garlan 95]**
D. Garlan, *"What is Style?"*. Proceedings of Dagshtul Workshop on Software Architecture, 1995.

**[Garlan et al. 97]**
D. Garlan, R. Monroe, D. Wile, *"ACME: an Architectural Descripion Interchange Language"*, Proceedings of CASCON'97, pp. 169-183, Toronto, novembre 1997.

**[Garlan 00]**
D. Garlan. *"Software Architecture: a Road Map"*. Proc. of the conference on The future of Software engineering, mai 2000.

**[Garlan 01]**
D. Garlan, "*Software Architecture*", Wiley Encyclopedia of Software Engineering, J. Marciniak (Ed.), John Wiley & Sons, 2001.

**[Garlan 03]**
D. Garlan, *"Formal Modeling and Analysis of Software Architecture: Components, Connectors and Events"*, Présentation à the Third International School on Formal Methods for the Design of Computer, Communication and Software Architectures, SFM 2003 Bertinoro, Italy, septembre 2003.

**[Géhin 94]**
A.L. Géhin, *"Analyse fonctionnelle et modèle générique des capteurs intelligents - Application à la surveillance de l'anesthésie"*, Thèse de Doctorat, Université des Sciences et Technologies de Lille, 26 janvier 1994.

**[Harel 90]**
D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, *"STATEMATE : A Working Environment for the Development of Complex Reactive Systems"*, IEEE Transactions on Software Engineering, Vol. 16, No 4, 1990, pp. 403-414.

**[Hoare 85]**
C.A.R. Hoare, *"Communicating Sequential Processes"*, Prentice Hall, 1985.

**[ISO 88]**
ISO/IEC LOTOS, *"A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour"*. International Organization for Standardization – Information Processing Systems – Open Systems Interconnection. International Standard no. 8807, Genève, septembre 1989.

**[Johnson& Murray 75]**
S. C. Johnson, Murray Hill, "*YACC - Yet another compiler*", Technical Report, 1975.

**[Jones 91]**
C. B. Jones, *"Does the OO-Community Need Formal Methods"*, Éditions Bertand Meyer et Jean Bezivin, TOOLS'91, Paris, Mars 1991, pp. 15-18.

**[Klein&Kazman 99]**
M. Klein et R. Kazman, *"Attribute-Based Architectural Styles"*, Technical Report CMU/SEI-99-TR-022. ESC-TR-99-022, octobre 1999.

**[Kozen 83]**
D. Kozen, *"Results on the propositional Mu-Calculus"*, Theorotical Computer Science 27, pages 333-354, 1983.

**[Kramer 96]**
Kramer J.F., *"The Talking Glove: Hand-Gesture-to-Speech Using an Instrumented Glove and a Tree-Structured Neural Classifying Vector Quantizer"*, Thèse de doctorat, Université de Stanford, 1996, 197 pages.

**[Lano 93]**
K. Lano, H. Haughton, *"Object-Oriented Specification Case Studies"*, Object Oriented Series, Prentice Hall, 1993.

**[Leymonerie et al. 01]**
F. Leymonerie, S. Cîmpan, F. Oquendo, *"Extension d'un langage de description architecturale pour la prise en compte des styles architecturaux: Application à J2EE"*, Proceedings ICSSEA 14th International Conference on Software and Software Engineering and their Applications, Paris, décembre 2001.

**[Leymonerie et al. 02]**
F. Leymonerie, S. Cîmpan, F. Oquendo *"Etat de l'art sur les styles architecturaux : classification et comparaison des langages de description d'architectures logicielles"*, Revue Génie Logiciel, No. 62, septembre 2002

**[Leymonerie 04]**
F. Leymonerie, *"ASL : un langage et des outils pour les styles architecturaux. Contribution à la description d'architectures dynamiques"*, Thèse de Doctorat, LISTIC, Université de Savoie, Annecy, décembre 2004.

**[Lesk&Schmidt 75]**
M. E. Lesk, E. Schmidt, *"Lex - A Lexical Analyzer Generator",* Bell Laboratories Computing Science Technical Report # 39, October 1975.

**[Luckham et al. 95]**
D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann, *"Specification and analysis of system architecture using Rapide"* Proceedings of the IEEE Transaction on Software Engineering,1995.

**[Luttenbacher 97]**
D. Luttenbacher, *"Modélisation du concept capteur intelligent par une approche orienté objet : application à un capteur intelligent de température"*, Thèse de Doctorat, Université Henri Poincaré, Nancy I, 10 février 1997.

**[Manset et al. 05]**
D. Manset, R. McClatchey, F. Oquendo, H. Verjus, *"A Model-driven Approach for Grid Services Engineering"*, Proceedings of the 2005 International Conference on Software Engineering and their Applications - ICSSEA'05, (France) 2005.

**[Magee et al. 95]**
J. Magee, N. Dulay, S. Eisenbach, J. Kramer., *"Specifying Distributed Software Architectures"*, Proceedings of the Fifth European Engineering Conference (ESEC'95), Barcelona, septembre 1995.

**[Megzari 04]**
K. Megzari, *"REFINER : environnement logiciel pour le raffinement d'architectures logicielles fondé sur une logique de réécriture",* Thèse de Doctorat de l'Université de Savoie, décembre 2004.

**[Megzari&Oquendo 04]**
K. Megzari, F. Oquendo, *"ArchWare Architecture Refinement Tools"*, ArchWare European RTD Project IST-2001-32360, Deliverable D6.3b, avril 2004.

**[Mettala& Graham 92]**
E. Mettala, M. H. Graham, *"The Domain-Specific Software Architecture Program",* Technical report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, juin 1992.

**[Milner 89]**
R. Milner, "*Communication and Concurrency"*. Prentice-Hall, 1989.

**[Milner et al. 92]**
R. Milner, J. Parraw, D. Walker, *"A Calculus of Mobile Processes"*, Information and Computation, pp.1-40, septembre 1992.

**[Monroe&Garlan 96]**
R. Monroe, D. Garlan, *"Style-Based Reuse for Software Architectures"*. Proceedings of the 1996 International Conference on Software Reuse, avril 1996.

**[Monroe et al. 97]**
R.T. Monroe, A. Kompanek, R. Melton, D. Garlan, *"Architectural Styles, Design Patterns, and Objects"*. Proceedings of the IEEE Transactions on Software Enginering, 1997

**[Monroe 98]**
R.T. Monroe, *"Capturing Software Architecture Design Expertise With Armani"*, Technical Report CMU-CS-98-163, 1998.

**[Moriconi et al. 95]**
M. Moriconi, X. Qian, R. Riemenschneider, *"Correct Architecture Refinement"*, IEEE Transactions on Software Engineering, Special Isue on Software Architecture, 21(4) : 356-372, avril 1995.

**[Moriconi&Riemenschneider 97]**
M. Moriconi, R.A. Riemenschneider, *"Introduction to SADL 1.0, A Language for Specifying Software Architecture Hierarchies"*, mars 1997.

**[Oquendo et al. 02]**
F. Oquendo, I. Alloui, S. Cîmpan, H. Verjus, *"The ArchWare ADL: Definition of the Abstract Syntax and Formal Semantics"*, ARCHWARE European RTD Project IST-2001-32360. Deliverable D1.1b, décembre 2002

**[Oquendo 03a]**
F. Oquendo, *"The ArchWare Architecture Description Language: Tutorial"*, Report R1.1-1, ArchWare European RTD Project, IST-2001-32360, mars 2003.

**[Oquendo 03b]**
F. Oquendo, *"Final Definition of the ArchWare Architecture Refinement Language",* ArchWare European RTD Project IST-2001-32360, Deliverable D6.1b, décembre 2003.

**[Oquendo 04]**
F. Oquendo, *"π-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures"*, ACM Press, ACM SIGSOFT Software Engineering Notes, Volume 29, Issue 5. New York, NY, USA September 2004.

**[Oquendo et al. 04]**
F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, C. Occhipinti, *"ArchWare: Architecting Evolvable Software"*, Proceedings of the First European Workshop on Sofware Architecture (EWSA 2004), pp. 257-271, St-Andrews, Ecosse, mai 2004, Springer-Verlag.

**[Perry&Wolf 92]**
D. Perry, A. Wolf, *"Foundations for the Study of Software Architecture"*. ACM SIGSOFT, Software Engineering Notes, Vol. 17, no 4, pp. 40-52, octobre 1992.

**[Rapide 97]**
Rapide Design Team, *"Rapide 1.0. Pattern Language. Reference Manual"*, juillet 1997.

**[Robert et al. 93]**
M. Robert, M. Marchandiaux, M. Porte, *"Capteurs Intelligents et Méthodologie d'Évaluation"*, Paris, France, Septembre 1993, Édition Hermès, 2-86601-382-4, p.176.

**[Rumbaugh et al. 91]**
J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, *"Object Oriented Modeling and Design"*, Prentice Hall International, 1991.

**[Shaw 90]**
M. Shaw, *"Prospects for an Engineering Discipline of Software"*. IEEE Software, novembre 1990.

**[Shaw et al. 95]**
M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, G. Zelesnik, *"Abstraction for Software Architecture and Tools to Support Them"*, IEEE Transactions on Software Engineering: Special Issue on Software Architecture, avril 1995.

**[Shneeman&Lee 00]**
R. D. Shneeman, K. B. Lee, *"Distributed Measurement and Control Based on the IEEE1451 Smart Transducer Interface Standards"*. IEEE Transactions on Instrumentation and Measurement, 49(3), 2000.

**[Stafford et al. 93]**
J. A. Stafford, D. J. Richardson, A. L. Wolf, *"Aladdin: A Tool for Architecture-Level Dependence Analysis of Software"*. University of Colorado at Boulder, Technical Report CU-CS-858-98, avril 1993.

**[Staroswiecki&Bayart 94]**
M. Staroswiecki, M. Bayart, *"Actionneurs intelligents"*, Collection automatique, Éditions Hermès, 1994.

**[Staroswiecki&Bayart 96]**
M. Staroswiecki, M. Bayart, *"Models and Languages for the Interoperability of Smart Instruments"*, Automatica, Vol. 32, No 6, pp. 859-873, 1996.

**[Tailland 00]**
J. Tailland, *"Instruments intelligents : modèles et outils de conception"*, Thèse de Doctorat, Université de Savoie, juillet 2000, 184 pages.

