UNIVERSITE SAAD DAHLAB DE BLIDA

Faculté des Sciences

Département d'Informatique

MEMOIRE DE MAGISTER

Spécialité : Informatique Répartie et Mobile

SPECIFICATION HAUTEMENT FLEXIBLE D'ARCHITECTURE LOGICIELLE

Par

Dalila GUESSOUM

Devant le jury composé de :

H. ABED	Professeur, Université Saad Dahlab de Blida	Présidente
N. BOUSTIA	Maitre de conférences (B), Université Saad Dahlab de Blida	Examinatrice
W. HIDOUCI	Maitre de conférences (A), Ecole Nationale Supérieur d'Informatique	Examinateur
D. BENNOUAR	Maitre de conférences (A), Université Saad Dahlab de Blida	Promoteur

Blida, Septembre 2012

RESUME

Ce travail s'inscrit dans le cadre des architectures logicielles à base de composants. Il s'intéresse à la proposition d'une méthode de spécification d'architecture logicielle capable de prendre en charge les modèles mentaux d'architecture élaborés dans les premières phases d'un processus de développement de logiciel. Ce qui est observable aujourd'hui dans le processus de la formalisation de la spécification d'architecture logicielle, c'est le fait que les ADLs (langage de description d'architecture logicielle) ne permettent pas de supporter les spécifications d'architectures logicielles telles que faite dans les premières phases d'un cycle de développement de logiciel. Ceci est du principalement que les ADLs dépendent fortement dans la spécification de l'architectures des mécanismes logiciel de base, notamment les concepts d'interface et le concept d'appel de procédure ou d'opération. Nous proposant dans ce mémoire une méthode de spécification d'architecture logicielle dans le cadre de IASA (Integrated Approach for Softaware Architecture) indépendante de tous mécanismes logiciels, notamment ceux dédiés aux interactions, afin de supporter le modèle mental d'un architecte. Nous avons déterminé ensuite, comment une spécification architecturale sera transformée en un logiciel exécutable, en proposant une série de transformation de modèles.

Mots-clés : Architecture Logicielle, Composant, Connecteur, ADL, Transformation de modèle, Synthèse de code.

ABSTRACT

This work is part of the field of software architecture based on components. We are interested to propose a method for specifying software architecture able to support the mental models of architecture developed in the early stages of a process of software development. What is observable now in the process of formalizing the specification of software architecture is the fact that ADLs (Architecture Description Language) do not support specification of software architectures such that done in the early stages of a cycle of software development. This is due mainly that the ADL depend strongly in the specification of software architectures of basic mechanisms, including the concepts of interface and the concept of procedure call. We proposed in this thesis a method for specifying software architecture in the context of IASA (Integrated Approach for Softaware Architecture) independent of all software mechanisms, including those dedicated to interaction, in order to support the mental model of an architect. We determined then, how architectural specification will be transformed into executable software by proposing a series of model transformation.

Keywords: Software Architecture, Component, Connector, ADL, Model Transformation, code synthesis.

REMERCIEMENTS

Je tiens tout d'abord à remercier vivement mon promoteur M. Djamel BENNOUAR, d'avoir accepté de m'encadrer, ainsi que pour ses remarques constructives, ses recommandations, son grand soutien, et sa disponibilité.

Je remercie chaleureusement les membres du jury pour l'honneur qu'ils me font en acceptant d'évaluer ce travail.

Je remercie tout particulièrement Mme Bénina TOUAIBIA, Professeur à l'ENSH pour le temps qu'elle m'a consacré et pour ses conseils lumineux qui ont été d'une grande utilité.

J'adresse ma grande reconnaissance à tous les enseignants qui ont donné de leurs mieux pour ma formation durant ces sept ans d'études et qui ont su me faire apprécier l'informatique.

Enfin, je remercie de tout mon cœur tous mes proches et amis pour leur confiance, leur soutien et leur aide inconditionnels ainsi que leur présence inestimable à mes cotés durant toute la durée de ce travail.

TABLE DES MATIERES

CHAPITRE 4: VERS UNE SPECIFICATION FLEXIBLE D'ARCHITECTURE LOGCIELLE SELON L'APPROCHE IASA

LISTE DES ILLUSTRATIONS, GRAPHIQUES ETTABLEAUX

INTRODUCTION

De nos jours, les systèmes informatiques sont de plus en plus complexes et difficiles à réaliser. Pour cette raison leur production doit être de plus en plus assistée et rigoureuse. Face à cette complexité, il est nécessaire d'avoir un niveau d'abstraction élevé et de disposer de modèles qui s'approchent du modèle mental du développeur. Une réponse possible est la définition d'une architecture du système. Une architecture logicielle à base de composants décrit l'ensemble des composants qui la composent, donne la définition de leur assemblage et prend en compte les structures d'accueil nécessaires pour le déploiement et l'exploitation du système résultant. On peut dire que la définition de l'architecture d'un système correspond à l'établissement du plan de construction du logiciel. Elle permet la conception d'applications en se détachant de détails techniques propres à l'environnement et en respectant les conditions fixées par les futurs utilisateurs. En maîtrisant l'architecture conceptuelle, il est alors plus facile de gérer ses éventuelles évolutions. En effet la modification d'un plan est plus simple que la modification d'un système complet.

La spécification de l'architecture d'un logiciel représente une des premières étapes du processus de conception. Elle peut être un élément de la spécification des besoins. L'architecture d'un logiciel représente le cadre dans lequel se développera la phase de conception. Etant un aspect pouvant faire partie des besoins, l'architecture d'un logiciel peut représenter une étape dans laquelle l'interaction client– concepteur est intense. Les exigences d'un client peuvent être spécifiées au moment de la définition architecturale d'une application. La formalisation de cette étape devrait en principe prendre en charge les diverses manière de spécifications des besoins du client (spécification du problème) et les diverses manières de spécification de la solution par le concepteur.

La spécification d'architecture logicielle est restée pour longtemps informelle, spécifiée de manière informelle et intuitive par un diagramme de type « Box-and-line » sans sémantique associée. Des langages de description d'architecture (ou ADL pour Architecture Description Language) répondent en partie à cette problématique en permettant la définition d'un vocabulaire précis et commun pour les acteurs devant travailler autour la spécification liée à l'architecture (architectes, concepteurs, développeurs, intégrateurs et testeurs). Ils spécifient les composants de l'architecture de manière abstraite sans entrer dans des détails d'implantation, définissent de manière explicite les interactions entre composants d'un

système et fournissent un support de modélisation pour aider les concepteurs à structurer et composer les différents éléments. En fait, les ADLs sont un support pour la description de la structure de l'application. Ils offrent des facilités de réutilisation des composants et des moyens de description de la composition par description des dépendances entre composants et des règles de communication à respecter.

1. Problématique

Ce qui est observable aujourd'hui dans le processus de la formalisation de la spécification d'architecture logicielle, c'est le fait que les ADLs ne permettent pas de supporter les spécifications d'architectures logicielles telles que faite dans les premières phases d'un cycle de développement de logiciel. Ceci est du principalement que les ADLs dépendent fortement dans la spécification de l'architectures des mécanismes logiciel de base, notamment les concepts d'interface et le concept d'appel de procédure ou d'opération. Les spécifications dans les premières phases peuvent être indépendantes de ces deux mécanismes.

De plus, la majorité des ADLs proposent une notation spécifique, ce qui nécessite un effort de plus pour l'architecte non seulement pour comprendre le langage de spécification mais même pour pouvoir spécifier l'architecture de son système à concevoir, d'une manière qui se rapproche plus à son modèle mental. Cette liberté de spécification n'est pas bien supportée par les ADLs. En effet, ces ADLs permettent difficilement de prendre en compte le modèle mental de l'architecte élaboré dans les premières phases d'un processus de développement de logiciel. Ainsi, il n'est pas possible par exemple d'effectuer des opérations d'interconnexion entre les composants à l'improviste qui peuvent survenir surtout durant les premières phases d'un processus de développement de logiciel. Les ADLs, introduisent eux-mêmes des suppositions spécifiques sur les architectures. Par exemple, certains ADLs sont spécifiques à un domaine et à un style particulier, donc ces suppositions peuvent être incohérentes ou conflictuelles avec celles sous-jacentes au système que l'architecte veut concevoir.

Le rôle de l'architecte n'est pas seulement de créer une bonne architecture, mais également de créer une stratégie technique pour le développement du système. Cependant certains ADLs, modélisent les composants et les connecteurs à un haut niveau d'abstraction et n'assument pas ou ne prescrivent pas une relation particulière entre une description architecturale et sa mise en œuvre. Ce découplage entre la description architecturale d'un système et son implantation, peut engendrer des incohérences aux niveaux des applications.

Le raffinement qui devrait permettre le passage d'un niveau abstrait vers des niveaux plus concrets devant déboucher sur une implémentation, n'est pas suffisamment voire pas du tout supporté par les outils, obligeant ainsi les architectes à effectuer la transformation manuellement.

2. Objectifs du travail de recherche

L'objectif du travail est la recherche d'une nouvelle approche de spécification d'architecture logicielle capable de supporter la spécification directe des modèles mentaux d'architecture élaborés par les architectes dans les premières phases d'un processus de développement de logiciel. Le modèle mental dépend beaucoup des comportements, de la manière d'élaboration d'un architecte. En bref il dépend beaucoup de l'art de l'élaboration d'une solution basée sur une décomposition architecturale. Etant en réalité un art, la résolution d'un même problème par la spécification architecturale peut être très différente d'un architecte à un autre. Les différences résident principalement dans l'approche d'élaboration (du global vers le détail ou du détail vers le global), le choix des composants et les techniques utilisés pour spécifier les diverses interconnexion entre composants. Ce travail de recherche se concrétiser par la proposition d'un modèle de composant, de connecteur et d'un ADL très flexible capable de supporter le modèle mental d'un architecte.

Une fois l'ADL élaboré, il serait question de déterminer comment une spécification architecturale sera transformée en un logiciel exécutable. Un ADL décrit une spécification de haut niveau d'abstraction qui est par la suite raffinée jusqu'à ce que les composant utilisés soient des composant dit primitifs, ayant au moins une réalisation bien précise dans une technologie d'implémentation particulière. Le processus de transformation est en lui-même un axe recherche récent dans le génie logiciel. Le raffinement de l'architecture logicielle est considéré comme une étape en cours de la recherche à cause de la complexité de la conversion de l'architecture abstraite à l'architecture spécifique. Le raffinement de l'architecture logicielle est devenu donc un domaine de recherche important en génie logiciel pour la description de l'architecture logicielle

La validation de ces travaux de recherche se fera dans le contexte d'un IDE qui permettrait la spécification graphique d'architecture logicielle. En réalité ce sera l'IDE qui permettrait d'accueillir les éléments du modèle mental. Ces éléments seront par la suite

transformé en un ADL qui lui-même sera transformé en un modèle basé sur un langage de programmation.

3. Les éléments fondamentaux de notre approche

Afin d'atteindre les objectifs cités, une étude approfondie des langages de spécification d'architecture logicielle et des modèles de composant existants, doit être réalisée. Cette étude a fait ressortir plusieurs limites qui font face à une spécification flexible des architectures logicielles.

Nous constatons par conséquent que la conception architecturale n'est pas une tâche triviale. Elle s'appuie sur l'expertise des architectes et sur la façon dont ils procèdent pour la résolution d'un même problème par la spécification architecturale. Pour rendre cette tâche plus systématique, nous proposons une approche de spécification d'architecture logicielle capable de supporter la spécification directe des modèles mentaux d'architecture élaborés par les architectes dans les premières phases d'un processus de développement de logiciel.

Concernant le raffinement de l'architecture logicielle qui devrait permettre le passage d'un niveau abstrait vers des niveaux plus concrets devant déboucher sur une implémentation dans un langage de programmation. La démarche que nous préconisons est inspirée de l'approche MDA (Model Driven Architecture), essor de l'Ingénierie Dirigée par les Modèles (IDM). Nos techniques correspondent aux deux artefacts de cette approche à savoir : modélisation et transformations de modèles. Le principe clé de MDA étant la séparation des préoccupations à savoir la logique métier et la logique d'implémentation. Elle conçoit l'intégralité du cycle de vie du logiciel comme un processus de production, de raffinement itératif des modèles. Un défi, lors de l'utilisation des approches de transformation de modèles, est de définir les règles de transformation qui guident le développeur du logiciel à travers le processus de transformation. Nous proposons dans ce travail un ensemble de règles permettant de produire des composants logiciels réalisés dans des langages de programmation cibles tel que Java.

4. Organisation du document

Le document est organisé en cinq chapitres, qui sont organisés comme suit :

- Chapitre 1 : ce chapitre situe le cadre général de notre travail. Il aborde le domaine des architectures logicielles. Il expose les concepts, les définitions et la terminologie des architectures logicielles.
- Chapitre 2 : dans ce chapitre, nous passons en revue les différents langages de spécification d'architecture logicielle. A la fin de ce chapitre, nous analyserons les avantages et les inconvénients de ces différents langages en proposant une vision synthétique.
- Chapitre 3 : dans ce chapitre, nous présentons le principe et les concepts fondamentaux de cette approche MDA, essor de l'Ingénierie Dirigée par les Modèles.
- Chapitre 4 : est consacré à la présentation des différents concepts et techniques développés pour de notre approche afin de supporter la spécification directe des modèles mentaux de l'architecte et à la présentation du processus de transformation capable de produire des composants logiciels réalisés dans un langage cible.
- Chapitre 5 : est dédié à la présentation de l'environnement de développement que nous avons développé pour valider notre travail.
- Ce manuscrit se termine par une conclusion générale et de perspectives des travaux en cours et à venir.

CHAPITRE 1
INTRODUCTION A L'ARCHITECTURE LOGICIELLE

1.1 Introduction

Face à la taille et à la complexité croissantes des systèmes logiciels, les programmeurs ont naturellement été amenés à étager le développement logiciel. Ainsi, avant de programmer un système complexe, il apparaît important de bien comprendre, à un plus haut niveau d'abstraction, ce que le système doit faire. L'architecture logicielle cherche à répondre à cette attente **[1]**. Le concept d'architecture logicielle a été introduit pour la première fois à la conférence NATO en 1969 **[2]**. Depuis, l'architecture logicielle a suscité une attention croissante, à la fois de la part de la communauté scientifique et de celle des industriels du domaine du génie logiciel **[3]**. En effet, des diagrammes informels et ad hoc en «boîtes et flèches» permettant de raisonner et de générer du support de programmation à partir de descriptions architecturales, en passant par des techniques et des méthodologies de conception architecturale (e.g., Rational Unified Process **[4]**). Dans ce chapitre, nous présentons une introduction à l'architecture logicielle, les principaux concepts et terminologies, ainsi qu'un aperçu des principaux avantages de son utilisation dans le développement logiciel. Nous donnons ensuite un éventail des approches existantes pour décrire une architecture logicielle. Enfin, nous nous penchons sur les limites actuelles de ces approches pour tirer pleinement profit des décisions architecturales au niveau de l'implémentation des systèmes.

1.2 Concepts et terminologie de l'Architecture Logicielle

L'architecture logicielle connaît aujourd'hui son âge d'or **[5]**. En effet, plusieurs travaux sur les architectures logicielles soulignent l'importance d'avoir une structure globale à un niveau d'abstraction élevé d'un système avant sa construction. Bien que tous ces travaux s'accordent sur l'importance de l'architecture logicielle pour le développement des systèmes, il n'existe pas de définition universelle de l'architecture logicielle. En effet, on peut trouver dans la littérature de nombreuses définitions **[6]** qui, bien qu'elles présentent des similitudes, correspondent à autant de vues différentes du domaine. La page web dédiée à ce sujet du

Software Engineering Institute[1] **[7]** témoigne de la diversification de ces définitions. Mais d'une manière générale on peut considérer l'architecture logicielle comme une vue abstraite d'un système en terme d'éléments architecturaux **[8, 29]**. Ces éléments sont (Figure 1.1) :

- Les composants qui décrivent les fonctionnalités métier de l'application ;
- Les connecteurs qui décrivent les communications et connexions entre les composants;
- La configuration qui décrit la topologie des connexions entre composants et connecteurs.

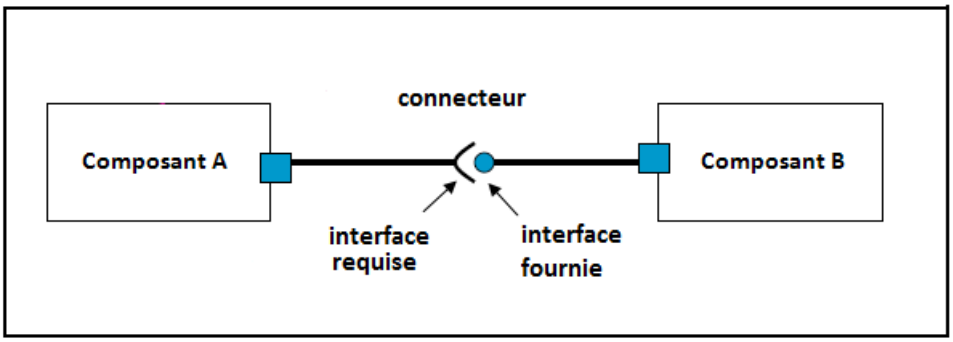


Figure 1.1: Concepts de base de l'architecture logicielle (diagramme selon UML 2.0)

Dans ce qui suit, nous présentons les différents éléments architecturaux. Pour chacun, nous proposons une synthèse des points communs entre les définitions couramment utilisées. Ensuite, nous présentons la notion de style architectural qui constitue un autre élément essentiel d'une architecture.

1.2.1 Le composant

Un composant logiciel est une entité responsable de la réalisation d'une ou plusieurs fonctionnalités bien précises dans une architecture à un certain niveau d'abstraction. Il peut être aussi petit qu'une procédure simple ou aussi grand qu'une application **[5]**. Un serveur, une base de données, une fonction mathématique sont des exemples de composants. Le composant offre une meilleure structuration de l'application et permet de construire un système par assemblage de briques élémentaires en favorisant la réutilisation de ces briques**[9]**.