**[Vestal 92]**
S. Vestal, *"MetaH Reference Manual"*, Honeywell Technology Center, Minneapolis MN, 1992.

**[Wile 99]**
D. Wile, *"AML: An Architecture Meta Language"*, Proceedings of the 14th International Conference on Automated Software Engineering, pp. 183-190, Cocoa Beach, Floride, octobre 1999.

**[Wile 01]**
D. Wile, *"Using Dynamic ACME"*, In Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Australie, décembre 2001.

# Annexe 1 : LES STYLES ARCHITECTURAUX

```
SInEvent is style where {
   constructors
   {
      SInEvent is constructor(InOutType:Type);
      {
         value bActiverElement is location(false);
         value activeElement is free connection(Boolean);
         value inputChannel is free connection(InOutType);
         value outputChannel is free connection(InOutType);
         recursive value actif is abstraction();
         {
           via activeElement receive bVal:Boolean;
           bActiverElement := bVal;
           actif()
         };
         recursive value protocole is abstraction();
         {
           if ('bActiverElement) do {
                 via inputChannel receive data:InOutType;
                 via outputChannel send data
           };
           protocole()
         };
         compose{
           protocole()
           and
           actif()
         }
      }
      as
      {
         input event with $InOutType type
      }
   }

   Constraints
   {
      to connections apply {
         forall (c1| c1.name=inputchannel and every sequence {true*.via c2 send any}
                                             leads to state {false})
      },
      to connections apply {
         forall (c2| c2.name=outputchannel and every sequence {true*.via c2 receive any}
                                             leads to state {false})
      },
      to connections apply {
           forall (c3| c3.name=activeElement and every sequence {true*.via c3 receive any}
                                                  leads to state {false})
      },
      to connections apply {
           exist([1]c4| c4.mane=inputchannel) and exist([1]c5| c5.mane=outputchannel)
           and exist([1]c6| c6.mane=activeElement))
      }
   }
}
```

```
SOutEvent is style where {
   Constructors
   {
         SOutEvent is constructor(InOutType:Type);
         {
               value bActiverElement is location(false);
               value activeElement is free connection(Boolean);
               value inputChannel is free connection(InOutType);
               value outputChannel is free connection(InOutType);
               recursive value actif is abstraction();
               {
                     via activeElement receive bVal:Boolean;
                     bActiverElement := bVal;
                     actif()
               };
               recursive value protocole is abstraction();
               {
                     if ('bActiverElement) do{
                           via inputChannel receive data:InOutType;
                           via outputChannel send data
                     };
                     protocole()
               };
               compose{
                     protocole()
                     and
                     actif()
               }
         }
         as
         {
               output event with $InOutType type
         }
   }
   Constraints
   {
      to connections apply {
         forall (c1| c1.name=inputchannel and every sequence {true*.via c2 send any}
                                             leads to state {false})
      },
      to connections apply {
         forall (c2| c2.name=outputchannel and every sequence {true*.via c2 receive any}
                                             leads to state {false})
      },
      to connections apply {
            forall (c3| c3.name=activeElement and every sequence {true*.via c3 receive any}
                                                   leads to state {false})
      },
      to connections apply {
            exist([1]c4| c4.mane=inputchannel) and exist([1]c5| c5.mane=outputchannel)
            and exist([1]c6| c6.mane=activeElement))
   }
}
```

```
SInPort is style where {
   constructors
   {
         SInPort is constructor(InOutType:Type);
         {
               value bActiverElement is location(false);
               value activeInPort is free connection(Boolean);
               value inputChannel is free connection(InOutType);
               value outputChannel is connection(InOutType);
               recursive value actif is abstraction();
               {
                     via activeInPort receive bVal:Boolean;
                     bActiverElement := bVal;
                     actif()
               };
               recursive value protocole is abstraction();
               {
                     if ('bActiverElement) do{
                           via inputChannel receive data:InOutType;
                           via outputChannel send data
                     };
                     protocole()
               };
               compose{
                     protocole()
                     and
                     actif()
               }
         }
   }
   Constraints
   {
      to connections apply {
            forall (c1| c1.name=inputchanel and every sequence {true*.via c2 send any}
                                                leads to state {false})
      },
      to connections apply {
            forall (c2| c2.name=outputchanel and every sequence {true*.via c2 receive any}
                                                leads to state {false})
      },
      to connections apply {
            forall (c3| c3.name=activeInPort and every sequence {true*.via c3 receive any}
                                                leads to state {false})
      },
      to connections apply {
            exist([1]c4| c4.mane=inputchanel) and exist([1]c5| c5.mane=outputchanel)
            and exist([1]c6| c6.mane=activeInPort))
   }
}
```

```
SOutPort is style where {
   constructors
   {
         SOutPort is constructor(InOutType:Type);
         {
               value bActiverElement is location(false);
               value activeOutPort is free connection(Boolean);
               value inputChannel is connection(InOutType);
               value outputChannel is free connection(InOutType);
               recursive value actif is abstraction();
               {
                     via activeOutPort receive bVal:Boolean;
                     bActiverElement := bVal;
                     actif()
               };
               recursive value protocole is abstraction();
               {
                     if ('bActiverElement) do{
                           via inputChannel receive data:InOutType;
                           via outputChannel send data
                     };
                     protocole()
               };
               compose{
                     protocole()
                     and
                     actif()
               }
         }
   }
   Constraints
   {
      to connections apply {
            forall (c1| c1.name=inputchanel and every sequence {true*.via c2 send any}
                                                leads to state {false})
      },
      to connections apply {
            forall (c2| c2.name=outputchanel and every sequence {true*.via c2 receive any}
                                                leads to state {false})
      },
      to connections apply {
            forall (c3| c3.name=activeOutPort and every sequence {true*.via c3 receive any}
                                                leads to state {false})
      },
      to connections apply {
            exist([1]c4| c4.mane=inputchanel) and exist([1]c5| c5.mane=outputchanel)
            and exist([1]c6| c6.mane=activeOutPort))
   }
}
```

```
SCompServiceInterne is style where {
   constructors
   {
         SCompServiceInterne is constructor(inType:Type, outType:Type, op_Java:String);
         {
               value bActiverElement is location(false);
               value outPort is SOutPort(outType);
               value inPort is SInPort(inType);
               value operation is SOperation(inType, outType, op_Java);
               value activeElement is free connection(Boolean);
               value outputChannel is connection(outType);
               value inputChannel is connection(inType);
               recursive value actif is abstraction();
               {
                     via activeElement receive bVal:Boolean;
                     bActiverElement := bVal;
                     via operation::activeOperation send bVal;
                     via inPort::activeInPort send bVal;
                     via outPort::activeOutPort send bVal;
                     actif()
               };
               recursive value protocole is abstraction();
               {
                     if ('bActiverElement) do{
                           via inputChannel receive var:inType;
                           via outputChannel send outType()
                     }
                     protocole()
               };
               compose
               {
                     actif()
                     and
                     inPort()
                     and
                     outPort()
                     and
                     operation()
                     and
                     protocole()
               }
               where
               {
                     inport::outputChannel unifies inputChannel
                     outport::inputChannel unifies outputChannel
               }
         }
         as
         {
               internal service with $inType input type $outType output type
                                                   operating with $operation_Java
         }
   }
}
```

```
SOperation is style where {
   constructors
   {
         SOperation is constructor(inType:Type, outType:Type, op_Java:String);
         {
               value bActiverElement is location(false);
               value outOpPort is SOutPort(outType);
               value inOpPort is SInPort(inType);
               value activeOperation is connection(Boolean);
               value outputChannel is connection(outType);
               value inputChannel is connection(inType);
               value nomOpJava is location(op_Java);
               recursive value actif is abstraction();
               {
                     via activeOperation receive bVal:Boolean;
                     bActiverElement := bVal;
                     via inOpPort::activeInPort send bVal;
                     via outOpPort::activeOutPort send bVal;
                     actif()
               };
               recursive value protocole is abstraction();
               {
                     if ('bActiverElement) do{
                           via inputChannel receive var:inType;
                           unobservable;
                           via outputChannel send
                     };
                     protocole()
               };
               compose
               {
                     actif()
                     and
                     outOpPort()
                     and
                     inOpPort()
                     and
                     protocole()
               }
               where
               {
                     inport::outputChannel unifies inputChannel;
                     outport::inputChannel unifies outputChannel
               }
         }
   }
}
```

```
SConnectorMulticast is style where {
   styles
   {
         SOutPortMulticast is style where
         {
               constructors
               {
                     SOutPortMulticast is constructor(inOutType:Type, nbchannels:Integer);
                     {
                           value bActiverElement is location(false);
                           value activeOutPortMulticast is connection(Boolean);
                           value inputChannel is  sequence using nbChannels values
                                                      connection(inOutType);
                           value outputChannel is  free sequence using nbChannels values
                                                      connection(inOutType);
                           recursive value actif is abstraction();
                           {
                                 via activeOutPortMulticast receive bVal:Boolean;
                                 bActiverElement := bVal;
                                 actif()
                           };
                           recursive value protocole is abstraction();
                           {
                                 if ('bActiverElement) do{
                                       iterate sequence (#1..nbChannels) by i:Natural do
                                       {
                                             via inputChannel::i receive data:inOutType();
                                             via outputChannel::i send data
                                       }
                                 };
                                 protocole()
                           };
                           compose{
                                 protocole()
                                 and
                                 actif()
                           }
                     }
               }
         }
   }
   constructors
   {
         SConnectorMulticast is constructor(inOutType:Type, nbChannels:Integer);
         {
               value bActiverElement is location(false);
               value outPort is SOutPortMulticast(inOutType, nbChannels);
               value inPort is SInPort(inOutType);
               value activeElement is free connection(Boolean);
               value inputChannel is connection(inOutType);
               value outputChannel is  sequence using nbChannels values
                                                   connection(inOutType);

               recursive value actif is abstraction();
               {
                     via activeElement receive bVal:Boolean;
                     bActiverElement := bVal;
                     via inPort::activeInPort send bVal;
                     via outPort::activeOutPortMulticast send bVal;
                     actif()
```

```
            };
            recursive value protocole is abstraction();
            {
                  if ('bActiverElement) do{
                        via inputChannel receive data:inOutType;
                        iterate sequence (#1..nbChannels) by i:Natural do
                        {
                              via outputChannel::i send data
                        }
                  };
                  protocole()
            };
            compose
            {
                  actif()
                  and
                  inPort()
                  and
                  outPort()
                  and
                  protocole()
            }
            where
            {
                  inport::outputChannel unifies inputChannel;
                  iterate sequence (#1..nbChannels) by j:Natural do
                  {
                        outport::inputChannel::j unifies outputChannel::j
                  }
            }
      }
      as
      {
            multicast connector with $inOutType type and  $nbChannels output channels
      }
   }
}
```

```
SConnectorRDV2 is style where {
   styles {
         SInPortConnectorRDV2 is style where {
               constructors
               {
                  SInPortConnectorRDV2 is constructor(type1:Type, type2:Type);
                  {
                              value bActiverElement is location(false);
                              value activeInPortConnectorRDV is connection(Boolean);
                              value inputChannel1 is free connection(type1);
                              value inputChannel2 is free connection(type2);
                              value outputChannel1 is connection(type1);
                              value outputChannel2 is connection(type2);
                              recursive value actif is abstraction();
                              {
                                    via activeInPortConnectorRDV receive bVal:Boolean;
                                    bActiverElement := bVal;
                                    actif()
                              };
                              recursive value protocole is abstraction();
                              {
                                    if ('bActiverElement) do{
                                          via inputChannel1 receive Data1:type1;
                                          via inputChannel2 receive Data2:type2;
                                          via outputChannel1 send Data1;
                                          via outputChannel2 send Data2
                                    };
                                    protocole()
                              };
                              compose{
                                    protocole()
                                    and
                                    actif()
                              }
                        }
                }
         }
   }
   constructors
   {
         SConnectorRDV2 is constructor(type1:Type, type2:Type);
         {
               value bActiverElement is location(false);
               value inPort is SInPortConnectorRDV2(type1, type2);
               value outPort is SOutPort(tuple[type1, type2]);
               value activeElement is connection(Boolean);
               value outputChannel is free connection(tuple[type1, type2]);
               value inputChannel1 is connection(type1);
               value inputChannel2 is connection(type2);
               recursive value actif is abstraction();
               {
                     via activeElement receive bVal:Boolean;
                     bActiverElement := bVal;
                     via inPort::activeInPortConnectorRDV send bVal;
                     via outPort::activeOutPort send bVal;
                     actif()
               }
               recursive value protocole is abstraction();
               {
                     if ('bActiverElement) do{
```

```
                        via inputChannel1 receive Data1:type1;
                        via inputChannel2 receive Data2:type2;
                        via outputChannel send tuple(Data1, Data2)
                  };
                  protocole()
            };
            compose
            {
                  actif()
                  and
                  inPort()
                  and
                  outPort()
                  and
                  protocole()
            }
            where
            {
                  inport::outputChannel1 unifies inputChannel1;
                  inport::outputChannel2 unifies inputChannel2;
                  outport::inputChannel unifies outputChannel
            }
      }
      as
      {
            rdv connector 2 with $type1, $type2 input types
      }
   }
}
```

```
SConnectorRDV3 is style where {
   styles {
         SInPortConnectorRDV3 is style where {
               constructors
               {
                  SInPortConnectorRDV3 is constructor(type1:Type, type2:Type, type3:Type);
                  {
                              value bActiverElement is location(false);
                              value activeInPortConnectorRDV is connection(Boolean);
                              value inputChannel1 is free connection(type1);
                              value inputChannel2 is free connection(type2);
                              value inputChannel3 is free connection(type3);
                              value outputChannel1 is connection(type1);
                              value outputChannel2 is connection(type2);
                              value outputChannel3 is connection(type3);
                              recursive value actif is abstraction();
                              {
                                    via activeInPortConnectorRDV receive bVal:Boolean;
                                    bActiverElement := bVal;
                                    actif()
                              };
                              recursive value protocole is abstraction();
                              {
                                    if ('bActiverElement) do{
                                          via inputChannel1 receive Data1:type1;
                                          via inputChannel2 receive Data2:type2;
                                          via inputChannel3 receive Data3:type3;
                                          via outputChannel1 send Data1;
                                          via outputChannel2 send Data2;
                                          via outputChannel3 send Data3
                                    };
                                    protocole()
                              };
                              compose{
                                    protocole()
                                    and
                                    actif()
                              }
                        }
               }
         }
   }
   constructors
   {
         SConnectorRDV3 is constructor(type1:Type, type2:Type, type3:Type);
         {
               value bActiverElement is location(false);
               value inPort is SInPortConnectorRDV3(type1, type2, type3);
               value outPort is SOutPort(tuple[type1, type2, type3]);
               value activeElement is connection(Boolean);
               value outputChannel is free connection(tuple[type1, type2, type3]);
               value inputChannel1 is connection(type1);
               value inputChannel2 is connection(type2);
               value inputChannel3 is connection(type3);
               recursive value actif is abstraction();
               {
                     via activeElement receive bVal:Boolean;
                     bActiverElement := bVal;
                     via inPort::activeInPortConnectorRDV send bVal;
                     via outPort::activeOutPort send bVal;
```

```
                actif()
            }
            recursive value protocole is abstraction();
            {
                if ('bActiverElement) do{
                    via inputChannel1 receive Data1:type1;
                    via inputChannel2 receive Data2:type2;
                    via inputChannel3 receive Data3:type3;
                    via outputChannel send tuple(Data1, Data2, Data3)
                };
                protocole()
            };
            compose
            {
                actif()
                and
                inPort()
                and
                outPort()
                and
                protocole()
            }
            where
            {
                inport::outputChannel1 unifies inputChannel1;
                inport::outputChannel2 unifies inputChannel2;
                inport::outputChannel3 unifies inputChannel3;
                outport::inputChannel unifies outputChannel
            }
        }
        as
        {
            rdv connector 3 with $type1, $type2, $type3 input types
        }
    }
}
```

```
SConnectorRDV4 is style where {
   styles {
         SInPortConnectorRDV4 is style where {
               constructors
               {
                  SInPortConnectorRDV4 is constructor(type1:Type, type2:Type, type3:Type,

   type4:Type);
                  {
                              value bActiverElement is location(false);
                              value activeInPortConnectorRDV is connection(Boolean);
                              value inputChannel1 is free connection(type1);
                              value inputChannel2 is free connection(type2);
                              value inputChannel3 is free connection(type3);
                              value inputChannel4 is free connection(type4);
                              value outputChannel1 is connection(type1);
                              value outputChannel2 is connection(type2);
                              value outputChannel3 is connection(type3);
                              value outputChannel4 is connection(type4);
                              recursive value actif is abstraction();
                              {
                                    via activeInPortConnectorRDV receive bVal:Boolean;
                                    bActiverElement := bVal;
                                    actif()
                              };
                              recursive value protocole is abstraction();
                              {
                                    if ('bActiverElement) do{
                                          via inputChannel1 receive Data1:type1;
                                          via inputChannel2 receive Data2:type2;
                                          via inputChannel3 receive Data3:type3;
                                          via inputChannel4 receive Data4:type4;
                                          via outputChannel1 send Data1;
                                          via outputChannel2 send Data2;
                                          via outputChannel3 send Data3;
                                          via outputChannel4 send Data4;
                                    };
                                    protocole()
                              };
                              compose{
                                    protocole()
                                    and
                                    actif()
                              }
                        }
                }
         }
   }
   constructors
   {
         SConnectorRDV4 is constructor(type1:Type, type2:Type, type3:Type, type4:Type);
         {
               value bActiverElement is location(false);
               value inPort is SInPortConnectorRDV3(type1, type2, type3, type4);
               value outPort is SOutPort(tuple[type1, type2, type3, type4]);
               value activeElement is connection(Boolean);
               value outputChannel is free connection(tuple[type1, type2, type3, type4]);
               value inputChannel1 is connection(type1);
               value inputChannel2 is connection(type2);
               value inputChannel3 is connection(type3);
```

```
            value inputChannel4 is connection(type4);
            recursive value actif is abstraction();
            {
                  via activeElement receive bVal:Boolean;
                  bActiverElement := bVal;
                  via inPort::activeInPortConnectorRDV send bVal;
                  via outPort::activeOutPort send bVal;
                  actif()
            }
            recursive value protocole is abstraction();
            {
                  if ('bActiverElement) do{
                        via inputChannel1 receive Data1:type1;
                        via inputChannel2 receive Data2:type2;
                        via inputChannel3 receive Data3:type3;
                        via inputChannel4 receive Data4:type4;
                        via outputChannel send tuple(Data1, Data2, Data3, Data4)
                  };
                  protocole()
            };
            compose
            {
                  actif()
                  and
                  inPort()
                  and
                  outPort()
                  and
                  protocole()
            }
            where
            {
                  inport::outputChannel1 unifies inputChannel1;
                  inport::outputChannel2 unifies inputChannel2;
                  inport::outputChannel3 unifies inputChannel3;
                  inport::outputChannel4 unifies inputChannel4;
                  outport::inputChannel unifies outputChannel
            }
      }
      as
      {
            rdv connector 4 with $type1, $type2, $type3, $type4 input types
      }
   }
}
```

```
SConnectorRDV5 is style where {
   styles {
         SInPortConnectorRDV5 is style where {
               constructors
               {
                  SInPortConnectorRDV5 is constructor(type1:Type, type2:Type, type3:Type,
                                                      type4:Type, type5:Type);
                  {
                           value bActiverElement is location(false);
                           value activeInPortConnectorRDV is connection(Boolean);
                           value inputChannel1 is free connection(type1);
                           value inputChannel2 is free connection(type2);
                           value inputChannel3 is free connection(type3);
                           value inputChannel4 is free connection(type4);
                           value inputChannel5 is free connection(type5);
                           value outputChannel1 is connection(type1);
                           value outputChannel2 is connection(type2);
                           value outputChannel3 is connection(type3);
                           value outputChannel4 is connection(type4);
                           value outputChannel5 is connection(type5);
                           recursive value actif is abstraction();
                           {
                                 via activeInPortConnectorRDV receive bVal:Boolean;
                                 bActiverElement := bVal;
                                 actif()
                           };
                           recursive value protocole is abstraction();
                           {
                                 if ('bActiverElement) do{
                                       via inputChannel1 receive Data1:type1;
                                       via inputChannel2 receive Data2:type2;
                                       via inputChannel3 receive Data3:type3;
                                       via inputChannel4 receive Data4:type4;
                                       via inputChannel5 receive Data5:type5;
                                       via outputChannel1 send Data1;
                                       via outputChannel2 send Data2;
                                       via outputChannel3 send Data3;
                                       via outputChannel4 send Data4;
                                       via outputChannel5 send Data5;
                                 };
                                 protocole()
                           };
                           compose{
                                 protocole()
                                 and
                                 actif()
                           }
                     }
               }
         }
   }
   constructors
   {
         SConnectorRDV5 is constructor(type1:Type, type2:Type, type3:Type, type4:Type,

type5:Type);
         {
               value bActiverElement is location(false);
               value inPort is SInPortConnectorRDV3(type1, type2, type3, type4, type5);
               value outPort is SOutPort(tuple[type1, type2, type3, type4, type5]);
```

```
            value activeElement is connection(Boolean);
            value outputChannel is free connection(tuple[type1, type2, type3, type4,
                                                                        type5]);
            value inputChannel1 is connection(type1);
            value inputChannel2 is connection(type2);
            value inputChannel3 is connection(type3);
            value inputChannel4 is connection(type4);
            value inputChannel5 is connection(type5);
            recursive value actif is abstraction();
            {
                  via activeElement receive bVal:Boolean;
                  bActiverElement := bVal;
                  via inPort::activeInPortConnectorRDV send bVal;
                  via outPort::activeOutPort send bVal;
                  actif()
            }
            recursive value protocole is abstraction();
            {
                  if ('bActiverElement) do{
                        via inputChannel1 receive Data1:type1;
                        via inputChannel2 receive Data2:type2;
                        via inputChannel3 receive Data3:type3;
                        via inputChannel4 receive Data4:type4;
                        via inputChannel5 receive Data5:type5;
                        via outputChannel send tuple(Data1, Data2, Data3, Data4, Data5)
                  };
                  protocole()
            };
            compose
            {
                  actif()
                  and
                  inPort()
                  and
                  outPort()
                  and
                  protocole()
            }
            where
            {
                  inport::outputChannel1 unifies inputChannel1;
                  inport::outputChannel2 unifies inputChannel2;
                  inport::outputChannel3 unifies inputChannel3;
                  inport::outputChannel4 unifies inputChannel4;
                  inport::outputChannel5 unifies inputChannel5;
                  outport::inputChannel unifies outputChannel
            }
      }
      as
      {
            rdv connector 5 with $type1, $type2, $type3, $type4, $type5 input types
      }
   }
}
```

```
SGraphServiceInterne is style where {
   constructors
   {
         SGraphServiceInterne is constructor(Num:Integer, seqPorts:sequence(meta_port),
                                             seqConstituents:sequence(meta_constituents),
                                             seqAttach:Expression[Behaviour]);
         {
               value InstrumentNumber is location(Num);
               seqPorts;
               seqConstituents;
               iterate seqPorts by i:Natural do
               {
                     new seqPorts::i::name
               };
               iterate seqConstituents by i:Natural do
               {
                     new seqConstituents::i::name
               };
               seqAttach
         }
         as
         {
               internal service graph with{
                     Num($Num)
                     Interfaces{$seqPorts}
                     Constituents{$seqConstituents}
                     Links{$seqAttach}
               }
         }
   }
   constraints
   {
         forall(x | ((x in style "SOutEvent")
         or (x in style "SInEvent")
         or (x in style "SConnectorRDV2")
         or (x in style "SConnectorRDV3")
         or (x in style "SConnectorRDV4")
         or (x in style "SConnectorRDV5")
         or (x in style "SConnectorMulticast")
         or (x in style "SCompServiceInterne"));

         ForAll((O1 in style SOutPort or O1 in style SOutPortMulticast or
               O1 in style SInEvent), (O2 in style SOutPort or
               O2 in style SOutPortMulticast or O1 in style SInEvent)).
         ForAll C1, C2 : Channel .
         ((C1 isin Channels(O1)) and (C2 isin Channels(O2))) implies not connect(C1,C2);

         ForAll (( I1 in style SInPort or I1 in style SinPortRdv2 or
               I1 in style SinPortRdv3 or I1 in style SinPortRdv4 or
               I1 in style SinPortRdv5 or I1 in style SOutEvent),
               I2 in style SInPort or I2 in style SinPortRdv2 or I2 in style SinPortRdv3
               or I2 in style SinPortRdv4 or I2 in style SinPortRdv5
               or I2 in style SOutEvent)) .
         ForAll C1, C2 : Channel .
         (C1 isin Channels(I1)) and (C2 isin Channels(I2)) implies not connect(C1,C2);


         ForAll (O in style SOutPort or O in style SOutPortMulticast or
               O in style SInEvent) .
         Exists C1, C2 : Channel .
```

```
        Exists (I in style SInPort or I in style SinPortRdv2 or I in style SinPortRdv3
            or I in style SinPortRdv4 or I in style SinPortRdv5
            or I in style SOutEvent) .
        (C1 isin Channels(O)) and (C2 isin Channels(I)) and connect(C1,C2);

        ForAll (I in style SInPort or I in style SinPortRdv2 or I in style SinPortRdv3
            or I in style SinPortRdv4 or I in style SinPortRdv5
            or I in style SOutEvent) .
        Exists C1, C2 : Channel .
        Exists (O in style SOutPort or O in style SOutPortMulticast or
            O in style SInEvent) .
        (C1 isin Channels(O)) and (C2 isin Channels(I)) and connect(C1,C2);

        ForAll (I in style SInPort or I in style SinPortRdv2 or I in style SinPortRdv3
              or I in style SinPortRdv4 or I in style SinPortRdv5
              or I in style SOutEvent) .
        Exist C:Channel .
        C isin Channels(I) .
        Some sequence{
              True*.via I receive.True*
        }leads to state{True};

        ForAll I in style SInPort or I in style SinPortRdv2 or I in style SinPortRdv3
              or I in style SinPortRdv4 or I in style SinPortRdv5 .
        Exist C1:Channel .
        C1 isin Channels(I1) .
        Every sequence{
              True*.via I1 receive.True*.via I1 receive
        }leads to state{False};

        Exists IE in style SInEvent and exists OE in style SOutEvent;
   }
}
```