[1] Carnegie Melon University, USA

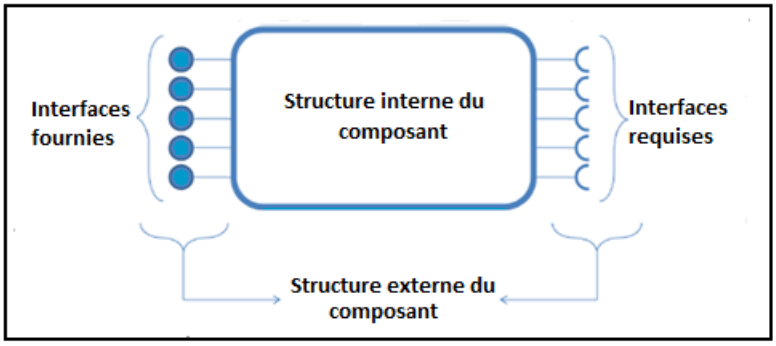


Figure 1.2: Représentation d'un composant

Il existe principalement deux parties dans un composant. Une première partie, dite extérieure, correspond à son interface. Elle comprend la description des interfaces fournies et requises par le composant (Figure 1.2). Elle définit les interactions du composant avec son environnement. La seconde partie correspond à son implantation et permet la description du fonctionnement interne du composant. Les caractéristiques globales d'un composant définies par Medvidovic et Taylor **[19]** sont les suivantes :

- l'interface,
- le type,
- la sémantique,
- les contraintes,
- les propriétés non fonctionnelles.

1.2.1.1 L'interface d'un composant

L'interface d'un composant est la description de l'ensemble des services offerts et requis par le composant sous la forme de signature de méthodes, de type d'objets envoyés et retournés, d'exceptions et de contexte d'exécution. L'interface est un moyen d'expression des liens du composant ainsi que ses contraintes avec l'extérieur. L'interface peut englober une description comportementale, souvent référencée par le terme typage comportemental. Le typage comportemental indique les règles de mise en œuvre d'une interface.

1.2.1.2 Le type d'un composant

Le type d'un composant est un concept représentant l'implantation des fonctionnalités fournies par le composant. Il s'apparente à la notion de classe que l'on trouve dans le modèle orienté objet. Ainsi, un type de composant permet la réutilisation d'instances de même fonctionnalité soit dans une même architecture, soit dans des architectures différentes. En

fournissant un moyen de décrire, de manière explicite, les propriétés communes à un ensemble d'instances d'un même composant, la notion de type de composant introduit un classificateur qui favorise la compréhension d'une architecture et sa conception.

1.2.1.3 La sémantique d'un composant

La sémantique du composant est exprimée en partie par son interface. Cependant, l'interface telle que décrite ci-dessus ne permet pas de préciser complètement le comportement du composant. La sémantique doit être enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liées à l'architecture. Ce modèle doit garantir une projection cohérente de la spécification abstraite de l'architecture vers la description de son implantation avec différents niveaux de raffinements.

1.2.1.4 Les contraintes d'un composant

Les contraintes définissent les limites d'utilisation d'un composant et ses dépendances intra composants. Une contrainte est une propriété devant être obligatoirement vérifiée sur un système ou une de ces parties. Si celle-ci est violée, le système est considéré comme un système incohérent et inacceptable. Elles permettent ainsi de décrire de manière explicite les dépendances des parties internes d'un composant comme la spécification de la synchronisation entre composants d'une même application (dépendance intra composant).

1.2.1.5 Les propriétés non fonctionnelles d'un composant

Les propriétés non fonctionnelles (propriétés liées à la sécurité, la performance, la portabilité, etc.) doivent être exprimées à part, permettant ainsi une séparation dans la spécification du composant des aspects fonctionnels (aspects métiers de l'application) et des aspects non fonctionnels ou techniques (aspects transactionnel, de cryptographie, de qualité de service).

1.2.2 Connecteur

Le connecteur correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui gouvernent ces interactions **[11]**. Par exemple, un connecteur peut décrire des interactions simples de type appel de procédure ou accès à une variable partagée, mais aussi des interactions complexes telles que des protocoles d'accès à des bases de données avec gestion

des transactions, la diffusion d'événements asynchrones ou encore l'échange de données sous forme de flux **[23]**.

Un connecteur comprend également deux parties. La première correspond à la partie visible du connecteur, c'est-à-dire son interface, qui permet la description des rôles des participants à une interaction. La seconde partie correspond à la description de son implantation. Il s'agit là de la définition du protocole permettant la mise en œuvre du protocole associé à l'interaction. Les exemples de connecteurs incluent des formes simples d'interaction, comme des pipes, des appels de procédure et l'émission d'évènements. Les connecteurs peuvent également représenter des interactions complexes, comme un protocole client/serveur ou un lien SQL entre une base de données et une application. Six caractéristiques importantes définies par Medvidovic et Taylor **[19]** sont à prendre en compte pour spécifier de manière exhaustive un connecteur. Ces caractéristiques sont les suivantes :

- l'interface,
- le type,
- la sémantique,
- les contraintes,
- l'évolution,
- les propriétés non fonctionnelles.

1.2.2.1 L'interface

L'interface d'un connecteur, appelée aussi rôles dans certains langages de description d'architecture tel que Wright **[15]**, définit les points de connexions entre connecteurs et composants. Elles servent à déclarer les participants à l'interaction décrite par le connecteur. Comme celles des composants. Néanmoins, à la différence des composants, les interfaces ne décrivent pas de services fonctionnels mais des mécanismes de connexion. Elles décrivent également le rôle de chacun des composants impliqués.

1.2.2.2 Le type

Le type d'un connecteur correspond à sa définition abstraite qui reprend les mécanismes de communication entre composants. Il permet la description d'interactions simples ou complexes de manière générique et offre ainsi des possibilités de réutilisation de protocoles. Par exemple, la spécification d'un connecteur de type RPC qui relie deux composants définit les règles du protocole RPC.

1.2.2.3 La sémantique

Comme pour les composants, la sémantique des connecteurs est définie par un modèle de haut niveau spécifiant le comportement du connecteur. A l'opposé de la sémantique du composant qui doit exprimer les fonctionnalités déduites des buts ou des besoins de l'application, la sémantique du connecteur doit spécifier le protocole d'interaction. De plus, celui-ci doit pouvoir être modélisé et raffiné lors du passage d'un niveau de description abstraite à un niveau d'implantation.

1.2.2.4 Les contraintes

Les contraintes permettent de définir les limites d'utilisation d'un connecteur, c'est-à dire les limites d'utilisation du protocole de communication associé. Une contrainte est une propriété devant être vérifiée sur un système ou sur l'une de ses parties. Si celle-ci est violée, le système est considéré comme un système incohérent et inacceptable. Par exemple, le nombre maximum de composants interconnectés à travers le connecteur peut être fixé et correspond alors à une contrainte.

1.2.2.5 L'évolution d'un connecteur

Le changement des propriétés (interface, comportement) d'un connecteur doit pouvoir évoluer sans perturber son utilisation et son intégration dans les applications existantes. Il s'agit de maximiser la réutilisation par modification ou raffinement des connecteurs existants.

1.2.2.6 Les propriétés non fonctionnelles

Les propriétés non fonctionnelles d'un connecteur concernent tout ce qui ne découle pas directement de la sémantique du connecteur. Elles spécifient des besoins qui viennent s'ajouter à ceux déjà existants et qui favorisent une implantation correcte du connecteur. Par exemple, elles peuvent concerner la performance ou la sécurité. La spécification de ces propriétés est importante puisqu'elle permet de simuler le comportement à l'exécution, l'analyse, la définition de contraintes et la sélection des connecteurs.

1.2.3 La configuration d'architecture

La configuration d'architecture d'application, encore appelée topologies, définit la façon dont les composants et les connecteurs sont reliés entre eux. Cette notion est nécessaire pour déterminer si les composants sont bien reliés, que leurs interfaces s'accordent, que les connecteurs correspondants permettent une communication correcte et que la combinaison de

leurs sémantiques aboutit au comportement désiré. Une composition de composants, appelée dans certains contextes composite, est une configuration.

Le rôle clé d'une configuration est de faciliter la communication entre les différents intervenants dans le développement d'un système. En effet, en faisant abstraction des détails des composants et des connecteurs, les configurations offrent une vision du système à un haut niveau d'abstraction qui peut être potentiellement comprise par des personnes avec différents niveaux d'expertise et de connaissance techniques **[10]**.

Dans **[19]**, Medvidovic et Taylor ont précisé les caractéristiques des configurations, on peut citer :

- un formalisme commun,
- la composition,
- le raffinement et la traçabilité,
- l'hétérogénéité,
- l'évolution de la configuration,
- les contraintes,
- les propriétés non-fonctionnelles.

1.2.3.1 Un formalisme commun

Une configuration doit permettre de fournir une syntaxe simple et une sémantique permettant de :

- faciliter la communication entre les différents partenaires d'un projet (concepteurs, développeurs, testeurs, architectes),
- rendre compréhensible la structure d'une application à partir de la configuration sans entrer dans le détail de chaque composant et de chaque connecteur,
- spécifier la dynamique d'un système, c'est-à-dire l'évolution de celui-ci au cours de son exécution.

1.2.3.2 La composition

La définition de la configuration d'une application doit permettre la modélisation et la représentation de la composition à différents niveaux de détail. La notion de configuration spécifie une application par composition hiérarchique. Ainsi un composant peut être composé de composants, chaque composant étant spécifié lui-même de la même manière, jusqu'au

composant dit primitif, c'est-à-dire non décomposable. L'intérêt de ce concept est qu'il permet la spécification de l'application par une approche descendante par raffinement, allant du niveau le plus général formé par les composants et les connecteurs principaux, définis eux mêmes par des groupes de composants et de connecteurs, jusqu'aux détails de chaque composant et de chaque connecteur primitifs.

1.2.3.3 Le raffinement et la traçabilité

La configuration est également un moyen de permettre le raffinement de l'application d'un niveau abstrait de description général vers un niveau de description de plus en plus détaillé, et, ceci, à chaque étape du processus de développement (conception, implantation, déploiement).

1.2.3.4 L'hétérogénéité

La configuration d'une architecture doit permettre le développement de grands systèmes avec des éléments préexistants de caractéristiques différentes. Une configuration doit être capable de spécifier une application indépendamment du langage de programmation, du système d'exploitation et du langage de modélisation.

1.2.3.5 L'évolution de la configuration

La configuration doit être capable d'évoluer pour prendre des nouvelles fonctionnalités impliquant une modification ou une évolution de la structure de l'application. Elle doit permettre de faire évoluer l'architecture d'une application de manière incrémentale, c'est-à dire par ajout ou retrait de composants et des connecteurs.

1.2.3.6 Les contraintes

Les contraintes liées à la configuration viennent en complément des contraintes définies pour chaque composant et chaque connecteur. Elles décrivent les dépendances entre les composants et les connecteurs et concernent des caractéristiques liées à l'assemblage de composants que l'on qualifie de contraintes inter composants. La spécification de ces contraintes permet de définir des contraintes dites globales, s'appliquant à tous les éléments de l'application.

1.2.3.7 Les propriétés non fonctionnelles

Certaines propriétés non fonctionnelles ne concernant ni les connecteurs et ni les composants doivent être exprimées au niveau de la configuration. Ces contraintes sont liées à l'environnement d'exécution.

1.2.4 Le style architectural

Dans n'importe quelle activité relative au domaine du génie logiciel, une question qui revient souvent est : comment bénéficier des expériences antérieures pour produire des systèmes plus performants? Dans le domaine des architectures logicielles, une des manières, selon **[12]**, est de classer les architectures par catégories et de définir leurs caractéristiques communes. En effet, un style architectural définit une famille d'architectures logicielles qui sont caractérisées par des propriétés structurelles et sémantiques communes.

Les styles d'architecture **[13]** sont des schémas d'architectures logicielles qui sont caractérisés par :

- Un ensemble de types de composants.
- Un ensemble de connecteurs.
- Une répartition topologique de ces composants indiquant leurs relations.
- Un ensemble de contraintes sémantiques.

Un style n'est pas une architecture mais une aide générique à l'élaboration de structures architecturales **[14]**. Parmi les principaux styles architecturaux, nous citons le style "client-serveur", le style "publier-souscrire", le style "pipe and filtre" et le style "tableau noir " (Figure 1.3).

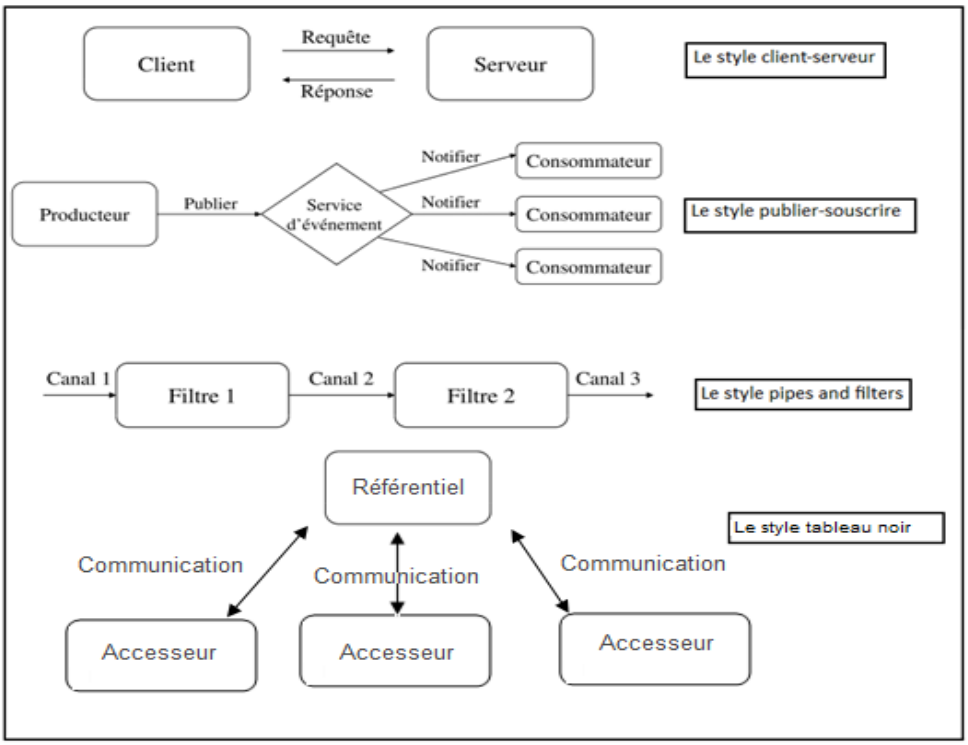


Figure 1.3: Exemples de style architectural

L'utilisation des styles architecturaux a un certain nombre d'avantages significatifs. D'abord, elle favorise la réutilisation de la conception. En effet, des applications modélisées selon un style et des propriétés architecturales bien définies peuvent être réappliquées à des nouvelles applications. Elle facilite, pour les autres, la compréhension de l'organisation de l'architecture de l'application si les structures conventionnelles sont bien appliquées et utilisées. Par exemple, concevoir une application selon le style Client-Serveur peut donner une idée claire sur le type de composants utilisés et les interactions entre les différents composants. Finalement, l'utilisation des styles architecturaux peut mener à la réutilisation significative du code : souvent les aspects invariants d'un style architectural se prétendent à des implémentations partagées.

1.3 Importance de l'architecture logicielle

La description de l'architecture logicielle s'impose de plus en plus comme une étape indispensable du développement des systèmes logiciels en permettant au concepteur de raisonner sur les propriétés fonctionnelles et non fonctionnelles du système à un haut niveau d'abstraction. Il est bien admis aujourd'hui qu'une bonne architecture peut amener à un produit qui répond aux besoins des utilisateurs et qui peut être modifié facilement et qu'une mauvaise architecture peut avoir des conséquences désastreuses sur le système **[60]**.

D'autres avantages ont été reconnus pour les architectures logicielles et qui ont été identifiés par **[17]**, repris par **[8]** :

- **La compréhension du système**: l'architecture fournie une représentation d'un système à un haut niveau d'abstraction. Cette vue synthétique du système met en valeur la plupart des décisions de conception ainsi que les conséquences de ces mauvaises décisions. En effet, l'architecture met en valeur les contraintes de haut niveau qui justement intéressent les concepteurs.

- **La réutilisation**: les descriptions architecturales favorisent la réutilisation à plusieurs niveaux. Les travaux en cours qui traitent la réutilisation se concentrent surtout sur les bibliothèques de composants. La conception architecturale supporte la réutilisation de grands composants *(large components)*, ainsi que les frameworks ou les composants peuvent être intégrés. Elle permet également, à travers les connecteurs, d'identifier les dépendances existantes entre ces parties réutilisables d'un système.

- **L'évolution**: l'architecture fournie un squelette du système. Ce squelette permet d'identifier les parties fortement utilisées ainsi que les parties potentiellement fragiles. L'architecture permet ainsi de mettre en valeur les parties nécessitant une attention particulière lors de l'évolution du système. Mais l'architecture permet également de révéler une image précise des dépendances entre les composants. Cette image est nécessaire pour connaitre les impacts de la modification d'un composant sur les autres composants du système et donc les conséquences des différentes évolutions. Elle permet aussi de modifier ces dépendances pour améliorer certains attributs du système tels que la performance ou l'interopérabilité. Enfin, l'architecture permet de corriger les erreurs à la source plutôt que là où elles apparaissent. Ceci peut être réalisé en localisant le composant fautif ou encore certaines dépendances ou contraintes non documentées ;

- **L'analyse** : la vue abstraite fournie par l'architecture permet de mesurer différents attributs tels que la consistance du système, son respect du style architectural ou encore d'autres attributs de qualité. Elle permet également de vérifier que les changements prévus dans le système sont conformes au style et aux objectifs de qualité fixés à la conception ;

- **La gestion de projets** : l'architecture permet une gestion plus précise des coûts et des risques de modifications, en particulier en soulignant les dépendances entre les

composants. Elle permet également une évaluation des qualités du système dans son ensemble, mais aussi des qualités de chaque composant. Ceci permet d'identifier les parties les plus faibles du système et ainsi d'examiner et de cibler précisément leurs faiblesses. Cette identification des composants les plus faibles permet de mettre en valeur les composants les plus problématiques et de décider de leur réingénierie. Enfin, la connaissance de la valeur de chaque composant et de leurs dépendances permet de planifier la réingénierie d'un système complexe en ordonnant les modifications selon leurs impacts sur la qualité du logiciel et le risque qu'elles soulèvent.

1.4 Description d'une architecture logicielle

De nombreuses approches ont été proposées pour décrire et analyser, plus ou moins précisément, une architecture logicielle.

1.4.1 Diagrammes en «boîtes et flèches»

Les diagrammes en «boîtes et flèches»[2] sont le moyen le plus simple pour décrire la structure d'un système. En effet, ils permettent de facilement identifier les éléments importants du système (i.e., les boîtes) ainsi que leurs relations (i.e., les flèches). Toutefois, ces diagrammes nécessitent d'être surchargés avec des informations informelles expliquant les calculs effectués par les boîtes et la nature des interactions représentées par les flèches. Par exemple, une flèche peut aussi bien représenter un lien de dépendance entre deux composants qu'un appel de procédure. Le manque de sémantique explicite limite sévèrement l'utilité de ces diagrammes dans le processus de développement logiciel.