# Annexe 2 : CODE SOURCE DES COMPOSANTS ET DES CONNECTEURS

# 1. Classe CInEvent.java

```
public final class CInEvent {
	protected COutPort outPort;
	protected Object variableEntree;
	protected boolean bActiverElement = false;

	//Constructor
	public CInEvent() {
		this.outPort = new COutPort(1);
	}
	//Methodes
	public void reset_element() {
		if (this.variableEntree != null) {
			this.variableEntree = null;
		}
	}
	public void active_element() {
		if (this.variableEntree != null) {
			this.bActiverElement = true;
		}
	}
	public void run_element() {
		if (this.bActiverElement == true) {
			this.outPort.iTabOutChannel[0].set_Value(this.variableEntree);
			this.variableEntree = null;
			this.bActiverElement = false;
		}
	}
}
```

# 2. Classe COutEvent.java

```
public final class COutEvent {
	protected CInPort inPort;
	protected Object variableSortie;
	protected boolean bActiverElement = false;
	//Constructor
	public COutEvent() {
		this.inPort = new CInPort(1);
	}
	//Methodes
	public void reset_element() {
		if (this.inPort.iTabInChannel[0].get_Value() != null) {
			this.inPort.iTabInChannel[0].set_Value(null);
		}
	}
	public void active_element() {
		if (this.inPort.iTabInChannel[0].get_Value() != null) {
			this.bActiverElement = true;
		}
	}
	public void run_element() {
		if (this.bActiverElement == true) {
			this.variableSortie = this.inPort.iTabInChannel[0].get_Value();
			System.out.println("Sortie sur COutEvent. Valeur de sortie : "
					+ this.variableSortie);
			this.inPort.iTabInChannel[0].set_Value(null);
			this.bActiverElement = false;
		}
	}
}
```

# 3. Classe COutPort.java

```
public final class COutPort {
	//variables
	protected CChannel iTabOutChannel[];		//tableau contenant les canaux du
							// port d'entrée
	//Constructor
	COutPort(int iNbOutChannel) {
		//instantiation du tableau des cannaux d'entree
		this.iTabOutChannel = new CChannel[iNbOutChannel];
	}
}
```

# 4. Classe CInPort.java

```
public final class CInPort {
	//variables
	protected CChannel iTabInChannel[];  //tableau contenant les canaux du port
						// de sortie
	//Constructor
	CInPort(int iNbInChannel) {
		//instantiation du tableau des cannaux de sortie
		this.iTabInChannel = new CChannel[iNbInChannel];
		//remplissage du tableau avec des objets COutChannel
		for (int i = 0; i < iTabInChannel.length; i++) {
			iTabInChannel[i] = new CChannel();
		}
	}

	//	Methode
	public void setInChannel(int numChannel, CChannel channel) {
		iTabInChannel[numChannel] = channel;
	}
}
```

# 5. Classe InterfaceOperation.java

```
public interface InterfaceOperation {
	public Object run(Object valIn);
}
```

# 6. Classe CChannel.java

```
public final class CChannel {
	//Variables
	private Object value;
	//Constructor
	public CChannel() {
	}
	//Methode
	public Object get_Value() {
		return value;
	}
	public void set_Value(Object iObjVal) {
		value = iObjVal;
	}
}
```

# 7. Classe CMulticastConnector.java

```
public final class CMulticastConnector {
	protected CInPort inPort;
	protected COutPort outPort;
	protected boolean bActiverElement = false;
	protected int intNumberOfChannels;
	//Constructor
	public CMulticastConnector(int iNbOutChannel) {
		this.inPort = new CInPort(1);
		this.outPort = new COutPort(iNbOutChannel);
		this.intNumberOfChannels = iNbOutChannel;
	}
	//Methodes
	public void reset_element() {
		if (this.inPort.iTabInChannel[0].get_Value() != null) {
			this.inPort.iTabInChannel[0].set_Value(null);
		}
	}
	public void active_element() {
		if (this.inPort.iTabInChannel[0].get_Value() != null) {
			this.bActiverElement = true;
		}
	}
	public void run_element() {
		int j = 0;
		if (this.bActiverElement == true) {
			while (j <= (intNumberOfChannels - 1)) {
				this.outPort.iTabOutChannel[j]

	.set_Value(this.inPort.iTabInChannel[0].get_Value());
				j = j + 1;
			}
			this.inPort.iTabInChannel[0].set_Value(null);
			this.bActiverElement = false;
		}
	}
}
```