1.4.2 Langages de description d'architecture

Les langages de description d'architecture (Architecture Description Languages ou ADL) fournissent des notations formelles pour décomposer un système en composants et connecteurs (i.e., des mécanismes d'interaction), et pour spécifier comment ces éléments sont combinés afin de former une configuration **[19]**. Les ADLs couvrent de nombreux domaines d'application et la plupart d'entre eux se focalisent sur certains aspects spécifiques de l'architecture logicielle pour des domaines particuliers.

[2] On retrouve dans la littérature le terme anglophone "Box and line".

1.4.3 Techniques formelles

Le troisième axe de recherche dans le domaine des architectures logicielles consiste à tirer profit des avantages des techniques formelles existantes en les appliquant à la conception architecturale. Les techniques formelles peuvent être classés principalement en trois catégories: les techniques basées sur les graphes **[18, 19, 20]**, les techniques basées sur les algèbres de processus **[21,22]** et les techniques basées sur la logique **[23,24]**. Les approches formelles élaborées dans le domaine de l'architecture logicielle ne possèdent pas les mêmes visions. Néanmoins, ils cherchent tous à pouvoir offrir aux développeurs la possibilité de raisonner et d'analyser une architecture. Cependant, les grammaires de graphe, comme toute technique formelle, sont basées sur des notations mathématiques qui exigent des connaissances et des expertises assez importantes.

1.5 Limitations actuelles

Bien que l'architecture logicielle est une approche très prometteuse dans le processus de conception des systèmes informatique, de nombreux problèmes ouverts demeurent **[5, 25].** Nous citerons ici deux problèmes majeurs. Le premier est le fossé qui sépare la définition d'une architecture faite à un haut niveau d'abstraction et le niveau implémentation. Ce fossé rend difficile non seulement la communication entre l'architecte et le développeur d'un système, mais aussi la garantie que l'implémentation du système respecte bien toutes les décisions architecturales prises, à la fois fonctionnelles et non fonctionnelles. Ce fossé pourrait avoir un impact sur la possibilité d'effectuer des opérations d'ingénierie inverse (reverse engineering) pour retrouver l'architecture de départ à partir du code. L'architecte et le développeur d'un système sont généralement deux personnes différentes. Pourtant, les approches existantes ne permettent pas de les faire communiquer de manière efficace, chacun dans leurs termes **[26]**. En effet, les langages de description d'architecture sont faiblement couplés avec les langages de programmation utilisés pour implémenter les systèmes logiciels. Ce faible couplage permet de favoriser la portabilité ainsi que de différer certains choix d'implémentation bas niveau. Toutefois, il ne permet pas de s'assurer que les développeurs d'un système ont bien compris tout ce que l'architecte avait à l'esprit, ni même de les aider ou de les guider dans ce but. À la place, les ADLs attendent des développeurs qu'ils suivent les bonnes pratiques de programmation (e.g, évité d'utiliser des données partagées) et le plan de construction donné par l'architecture. Les développeurs sont alors amenés soit à deviner les intentions de l'architecte, essentiellement en refaisant le travail de conception, soit à consulter

fréquemment l'architecte, faisant de ce dernier le goulot d'étranglement dans le processus de développement logiciel. Cette situation peut finalement conduire les développeurs à introduire involontairement des incohérences dans l'implémentation du système, rendant de surcroît l'architecture inutile **[27, 28]**. Le deuxième problème concerne la capacité des techniques actuelles de spécification à supporter les diverses approches de définition d'une architecture, surtout que l'architecture est en réalité un art qui dépend fortement du comportement de l'architecte, notamment de ce qu'on appelle les modèles mentaux de l'architecte. Actuellement la spécification d'une architecture est fortement impactée par les mécanismes logiciels de base, notamment l'appel de procédure synchrone ou asynchrone, à travers lesquels les connexions ne peuvent être établie qu'entre interface.

1.6 Conclusion

Nous avons présenté dans ce chapitre le contexte global dans lequel s'intègre ce travail de recherche: le domaine des architectures logicielles. Pour cerner la notion d'architecture logicielle nous avons présenté dans la section 1.2 les principaux concepts et terminologies de l'architecture logicielle. L'étude de cette terminologie nous a permis de mettre en lumière les différentes définitions et notions que nous jugeons nécessaires pour aborder notre problématique. Nous avons abordé dans la section 1.3, l'importance de l'architecture logicielle qui s'impose de plus en plus comme une étape indispensable du développement des systèmes logiciels. Pour consolider le développement de telles architectures, il est nécessaire de disposer de notations formelles et d'outils d'analyse de spécifications architecturales, dans la section 1.4 nous avons présenté quelques approches existantes pour la description de l'architecture logicielle. Les ADLs proposent donc une bonne réponse pour cette tache. Une étude assez approfondie sur les ADLs fait l'objet du chapitre suivant.

CHAPITRE 2
SPECIFICATION D'ARCHITECTURE LOGICIELLE : UN ETAT DE L'ART

2.1 Introduction

Dans ce chapitre, nous nous passerons en revue les travaux les plus représentatifs des langages de spécification d'architecture logicielle et des modèles de composant. Nous commençons par une introduction rappelant les principaux défis auxquelles fait face l'architecture logicielle. Nous présentons par la suite une évaluation de ces langages en se basant sur un ensemble de critères que nous jugeons pertinents pour notre problématique. Nous concluons ce chapitre en proposant une vision synthétique des différents langages.

2.2 Critères d'évaluation

Si la description de l'architecture logicielle prend maintenant une place prépondérante dans la phase de conception d'un logiciel, cette tâche reste pour le moment une opération complexe pour l'architecte. Afin de mener une étude bibliographique dans ce contexte, nous avons fixé un ensemble de critères d'évaluation, ces critères concernent les caractéristiques essentielles que l'on souhaite trouver dans un langage de spécification d'architecture logicielle. Dans ce qui suit nous allons lister ces critères d'évaluation.

2.2.1 L'abstraction

L'ingénierie dirigée par les modèles montre le besoin de définir les fonctionnalités des entités de son système d'information indépendamment de toute plate-forme d'exécution pour garantir la pérennité et l'interopérabilité des entités du système d'information. Les approches par composants offrent de leur côté une structure pour la définition de l'architecture logicielle séparant clairement les interactions entre les entités logicielles de la mise en œuvre de ces entités. Ainsi, la notion d'architecture logicielle véhicule principalement une distinction nette entre la structure conceptuelle d'une application (modèle abstrait) et ses possibles réalisations techniques (implantations) sur des cibles particulières (langage et système d'exploitation).

Le premier critère d'évaluation étudie la présence d'une description claire de l'architecture, le niveau d'abstraction du formalisme utilisé pour effectuer cette description et son indépendance vis-à-vis d'une plate-forme d'exécution. L'idéal consiste à proposer à

l'architecte un modèle de haut niveau de l'architecture qui se concentre sur les éléments caractéristiques d'une description d'architecture. Cette abstraction doit lui permettre de travailler avec un formalisme simple, facilement compréhensible par l'ensemble des acteurs d'un projet informatique.

2.2.2 Le typage comportemental des interfaces

L'architecture logicielle décrit, outre la structure du système en un assemblage de composants, l'interface de chaque composant. Or, les composants cherchent à être, en plus d'une unité de structuration, une unité de réutilisation. Dès lors, il est indispensable de disposer au niveau de l'interface d'un composant de suffisamment d'informations sur celui-ci pour détecter certaines incompatibilités entre composants liés, ceci dès la phase de description de l'architecture. Dès lors, les différents travaux sur la conception par contrats démontrent qu'une interface de composant ne peut se limiter à la description syntaxique des services offerts et requis, mais doit contenir des informations précises sur la façon dont le composant interagit avec son environnement. La description d'architecture logicielle doit permettre de valider l'assemblage de composant à la vue de certaines propriétés.

Le deuxième critère d'évaluation se concentre sur le pouvoir d'expression associé à la spécification des interfaces d'un composant dans les différentes approches de l'architecture logicielle. Ce pouvoir d'expression est très important, car il conditionne la vérification de la cohérence d'un assemblage de composants par rapport à certaines propriétés structurelles ou comportementales. La richesse de description des interfaces d'un composant améliore la réutilisabilité de ce composant en garantissant sa bonne intégration dans un contexte différent de celui pour lequel il a été conçu.

2.2.3 La conception par aspect

Les notions de composant, connecteur et configuration offrent un premier niveau de modularité dans la description d'une application. Si ces notions permettent une bonne séparation entre la mise en œuvre des entités de base du système et leurs interactions, elles proposent une unique dimension de décomposition, à savoir le composant. Or différents travaux ont montré la difficulté de structurer une application à l'aide d'une unique dimension. Ce problème a été identifié comme la tyrannie de la décomposition dominante **[31]**.

Un des problèmes résultant de cette unique dimension de modularité consiste à introduire par exemple de nombreuses dépendances entre les composants et les composants techniques rendant la spécification des composants métiers très difficile à établir. En effet, le travail de conception d'une description d'architecture logicielle demande la spécification des interfaces des composants de l'architecture et la spécification de l'assemblage de ces composants. Même si cette description propose un formalisme abstrait permettant de simplifier la conception d'une telle description, le nombre de préoccupations à prendre en compte dès la spécification des interfaces des composants rend ce travail très complexe et propice aux erreurs. La conception d'un ADL moderne doit s'appuyer sur différentes dimensions de structuration afin de simplifier la définition des composants métiers et doit permettre la construction du système par ajout successif de nouvelles préoccupations.

2.2.4 L'anticipation du changement

L'anticipation du changement consiste dans le domaine de l'architecture logicielle à construire un modèle conscient du besoin permanent de changement du système, capable de spécifier et de « cadrer » ces changements, c'est-à-dire de préciser l'évolution de l'architecture. Nous considérons deux principaux types de changement associés à une description d'architecture logicielle. Le premier concerne la dynamique de l'architecture appelée aussi évolution interne. Le deuxième concerne l'adaptation de l'architecture aux nouveaux besoins de l'utilisateur appelée aussi évolution externe.

La description de l'architecture logicielle doit donc prendre en compte ces deux types d'évolution au niveau du modèle pour permettre à l'architecte d'intégrer dans sa spécification ces possibilités d'évolution.

2.2.5 La composition hiérarchique

Les architectures sont nécessaires pour décrire des systèmes logiciels à différents niveaux de détails, où les comportements complexes sont soit explicitement représentés soit encapsulés dans des composants et des connecteurs. La composition est ainsi la capacité à pouvoir construire des éléments architecturaux à partir d'éléments déjà existants, en les encapsulant dans une plus grande structure. On parle alors de composants-composites ou connecteurs-composites. Un ADL doit pouvoir prendre en compte le fait qu'une architecture entière devienne un composant simple dans une plus grande architecture. Par conséquent, la prise en compte de la composition ou de la composition hiérarchique est cruciale.

2.2.6 La séparation des propriétés non fonctionnelles des propriétés fonctionnelles

Certaines propriétés non fonctionnelles ne concernant ni les connecteurs et ni les composants doivent être exprimées au niveau de la configuration **[50]**. Ces contraintes sont liées à l'environnement d'exécution, regroupant les aspects non fonctionnels liés à la sécurité, la performance, la portabilité, etc. Un ADL doit donc être capable de spécifier ces contraintes au niveau de la configuration. Une architecture logicielle de qualité devrait distinguer d'une façon explicite les propriétés fonctionnelles des propriétés non fonctionnelles. Le principe de cette séparation des préoccupations (*separation of concerns*) préconise le découpage d'une application en entités de taille plus réduite et aussi indépendantes que possible les unes des autres afin de faciliter la maintenance, la compréhension et la réutilisation.

2.2.7 Le Concept de connecteur (First class ou Non)

Il est largement accepté en architecture logicielle qu'un connecteur doit être une entité à part entière, indépendante de toute logique de composant. Pendant longtemps, les connecteurs ont été considérés comme implicites et leurs sémantiques étaient souvent incorporées dans les composants. Dans ce cas, le connecteur n'existe pas en tant qu'entité manipulable. Il a ni structure (pas de concept de rôles) ni comportement et de ce fait, il n'est ni réutilisable ni échangeable. Pour que des connecteurs soient réutilisables et extensibles, il est nécessaire qu'ils soient considérés comme des entités de première classe, i.e. au même titre que les composants.

2.2.8 Le support de styles architecturaux

Le style d'architecture est une notion très répandue dans l'architecture logicielle. L'utilisation des styles architecturaux favorise la réutilisation de la conception, facilite la compréhension de l'organisation de l'architecture de l'application et peut mener à la réutilisation significative du code. Le huitième critère d'évaluation concerne la capacité des langages de spécification d'architecture logicielle à supporter la définition et l'utilisation des styles architecturaux.

2.2.9 Les outils de spécification d'architecture

Les outils de spécification d'architecture logicielle sont aussi importants que les notations. Ces outils devraient soutenir plusieurs fonctionnalités, comme l'édition, la visualisation, la création d'extensions, l'interopérabilité, l'analyse, etc. Le neuvième

d'évaluation concerne la capacité des langages de spécification d'architecture logicielle à fournir des outils qui soutiennent activement la spécification des architectures.

2.2.10 Le raffinement (processus de transformation)

Enfin, le dernier critère concerne le raffinement de l'architecture logicielle. Le raffinement architectural est le passage d'une architecture abstraite vers une architecture plus concrète. Cette dernière contient plus d'informations qui sont consistantes avec les informations de l'architecture abstraite **[61]**. L'argument le plus souvent évoqué pour créer et utiliser les ADLs est qu'ils sont nécessaires pour établir le pont entre les diagrammes informels de haut niveau de type "boîtes-et-lignes" et les langages de programmation qui sont considérés comme des langages de bas niveau. Comme les modèles architecturaux peuvent être définis à différents niveaux d'abstractions ou de raffinement, les ADLs fournissent aux développeurs et aux concepteurs des outils expressifs et sémantiquement riches pour les spécifier. Les ADLs doivent également permettre une traçabilité des changements à travers ces niveaux de raffinement. Notons que le raffinement et la traçabilité sont les mécanismes les moins pris en compte par les ADLs actuels **[60].**

2.3 Les langages de description d'architectures

Décrire précisément une architecture logicielle est un problème qui a été souvent étudié dans la communauté de recherche en génie logiciel avec une effervescence importante dans les années 90 autour des langages de description d'architecture. L'objectif principal est la formalisation de l'approche Box and line qui représente la première activité dans le processus d'élaboration d'une architecture et qui correspond fortement aux premières idées de conceptions que l'on qualifie souvent par le terme modèle mental. Ce que nous devons retenir ici que la spécification d'une architecture est par essence graphique. C'est une topologie qui sera renforcée par du texte décrivant entre autres des attributs, des règles, des contraintes, des interactions etc. Ainsi toute description d'architecture doit contenir obligatoirement, de manière implicite ou explicite la description de la topologie.

En général les ADLs fournissent un modèle explicite de composants, de connecteurs et de leur configuration **[13]**, utilisé pour décrire au moins la structure d'un système comme un assemblage d'éléments logiciels. Ils ont en commun la capacité à décrire l'architecture des systèmes mais diffèrent largement quant à leurs motivations et leurs formalismes d'écriture.

Ils sont en général graphiques et textuels et possèdent des outils associés. Face une telle variété au manque de consensus Medvidovic et Taylor ont fait une classification **[36]**.

Dans ce sens, pour atteindre correctement notre objectif qui est la proposition d'une approche de spécification d'architecture logicielle capable de supporter aisément la spécification directe des modèles mentaux de l'architecte il est nécessaire de réaliser une étude approfondie des différents ADLs, L'évaluation des ADLs se focalise sur les critères d'évaluation que nous avons fixé précédemment.

2.3.1 Wright

2.3.1 1 Présentation

Wright **[32] [49]** est un langage d'architecture logicielle qui se focalise sur la spécification de l'architecture et de ses éléments en se basant sur un le langage formel CSP **[33]**. Comme les autres ADLs, Wright reprend les trois concepts de l'architecture logicielle à savoir le composant, le connecteur et la configuration.

Le composant

Un composant en Wright est une unité de traitement abstraite localisée et indépendante. La sémantique associée au composant est le processus. La description d'un composant contient deux parties importantes : l'interface et la partie calcul (computation). L'interface consiste à décrire les ports, c'est-à-dire les interactions auxquelles le composant peut participer. Par exemple, un composant représentant une station service peut avoir deux ports, un pour les requêtes du client et un autre pour la banque. À chaque port est associée une description formelle par le langage CSP (Communicating Sequential Processes) **[33]** spécifiant son comportement par rapport à l'environnement. La partie calcul, quant à elle, consiste à décrire le comportement du composant en indiquant comment celui-ci utilise les ports. Ainsi, les ports, qui sont décrits indépendamment dans l'interface, sont utilisés pour décrire le comportement du composant dans le calcul.

Le connecteur

Un connecteur représente une interaction entre une collection de composants. Il spécifie le patron d'une interaction de manière explicite et abstraite. Ce patron peut être réutilisé dans différentes architectures. Par exemple, un protocole de validation à deux phases (two-phase-commit) peut être un connecteur. Le connecteur contient deux parties importantes

qui sont un ensemble de rôles et la glue. Chaque rôle indique comment se comporte un composant qui participe à l'interaction. La sémantique associée au connecteur est donc aussi le processus. Le comportement du rôle est décrit par une spécification CSP. La glue décrit comment les participants (c'est-à-dire les rôles) interagissent entre eux pour former une interaction.

La configuration

La configuration permet de décrire l'architecture d'un système en regroupant des instances de composant et des instances de connecteur. La description d'une configuration est composée de trois parties qui sont la déclaration des composants et des connecteurs utilisés dans l'architecture, la déclaration des instances de composant et de connecteurs, les descriptions des liens entre les instances de composant par les connecteurs. Wright supporte la composition hiérarchique. Ainsi, un composant peut être composé d'un ensemble de composants. Il en va de même pour un connecteur.

Description du comportement

La mise en œuvre du composant et du connecteur Wright est décrite à l'aide du langage CSP. Ce langage a été proposé en 1978 par Hoare dans un souci de formalisation théorique. Il est basé sur une logique de composition des actions (Logique de Hoare) **[34]**.

Ainsi, à chaque comportement, est associé un *process* CSP. Le *process* est un patron de comportement formé d'événements observables et déclenchés par ce *process* (*events*). Chaque partie d'un élément de l'architecture sous Wright (glue du connecteur, rôles du connecteur, calcul ou ports d'un composant, configuration de l'architecture du système, interface de type) a une spécification décrivant le comportement de celle-ci.

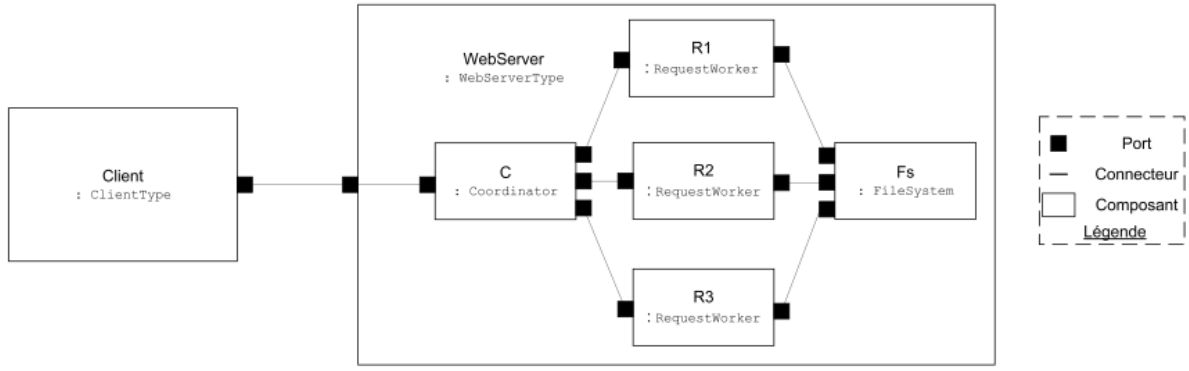
La notion de style

Wright permet de définir des types de connecteurs et de composants pour une famille d'architectures à la condition que ceux-ci respectent les propriétés de cette famille (le même style architecturale). Les propriétés et les contraintes communes à une architecture peuvent être définies selon trois caractéristiques qui sont les types d'interfaces, les paramètres et les contraintes. Les types d'interface permettent de typer le rôle d'un connecteur ou le port d'un composant pour un système donné. Les paramètres comprennent les informations de style pour définir des composants ou des connecteurs avec des parties de leurs descriptions qui

peuvent être en paramètre comme par exemple la partie calcul. Finalement, les contraintes sont des prédicats logiques de premier ordre qui doivent être satisfaits pour tous les éléments appartenant au style. Chaque partie d'un élément de l'architecture sous Wright a une spécification décrivant le comportement de celle-ci.

Exemple

Pour illustrer les principaux mécanismes de Wright, nous prenons l'exemple présenté dans **[35]** d'un serveur Web possédant trois composants : un composant prenant en charge la lecture/écriture sur un système de fichier, différents composants prenant en charge le traitement des requêtes et un composant servant de distributeur des différentes requêtes (voir Figure 2.1).




```
Configuration System
     Connector ...
     Component ClientType
     Component WebServerType
             Port ServiceHttp
             Computation
               Configuration WebServer
                     Component Coordinator
                             Port ServiceHttp (lire les données jusqu'à la
fin des données)
                             Port Dispatch1 (sortir les données de manière
permanente)
                           ...
                             Computation(Attendre les requêtes sur le port
Service http et les envoyé à un des trois ports Dispatch)
                    Component FileSystem
                    Component RequestWorker
                    Connector CSConn
                      RoleClient
                      RoleServer
                   ...
                  Instances
                    C: Coordinator
                    Fs: FileSystem
                    R1: RequestWorker
                    R2: RequestWorker
                    R3: RequestWorker
```

```
                    S1: CSConn
                    S2: CSConn
                    S3: CSConn
                    S4: CSConn
              Attachments
                   C.Dispatch1as S1.Client
                   R1.Serviceas S1.Server
                    ...
  End WebServer
  Bindings
     C.ServiceHttp = ServiceHttp
  End Bindings
  Instances
  ...
  Attachments
  ...
End System
```

Figure 2.1: Architecture d'un serveur Web décrit à l'aide de Wright [35]

Les liens entre les composants et les connecteurs sont définis dans la clause *Attachement*. Le mot-clé *Bindings* permet de spécifier les liens entre les ports du composite et ceux des composants faisant partie du composite. Dans cet exemple, le port *ServiceHttp* du composant *Coordinator* est lié au port *ServiceHttp* du composant *WebServerType.*

2.3.1.2 Evaluation

Wright est un langage de description orienté vers la vérification des protocoles entre les composants, plutôt que sur la correction fonctionnelle de l'architecture globale. Il décrit la structure et le comportement. Le principal avantage de Wright est de fournir un langage formel (CSP) pour la spécification des composants et des connecteurs. Ainsi, la description d'une architecture peut être analysée. En outre, les différents acteurs (concepteur, architecte, développeur) d'un projet peuvent communiquer sans ambiguïté leurs points de vue sur l'architecture d'une application. Un autre point essentiel de cet ADL est qu'il sépare la notion de composant et de connecteur en proposant un modèle de type de composant et de type de connecteur. Ainsi, un composant peut être spécifié de manière indépendante des connecteurs, ce qui le rend nécessairement plus indépendant par rapport à son contexte d'exécution.

Cependant, la description de l'architecture est assez pauvre et éloignée des habitudes des développeurs. Ainsi, un des inconvénients majeurs de Wright est qu'il est difficile à assimiler. En effet, l'outil de spécification (CSP couplé à Wright) n'est pas facile à comprendre pour un développeur débutant. Il peut donc s'avérer inefficace lorsque les délais d'un projet sont courts ou lorsque les compétences font défaut. Le langage Wright ne permet pas de spécifier les contraintes non fonctionnelles séparément de la spécification fonctionnelle de l'architecture. Ainsi, les contraintes fonctionnelles et non fonctionnelles sont exprimées de

la même manière et sans distinction. De plus, Wright ne possède pas d'environnement d'utilisation ou d'exécution. Il ne possède pas par exemple de générateur de code. En effet Ce langage n'est pas dédié à la production d'une image exécutable de l'application, il porte effectivement son accent sur la vérification formelle du système. En outre, Wright n'apporte pas de solution pour la hiérarchisation et la structuration de l'application. Les composants composites qui existent dans la plupart des langages n'ont pas d'équivalent ici. Malheureusement, toutes les architectures devant passer à l'échelle ont besoin d'encapsulation de sous-systèmes dans des composants. Cette limitation réduit donc grandement la classe d'applications utilisables.

2.3.2 Rapide

2.3.2.1 Présentation

Rapide **[37],** est un langage de description d'architecture dont le but est de vérifier, par la simulation, la validité d'une architecture logicielle donnée. Il fut proposé à l'origine au projet ARPA (Advanced Research Projects Agency) en 1990 par l'université de Stanford aux Etats Unis.

Avec le langage Rapide, une application est construite à partir de composants communiquant par échange de messages ou d'événements. Rapide fournit également un environnement composé d'un simulateur permettant de vérifier la validité de l'architecture. Les concepts de base du langage Rapide sont les événements, les composants, et l'architecture.

L'événement

L'événement est une information transmise. Il permet de construire des expressions appelées *event patterns* qui caractérisent les événements circulant entre composants. La construction de ces expressions se fait avec l'utilisation d'opérateurs qui définissent les dépendances entre événements. Parmi ces opérateurs on trouve l'opérateur de dépendance causale (A → B si l'événement B dépend causalement de A), l'opérateur d'indépendance A||B, l'opérateur de différence (A~B si A et B sont différents) et l'opérateur de simultanéité (A and B si A et B sont vérifiés). Ainsi, l'événement correspond à une information permettant de spécifier le comportement d'une application.

Le composant

Le composant est défini par une interface. Cette dernière est constituée d'un ensemble de services fournis et d'un ensemble de services requis. Les services sont de trois types :

- Les services *Provides*: fournis par le composant appelés de manière synchrone par d'autres composants ;
- Les services *Requires*: demandés par le composant appelés de manière synchrone ;
- Les *Actions*: qui correspondent à des appels asynchrones entre composants. Deux types d'actions existent : les actions in et out qui sont des événements acceptés et envoyés par un composant.

L'interface contient également une section de description du comportement (clause *behavior*) du composant. Cette dernière correspond au fonctionnement observable du composant comme, par exemple, l'ordonnancement des événements ou des appels aux services. Ainsi, l'environnement Rapide peut simuler le fonctionnement de l'application.

De plus, Rapide permet également de spécifier des contraintes (clause *constraint*) qui sont des patrons d'événements qui doivent ou non se produire pour un composant lors de son exécution. Par exemple, une contrainte peut fixer un ordre obligatoire pour une séquence d'événements d'un composant. En général, ces contraintes permettent de spécifier des restrictions sur le comportement des composants.

L'architecture

L'architecture contient la déclaration des instances de composants et les règles de connexions entre ces instances. Toutes les instances sont déclarées sous forme de variables. La règle d'interconnexion est composée de deux parties. La première est la partie gauche qui contient une expression d'événements qui doit être vérifiée, la seconde est la partie droite qui contient également une expression d'événements qui doivent être déclenchés après la vérification de l'expression de la partie de gauche. Les contraintes (*clause constraint*) peuvent être utilisées pour décrire l'architecture. Elles permettent de restreindre le comportement de l'architecture en définissant des patrons d'événements à appliquer pour certaines connexions entre composants. Les parties gauches et droites peuvent être connectées par trois types d'opérateurs :

- L'opérateur To : connecte deux expressions d'événements simples. Il ne peut y avoir qu'un événement possible vers un composant. Si la partie gauche est vérifiée alors l'expression de la partie droite permet le déclenchement de l'événement vers l'unique composant désigné par cette expression. Cet opérateur permet de spécifier un appel de type RPC.

- L'opérateur de diffusion ||> connecte deux expressions quelconques. Dès que la partie gauche est vérifiée, tous les événements contenus dans la partie droite sont déclenchés. Ils sont envoyés vers l'ensemble des destinataires désignés dans cette expression. L'ordre d'évaluation de cette règle de connexion est quelconque. Un déclenchement de cette règle est indépendant d'autres déclenchements antérieurs ou postérieurs.

- L'opérateur pipeline => est identique au précédent mais l'ordre d'´evaluation des règles est contrôlé. Un déclenchement de cette règle est causalement dépendant des déclenchements antérieurs de cette même règle.

Exemple

Un exemple d'architecture producteur/consommateur est décrit par la spécification suivante de la figure 2.2

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

type  Consumer is interface
   action  in  Receive (N: Integer ) ;
   action  out  Ack(N : Integer ) ;
   behavior
        (?X in Integer ) Receive (?X)  =>  Ack (?X) ;
end Consumer;

architecture   ProdCon ( )   return   SomeType   is
         Prod  :  Producer (100) ;    Cons  :  Consumer ;
        connect
        (?n  in Integer )    Prod.Send (?n) => Cons.Receive (?n ) ;
        Cons.Ack(?n)  =>  Prod.Reply (?n ) ;
end architecture ProdCon ;
```

Figure 2.2: Exemple d'architecture producteur/consommateur dans Rapide

2.3.2.2 Evaluation

Rapide est un langage permettant une expression forte de la dynamicité d'une application. Ceci est assuré grâce aux concepts de base de ce langage qui sont les composants, les évènements et l'architecture.

Le langage Rapide ne fournit pas d'outils de génération de code, il permet cependant la description du comportement dynamique et il offre des méthodes de validation (simulation) des architectures qu'il décrit. Il permet aussi de vérifier certaines propriétés comme l'interblocage lors de l'exécution de l'application.

L'inconvénient majeur de Rapide est à l'absence d'un modèle explicite pour exprimer les connecteurs. Même si les interactions entre composants peuvent être spécifiées de manière statique et dynamique, le langage ne fournit pas un moyen de typer un connecteur. En effet, un composant est décrit par une interface ; il est donc typé et son interface peut être réutilisée. Par contre, un connecteur n'est pas décrit de manière explicite. Ceci ne favorise pas la réutilisation des connecteurs. Il faut noter aussi que Rapide ne présente aucun élément de structuration de l'application. De plus, les propriétés non fonctionnelles ne peuvent pas être décrites par l'interface d'un composant. La spécification des propriétés non fonctionnelles se limite à la modélisation du temps (définie par le langage de contrainte).

2.3.3 L'ADL Darwin

2.3.3.1 Présentation

Le langage Darwin **[38] [39]** a été développé au Distributed Software Engineering Group de l'Imperial College de Londre. Il est considéré comme un langage de description d'architecture, bien que celui-ci soit souvent appelé langage de configuration. Un langage de configuration favorise la description de la configuration d'une application, c'est-à-dire la description des interactions entre composants. La particularité de ce langage est qu'un composant est une entité instanciable. La description d'un composant au niveau du langage permet de créer de multiples instances d'un composant lors de l'exécution. Ainsi, ce type de langage se centre sur l'expression du comportement d'une application plutôt que sur la description structurelle de l'architecture d'un système comme le font de nombreux ADLs **[40].** La particularité de Darwin est de permettre la spécification d'une partie de la dynamique de l'application en terme de schéma de création de composants logiciels avant, après ou en cours d'exécution.

Le composant

Comme pour la plupart des ADLs, un composant est une instance définie par une interface qui décrit les services fournis et requis pour lui permettre de fonctionner et de s'intégrer dans une architecture complexe. La sémantique associée au composant est le processus, chaque composant correspond à un processus créé.

Deux types de composants existent. Le premier correspond aux composants primitifs qui intègrent du code logiciel, le second aux composites qui sont des entités de configuration décrivant les interconnexions

Les composants primitifs

Les composants primitifs sont avant tout des entités d'encapsulation de fonctions et données d'un module logiciel. La description d'un composant est constituée de son nom et son interface qui déclare les services fournis et requis (**provide** ou **require**) par le composant. Le composant primitif sera représenté à l'exécution par une classe C++. Celle-ci hérite d'une autre classe *process*. Elle est issue des classes de base de **Regis [41]** support d'exécution répartis de Darwin.

Les composants composites

La structure finale du système est représentée par un composant composite. Ces composites peuvent être alors considérés comme des entités de configuration puisque ils contiennent la description complète de l'application. A l'exécution, ces composites sont représentés par un ensemble d'instances de composants qui s'exécutent en concurrence.

La description d'un composant composite dans le langage est constituée de son nom suivi d'une liste de paramètres typés. Il est alors possible d'utiliser la valeur des paramètres à l'intérieur du composite, pour décrire par exemple, le nombre d'instances de composants à créer. Vient ensuite l'implémentation du composant composite qui est constituée de déclarations d'instances et de schémas d'interconnexion. Deux constructions syntaxiques permettent de définir des schémas d'instanciation :

- l'opérateur *inst* qui déclare une instance de composant sur un site particulier. Cet opérateur permet de décrire la phase d'initialisation ;
- l'opérateur *bind* qui relie un port requis d'un composant à un port fourni d'un autre composant. Cet opérateur permet de décrire les liens entre composants au moment de

l'exécution. Cet opérateur peut servir à lier un port d'un composant composite avec un port d'un composant primitif faisant partie du composite.

Services de composants

Les composants interagissent avec les services. La notion de connecteurs n'apparait pas dans ce langage. En effet, chaque interaction est représentée par un lien entre un service requis et un service fourni entre des composants différents. Ces services désignent le type d'objet de communication utilisé, créé et géré par le système d'exécution *Regis*. Parmi les types d'objet, le port est le plus courant : il s'agit d'un objet envoyant des requêtes de manière synchrone ou asynchrone entre composants répartis ou non. La déclaration d'un composant suit la syntaxe suivante :

```
Component nom (liste de paramètres)
Provide nomPort <port, signature>
Require nomPort <port, signature>
```

Opérateurs particuliers

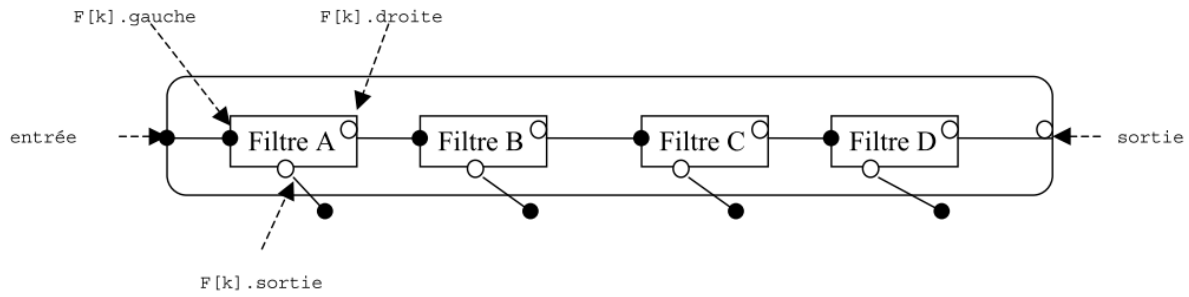
Darwin permet de décrire un schéma d'instanciation de composants très évolué. Par exemple, il est possible de définir des variables tels que des compteurs. Il existe également des constructions syntaxiques telles que l'itérateur *forall* qui permet de faire des itérations sur des variables entières et l'opérateur de test *when* qui permet d'évaluer une condition et de réagir selon cette évaluation. Ainsi, il est possible de spécifier le comportement de l'application au niveau comportement global en spécifiant la coopération des instances de composants.

L'environnement Regis

Nous rappelons que *Regis* **[41]** est le support d'exécution de Drawin permettant à des configurations de s'exécuter. Regis fournit un cadre écrit en C++ pour construire et exécuter des programmes distribués et spécifiés par Darwin. Chaque composant primitif est implanté sous forme d'objet actif et communique avec les autres composants grâce à des objets de communication qui correspondent aux services fournis et requis de Darwin; le code C++ implantant les liens entre ceux-ci est généré à partir des descriptions faites par le langage de description. L'environnement Regis permet aux programmeurs d'utiliser plusieurs types d'objets de communication tel que le port ou la diffusion multiple d'événements et autorise la création de nouveaux styles d'interaction.

Exemple

L'exemple de la figure 2.3 tiré de **[40]** illustre un composant de type *pipeline* composé par une liste d'instances de composant *filter*. L'entrée input de chaque instance est connectée à la sortie output de son prédécesseur. L'itérateur *forall* permet d'instancier chacune des instances (inst) et de les connecter (bind). Lorsqu'une interface n'est pas satisfaite à l'intérieur du composant, le composant peut l'exposer comme un besoin à satisfaire par l'extérieur, c'est-à-dire par d'autres composants compatibles. Ceci est le cas, par exemple dans : F[n-1].entrée – sortie.