# 8. Classe CConnectorRDV.java

```
public final class CConnectorRDV {
	protected CInPort inPort;
	protected COutPort outPort;
	protected boolean bActiverElement = false;
	protected int intNumberOfChannels;
	//Constructor
	public CConnectorRDV(int iNbInChannel) {
		this.inPort = new CInPort(iNbInChannel);
		this.outPort = new COutPort(1);
		this.intNumberOfChannels = iNbInChannel;
	}
	//Methodes
	public void reset_element() {
		for (int i = 0; i < intNumberOfChannels; i++) {
			if (this.inPort.iTabInChannel[i].get_Value() != null) {
				this.inPort.iTabInChannel[i].set_Value(null);
			}
		}
	}
	public void active_element() {
		int intValTemp;

		intValTemp = 0;
		for (int j = 0; j < intNumberOfChannels; j++) {
			if (this.inPort.iTabInChannel[j].get_Value() != null) {
				intValTemp = intValTemp + 1;
			}
		}
```

```
                if (intValTemp == intNumberOfChannels) {
                        this.bActiverElement = true;
                }
        }
        public void run_element() {
                int i, j = 0;
                String objValue = new String();

                if (this.bActiverElement == true) {
                        while (j <= (intNumberOfChannels - 1)) {
                                if (j == 0) {
                                        objValue = "[" +
                                         this.inPort.iTabInChannel[j].get_Value().toString();
                                } else {
                                        objValue = objValue.toString()+ "||" +
                                         this.inPort.iTabInChannel[j].get_Value().toString();
                                }
                                j = j + 1;
                        }
                        objValue = objValue.toString() + "]";
                        this.outPort.iTabOutChannel[0].set_Value(objValue);
                        i = 0;
                        while (i <= (intNumberOfChannels - 1)) {
                                this.inPort.iTabInChannel[i].set_Value(null);
                                i = i + 1;
                        }
                        this.bActiverElement = false;
                }
        }
}
```

# 9. Classe CServiceInterne.java

```
public final class CServiceInterne {
        protected CInPort inPort;
        protected COutPort outPort;
        protected boolean bActiverElement = false;
        private InterfaceOperation operation;
        protected Object objRetour;
        //Constructor
        public CServiceInterne(String iStrClassName) {
                this.inPort = new CInPort(1);
                this.outPort = new COutPort(1);
                try {
                        operation = (InterfaceOperation) (Class.forName("interpreter."
                                        + iStrClassName).newInstance());
                } catch (Exception e) {
                        System.out.println(e);
                }
        }
        //Methodes
        public void reset_element() {
                if (this.inPort.iTabInChannel[0].get_Value() != null) {
                        this.inPort.iTabInChannel[0].set_Value(null);
                }
        }
        public void active_element() {
                if (this.inPort.iTabInChannel[0].get_Value() != null) {
                        this.bActiverElement = true;
                }
        }
        public void run_element() {
                if (this.bActiverElement == true) {
                        this.objRetour = this.operation.run(this.inPort.iTabInChannel[0]
                                        .get_Value());
                        this.outPort.iTabOutChannel[0].set_Value(objRetour);
                        this.inPort.iTabInChannel[0].set_Value(null);
                        this.bActiverElement = false;
                }
        }
```