```
component pipeline (int n) {
    // Interface
    provide entrée;
    require sortie;
    // Implantation
    array F[n] : filtre;
    // Définition d'un ensemble d'instances de filtre
    forall k: 0..n-1 {
        inst F[k]; // création d'une instance de filtre
        bind F[k].sortie -- sortie;
        // Lien avec le composant pipeline
            when k<n-1
                bind F[k].droite -- F[k+1].gauche;
                // Lien des composants entre eux
    }
    bind entrée -- F[0].gauche;
    F[n-1].entrée -- sortie;
}
```

Figure 2.3: Architecture pipeline à base d'un composant composite dans Darwin

2.3.3.2 Evaluation

Darwin propose une vision de l'architecture claire et fonctionnelle. En effet, le langage fournit une syntaxe riche permettant de décrire les interactions entre composants. Ce langage considère un élément de l'architecture (le composant) comme une entité pouvant être instanciée. Ceci a pour conséquence de spécifier de manière plus précise les interactions entre composants. Darwin permet également de décrire de manière explicite la répartition des composants d'une application en spécifiant pour chaque instance le site sur lequel elle

s'exécute. Et finalement, la séparation entre le code fonctionnel et les codes de communication et de prise en charge de la répartition semble être complète et configurable.

Cependant, Darwin est limité par le fait qu'un composant n'est associé qu'à une seule sémantique qui est le processus. Ainsi, un composant ne permet pas d'exprimer un autre élément tel qu'un fichier ou une mémoire partagée.

Le code d'utilisation de l'environnement *Regis* n'est pas transparent pour le programmeur. Le programmeur doit explicitement utiliser les objets de type port pour réaliser ses communications et effectuer les créations de composants lorsqu'ils les sollicitent.

2.3.4 ACME

2.3.4.1 Présentation

ACME **[42] [43]** est un langage de description d'architecture établi par la communauté scientifique dans le domaine des architectures logicielles. Il a pour buts principaux de fournir un langage pivot qui prend en compte les caractéristiques communes de l'ensemble des ADLs, qui soit compatible avec leurs terminologies et qui propose un langage permettant d'intégrer facilement de nouveaux ADLs. Il a été créé sur le fait que beaucoup d'ADLs ont des points communs dans leur manière d'analyser une architecture. En effet, la plupart des langages fournissent des notions similaires comme le composant ou le connecteur. ACME apparaît alors plus comme un langage fédérateur de ce qui existe que comme un langage réellement novateur. Les principaux éléments de ACME sont les suivants :

Les composants

Le composant représente l'élément primitif de calcul et d'enregistrement des données du système. Des exemples typique de composants sont les processus, les serveurs, les bases de données, etc.

Les ports

Permettent aux composants d'exposer leurs fonctionnalités. Chaque port identifie un point d'interaction entre le composant et son environnement. Il peut s'agir, par exemple, d'une simple signature d'une méthode ou d'un ensemble de procédures qui doivent être invoqués dans un ordre défini.

Les connecteurs

Le connecteur représente l'interaction entre composants. Il s'agit d'un médiateur de communication qui coordonne les connexions entre composants. Des exemples simples d'interaction sont les pipes, appel de procédure, diffusion d'événements, etc. Mais des interactions plus complexes peuvent être représentées, comme une requête SQL entre une base de données et une application

Les rôles

Définissent des interfaces des connecteurs. Chaque rôle d'un connecteur définit un participant à une interaction. La plus part des connecteurs sont binaires, ils possèdent deux rôles comme appelé et appelant dune communication RPC, ou émetteur et récepteur pour un envoi de messages.

Les systèmes

Le système représente la configuration d'une application, c'est-à-dire l'assemblage structurel entre les composants et les connecteurs, décrivant les entités de l'application et la nature de leurs interactions.

Les représentations

La représentation et la carte de représentation (Figure 2.4) permettent à ACME de supporter la description hiérarchique d'une architecture. Ainsi, un composant ou un connecteur peut être décrit d'un niveau général à un niveau plus détaillé et peut donc être raffiné. Chaque nouvelle description (sous élément) d'un élément est appelée une représentation. La correspondance entre l'élément et ses représentations est spécifiée grâce à la carte de représentation. Ainsi, la carte de représentation permet d'établir la correspondance entre les ports de l'interface d'un composant et ceux définis dans les interfaces de ses sous composants.

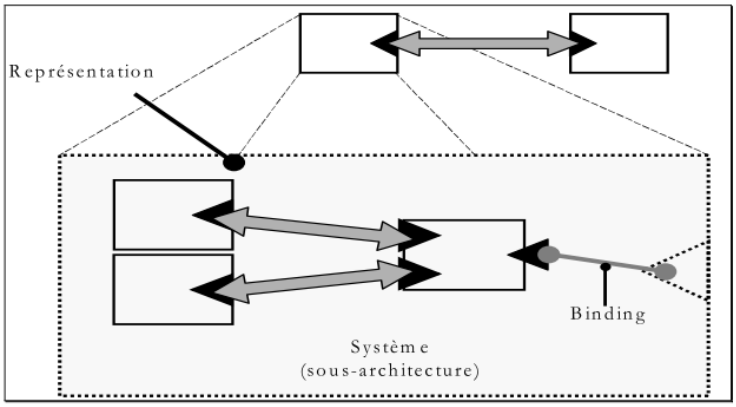


Figure 2.4 : Représentation sous Acme

Les propriétés

Les concepts présentés ci-dessus permettent de décrire l'aspect structurel de l'architecture de logiciel. Pour améliorer la description des composants, des connecteurs et des systèmes, chaque ADL offre son propre ensemble d'informations supplémentaires pour définir la sémantique à l'exécution. Acme propose un moyen d'intégrer ces informations travers la notion de propriété (Figure 2.5). Une propriété a un *nom* qui l'identifie, un *type* optionnel et une *valeur*. Chaque entité (composant, connecteur) peut avoir une ou plusieurs propriétés.

Le type optionnel de la propriété ACME peut indiquer :

- un type simple comme un entier, une chaîne de caractère, ou un booléen,

- un type indiquant une propriété d'un sous-langage ADL comme Wright, UniCon, Rapide; dans ce cas, le nom de la propriété indique le nom du langage,

- un type indiquant un lien externe (type « *external* ») avec une implantation comme par exemple un programme.

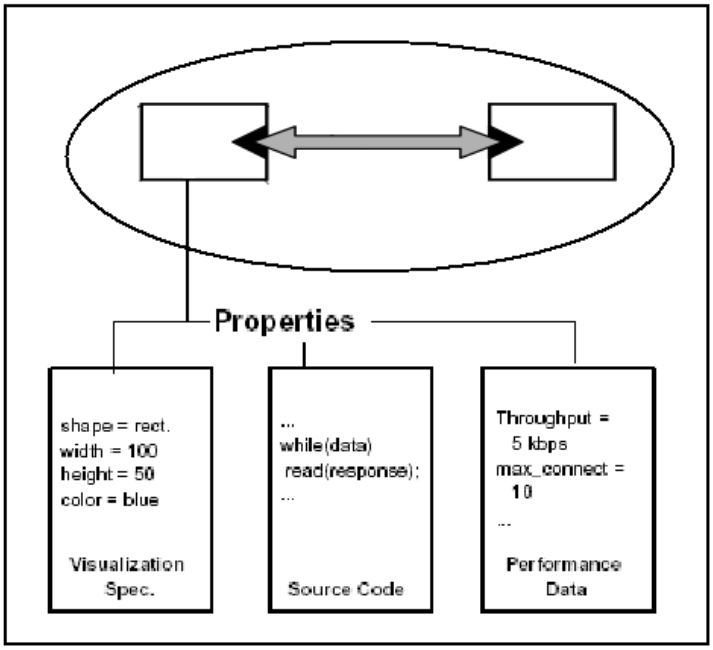


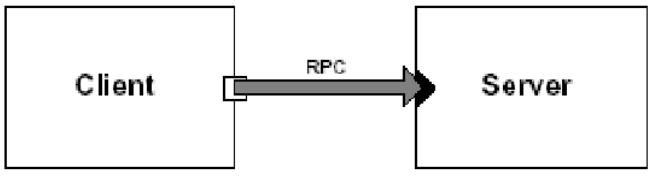
Figure 2.5: Propriétés sous Acme

Les styles

ACME fournit un moyen de décrire des gabarits de conception (*templates*). Cette notion est équivalente à la notion de style d'architecture que l'on peut trouver dans la plupart des ADLs. Le langage permet de créer des gabarits de conception paramétrables et réutilisables permettant de spécifier des patrons de conception.

Exemple

Pour illustrer la spécification d'architecture logicielle sous Acme un exemple d'architecture Client / Serveur est présenté ci-dessous (Figure 2.6).




```
System simple_cs = {
    Component client = {
    Port send-request;
Properties { Aesop-style : style-id = client-server; /*
propriété permettant
/* d'indiquer que le composant client dans ACME est
/* défini comme un style dans Aesop
UniCon-style : style-id = cs;
source-code : external = "CODE-LIB/client.c" }}
Component server = {
```

```
Port receive-request;
Properties { idempotence : boolean = true;
max-concurrent-clients : integer = 1;
source-code : external = "CODE-LIB/server.c" }}
Connector rpc = {
Roles {caller, callee}
Properties { synchronous : boolean = true;
max-roles : integer = 2;
protocol : Wright = "..." }}
Attachments {
client.send-request to rpc.caller ;
server.receive-request to rpc.callee }
}
```

Figure 2.6: Exemple d'architecture Client / Serveur sous Acme

2.3.4.2 Evaluation

Acme présente un atout en offrant un langage et une boite à outils servant de base à l'élaboration de nouveaux outils de construction. De plus, Acme, par son origine, est un langage générique permettant d'intégrer des notions d'ADLs existants plus spécifiques à un domaine ou à la spécification du comportement d'un composant, les architectes gagneront donc plus de force d'expression. Un autre atout de Acme est le fait que ce langage permet de séparer les préoccupations métiers des préoccupations techniques grâce une extension nommé AspectualACME **[48]** pour assurer la représentation modulaire des préoccupations transversales.

Malgré ses atouts, le langage Acme est sujet à quelque limite: Acme n'offre pas la possibilité d'avoir une intégration complète des différents langages de description. Ne prévoyant pas dans sa conception toutes les structures architecturales, certains systèmes décrits dans un ADL ne pourront pas à cet effet, être transcrit dans un autre. De plus Il ne fournit pas de moyen automatique pour raffiner la spécification d'application et n'encourage pas une description de l'application par une approche modulaire. Pourtant, la notion de représentation aurait pu servir de base à l'automatisation de cet aspect. La génération de code n'est pas non plus proposée par l'environnement du langage.

2.3.5 Fractal

2.3.5.1 Présentation

Le modèle de composant Fractal **[51]** a été défini par France Télécom R&D et l'INRIA dans le cadre du consortium ObjectWeb[3] pour le middleware open source. Le modèle FRACTAL est indépendant de son implémentation ainsi que des langages qui en

[3] http://www.objectweb.org

découlent. Il existe plusieurs implémentations du modèle dans divers langages. Cela permet de viser des contextes applicatifs différents. Parmi ces implémentations, nous pouvons citer :

– Julia **[52]** en Java,

– AOKell **[53]** en AspectJ,

– Think (THink Is Not a Kernel) **[54]** en C,

– FractNet **[55]** en .Net.

ADL Fractal

En amont du modèle de composant Fractal, il est possible de décrire l'architecture d'une application à base de composants à l'aide de l'ADL Fractal. Basé sur XML, ce langage définit une syntaxe abstraite indépendante de tout langage de programmation pour la description de l'architecture en termes de composants, d'interfaces, de liaisons et d'attributs. L'ADL Fractal, dans la lignée d'ADL comme xADL **[56]**, est un ADL extensible. Il permet d'intégrer facilement de nouveaux concepts au niveau de la description de l'architecture en modifiant la grammaire du langage. Cette grammaire est en effet construite à partir d'un ensemble de modules prenant en charge chacun une caractéristique du modèle de composant comme les interfaces, les liaisons ou les attributs. Il est possible de définir son propre module pour ces propres caractéristiques ou simplement redéfinir des caractéristiques existantes pour définir la syntaxe de son langage de description d'architecture. On retrouve au niveau du modèle concret de composant Fractal les concepts de composant, connecteur et configuration.

Le composant

Les composants possèdent une membrane qui délimite clairement leur contenu de leur environnement extérieur afin de structurer l'application. Cette membrane dispose d'interfaces externes (resp. internes) permettant la communication des composants avec l'extérieur (resp. au sein de leur contenu). Leur contenu consiste en un ensemble fini de sous-composants. Le modèle est donc hiérarchique permet ainsi une construction récursive qui s'arrête aux composants primitifs qui sont directement programmés dans un langage de programmation donné (Figure 2.7).

Les interfaces jouent un rôle central dans Fractal. Elles appartiennent à deux catégories distinctes : les interfaces métier et les interfaces de contrôle (la membrane

correspond à l'ensemble de ces interfaces de contrôle). L'ensemble des interfaces métier et de contrôle d'un composant définit son type. Les interfaces métier sont les points d'accès externes au composant alors que les interfaces de contrôle prennent en charge des propriétés non fonctionnelles du composant comme la gestion de son cycle de vie ou de ses liaisons. Une interface Fractal est composée d'un nom, d'une signature et d'un type. Dans la projection du modèle en Java, la signature d'une interface est une interface Java.

Une interface Fractal métier est du type client ou serveur. Une interface serveur identifie les services offerts par un composant alors qu'une interface client spécifie les fonctionnalités qu'un composant requiert pour son fonctionnement.

Le connecteur

Fractal ne possède pas de notion de connecteur explicite avec une sémantique de processus. Cependant, il utilise la notion de liaison pour spécifier les interactions entre composants. Une liaison Fractal est définie comme un lien orienté entre une interface client et une interface serveur. Ce lien permet aux composants d'interagir. Le modèle étant fortement typé, le type d'une interface serveur doit être obligatoirement du type ou du sous-type de l'interface cliente à laquelle elle est reliée. La liaison est assimilable à un connecteur implicite, elle ne possède pas de comportement propre.

La membrane possède à la fois des interfaces externes, qui sont accessibles de l'extérieur du composant, et des interfaces internes, accessibles seulement par son contenu. Dans le modèle Fractal, une interface interne ne peut exister que symétriquement à une interface externe. Ce mécanisme d'interface interne sert principalement à permettre les traversées de membranes des composites en conservant la sémantique de la liaison. Ainsi la liaison permet de définir à la fois les interactions entre deux composants mais permet aussi de définir les liens de délégation entre les interfaces d'un composite et les composants de son contenu.

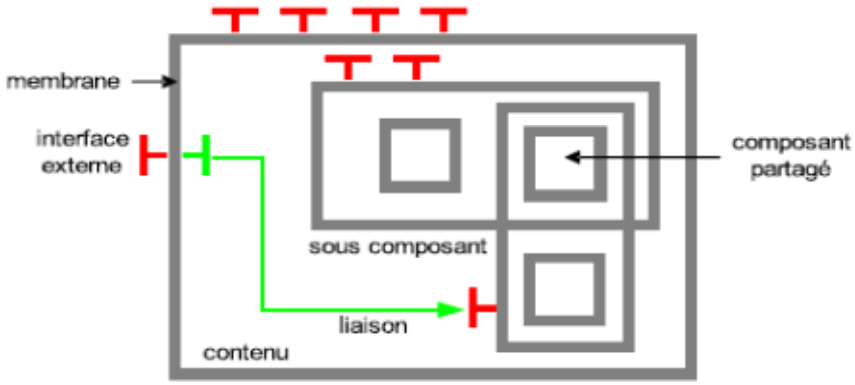


Figure 2.7 Structure d'un composant Fractal

La configuration

Fractal est un modèle hiérarchique, un composant peut posséder au sein de son contenu d'autres composants. Il est identifié dès lors comme un composite. Le contenu d'un composite définit une configuration pour la mise en œuvre des services définis au niveau de ses interfaces métiers. Enfin, et c'est une spécificité du modèle Fractal, un composant peut appartenir au contenu de deux composites qui ne sont pas imbriqués l'un dans l'autre. Il est alors dit *partagé*.

Exemple

L'exemple ci-dessous montre une application très simple constituée d'un composant composite contenant deux composants primitifs. Le premier composant primitif est *"server"* et qui fournit une interface *s*. Le deuxième composant primitif est *"client"* lié à l'interface du serveur précédent (Figure 2.8).

```
//Le composant Serveur
@Component(provides=@Interface(name="s",signature=Service.class) )

public class ServeurImpl implements Service {
   public void print( String msg ) {
     System.out.println(msg);
     }
 }
//Le composant Client
@Component(provides=@Interface(name="r",signature=Runnable.class) )
public class ClientImpl implements Runnable {
@Requires(name="s")
private Service service;
     public void run() {
        service.print("Hello world!");
      }
}
//L'assemblage
<definition name="HelloWorld">
```

```
<interface name="r" role="server" signature="java.lang.Runnable" />
<component name="client" definition="ClientImpl" />
<component name="serveur" definition="ServeurImpl" />
<binding client="this.r" server="client.r" />
<binding client="client.s" server="serveur.s" />
</definition>
```

Figure 2.8: Exemple de description Fractal

2.3.5.2 Evaluation

Fractal est un modèle simple et léger. Sa prise en main est aisée pour les programmeurs et les architectes issus de l'objet. Fractal permet de construire des applications par composition et son modèle de composant est indépendant de toute technologie. Il fournit plusieurs implémentations du modèle dans divers langages.

Néanmoins, le modèle Fractal n'intègre pas explicitement la notion de port. Elle est en réalité complètement intégrée dans la notion d'interface. Une interface est ainsi considérée en Fractal à la fois comme un point de connexion d'un composant et comme un contrat fonctionnel. Cela provoque bien souvent des ambiguïtés. De plus, le support des connecteurs via les binding components semble assez primitif et peu étudié contrairement à d'autres approches. Enfin Fractal ne permet pas une séparation claire entre les propriétés fonctionnelles et non fonctionnelles prises en charge par la membrane.

2.3.6 ArchJava

2.3.6.1 Présentation

ArchJava **[44, 45]** est un langage de description d'architectures logicielles, créé à l'université de Washington par Jonathan Aldrich et Craig Chambers. ArchJava a pour objectif d'améliorer la compréhension des programmes, de garantir l'architecture de l'application, de permettre une meilleure évolutivité des applications. En effet ArchJava étend le langage Java afin d'incorporer les éléments d'architectures à l'intérieur du code. La prise en compte des concepts architecturaux au niveau des langages de programmation permet une traçabilité tout au long de la phase de développement des concepts architecturaux, ce qui permet une meilleure évolutivité des applications et d'encourager les développeurs à se servir des avantages offerts par les concepts de composant, connecteur et configuration.

Le modèle de composant d'ArchJava définit, comme la plupart des ADLs, les trois concepts de composant, connecteur et configuration. ArchJava reprend ces concepts, comme

élément de structure du code de l'application. Il ajoute de nouveaux éléments de syntaxe au langage Java pour gérer les composants, les connecteurs et les ports.

Le composant

Les composants sont les seules unités d'encapsulation définies dans le langage ArchJava. Les composants communiquent avec d'autres composants via des ports. Un port définit un ensemble de méthodes, au sens Java classique, qui définissent des points d'accès à un composant. Dans ce sens, les ports ArchJava peuvent être considérés comme des interfaces des composants au sens large. Un composant peut avoir plusieurs ports. Contrairement à la plupart des modèles qui définissent un sens associé à chaque interface (serveur/client ou fourni/requis), aucune propriété de ce type n'est associée aux ports dans ArchJava. En effet, ce sont les méthodes associées à un port qui vont donner ce type d'informations. Dans ces termes, une méthode peut être soit de type fourni (implantée par le composant), soit de type requis ou diffusion (appelée par le composant).

Le connecteur

La communication entre les composants se fait à l'aide d'un appel de méthode. ArchJava garantit que les communications entre composants respecteront les connexions définies au niveau de l'architecture et qu'aucune autre interaction entre composants ne sera possible. La vérification de cette propriété est réalisée à la compilation, évitant l'apparition impromptue de messages d'erreur à l'exécution. Ainsi, les interactions entre composants connectés et entre un parent et ses sous-composants immédiats sont autorisées. Par contre, les appels externes à un sous-composant et les appels entre composants non connectés sont interdits. ArchJava interdit aussi les appels violant la structure hiérarchique de l'architecture.

ArchJava permet enfin de définir des classes de connexion appelées plus communément connecteurs complexes. En effet, lorsque le programmeur désire découpler l'implémentation des composants de la façon dont ils communiquent, il peut transformer les connexions entre les composants en objets à part entière. Des objets dont le seul but est de gérer la communication en tant que telle. Ces classes de connexion offrent un mécanisme d'extension simple permettant à l'architecte de transformer le simple appel de méthode (sémantique de base de la connexion en ArchJava) en un appel de procédure à distance ou en une interaction chiffrée. Au niveau langage, quand l'architecte utilise de tels connecteurs, il

doit utiliser le mot clé *with* à la suite du mot clé *connect* pour préciser la classe qui permettra le nouveau type de connexion, cette classe étendant nécessairement la classe *Connector*.

Configuration

Les composants composites sont les unités de configuration. En effet, ce sont eux qui contiennent la description des composants d'une application ainsi que leurs interconnections. Une application est donc un composant composite. La communication entre les composants se fait par appel de méthodes.

Exemple

Pour illustrer la spécification en ArchJava, nous proposons de reprendre l'exemple du compilateur présenté dans **[44]**. La figure 2.9 présente l'architecture d'un compilateur formé de trois composants. L'implantation d'un composant primitif, tel que l'exemple du composant parser (Figure 2.9 (b)) est constituée de la définition de ses ports d'entrée et de sortie, et de l'implantation des méthodes fournies.

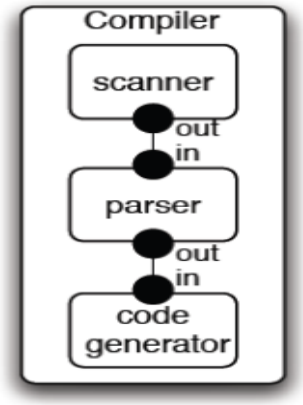
Dans le cas d'un composant composite, tel que l'application compilateur (Figure 2.9 (a) et (c)), les sous-composants sont déclarés comme des champs privés et finaux. De plus, les interconnexions entre les ports des sous-composants sont définies à ce niveau. Il est à noter qu'un composant composite peut exporter ou importer les interfaces de ses sous-composants et peut invoquer directement des opérations de ses sous-composants de premier ordre.

```
public component class Parser {
    public port in {
        provides void setInfo ( Token symbol , SymTabEntry e ) ;
        requires Token nextToken ( ) throws ScanException ;
    }

    public port out {
        provides SymTabEntry getInfo ( Token t ) ;
        requires void compile (AST ast ) ;
    }

    void setInfo ( Token t , SymTabEntry e ) { . . . } ;
    SymTabEntry getInfo ( Token t ) { . . . } ;
    . . .
}
```

(b)



(a)

```
public component class Compiler {
    private final Scanner scanner = . . . ;
    private final Parser parser = . . . ;
    private final CodeGen codegen = . . . ;

    connect scanner. out, parser.in;
    connect parser. out , codegen.in ;
    public static void  main ( String args [ ] ) {
        new Compiler( ).compile ( args ) ;
    }

    public void compile ( String args [ ] ) {
    / / for each file in args do :
        . . . parser.parse ( file ) ; . . .
    }
}
```

(c)

Figure 2.9 : L'architecture d'un compilateur à base de trois composants principaux (a), et la description en ArchJava du composant primitif Parser (b) et le composant composite Compiler (C)

2.3.6.2 Evaluation

ArchJava est l'une des rares approches à tenter d'intégrer dans l'implémentation les descriptions architecturales normalement décrites par les ADLs. L'atout de ArchJava est certainement le fait qu'il est une extension de Java dont la syntaxe est largement connue ce qui permet un apprentissage rapide **[46]** et contribue à son essor comparé à d'autres propositions bien plus lourdes à mettre en œuvre **[47]**.

Toutefois ArchJava ne permet pas de représenter des modèles conceptuels, il reste encore trop proche du code. En effet, la principale faiblesse d'ArchJava est justement d'être une extension de Java, cela suppose que tous les concepts et mécanismes objets (de Java) sont compatibles avec l'objectif visé ce qui réduit la flexibilité de ce langage. Le modèle d'ArchJava ne s'applique pour le moment qu'aux programmes écrits en Java et s'exécutant à l'intérieur d'une unique JVM (Java Virtual Machine). De plus La communication entre les composants se fait qu'à l'aide d'un appel de méthode cela ne correspond pas forcément avec les violentés des architectes.

2.3.7 UML 2.0

2.3.7.1 Présentation

Après le succès de la première version d'UML **[62]**, qui offre une notation graphique unifiée connue par la majorité des architectes et apportant une solution pour décrire l'architecture logicielle, une deuxième version notée UML 2.0 **[63]** introduit la notion de diagramme d'architecture, aussi appelée diagramme de structure composite. UML propose une notation unique pour couvrir toutes les phases du développement logiciel. Inspiré des

ADLs issus de la recherche académique, UML 2.0 propose les trois concepts fondamentaux des ADLs à savoir le composant, le connecteur et la configuration.

Le composant

Un composant UML2.0 est une entité modulaire et réutilisable fournissant et requérant des interfaces qui peuvent être potentiellement exposées par l'intermédiaire de ports. Un composant est vu comme une boîte noire. La partie interne d'un composant n'est accessible qu'au travers de ses interfaces. Le composant est une entité d'encapsulation. Ses dépendances sont explicitement définies. Il peut donc être facilement réutilisé au sein de plusieurs développements.

UML 2.0 définit deux types de composant : le composant basique (*Basic Component*) et le composant empaqueté (*Packaging Component*). Un composant basique est une entité instanciable qui interagit avec son environnement par l'intermédiaire de ports ou d'interfaces. La notion de composant empaqueté étend la notion de composant basique. Le composant empaqueté est une entité composée d'autres éléments liés de façon cohérente tout au long du processus de développement du logiciel.

Les interfaces

Une interface définit un type. Elle contient un ensemble de méthodes et/ou de contraintes. Une interface peut être attachée à un port fourni ou requis.

Le port

Un port est une entité qui émerge d'un composant. Le port se comporte comme un point d'interaction du composant avec l'environnement typé par plusieurs interfaces. Un port peut être requis, fourni ou complexe. Un port requis ne possède que des interfaces requises et signifie que l'instance du composant doit être connectée à un port de l'environnement fournissant les méthodes requises. Un port fourni ne contient que des interfaces fournies. Un port complexe peut contenir des interfaces requises ou fournies. Ce type de port est très utile pour décrire l'interaction d'un composant avec son environnement dans le cas de service complexe. Un port peut se voir attacher un comportement pour décrire les interactions qui doivent se produire sur le port. Les ports peuvent être connectés si et seulement si leurs spécifications de structure et de comportement sont compatibles.

Le connecteur

Un connecteur est une entité qui relie des ports ou des interfaces de composant. Il existe plusieurs types de connecteur :

– le connecteur d'assemblage qui permet d'assembler deux instances de composant en connectant un port fourni d'un composant au port requis de l'autre composant,

– le connecteur de délégation qui permet de connecter un port externe au port d'un sous-composant interne. L'invocation d'un service sur le port externe sera transmise au port interne auquel il est connecté.

Plusieurs connecteurs peuvent être attachés au même port ou à la même interface.

La configuration

Il y a deux façons de voir le composant en UML : la vision boîte noire présentée précédemment et une vision boîte blanche. Dans la vision boîte noire, le composant n'est défini qu'en fonction des éléments lui permettant d'interagir avec son environnement : ports, interfaces. Au contraire, dans la vision boîte blanche l'attention est portée sur l'organisation interne du composant en termes de sous-component, connecteur, etc.

Exemple

Nous illustrons la description structurelle de l'architecture d'un serveur Web. On retrouve sur ce schéma dans le formalisme défini par UML 2.0 le fait que le serveur Web est composé de cinq composants comme le montre la Figure 2.10.

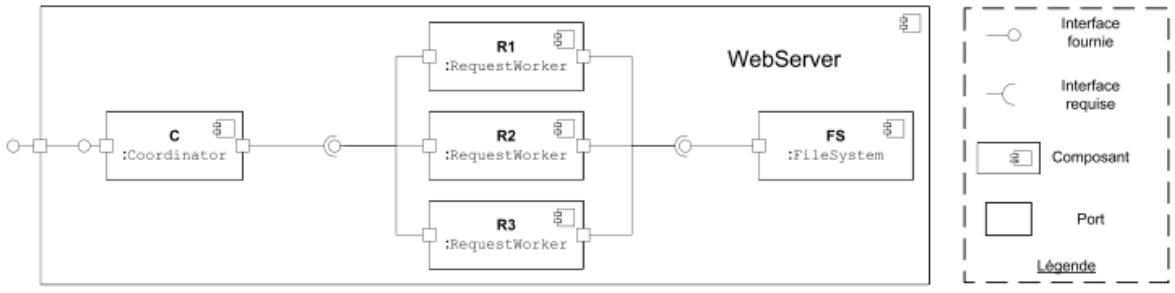


Figure 2.10: Architecture d'un serveur Web décrit àl'aide d'UML 2.0

2.3.7.2 Evaluation

Le point fort de UML 2.0 en tant que ADL est le fait qu'il est accepté et connu par la majorité des architectes, en effet, plusieurs travaux de recherche ont considérer UML 2.0 en

tant que langage de description d'architecture où il est intéressant de mapper vers cette notation **[64, 65, 66, 67, 68, 69, 70]**. De plus UML 2.0 est un langage à usage général de modélisation qui a été utilisé efficacement dans presque tous les domaines de l'ingénierie logiciels grâce aux nombreux outils qui sont disponibles, notamment pour la génération de code, nous citons à titre d'exemples les outils : BOUML[4], Modelio[5], Acceleo[6], Rational Rose[7] et Umbrello[8], etc.

Cependant UML 2.0 présente quelques inconvénients. Tout d'abord, il ne convient pas pour l'analyse automatisée de vérification et de validation de l'architecture. Les notations graphiques dans UML 2.0 manquent de sémantique formelle et peuvent donc devenir une source d'ambiguïté et d'incohérence dans certains cas **[71]**.De plus le connecteur dans UML est implicite ce qui empêche sa réutilisation.

2.3.8 AADL

2.3.8.1 Présentation

Le langage AADL (anciennement Avionics Architecture Description Language **[57]**), signifiant Architecture Analysis and Design Language **[58]**, est un ADL qui a été normalisé par le SAE (Society of Automotive Engineers) en 2004 pour la version 1, puis en 2009 pour la version 2.0.

AADL est à l'origine défini par différents partenaires du domaine de l'avionique, mais il s'adresse dans les faits à tous les systèmes embarqués temps réel, et pas uniquement aux systèmes avioniques. AADL est d'abord un langage textuel. Une annexe au standard définit une notation graphique associée. Une autre annexe définira un profil UML pour AADL.

Le composant

AADL considère deux niveaux de description du composant. Le premier, le type, correspond à l'interface fonctionnelle du composant, ce qui est visible des autres composants. Le second, l'implémentation, décrit le contenu du composant. Le modèle est hiérarchique, le contenu d'un composant consiste à définir ses sous-composants, les connexions en son sein et les délégations.

---

[4] www.bouml.free.fr
[5] www.modeliosoft.com
[6] www.Accelo.fr
[7] www.ibm.com/software/rational/
[8] uml.sourceforge.net/

En plus d'être typé, chaque composant appartient à une catégorie prédéfinie. Ces catégories représentent la nature des éléments manipulés. On retrouve dix grandes catégories dans AADL. Ainsi un composant peut être une mémoire (*memory*), un périphérique (*device*), un processeur (*processor*), un bus, une donnée (*data*), un sous-programme (*subprogram*), un processus léger (thread), un groupe de processus léger (*thread group*), un processus (*process*), un système (*system*).

Les composants communiquent les uns avec les autres par l'intermédiaire de ports. Ces ports sont des points d'entrée et/ou de sortie d'un composant. Il y a trois catégories de port:

– data : transportent des données;

– *event* : semblables aux signaux;

– *data event* :des signaux qui transportent des données.

Les connecteurs

Les interactions entre les composants sont matérialisées par la définition de connexions entre leurs ports. Une connexion permet de relier deux ports. Les ports peuvent être déclarés en entrée (*in*), sortie (*out*), ou entrée-sortie (*in out*). Une vérification est faite qu'il y a bien conformité de type et de sens entre les ports connectés. Les connecteurs n'ont pas de comportement associé, mais il est possible de leur ajouter des propriétés sur le délai de transmission des données (voir le paragraphe propriétés).

La configuration

De par la nature même des composants, le modèle d'AADL est hiérarchique. Par exemple, un processus peut contenir des processus légers. En outre, la notion de système qui peut lui même être composé de sous-systèmes permet de décomposer le système sur une infinité de niveaux.

Exemple

La figure 2.11 montre un système contenant deux processus, eux-mêmes contenant chacun un processus léger. Une succession de ports et de connexions établit une liaison entre les deux processus légers.

```
system system1
end system1;

system implementation system1.impl
    subcomponents
        p1: process process1.impl;
        p2: process process2.impl;
    connections
        cn: data port p1.outport ->
p2.inport;
end system1.impl;

process process1
    features
        outport: out data port;
end process1;