# Annexe 3 : LE GANT NUMERIQUE

# 1 Le fichier π-ADL du gant numérique

```
value SInEventReal is abstraction();
{
   value bActiverElement is location(false);
   value activeElement is free connection(Boolean);
   value inputChannel is free connection(Real);
   value outputChannel is free connection(Real);
   recursive value actif is abstraction();
   {
         via activeElement receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do {
               via inputChannel receive data:Real;
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SOutEventString is abstraction();
{
   value bActiverElement is location(false);
   value activeElement is free connection(Boolean);
   value inputChannel is free connection(String);
   value outputChannel is free connection(String);
   recursive value actif is abstraction();
   {
         via activeElement receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive data:String;
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SOutPort_tuple-Real-Real-Real is abstraction();
{
   value bActiverElement is location(false);
```

```
   value activeOutPort is free connection(Boolean);
   value inputChannel is connection(tuple[Real, Real, Real]);
   value outputChannel is free connection(tuple[Real, Real, Real]);
   recursive value actif is abstraction();
   {
         via activeOutPort receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive data:tuple[Real, Real, Real];
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SOutPort_tuple-String-String is abstraction();
{
   value bActiverElement is location(false);
   value activeOutPort is free connection(Boolean);
   value inputChannel is connection(tuple[String, String]);
   value outputChannel is free connection(tuple[String, String]);
   recursive value actif is abstraction();
   {
         via activeOutPort receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive data:tuple[String, String];
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SOutPort_tuple-String-String-String is abstraction();
{
   value bActiverElement is location(false);
   value activeOutPort is free connection(Boolean);
   value inputChannel is connection(tuple[String, String, String]);
   value outputChannel is free connection(tuple[String, String, String]);
   recursive value actif is abstraction();
   {
         via activeOutPort receive bVal:Boolean;
         bActiverElement := bVal;
```

```
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive data:tuple[String, String, String];
             via outputChannel send data
        };
        protocole()
   };
   compose{
        protocole()
        and
        actif()
   }
};

value SOutPort_tuple-Real-Real is abstraction();
{
   value bActiverElement is location(false);
   value activeOutPort is free connection(Boolean);
   value inputChannel is connection(tuple[Real, Real]);
   value outputChannel is free connection(tuple[Real, Real]);
   recursive value actif is abstraction();
   {
        via activeOutPort receive bVal:Boolean;
        bActiverElement := bVal;
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive data:tuple[Real, Real];
             via outputChannel send data
        };
        protocole()
   };
   compose{
        protocole()
        and
        actif()
   }
};

value SOutPort_String is abstraction();
{
   value bActiverElement is location(false);
   value activeOutPort is free connection(Boolean);
   value inputChannel is connection(String);
   value outputChannel is free connection(String);
   recursive value actif is abstraction();
   {
        via activeOutPort receive bVal:Boolean;
        bActiverElement := bVal;
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive data:String;
             via outputChannel send data
```

```
        };
        protocole()
   };
   compose{
        protocole()
        and
        actif()
   }
};

value SOutPort_Real is abstraction();
{
   value bActiverElement is location(false);
   value activeOutPort is free connection(Boolean);
   value inputChannel is connection(Real);
   value outputChannel is free connection(Real);
   recursive value actif is abstraction();
   {
        via activeOutPort receive bVal:Boolean;
        bActiverElement := bVal;
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive data:Real;
             via outputChannel send data
        };
        protocole()
   };
   compose{
        protocole()
        and
        actif()
   }
};

value SOutPort_tuple-String-Real is abstraction();
{
   value bActiverElement is location(false);
   value activeOutPort is free connection(Boolean);
   value inputChannel is connection(tuple[String, Real]);
   value outputChannel is free connection(tuple[String, Real]);
   recursive value actif is abstraction();
   {
        via activeOutPort receive bVal:Boolean;
        bActiverElement := bVal;
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive data:tuple[String, Real];
             via outputChannel send data
        };
        protocole()
   };
   compose{
        protocole()
        and
        actif()
```

```
   }
};

value SInPort_tuple-Real-Real-Real is abstraction();
{
   value bActiverElement is location(false);
   value activeInPort is free connection(Boolean);
   value inputChannel is free connection(tuple[Real, Real, Real]);
   value outputChannel is connection(tuple[Real, Real, Real]);
   recursive value actif is abstraction();
   {
         via activeInPort receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive data:tuple[Real, Real, Real];
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SInPort_tuple-Real-Real is abstraction();
{
   value bActiverElement is location(false);
   value activeInPort is free connection(Boolean);
   value inputChannel is free connection(tuple[Real, Real]);
   value outputChannel is connection(tuple[Real, Real]);
   recursive value actif is abstraction();
   {
         via activeInPort receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive data:tuple[Real, Real];
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SInPort_String is abstraction();
{
   value bActiverElement is location(false);
   value activeInPort is free connection(Boolean);
```

```
   value inputChannel is free connection(String);
   value outputChannel is connection(String);
   recursive value actif is abstraction();
   {
         via activeInPort receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive data:String;
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SInPort_Real is abstraction();
{
   value bActiverElement is location(false);
   value activeInPort is free connection(Boolean);
   value inputChannel is free connection(Real);
   value outputChannel is connection(Real);
   recursive value actif is abstraction();
   {
         via activeInPort receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive data:String;
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SInPort_tuple-String-String is abstraction();
{
   value bActiverElement is location(false);
   value activeInPort is free connection(Boolean);
   value inputChannel is free connection(tuple[String, String]);
   value outputChannel is connection(tuple[String, String]);
   recursive value actif is abstraction();
   {
         via activeInPort receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
```

```
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive data:tuple[String, String];
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SInPort_tuple-String-Real is abstraction();
{
   value bActiverElement is location(false);
   value activeInPort is free connection(Boolean);
   value inputChannel is free connection(tuple[String, Real]);
   value outputChannel is connection(tuple[String, Real]);
   recursive value actif is abstraction();
   {
         via activeInPort receive bVal:Boolean;
         bActiverElement := bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive data:tuple[String, Real];
               via outputChannel send data
         };
         protocole()
   };
   compose{
         protocole()
         and
         actif()
   }
};

value SConnectorRDV3_Real-Real-Real is abstraction();
{
   value SInPortConnectorRDV3_Real-Real-Real is abstraction();
   {
         value bActiverElement is location(false);
               value bActiverElement is location(false);
               value activeInPortConnectorRDV is connection(Boolean);
               value inputChannel1 is free connection(Real);
               value inputChannel2 is free connection(Real);
               value inputChannel3 is free connection(Real);
               value outputChannel1 is connection(Real);
               value outputChannel2 is connection(Real);
               value outputChannel3 is connection(Real);
               recursive value actif is abstraction();
               {
                     via activeInPortConnectorRDV receive bVal:Boolean;
                     bActiverElement := bVal;
                     actif()
```

```
            };
            recursive value protocole is abstraction();
            {
                  if ('bActiverElement) do{
                        via inputChannel1 receive Data1:Real;
                        via inputChannel2 receive Data2:Real;
                        via inputChannel3 receive Data3:Real;
                        via outputChannel1 send Data1;
                        via outputChannel2 send Data2;
                        via outputChannel3 send Data3
                  };
                  protocole()
            };
            compose{
                  protocole()
                  and
                  actif()
            }
      }
}
value bActiverElement is location(false);
value inPort is SInPortConnectorRDV3_Real-Real-Real();
value outPort is SOutPort_tuple-Real-Real-Real();
value activeElement is connection(Boolean);
value outputChannel is free connection(tuple[Real, Real, Real]);
value inputChannel1 is connection(Real);
value inputChannel2 is connection(Real);
value inputChannel3 is connection(Real);
recursive value actif is abstraction();
{
      via activeElement receive bVal:Boolean;
      bActiverElement := bVal;
      via inPort::activeInPortConnectorRDV send bVal;
      via outPort::activeOutPort send bVal;
      actif()
};

recursive value protocole is abstraction();
{
      if ('bActiverElement) do{
            via inputChannel1 receive Data1:Real;
            via inputChannel2 receive Data2:Real;
            via inputChannel3 receive Data3:Real;
            via outputChannel send tuple(Data1, Data2, Data3)
      };
      protocole()
};
compose
{
      actif()
      and
      inPort()
      and
      outPort()
      and
      protocole()
}
where
{
      inport::outputChannel1 unifies inputChannel1;
      inport::outputChannel2 unifies inputChannel2;
```

```
        inport::outputChannel3 unifies inputChannel3;
        outport::inputChannel unifies outputChannel
   }
};

value SConnectorRDV2_String-String is abstraction();
{
   value SInPortConnectorRDV2_String-String is abstraction();
   {
        value bActiverElement is location(false);
        value activeInPortConnectorRDV is connection(Boolean);
        value inputChannel1 is free connection(String);
        value inputChannel2 is free connection(String);
        value outputChannel1 is connection(String);
        value outputChannel2 is connection(String);
        recursive value actif is abstraction();
        {
              via activeInPortConnectorRDV receive bVal:Boolean;
              bActiverElement := bVal;
              actif()
        };
        recursive value protocole is abstraction();
        {
              if ('bActiverElement) do{
                    via inputChannel1 receive Data1:String;
                    via inputChannel2 receive Data2:String;
                    via outputChannel1 send Data1;
                    via outputChannel2 send Data2
              };
              protocole()
        };
        compose{
              protocole()
              and
              actif()
        }
   }
   value bActiverElement is location(false);
   value inPort is SInPortConnectorRDV2_String-String();
   value outPort is SOutPort_tuple-String-String();
   value activeElement is connection(Boolean);
   value outputChannel is free connection(tuple[String, String]);
   value inputChannel1 is connection(String);
   value inputChannel2 is connection(String);

   recursive value actif is abstraction();
   {
        via activeElement receive bVal:Boolean;
        bActiverElement := bVal;
        via inPort::activeInPortConnectorRDV send bVal;
        via outPort::activeOutPort send bVal;
        actif()
   };

   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
              via inputChannel1 receive Data1:String;
              via inputChannel2 receive Data2:String;
              via outputChannel send tuple(Data1, Data2)
        };
```

```
        protocole()
   };
   compose
   {
        actif()
        and
        inPort()
        and
        outPort()
        and
        protocole()
   }
   where
   {
        inport::outputChannel1 unifies inputChannel1;
        inport::outputChannel2 unifies inputChannel2;
        outport::inputChannel unifies outputChannel
   }
};

value SConnectorRDV3_String-String-String is abstraction();
{
   value SInPortConnectorRDV3_String-String-String is abstraction();
   {
        value bActiverElement is location(false);
             value bActiverElement is location(false);
             value activeInPortConnectorRDV is connection(Boolean);
             value inputChannel1 is free connection(String);
             value inputChannel2 is free connection(String);
             value inputChannel3 is free connection(String);
             value outputChannel1 is connection(String);
             value outputChannel2 is connection(String);
             value outputChannel3 is connection(String);
             recursive value actif is abstraction();
             {
                  via activeInPortConnectorRDV receive bVal:Boolean;
                  bActiverElement := bVal;
                  actif()
             };
             recursive value protocole is abstraction();
             {
                  if ('bActiverElement) do{
                       via inputChannel1 receive Data1:String;
                       via inputChannel2 receive Data2:String;
                       via inputChannel3 receive Data3:String;
                       via outputChannel1 send Data1;
                       via outputChannel2 send Data2;
                       via outputChannel3 send Data3
                  };
                  protocole()
             };
             compose{
                  protocole()
                  and