process              implementation
process1.impl
    subcomponents
        t1: thread thread1.impl;
    connections
        ……
thread thread1
    features
        outport: out data port;
    properties
        Deadline => 10O ms;
end thread1;
……
```

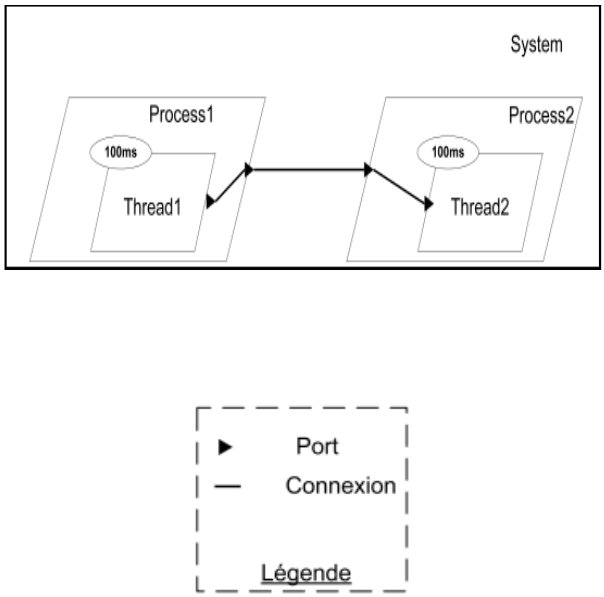


Figure 2.11: Exemple simple AADL

Les propriétés

À chaque élément de la description architecturale, on peut associer des propriétés et leur attribuer des valeurs. Celles-ci permettent de caractériser le composant. Certaines propriétés sont prédéfinies, c'est-à-dire qu'elles sont identifiées par un nom, un type et la liste des catégories de composant sur lesquelles elles s'appliquent. Par exemple, les tâches (processus léger) disposent de propriétés temps-réel telles que la période, l'échéance ou la durée d'exécution. De nouvelles propriétés ainsi que de nouveaux types de propriétés peuvent également être définis par l'utilisateur et associés à des catégories de composant. Ce mécanisme de propriétés permet de prendre en compte toute notion propre aux besoins de l'utilisateur.

Les annexes

Un autre mécanisme d'extension, appelé l'annexe, existe dans AADL. Elle permet d'utiliser des déclarations écrites dans d'autres sous-langages pour ajouter de l'information

sur un élément de l'architecture. Complémentaire des propriétés, une annexe sert à ajouter des précisions facultatives, alors qu'une propriété est très liée à la description architecturale.

2.3.8.2 Evaluation

Le Principal atout d'AADL réside dans sa prise en compte de plusieurs sémantiques au niveau du composant, entre autres, la description des éléments matériels qui sont peu définis dans les modèles académiques. De plus AADL offre la possibilité de définir les concepts de propriétés et d'annexes qui permettent d'ajouter de l'information non fonctionnelle à la description de l'interface du composant tel que des informations liées à la qualité de service fournie ou requise par un composant ou bien des informations supplémentaires (sélection du langage cible, la plate- forme cible etc.) pour la génération du code grâce au générateur de code de AADL(vers C++ et Ada) nommé Ocarina **[59]** .

Cependant, il manque à AADL une abstraction du comportement des composants qui permettrait de spécifier la relation entre les sorties d'un composant et ses entrées. De plus on a observé que AADL est un langage de description d'architecture déjà très orienté mise en œuvre. Les concepts de base du langage ont une forte implication quant aux choix liés à la réalisation des modules logiciels.

2.4 Récapitulatif des ADLs:

Les langages de description d'architecture logicielle que nous venons de présenter dans ce chapitre ont tous la même vocation qui est la proposition une base formelle pour spécifier les architectures logicielles en modélisant les composants, les connecteurs, et les configurations. Avec chacun leurs spécificités, leurs avantages et leurs inconvénients.

2.4 .1 Récapitulatif des Principaux avantages et inconvénients

Bien qu'ils aient beaucoup de points en commun, les buts recherchés par les ADLs, ne sont pas toujours les mêmes. Par exemple, certains s'intéressent plus particulièrement à la sémantique des composants et des connecteurs alors que d'autres s'attachent plutôt à définir les interconnexions entre composants et connecteurs. Chaque ADL possède alors ses propres atouts et ses propres inconvénients. Dans le tableau suivant (voir Tableau 2.1) nous résumons les principaux avantages et inconvénients des ADLs que nous avons présentés dans ce chapitre.

ADL	Principaux avantages	Principaux inconvénients
Wright	- Permet de spécifier une architecture logicielle de manière formelle et totalement abstraite (composant, connecteur, configuration), - Présence d'un modèle abstrait de composant et d'un modèle abstrait de connecteur. Les deux modèles sont indépendants.	- Difficile à assimiler, - Wright n'apporte pas de solution pour la hiérarchisation et la structuration de l'application, - Wright ne possède pas de moyen de projeter l'architecture logicielle vers un système concret (le passage de l'abstrait au concret est difficile), - Peu de moyens pour séparer les spécifications fonctionnelles et non fonctionnelles.
Rapide	- Rapide permet d'exprimer la dynamique d'une application de manière précise et détaillée, - Possibilité de simuler une application grâce à la création d'événements causals et grâce à l'environnement d'exécution	- Pas de représentation explicite de connecteur, - Ne fournit pas d'outils de génération de code, - Pas de moyens pour séparer les spécifications fonctionnelles et non fonctionnelles.
Darwin	- Darwin propose une syntaxe riche pour décrire les interactions entre composants, - Darwin permet de décrire de manière explicite la répartition des composants d'une application en spécifiant pour chaque instance le site sur lequel elle s'exécute, - L'environnement Regis sépare totalement lors de sa génération, le code fonctionnel (composant) et le code de communication (connecteur).	- Le composant n'est associé qu'à une seule sémantique, le processus, - L'environnement Régis n'est pas transparent : le programmeur doit intégrer manuellement dans son code fonctionnel les classes générées par l'environnement Regis correspondant aux connecteurs appelés « objets de communication » dans l'environnement Regis, - L'utilisation de divers modes de communication (synchrone, asynchrone) n'est pas configurable, - La description de la dynamique est limitée, - Impossibilité de décrire les propriétés non fonctionnelles.

Acme	- Langage pivot et fédérateur de l'ensemble des concepts utilisés par les ADLs, - Langage d'intégration des ADLs existants, - Il est possible de séparer les préoccupations métiers des préoccupations techniques des applications grâce une extension nommé Aspectual ACME.	- Langage orienté vers la spécification structurelle d'une architecture logicielle : très peu de moyen pour exprimer la dynamique d'un système, - Langage de modélisation qui n'offre pas de moyen de projeter la spécification d'une architecture logicielle vers un système.
Fractal	- Fractal est un modèle simple et léger. Sa prise en main est aisée pour les programmeurs et les architectes issus de l'objet. - Fractal permet de construire des applications par composition et son modèle de composant est indépendant de toute technologie, - Il fournit plusieurs implémentations du modèle dans divers langages.	- Fractal n'intègre pas explicitement la notion de port. Elle est intégrée dans la notion d'interface cela provoque bien souvent des ambiguïtés, - Le support des connecteurs via les binding components semble assez primitif et peu étudié, - Fractal ne permet pas une séparation claire entre les propriétés fonctionnelles et non fonctionnelles prises en charge par la membrane.
Archjava	- ArchJava est une extension de Java dont la syntaxe est largement connue ce qui permet un apprentissage rapide et une bonne lisibilité du programme.	- ArchJava ne permet pas de représenter des modèles conceptuels, il reste encore trop proche du code ce qui réduit la flexibilité de ce langage. - Le modèle d'ArchJava ne s'applique pour le moment qu'aux programmes écrits en Java et s'exécutant à l'intérieur d'une unique JVM, - La communication entre les composants se fait qu'à l'aide d'un appel de méthode cela ne correspond pas forcément avec les violentés des architectes, - Manque de séparation claire entre les propriétés fonctionnelles et non

		fonctionnelles.
AADL	- AADL propose plusieurs sémantiques au niveau du composant, entre autres, la description des éléments matériels, - AADL offre la possibilité de définir les concepts de propriétés et d'annexes qui permettent d'ajouter de l'information non fonctionnelle à la description de l'interface du composant, - Possibilité de Génération du code vers C++ et Ada.	- Il manque à AADL une abstraction du comportement des composants qui permettrait de spécifier la relation entre les sorties d'un composant et ses entrées, - AADL est un langage de description d'architecture déjà très orienté mise en œuvre en effet les concepts de base du langage ont une forte implication quant aux choix liés à la réalisation des modules logiciels, - AADL s'adresse plus particulièrement pour les systèmes embarqués temps réel.
UML 2.0	- UML 2.0 est accepté et connu par la majorité des architectes, - UML 2.0 une représentation graphique de l'architecture logicielle, - UML 2.0 est un langage à usage général de modélisation qui a été utilisé efficacement dans presque tous les domaines de l'ingénierie logiciels, - Disponibilité de nombreux outils notamment pour la génération de code.	- UML 2.0 ne convient pas pour l'analyse automatisée de vérification et de validation de l'architecture, - Les notations graphiques dans UML 2.0 manquent de sémantique formelle et peuvent donc devenir une source d'ambiguïté et d'incohérence, - Le langage définie une version très restreinte de la notion de connecteur, - Pas de moyens pour séparer les spécifications fonctionnelles et non fonctionnelles.

Tableau 2.1 : principaux avantages et inconvénients des ADLs.

Nous avons récapitulé les principaux avantages et inconvénients de chaque ADL. Il convient maintenant de les comparer. Nous proposons une comparaison des langages décrits en mettant en avant les principaux critères que doivent satisfaites un ADL pour aboutir à une description architecturelle flexible.

2.4 .2 Evaluation des différents ADLs présentés

Cette étude et analyse des différents ADLs, nous a permis de constater la grande diversité dans les caractéristiques et les objectifs de ces approches. De ces différences d'objectifs et de caractéristiques découlent de nombreux critères d'évaluation possibles d'une approche de spécification d'architecture logicielle capable de supporter la spécification directe des modèles mentaux d'architecture. Nous en retenons les critères suivants :

- Une bonne abstraction par rapport aux plates-formes d'exécution. En effet, une description d'architecture sert de pivot aux différents acteurs d'un projet informatique. En doit surtout représenté le modèle mentale de l'architecte .Le langage doit donc être suffisamment abstrait pour ne pas être noyé dans des considérations techniques. En outre, cette description étant généralement définie avant même le choix de la plate-forme cible, il est alors intéressant de pouvoir utiliser une partie de cette description pour générer du code vers la plate-forme cible.
- Une décomposition du système à l'aide de plusieurs dimensions. En effet, la notion de composant permet de structurer le système de manière fonctionnelle selon les services fournis et requis. D'autres niveaux de structuration doivent permettre de conserver la modularité de l'application.
- La description de l'assemblage et des interfaces des composants doit être la plus précise possible. Dans ce sens, un langage de description d'architecture idéal doit fournir suffisamment d'informations au niveau de l'interface du composant pour permettre une réutilisation sûre des composants logiciels.
- Enfin, l'architecte doit pouvoir construire son architecture de manière incrémentale en intégrant au fur et à mesure les préoccupations de son application.

Cette section dresse un bilan récapitulatif des différentes caractéristiques des différents ADLs et modèles de composant abordés dans ce chapitre en se focalisant sur les caractéristiques précitées.

2.4 .1.1 Abstraction : indépendance vis-à-vis de la plate-forme d'exécution

- La plupart des ADLs proposent une abstraction vis-à-vis de la plate-forme d'exécution. Cependant, ils ne proposent pas tous des mécanismes permettant de profiter de la description d'architecture du système dans la production de l'application. Ainsi Wright dont l'objectif était la production d'une spécification du système en vue de son analyse ne propose pas de mécanismes de projection de code. Le langage Rapide ne fournit pas

d'outils de génération de code, il permet cependant la description du comportement dynamique et il offre des méthodes de validation (simulation) des architectures qu'il décrit. Acme n'offre pas de moyen de projeter la spécification d'une architecture logicielle vers un système. La traçabilité des éléments d'architecture entre la spécification et la projection vers une plate-forme d'exécution est peu intuitive. Darwin propose une unique plate-forme d'exécution pour leur modèle de composant. Là encore, ArchJava est un cas à part. Comme il ne sépare pas la description de l'architecture de l'implantation des services fournis et requis par ses composants, la génération de code vers une autre plate-forme à composant semble difficile. Fractal est un bon modèle d'abstraction vis-à-vis de la plate-forme d'exécution. Défini lui-même à partir d'un modèle de composant abstrait, plusieurs implantations de Fractal existent et permettent de choisir sa plate-forme cible en fonction du contexte du projet. Ainsi, une implantation en C permet de construire des applications à composants pour de l'embarqué alors que plusieurs implantations en Java offrent davantage de fonctionnalités et permettent la construction de système d'information d'entreprise. AADL et UML 2.0 sont des modèles abstraits, ils ne possèdent pas de plate-forme propre, mais proposent des mécanismes de génération de code vers des plates-formes à composants existantes. Le **Tableau 2** résume, pour chacune des approches étudiées dans ce chapitre, l'existence d'un modèle abstrait de description d'architecture logicielle et l'existence d'une plate-forme d'exécution ou de mécanismes de génération de code.

ADL	Abstraction	Génération de code / plate-forme d'exécution
Wright	oui	non/non
Rapide	oui	non/non
Darwin	non	non/oui (Regis)
Acme	oui	non/non
Fractal	oui	non/oui (Julia,Think)
Archjava	non	oui (java)/oui (Machine virtuelle Java)
UML 2.0	oui	oui/non
AADL	oui	oui (ADA, C) / non

Tableau 2.2 : Indépendance vis-à-vis de la plate-forme d'exécution

2.4 .1.2 Modularité : structuration multi-dimensionnelle de l'application

Les architectures sont nécessaires pour décrire des systèmes logiciels à différents niveaux de détails. Par conséquent, la prise en compte de la composition ou de la composition hiérarchique est cruciale. Si les notions de composant, connecteur et configuration offrent un premier niveau de modularité dans la description d'une application, on constate que ces mécanismes ne permettent pas à eux seuls l'obtention d'un bon découpage modulaire. Certaines préoccupations transverses à l'application engendrent de très nombreuses dépendances entre composants. C'est le cas dans une architecture logicielle d'un composant en charge de préoccupations dites techniques comme la trace ou la persistance qui sont liées à de nombreux composants métier.

Si ces dépendances multiples sont, d'un point de vue conceptuel, inévitables (les composants ayant de réelles dépendances entre eux), il faut pouvoir définir un nouvel élément de structuration afin de ne pas dupliquer une information dans de nombreuses parties de la description d'architecture logicielle. Cette caractéristique, qui est essentielle pour permettre la spécification d'architecture logicielle fournissant un bon découpage modulaire, est encore très peu prise en compte dans les approches de l'architecture logicielle présentées dans ce chapitre. Fractal propose à travers des interfaces de contrôle un moyen de regrouper certaines préoccupations techniques du composant au sein d'une même entité dupliquée sur l'ensemble des composants concernés. Cependant, ces interfaces de contrôle étant pour le moment des interfaces fournies, elles ne permettent pas d'injecter une dépendance vers un composant technique. Le Tableau 2.3 récapitule les caractéristiques des ADLs présentés sous les critères de la composition hiérarchique et le découpage modulaire des préoccupations techniques des applications.

ADL	Composition Hiérarchique	Séparation des préoccupations techniques
Wright	non	non
Rapide	non	non
Darwin	non	non
Acme	oui	oui
Fractal	oui	non
Archjava	oui	non
UML 2.0	oui	non
AADL	oui	non

Tableau 2.3: les caractéristiques de la composition hiérarchique et le découpage modulaire des préoccupations techniques.

2.4 .1.3 Réutilisation : définition étendue des interfaces de composant

Comme une partie des ADLs travaille sur l'analyse d'une description d'architecture, beaucoup de langages ont fourni une définition étendue des interfaces de composant afin de disposer d'un maximum d'information sur le composant lors de l'analyse de l'assemblage. Ainsi, Wright, Darwin et Rapide proposent, au niveau des interfaces des composants et des rôles des connecteurs, une description des messages entrants et sortants de ces composants ou de ces connecteurs. Fractal ne dispose au niveau de son modèle qu'un contrat d'interface de premier niveau à l'aide d'une description de services offerts et requis par le composant. AADL pour sa part ne propose pas d'abstraction du comportement du composant au niveau du modèle. Il dispose néanmoins de mécanismes d'extension, les annexes et les propriétés qui permettent d'ajouter de l'information pour caractériser le composant. Grâce à ce mécanisme d'annotation, les différents niveaux de contrats peuvent être décrits en AADL. De même ACME propose un moyen d'intégrer les informations concernant la description des composants, des connecteurs et des systèmes à travers la notion de propriété. UML 2.0 propose, quant à lui, de nombreux mécanismes au niveau du langage pour enrichir la définition de l'interface d'un composant. Cependant, la définition d'information relative au comportement à tous les niveaux du composant (composant, port, interface) et le manque de cohérence entre ces informations gène pour le moment l'analyse d'un assemblage de composants UML 2.0. Finalement, le cas du langage ArchJava est de nouveau à part. Relativement peu utilisée pour l'analyse, la définition des interfaces de composant se limite à une définition syntaxique des services qu'il fournit et qu'il requiert.

2.4 .1.4 Construction incrémentale de la description d'architecture logicielle