                  actif()
             }
        }
   }
   value bActiverElement is location(false);
   value inPort is SInPortConnectorRDV3_String-String-String();
   value outPort is SOutPort_tuple-String-String-String();
```

```
   value activeElement is connection(Boolean);
   value outputChannel is free connection(tuple[String, String, String]);
   value inputChannel1 is connection(String);
   value inputChannel2 is connection(String);
   value inputChannel3 is connection(String);

   recursive value actif is abstraction();
   {
         via activeElement receive bVal:Boolean;
         bActiverElement := bVal;
         via inPort::activeInPortConnectorRDV send bVal;
         via outPort::activeOutPort send bVal;
         actif()
   };

   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel1 receive Data1:String;
               via inputChannel2 receive Data2:String;
               via inputChannel3 receive Data3:String;
               via outputChannel send tuple(Data1, Data2, Data3)
         };
         protocole()
   };
   compose
   {
         actif()
         and
         inPort()
         and
         outPort()
         and
         protocole()
   }
   where
   {
         inport::outputChannel1 unifies inputChannel1;
         inport::outputChannel2 unifies inputChannel2;
         inport::outputChannel3 unifies inputChannel3;
         outport::inputChannel unifies outputChannel
   }

};

value SConnectorRDV2_String-Real is abstraction();
{
   value SInPortConnectorRDV2_String-Real is abstraction();
   {
         value bActiverElement is location(false);
         value activeInPortConnectorRDV is connection(Boolean);
         value inputChannel1 is free connection(String);
         value inputChannel2 is free connection(Real);
         value outputChannel1 is connection(String);
         value outputChannel2 is connection(Real);
         recursive value actif is abstraction();
         {
               via activeInPortConnectorRDV receive bVal:Boolean;
               bActiverElement := bVal;
               actif()
         };
```

```
        recursive value protocole is abstraction();
        {
              if ('bActiverElement) do{
                    via inputChannel1 receive Data1:String;
                    via inputChannel2 receive Data2:Real;
                    via outputChannel1 send Data1;
                    via outputChannel2 send Data2
              };
              protocole()
        };
        compose{
              protocole()
              and
              actif()
        }
   }
   value bActiverElement is location(false);
   value inPort is SInPortConnectorRDV2_String-Real();
   value outPort is SOutPort_tuple-String-Real();
   value activeElement is connection(Boolean);
   value outputChannel is free connection(tuple[String, Real]);
   value inputChannel1 is connection(String);
   value inputChannel2 is connection(Real);

   recursive value actif is abstraction();
   {
        via activeElement receive bVal:Boolean;
        bActiverElement := bVal;
        via inPort::activeInPortConnectorRDV send bVal;
        via outPort::activeOutPort send bVal;
        actif()
   };

   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
              via inputChannel1 receive Data1:String;
              via inputChannel2 receive Data2:Real;
              via outputChannel send tuple(Data1,Data2)
        };
        protocole()
   };
   compose
   {
        actif()
        and
        inPort()
        and
        outPort()
        and
        protocole()
   }
   where
   {
        inport::outputChannel1 unifies inputChannel1;
        inport::outputChannel2 unifies inputChannel2;
        outport::inputChannel unifies outputChannel
   }
};

value SConnectorMulticast_tuple-Real-Real is abstraction(nbChannels:Natural);
```

```
{
   value SOutPortMulticast_tuple-Real-Real is abstraction(nbChannels:Natural);
   {
         value bActiverElement is location(false);
         value activeOutPortMulticast is connection(Boolean);
         value inputChannel is  sequence using nbChannels values
                                                                  connection(tuple[Real,
Real]);
         value outputChannel is  free sequence using nbChannels values
                                                                  connection(tuple[Real,
Real]);
         recursive value actif is abstraction();
         {
               via activeOutPortMulticast receive bVal:Boolean;
               bActiverElement := bVal;
               actif()
         };
         recursive value protocole is abstraction();
         {
               if ('bActiverElement) do{
                     iterate sequence (#1..nbChannels) by i:Natural do
                     {
                           via inputChannel::i receive data:tuple[Real, Real];
                           via outputChannel::i send data
                     }
               };
               protocole()
         };
         compose{
               protocole()
               and
               actif()
         }
   }
   value bActiverElement is location(false);
   value outPort is SOutPortMulticast_tuple-Real-Real(nbChannels);
   value inPort is SInPort_tuple-Real-Real();
   value activeElement is connection(Boolean);
   value inputChannel is free connection(tuple[Real, Real]);
   value outputChannel is free sequence using nbChannels values
                                                                  connection(tuple[Real,
Real]);

   recursive value actif is abstraction();
   {
         via activeElement receive bVal:Boolean;
         bActiverElement := bVal;
         via inPort::activeInPort send bVal;
         via outPort::activeOutPortMulticast send bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive var:tuple[Real, Real];
               iterate sequence (#1..nbChannels) by i:Natural do
               {
                     via outputChannel::i send
               }
         };
         protocole()
   };
```

```
   compose
   {
         actif()
         and
         inPort()
         and
         outPort()
         and
         protocole()
   }
   where
   {
         inport::outputChannel unifies inputChannel;
         iterate sequence (#1..nbChannels) by j:Natural do
         {
               outport::inputChannel::j unifies outputChannel::j
         }
   }
};

value SConnectorMulticast_String is abstraction(nbChannels:Natural);
{
   value SOutPortMulticast_String is abstraction(nbChannels:Natural);
   {
         value bActiverElement is location(false);
         value activeOutPortMulticast is connection(Boolean);
         value inputChannel is  sequence using nbChannels values connection(String);
         value outputChannel is  free sequence using nbChannels values connection(String);
         recursive value actif is abstraction();
         {
               via activeOutPortMulticast receive bVal:Boolean;
               bActiverElement := bVal;
               actif()
         };
         recursive value protocole is abstraction();
         {
               if ('bActiverElement) do{
                     iterate sequence (#1..nbChannels) by i:Natural do
                     {
                           via inputChannel::i receive data:String;
                           via outputChannel::i send data
                     }
               };
               protocole()
         };
         compose{
               protocole()
               and
               actif()
         }
   }
   value bActiverElement is location(false);
   value outPort is SOutPortMulticast_String(nbChannels);
   value inPort is SInPort_String();
   value activeElement is connection(Boolean);
   value inputChannel is free connection(String);
   value outputChannel is free sequence using nbChannels values connection(String);
   recursive value actif is abstraction();
   {
         via activeElement receive bVal:Boolean;
         bActiverElement := bVal;
```

```
        via inPort::activeInPort send bVal;
        via outPort::activeOutPortMulticast send bVal;
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive var:String;
             iterate sequence (#1..nbChannels) by i:Natural do
             {
                  via outputChannel::i send
             }
        };
        protocole()
   };
   compose
   {
        actif()
        and
        inPort()
        and
        outPort()
        and
        protocole()
   }
   where
   {
        inport::outputChannel unifies inputChannel;
        iterate sequence (#1..nbChannels) by j:Natural do
        {
             outport::inputChannel::j unifies outputChannel::j
        }
   }
};

value SOperation_tuple-Real-Real-Real_tuple-Real-Real is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outOpPort is SOutPort_tuple-Real-Real();
   value inOpPort is SInPort_tuple-Real-Real-Real();
   value activeOperation is connection(Boolean);
   value outputChannel is connection(tuple[Real, Real]);
   value inputChannel is connection(tuple[Real-Real-Real]);
   value nomOpJava is location(op_Java);
   recursive value actif is abstraction();
   {
        via activeOperation receive bVal:Boolean;
        bActiverElement := bVal;
        via inOpPort::activeInPort send bVal;
        via outOpPort::activeOutPort send bVal;
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive var:tuple[Real,Real,Real];
             unobservable;
             via outputChannel send
        };
        protocole()
   };
```

```
   compose
   {
         actif()
         and
         outOpPort()
         and
         inOpPort()
         and
         protocole()
   }
   where
   {
         inport::outputChannel unifies inputChannel;
         outport::inputChannel unifies outputChannel
   }
}


value SCompServiceInterne_tuple-Real-Real-Real_tuple-Real-Real is

   abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outPort is SOutPort_tuple-Real-Real();
   value inPort is SInPort_tuple-Real-Real-Real();
   value operation is SOperation_tuple-Real-Real-Real_tuple-Real-Real(op_Java);
   value activeElement is connection(Boolean);
   value outputChannel is free connection(tuple[Real,Real]);
   value inputChannel is free connection(tuple[Real,Real,Real]);
   recursive value actif is abstraction();
   {
         via activeElement  receive bVal:Boolean;
         bActiverElement := bVal;
         via operation::activeOperation send bVal;
         via inPort::activeInPort send bVal;
         via outPort::activeOutPort send bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive var:tuple[Real,Real,Real];
               via outputChannel send
         };
         protocole()
   };
   compose
   {
         actif()
         and
         protocole()
   }
};

value SOperation_tuple-Real-Real_String is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outOpPort is SOutPort_String();
   value inOpPort is SInPort_tuple-Real-Real();
   value activeOperation is connection(Boolean);
   value outputChannel is connection(String);
```

```
   value inputChannel is connection(tuple[Real,Real]);
   value nomOpJava is location(op_Java);
   recursive value actif is abstraction();
   {
         via activeOperation receive bVal:Boolean;
         bActiverElement := bVal;
         via inOpPort::activeInPort send bVal;
         via outOpPort::activeOutPort send bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive var:tuple[Real,Real];
               unobservable;
               via outputChannel send
         };
         protocole()
   };
   compose
   {
         actif()
         and
         outOpPort()
         and
         inOpPort()
         and
         protocole()
   }
   where
   {
         inport::outputChannel unifies inputChannel;
         outport::inputChannel unifies outputChannel
   }
}

value SCompServiceInterne_tuple-Real-Real_String is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outPort is SOutPort_String();
   value inPort is SInPort_tuple-Real-Real();
   value operation is SOperation_tuple-Real-Real_String(op_Java);
   value activeElement is connection(Boolean);
   value activeElement is connection(Boolean);
   value outputChannel is free connection(String);
   value inputChannel is free connection(tuple[Real,Real]);

   recursive value actif is abstraction();
   {
         via activeElement  receive bVal:Boolean;
         bActiverElement := bVal;
         via operation::activeOperation send bVal;
         via inPort::activeInPort send bVal;
         via outPort::activeOutPort send bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive var:tuple[Real,Real];
               via outputChannel send
```

```
        };
        protocole()
   };
   compose
   {
        actif()
        and
        inPort()
        and
        outPort()
        and
        operation()
        and
        protocole()
   }
   where
   {
        inport::outputChannel unifies inputChannel
        outport::inputChannel unifies outputChannel
   }

};

value SOperation_String_String is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outOpPort is SOutPort_String();
   value inOpPort is SInPort_String();
   value activeOperation is connection(Boolean);
   value outputChannel is connection(String);
   value inputChannel is connection(String);
   value nomOpJava is location(op_Java);
   recursive value actif is abstraction();
   {
        via activeOperation receive bVal:Boolean;
        bActiverElement := bVal;
        via inOpPort::activeInPort send bVal;
        via outOpPort::activeOutPort send bVal;
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive var:String;
             unobservable;
             via outputChannel send
        };
        protocole()
   };
   compose
   {
        actif()
        and
        outOpPort()
        and
        inOpPort()
        and
        protocole()
   }
   where
   {
```

```
        inport::outputChannel unifies inputChannel;
        outport::inputChannel unifies outputChannel
   }
}

value SCompServiceInterne_String_String is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outPort is SOutPort_String();
   value inPort is SInPort_String();
   value operation is SOperation_String_String(op_Java);
   value activeElement is connection(Boolean);
   value outputChannel is free connection(String);
   value inputChannel is free connection(String);

   recursive value actif is abstraction();
   {
        via activeElement  receive bVal:Boolean;
        bActiverElement := bVal;
        via operation::activeOperation send bVal;
        via inPort::activeInPort send bVal;
        via outPort::activeOutPort send bVal;
        actif()
   };

   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive var:String;
             via outputChannel send
        };
        protocole()
   };
   compose
   {
        actif()
        and
        inPort()
        and
        outPort()
        and
        operation()
        and
        protocole()
   }
   where
   {
        inport::outputChannel unifies inputChannel
        outport::inputChannel unifies outputChannel
   }
};

value SOperation_Real_String is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outOpPort is SOutPort_String();
   value inOpPort is SInPort_Real();
   value activeOperation is connection(Boolean);
   value outputChannel is connection(String);
   value inputChannel is connection(Real);
   value nomOpJava is location(op_Java);
```

```
   recursive value actif is abstraction();
   {
         via activeOperation receive bVal:Boolean;
         bActiverElement := bVal;
         via inOpPort::activeInPort send bVal;
         via outOpPort::activeOutPort send bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive var:Real;
               unobservable;
               via outputChannel send
         };
         protocole()
   };
   compose
   {
         actif()
         and
         outOpPort()
         and
         inOpPort()
         and
         protocole()
   }
   where
   {
         inport::outputChannel unifies inputChannel;
         outport::inputChannel unifies outputChannel
   }
}

value SCompServiceInterne_Real_String is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outPort is SOutPort_String();
   value inPort is SInPort_Real();
   value operation is SOperation_Real_String(op_Java);
   value activeElement is connection(Boolean);
   value outputChannel is free connection(String);
   value inputChannel is free connection(Real);

   recursive value actif is abstraction();
   {
         via activeElement  receive bVal:Boolean;
         bActiverElement := bVal;
         via operation::activeOperation send bVal;
         via inPort::activeInPort send bVal;
         via outPort::activeOutPort send bVal;
         actif()
   };

   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive var:Real;
               via outputChannel send
         };
         protocole()
```

```
   };
   compose
   {
         actif()
         and
         inPort()
         and
         outPort()
         and
         operation()
         and
         protocole()
   }
   where
   {
         inport::outputChannel unifies inputChannel
         outport::inputChannel unifies outputChannel
   }
};

value SOperation_tuple-String-String_String is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outOpPort is SOutPort_String();
   value inOpPort is SInPort_tuple-String-String();
   value activeOperation is connection(Boolean);
   value outputChannel is connection(String);
   value inputChannel is connection(tuple[String, String]);
   value nomOpJava is location(op_Java);
   recursive value actif is abstraction();
   {
         via activeOperation receive bVal:Boolean;
         bActiverElement := bVal;
         via inOpPort::activeInPort send bVal;
         via outOpPort::activeOutPort send bVal;
         actif()
   };
   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive var:tuple[String, String];
               unobservable;
               via outputChannel send
         };
         protocole()
   };
   compose
   {
         actif()
         and
         outOpPort()
         and
         inOpPort()
         and
         protocole()
   }
   where
   {
         inport::outputChannel unifies inputChannel;
         outport::inputChannel unifies outputChannel
   }
```

```
}

value SCompServiceInterne_tuple-String-String_String is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outPort is SOutPort_String();
   value inPort is SInPort_tuple-String-String();
   value operation is SOperation_tuple-String-String_String(op_Java);
   value activeElement is connection(Boolean);
   value outputChannel is free connection(String);
   value inputChannel is free connection(tuple[String, String]);

   recursive value actif is abstraction();
   {
         via activeElement  receive bVal:Boolean;
         bActiverElement := bVal;
         via operation::activeOperation send bVal;
         via inPort::activeInPort send bVal;
         via outPort::activeOutPort send bVal;
         actif()
   };

   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive var:tuple[String, String];
               via outputChannel send
         };
         protocole()
   };
   compose
   {
         actif()
         and
         inPort()
         and
         outPort()
         and
         operation()
         and
         protocole()
   }
   where
   {
         inport::outputChannel unifies inputChannel
         outport::inputChannel unifies outputChannel
   }
};

value SOperation_String_Real is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outOpPort is SOutPort_Real();
   value inOpPort is SInPort_String();
   value activeOperation is connection(Boolean);
   value outputChannel is connection(Real);
   value inputChannel is connection(String);
   value nomOpJava is location(op_Java);
   recursive value actif is abstraction();
   {
         via activeOperation receive bVal:Boolean;
```

```
        bActiverElement := bVal;
        via inOpPort::activeInPort send bVal;
        via outOpPort::activeOutPort send bVal;
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive var:String;
             unobservable;
             via outputChannel send
        };
        protocole()
   };
   compose
   {
        actif()
        and
        outOpPort()
        and
        inOpPort()
        and
        protocole()
   }
   where
   {
        inport::outputChannel unifies inputChannel;
        outport::inputChannel unifies outputChannel
   }
}

value SCompServiceInterne_String_Real is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outPort is SOutPort_Real();
   value inPort is SInPort_String();
   value operation is SOperation_String_Real(op_Java);
   value activeElement is connection(Boolean);
   value outputChannel is free connection(Real);
   value inputChannel is free connection(String);

   recursive value actif is abstraction();
   {
        via activeElement  receive bVal:Boolean;
        bActiverElement := bVal;
        via operation::activeOperation send bVal;
        via inPort::activeInPort send bVal;
        via outPort::activeOutPort send bVal;
        actif()
   };

   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive var:String;
             via outputChannel send
        };
        protocole()
   };
   compose
   {
```

```
        actif()
        and
        inPort()
        and
        outPort()
        and
        operation()
        and
        protocole()
   }
   where
   {
        inport::outputChannel unifies inputChannel
        outport::inputChannel unifies outputChannel
   }
};

value SOperation_tuple-String-Real_String is abstraction(op_Java:String);
{
   value bActiverElement is location(false);
   value outOpPort is SOutPort_String();
   value inOpPort is SInPort_tuple-String-Real();
   value activeOperation is connection(Boolean);
   value outputChannel is connection(Real);
   value inputChannel is connection(tuple[String, Real]);
   value nomOpJava is location(op_Java);
   recursive value actif is abstraction();
   {
        via activeOperation receive bVal:Boolean;
        bActiverElement := bVal;
        via inOpPort::activeInPort send bVal;
        via outOpPort::activeOutPort send bVal;
        actif()
   };
   recursive value protocole is abstraction();
   {
        if ('bActiverElement) do{
             via inputChannel receive var:tuple[String, Real];
             unobservable;
             via outputChannel send
        };
        protocole()
   };
   compose
   {
        actif()
        and
        outOpPort()
        and
        inOpPort()
        and
        protocole()
   }
   where
   {
        inport::outputChannel unifies inputChannel;
        outport::inputChannel unifies outputChannel
   }
}

value SCompServiceInterne_tuple-String-Real_String is abstraction(op_Java:String);
```

```
{
   value bActiverElement is location(false);
   value outPort is SOutPort_String();
   value inPort is SInPort_tuple-String-Real();
   value operation is SOperation_String_Real(op_Java);
   value activeElement is connection(Boolean);
   value outputChannel is free connection(String);
   value inputChannel is free connection(tuple[String, Real]);

   recursive value actif is abstraction();
   {
         via activeElement  receive bVal:Boolean;
         bActiverElement := bVal;
         via operation::activeOperation send bVal;
         via inPort::activeInPort send bVal;
         via outPort::activeOutPort send bVal;
         actif()
   };

   recursive value protocole is abstraction();
   {
         if ('bActiverElement) do{
               via inputChannel receive var:tuple[String, Real];
               via outputChannel send
         };
         protocole()
   };
   compose
   {
         actif()
         and
         inPort()
         and
         outPort()
         and
         operation()
         and
         protocole()
   }
   where
   {
         inport::outputChannel unifies inputChannel
         outport::inputChannel unifies outputChannel
   }
};

value Gant is abstraction();
{
   value MPC_Pouce is SInEventReal();
   value CMC_Pouce is SInEventReal();
   value ABD_Pouce is SInEventReal();
   value IP_Pouce is SInEventReal();
   value MPC_Index is SInEventReal();
   value IP_Index is SInEventReal();
   value CMD_Static is SOutEventString();
   value CMD_Proportionnal is SOutEventString();
   value Projection_Pouce is
         SCompServiceInterne_tupleRealRealReal_tupleRealReal("Projection_pouce.java");
   value NtoS_Axe1_Pouce is
         SCompServiceInterne_tupleRealReal_String("NtoS_axe1_Pouce.java");
   value NtoS_Axe2_Pouce is
```

```
        SCompServiceInterne_tupleRealReal_String("NtoS_axe2_Pouce.java");
    value RuleBase_Pouce is SCompServiceInterne_String_String("RuleBase_Pouce.java");
    value NtoS_IP_Pouce is SCompServiceInterne_Real_String("NtoS_IP_Pouce.java");
    value NtoS_IP_Index is SCompServiceInterne_Real_String("NtoS_IP_Index.java");
    value NtoS_MPC_Index is SCompServiceInterne_Real_String("NtoS_MPC_Index.java");
    value RuleBase_Index is
        SCompServiceInterne_tupleStringString_String("RuleBase_Index.java");
    value StaticPosture is
        SCompServiceInterne_tupleStringString_String("StaticPosture.java");
    value Proportionnal_Command_Value is
        SCompServiceInterne_String_Real("Proportionnal_Command_Value.java");
    value Robot_Command_Static is
        SCompServiceInterne_String_String("Robot_Command_Static.java");
    value Robot_Command_Proportionnal is
        SCompServiceInterne_tupleStringReal_String("Robot_Command_Proportionnal.java");
    value Mult1 is SConnectorMulticast_tupleRealReal(2);
    value Mult2 is SConnectorMulticast_String(2);
    value Mult3 is SConnectorMulticast_String(2);
    value Rdv1 is SConnectorRDV3_RealRealReal();
    value Rdv2 is SConnectorRDV2_StringString();
    value Rdv3 is SConnectorRDV2_StringString();
    value Rdv4 is SConnectorRDV3_StringStringString();
    value Rdv5 is SConnectorRDV2_StringReal();

    compose{
        MPC_Pouce()
        and
        CMC_Pouce()
        and
        ABD_Pouce()
        and
        IP_Pouce()
        and
        MPC_Index()
        and
        IP_Index()
        and
        CMD_Static()
        and
        CMD_Proportionnal()
        and
        Projection_Pouce()
        and
        NtoS_Axe1_Pouce()
        and
        NtoS_Axe2_Pouce()
        and
        RuleBase_Pouce()
        and
        NtoS_IP_Pouce()
        and
        NtoS_IP_Index()
        and
        NtoS_MPC_Index()
        and
        RuleBase_Index()
        and
        StaticPosture()
        and
        Proportionnal_Command_Value()
        and
```

```
        Robot_Command_Static()
        and
        Robot_Command_Proportionnal()
        and
        Mult1()
        and
        Mult2()
        and
        Mult3()
        and
        Rdv1()
        and
        Rdv2()
        and
        Rdv3()
        and
        Rdv4()
        and
        Rdv5()
   }
   where{
        MPC_Pouce::outputChannel unifies Rdv1::inPort::inputChannel::1;
        CMC_Pouce::outputChannel unifies Rdv1::inPort::inputChannel::2;
        ABD_Pouce::outputChannel unifies Rdv1::inPort::inputChannel::3;
        IP_Pouce::outputChannel unifies NtoS_IP_Pouce::inPort::inputChannel;
        Rdv1::outPort::outputChannel unifies Projection_Pouce::inPort::inputChannel;
        Projection_Pouce::outPort::outputChannel unifies Mult1::inPort::inputChannel;
        Mult1::outPort::outputChannel::1 unifies NtoS_Axe1_Pouce::inPort::inputChannel;
        Mult1::outPort::outputChannel::2 unifies NtoS_Axe2_Pouce::inPort::inputChannel;
        NtoS_Axe1_Pouce::outPort::outputChannel unifies Rdv2::inPort::inputChannel::1;
        NtoS_Axe2_Pouce::outPort::outputChannel unifies Rdv2::inPort::inputChannel::2;
        Rdv2::outPort::outputChannel unifies RuleBase_Pouce::inPort::inputChannel;
        RuleBase_Pouce::outPort::outputChannel unifies Rdv4::inPort::inputChannel::1;
        NtoS_IP_Pouce::outPort::outputChannel unifies Rdv4::inPort::inputChannel::2;
        IP_Index::outputChannel unifies NtoS_IP_Index::inPort::inputChannel;
        MPC_Index::outputChannel unifies NtoS_MPC_Index::inPort::inputChannel;
        NtoS_IP_Index::outPort::outputChannel unifies Rdv3::inPort::inputChannel::1;
        NtoS_MPC_Index::outPort::outputChannel unifies Rdv3::inPort::inputChannel::2;
        Rdv3::outPort::outputChannel unifies Mult2::inPort::inputChannel;
        Mult2::outPort::outputChannel::1 unifies Rdv4::inPort::inputChannel::3;
        Mult2::outPort::outputChannel::2 unifies
                                    Proportionnal_Command_Value::inPort::inputChannel;
        Rdv4::outPort::outputChannel unifies StaticPosture::inPort::inputChannel;
        StaticPosture::outPort::outputChannel unifies Mult3::inPort::inputChannel;
        Mult3::outPort::outputChannel::1 unifies
                                    Robot_Command_Static::inPort::inputChannel;
        Mult3::outPort::outputChannel::2 unifies Rdv5::inPort::inputChannel::1;
        Proportionnal_Command_Value::outPort::outputChannel unifies
                                    Rdv5::inPort::inputChannel::2;
        Rdv5::outPort::outputChannel unifies
                                    Robot_Command_Proportionnal::inPort::inputChannel;
        Robot_Command_Static::outPort::outputChannel unifies CMD_Static::inputChannel;
        Robot_Command_Proportionnal::outPort::outputChannel unifies
                                    CMD_Proportionnal::inputChannel
   }
}
```

# 2 Les fichiers d'actions de raffinement

## 2.1 Les services externes

Le fichier *SE_Static.arl* :

```
archetype 'Static refines 'Gant using {
   Components excludes {('Proportionnal_Command_Value, 'Robot_Command_Proportionnal)}
   and
   connectors excludes{( 'Rdv5)}
   and
   Ports excludes {('CMD_Proportionnal)}
}
```

Le fichier SE_Proportionnal.arl :

```
archetype 'Proportionnal refines 'Gant using {
   Components excludes {('Robot_Command_Static)}
   and
   Ports excludes {('CMD_Static)}
}
```

## 2.1 Les modes

Le fichier *Mode_Static_Mode.arl* :

```
archetype 'Static refines 'Gant using {
   Components excludes {('Proportionnal_Command_Value, 'Robot_Command_Proportionnal)}
   and
   connectors excludes{( 'Rdv5)}
   and
   Ports excludes {('CMD_Proportionnal)}
}
```

Le fichier Mode_Proportionnal_Mode.arl :

```
archetype 'Proportionnal refines 'Gant using {
   Components excludes {('Robot_Command_Static)}
   and
   Ports excludes {('CMD_Static)}
}
```

On remarque que les fichiers d'actions de raffinements des modes sont identiques à ceux qui correspondent aux services externes. Ceci est tout à fait normal puisque chacun des modes ne contient qu'un seul service externe.