La construction incrémentale d'une description d'architecture logicielle vise à offrir à l'architecte la possibilité de définir son architecture en plusieurs étapes. L'intérêt d'un tel mécanisme est double. Il permet de maîtriser la complexité liée à la spécification d'une architecture en permettant de n'adresser qu'une problématique à la fois. En outre, la description de l'architecture logicielle doit être un document central compréhensible par l'ensemble des acteurs d'un projet informatique afin que chacun puisse identifier facilement son rôle au sein de ce projet. Dès lors, une construction incrémentale permet d'obtenir différents niveaux de granularité autour d'une architecture logicielle et permet au client d'avoir une vue de très haut niveau de son système et à l'expert en sécurité de voir en détailles dépendances entre les composants métiers et les composants techniques.

Cette construction incrémentale n'est proposée pour le moment par aucune des approches étudiées dans ce chapitre. L'architecte voulant spécifier son architecture en plusieurs étapes sera obligé de modifier manuellement sa spécification pour détruire des liaisons, ajouter des interfaces, du comportement, etc. Cette opération manuelle si elle peut très vite se révéler fastidieuse est en outre inefficace, car l'ensemble des vérifications effectuées à une étape *t* de la conception sont perdues et doivent être à nouveau effectuées à l'étape t+1.

2.5 Conclusion

Nous avons consacré ce chapitre à la présentation des langages de description des architectures logicielles à travers de leurs concepts descriptifs, leurs mécanismes opérationnels etc. Selon l'étude de ces ADLs que nous avons menée, nous avons identifié certaines lacunes. En effet, les points faibles rencontrés dans ces ADLs sont de plusieurs ordres, causant une rigidité des ADLs en empêchant de représenter d'une manière flexible le modèle mental de l'architecte. Nous avons illustré ce chapitre par une synthèse regroupant les principales avantages et inconvénients des ADLs étudié, ensuite nous avons terminé le chapitre par une comparaison des ces ADLs en mettant l'accent sur des critères que doivent satisfaire un ADL pour permettre de spécifier une architecture d'une manière flexible.

En conclusion, plusieurs efforts et tentatives étaient consacrés pour proposer des modèles abstraits pour la spécification d'architecture logicielle. Ces modèles abstraits ne sont pas complets, trop liés aux mécanismes logiciels ou bien ils manquent de liens avec les

modèles de réalisation. Une approche récente du génie logiciel : l'approche dirigée par les modèles qui conçoit l'intégralité du cycle de vie comme un processus de production, de raffinement itératif et d'intégration de modèles nous semble être intéressante. Dans le chapitre suivant nous allons aborder cette approche.

CHAPITRE 3
L'INGENIERIE DIRIGEE PAR LES MODELES

3.1 Introduction

Dans l'introduction de ce travail de recherche, nous avons indiqué que notre proposition repose sur la démarche de l'Ingénierie Dirigée par les Modèles (IDM)[9] et plus précisément de la variante que l'Object Management Group (OMG) a définie autour de son standard : le Model-Driven Architecture (MDA) **[87]**. Dans ce chapitre, nous présentons le principe et les concepts fondamentaux de cette approche.

3.2 Principes de l'approche

L'IDM spécifie l'intégralité du cycle de vie du logiciel comme un processus de production, de raffinement itératif et d'intégration de modèles. L'IDM a pour but d'apporter une nouvelle vision permettant de concevoir des applications en séparant la logique métier de l'entreprise, de toute plateforme technique. En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique **[88]**. Il est donc évident de séparer les deux pour faire face à la complexité des systèmes d'information et aux coûts excessifs de migration technologique. En ce sens, l'IDM fait évoluer l'usage des *modèles*, qui peuvent être interprétés ou transformés. Pour que ces modèles puissent être interprétés ou transformés, il faut que ces modèles soient définis dans un langage. Ce langage est appelé *métamodèle* dans la littérature relative à la démarche de l'IDM. Les métamodèles utilisés pour la définition de modèles peuvent être différents. Pour gérer cette diversité, l'IDM préconise d'utiliser un langage commun appelé *métamétamodèle* pour décrire tous les métamodèles impliqués dans la description des modèles afin de permettre leur intégration dans les outils de mise en œuvre du processus de construction.

Les travaux menés sur l'IDM font suite à la définition de MDA par l'OMG **[87]**. MDA a recours aux différents standards de l'OMG afin de décrire les démarches basées sur l'ingénierie des modèles. En effet, MDA est considéré comme un exemple particulier d'ingénierie dirigée par les modèles. Néanmoins, la plupart des travaux sur l'IDM font référence à MDA. Cet état de fait provient du fait que l'OMG catalyse, centralise, et synthétise bon nombre de travaux sur l'IDM. Pour cette raison, nous nous sommes basés dans

[9] Model Driven Engineering (MDE) en anglais.

la suite de ce travail sur l'approche MDA. La section suivante présente les concepts de base de l'IDM.

3.3 Définition des concepts fondamentaux

Nous proposons de définir les concepts de base de l'IDM afin d'appréhender de manière optimale cette approche. Cependant il n'existe pas de définitions standards de ces concepts .Cette section présente donc, les définitions qui nous semblent être acceptées par la plupart des utilisateurs de l'IDM dans la littérature.

3.3.1 Modèle

Un modèle est une abstraction simplifiée d'un système étudié, construite dans une intention particulière. Il doit pouvoir être utilisé pour répondre à des questions sur le système **[89]**. Un système est une construction théorique que forme l'esprit sur un sujet (ex. : une idée expliquant un phénomène physique et représentée par un modèle mathématique) **[90]**.

3.3.2 Métamodèle

Un métamodèle est un langage qui permet d'exprimer des modèles. Il définit les concepts ainsi que les relations entre concepts nécessaires à l'expression de modèles **[89]**. Un modèle est une construction possible du métamodèle dans lequel il est défini. Dans la littérature, un modèle est dit conforme au métamodèle dans lequel il est défini.

3.3.3 Métamétamodèle

Un métamétamodèle est un langage qui permet d'exprimer des métamodèles. C'est un langage unificateur et d'intégration pour les outils qui manipulent les métamodèles qui servent à décrire les modèles d'un même système. Dans le monde de l'IDM, il existe de nombreux langages tels que Kermeta **[92]**, MOF (Meta Object Facility) **[93, 94]**, EMOF (The Essential MOF Model) **[121]**, Ecore **[96]** etc. disponibles pour l'écriture de métamodèles.

3.3.4 Transformation de modèles

D'après Hubert Kadima **[97]** :

- Une transformation de modèles est la génération d'un ou plusieurs modèles cibles à partir d'un ou plusieurs modèles sources conformément à une définition de transformation.

- Une définition de transformation est un ensemble de règles de transformation qui décrivent globalement comment un modèle décrit dans un langage source peut être transformé en un modèle décrit dans un langage cible.
- Une règle de transformation est une description de la manière dont une ou plusieurs constructions dans un modèle source peuvent être transformées en une ou plusieurs constructions dans un modèle cible.

Dans l'IDM, un processus de construction d'un système est caractérisé par une séquence ordonnée (partiellement) de transformations de modèles.

3.4 Les quatre niveaux de MDA

Afin de mieux comprendre l'approche MDA, il convient de préciser plus en détail la nature des modèles utilisés tout au long du cycle de vie d'un logiciel, les divers usages de ces modèles, ainsi que les possibles intentions du concepteur au moment de leur construction. MDA a proposé une architecture à quatre niveaux qui structure les différents modèles pouvant être produits lors de l'application de l'approche de l'IDM **[99]**. Cette architecture (Figure 3.1) comporte quatre niveaux d'abstraction que nous allons détailler dans ce qui suit.

Le niveau M0

Le niveau M0, représente les objets du monde du réel. Il représente, par exemple, un compte bancaire avec son numéro et son solde actuel (Figure 3.1).

Le niveau M1

C'est au niveau M1 que les modèles sont édités. Ces modèles sont conformes aux métamodèles définis au niveau M2. Ainsi, MDA considère que si l'on veut décrire des informations appartenant au niveau M0, il faut d'abord construire un modèle appartenant au niveau M1. De ce fait, un modèle UML (comme le diagramme de classes ou le diagramme d'état/transition) est considéré comme appartenant au niveau M1. Il représenterait des objets manipulés dans le monde réel (décrits au niveau M0).

Le niveau M2

Le niveau M2, est le lieu de définition des métamodèles. Un métamodèle peut être considéré comme un langage spécialisé pour un aspect du système. Il peut aussi décrire les aspects spécifiques aux différents domaines, chaque aspect étant pris en compte dans un métamodèle spécifique. Les métamodèles contenus dans le niveau M2 sont tous des instances

du niveau M3 (notons qu'au niveau M3, il ne peut y avoir qu'un seul métamétamodèle). Dans le cadre de MDA, c'est le métamodèle d'UML **[100]** qui est le plus utilisé, celui-ci définit la structure interne des modèles UML.

Le niveau M3

Le niveau supérieur, M3 correspond au métamétamodèle. Il définit les notions de base permettant l'expression des métamodèles (niveau M2), et des modèles (M1). Pour éviter la multiplication des niveaux d'abstraction, le niveau M3 est réflexif, c'est-à-dire qu'il se définit par lui-même. Le plus souvent, c'est le métamétamodèle MOF (*Meta Object Facility*) qui est utilisé. Celui-ci est standardisé par l'OMG **[93]**. Cependant d'autres méta-métamodèles ont été proposés tels que Ecore (EMF) **[96]** et OWL **[101]**.

Afin d'avoir une idée plus précise de l'architecture à quatre niveaux d'OMG, nous illustrons par la figure 3.1, les modèles pouvant appartenir à chaque couche.

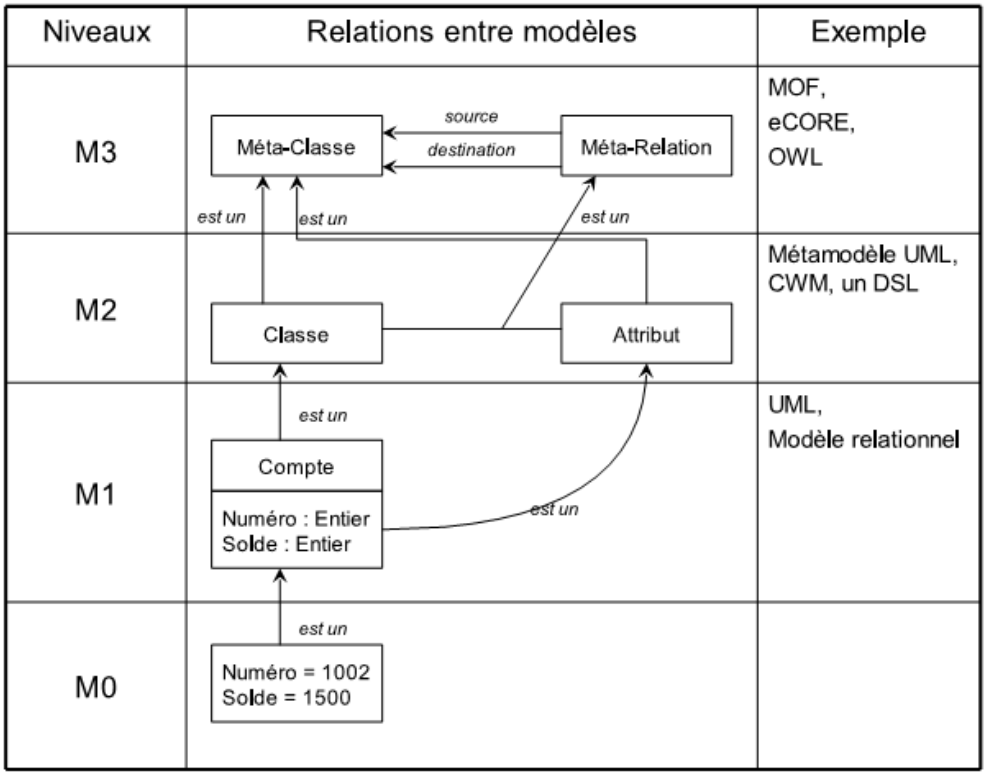


Figure 3.1:Les quatre niveaux de MDA

3.5 Les relations entre les modèles

Les flèches que nous avons annotées "*est un*" sur la figure 3.1 méritent un approfondissement de leur sémantique. Actuellement, trois relations fondamentales sont identifiées : "*représentation de*", "*être conforme à*" et "*être instance de*".

3.5.1 La relation "*représentation de*"

Cette relation (notée μ) lie le modèle au système qu'il représente. Cette relation traduit la sémantique qui existe entre un système et un modèle. En effet, il est communément admis qu'un modèle est la simplification subjective d'un système. Dans ce contexte, le modèle sert à obtenir des réponses par rapport au système qu'il représente. Par exemple, une carte géographique peut jouer le rôle de modèle, alors que la région étudiée jouera celui de système modélisé. De même qu'une carte elle-même peut jouer le rôle du système modélisé comme le montre la figure 3.2. Dans cette figure, la carte est elle même représentée par un schéma XML.

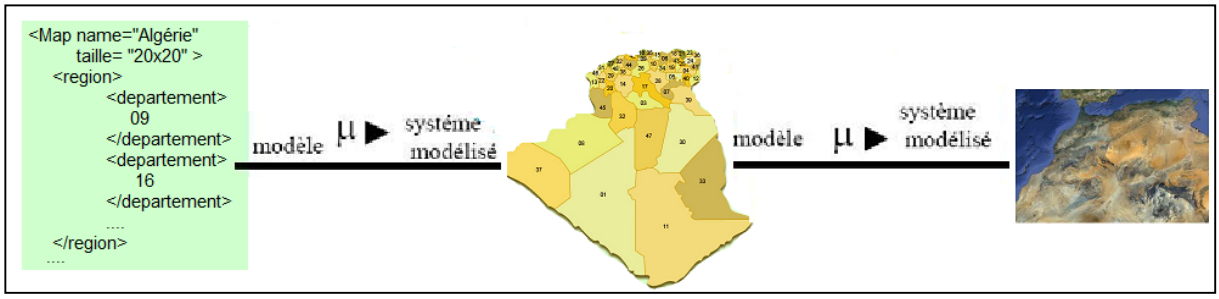


Figure 3.2: La relation μ

Il faut spécifier par ailleurs, que dans la figure précédente, le fichier modélisant la carte ne peut être considéré comme un métamodèle. En effet, il existe une nuance entre la définition d'un modèle et la définition d'un métamodèle. Alors qu'un modèle est une description d'un système ou d'une partie du système, un métamodèle est un modèle qui définit le langage qui exprime le modèle. Ainsi, la relation "représentation de" peut décrire le lien entre la couche M0 et M1, mais ne décrit pas le lien entre les couches M1, M2 et M3.

3.5.2 La relation "*être conforme à*"

La relation "*être conforme à*" (notée χ) est plus appropriée pour la description de liens entre les couches M1, M2 et M3. Elle spécifie en effet la relation existante entre les modèles et leur métamodèle. Manipuler informatiquement et opérationnaliser des modèles nécessitent qu'ils soient exprimés dans un langage clairement défini. L'architecture à quatre niveaux

permet d'avoir un cadre permettant cette définition : un modèle appartenant au niveau M1 est conforme à un métamodèle défini au niveau M2, qui est lui même conforme à un métamétamodèle défini au niveau M3.

La relation χ permet d'assurer qu'un modèle est correctement construit. A partir de ce moment, il devient envisageable de lui appliquer des transformations automatisées. La figure 3.3 démontre la relation entre une carte géographique et son métamodèle qui est constitué d'une notation graphique représentant les régions et les départements.

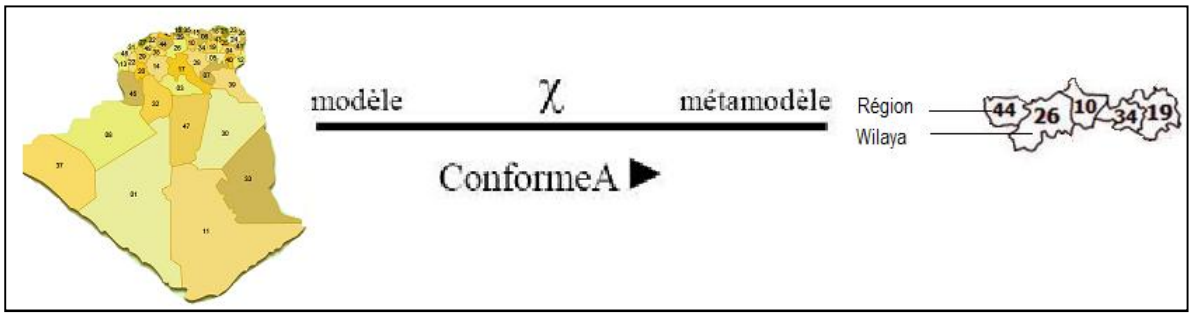


Figure 3.3: La relation χ

3.5.3 La relation "*InstanceDe*"

MDA est principalement fondée sur l'approche orientée objet. Ceci a généré de nombreuses confusions par rapport aux relations existantes entre les différentes couches de modèles. La relation "*InstanceDe*" est un exemple qui incarne cette confusion. Au niveau du standard MDA, il a été largement relayé par l'OMG qu'un modèle serait une "instance d'un" métamodèle. Or, ceci fait référence à l'hypothèse implicite que modèles et métamodèles sont exprimés dans une approche orientée-objet. Il n'existe aucune contrainte de cette sorte dans l'IDM. L'approche IDM se veut indépendante de toute contrainte technologique, pour cette raison, la relation "*InstanceDe*" est considérée comme une incarnation particulière de la relation "ConformeA" en orienté objet **[99]**.

3.6 Les différents modèles de MDA

Le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Ces modèles vont servir dans un premier temps à modéliser l'application, puis par transformations successives à générer du code. Aujourd'hui, la frontière entre les différents modèles n'est pas encore bien explicitée, ni formalisée. Néanmoins, il est possible de donner une description de chacun d'eux.

3.6.1 Computational Independent Model (CIM)

Dans la démarche du MDA **[87]**, la construction d'un système commence par l'élaboration du modèle métier ou modèle de domaine : le CIM. L'élaboration du CIM consiste à décrire le système indépendamment de tout système informatique en respectant les exigences des utilisateurs. Le CIM précise ce que le système doit faire dans l'environnement dans lequel il opère sans rentrer dans les détails de sa structure et de son implantation. Cette indépendance technique permet de capitaliser le savoir faire dans le CIM en faisant abstraction de la technologie d'implantation. De ce fait, le CIM peut être utilisé pour spécifier les liens de traçabilité avec les modèles (PIM ou PSM) qui sont décrits dans les autres étapes du processus de construction du logiciel. Il peut être utilisé également comme source de vocabulaires partagés avec ces modèles. Le MDA **[87]** souligne qu'il doit être possible de suivre les spécifications du CIM qui précisent les conditions d'utilisation du système dans la définition des PIM et PSM et inversement.

Avec UML, un CIM peut se résumer à un diagramme de cas d'utilisation. Ces derniers contiennent en effet les fonctionnalités fournies par l'application (cas d'utilisation) ainsi que les différentes entités qui interagissent avec elle (acteurs) sans apporter d'information sur le fonctionnement de l'application.

3.6.2 Platform Independent Model (PIM)

Une fois le CIM établi, il est possible de concevoir les modèles d'analyse et de conception ou PIM en respectant les conditions d'utilisation du système spécifiées dans le CIM. Le MDA stipule que le PIM doit être indépendant des plateformes techniques. Le PIM décrit le fonctionnement des entités et des services, intègre les aspects technologiques et architecturaux sans montrer les détails de l'utilisation du système sur sa plateforme. C'est un modèle informatique qui représente une vue partielle d'un CIM. MDA recommande d'utiliser UML **[102]** pour représenter les PIM. A ce niveau, le formalisme utilisé pour exprimer un PIM est un diagramme de classes en UML qui peut être couplé avec un langage de contrainte comme OCL (Object Constraint Language). Ceci étant, quelque soit le formalisme utilisé, le PIM doit être productif et peut contenir des informations sur la persistance, les transactions, la sécurité, etc. qui permettent d'établir le lien entre les exigences métiers et le code d'implantation effective du système **[91, 87].**

3.6.4 Platform Model ou Platform Description Model (PDM)

Après la définition du PIM, l'architecte doit choisir une plateforme technique qui perme d'implanter le système avec la qualité architecturale décrite dans le PIM. Cette plateforme doit être décrite dans un modèle afin de permettre son intégration dans la construction du système. Ce modèle est appelé PDM. L'OMG ne donne pas plus de précisions sur le PDM dans la description du MDA **[87]**. Dans la littérature **[103]**, on pense que chaque fournisseur de plateforme devrait proposer le PDM (Figure 3.4).

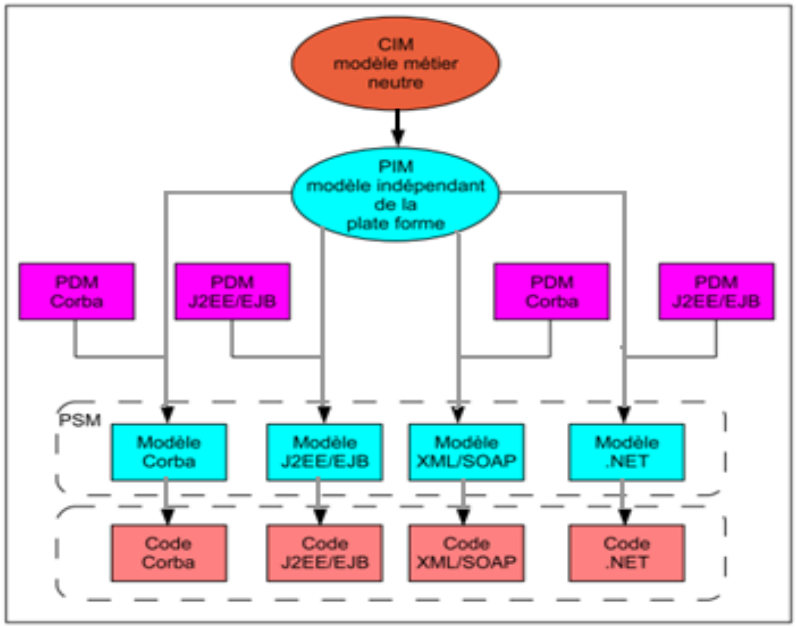


Figure 3.4: Principe de la construction d'un système à base de modèles dans le MDA [91]

3.6.5 Platform Specific Model (PSM)

Une fois le PIM et le PDM connus, on peut construire le modèle du code ou modèle de conception effective (concrète) du système: le PSM. Le PSM dépend de la plateforme technique choisie par l'architecte. Il indique comment le système sera utilisé sur cette plateforme. Le MDA estime que le code d'implantation effective du système peut être construit à partir du PSM (Figures 3.4). A ce titre, le PSM sert essentiellement de base à la génération du code exécutable vers la plateforme choisie. Il existe différents niveaux de PSM. Le premier est issu de la transformation d'un PIM (Figure 3.4). Le MDA recommande de le représenter à l'aide d'un profil (ou schéma) UML spécifique à la plateforme cible. Les autres PSM sont obtenus par transformations successives de modèles jusqu'à l'obtention du code d'implantation effective du système dans la plateforme considérée (Figure 3.4). Parmi les informations d'un PSM d'implémentation (Code) on peut trouver par exemple le code du

programme, les types pour l'implémentation, les programmes liés, les descripteurs de déploiement, etc. **[87]**. La section suivante présente les catégories de transformations de modèle qu'il est possible de réaliser dans le MDA.

3.7 Transformations de modèles

L'idée du MDA consiste à passer progressivement des PIM aux PSM pour préparer et faciliter la génération de code vers la plateforme technique choisie. Ce passage des PIM aux PSM est une transformation de modèles **[91, 87]**.

3.7.1 Type de transformation de MDA

Le MDA identifie différents types de transformations de modèles dans le cycle de vie et de développement d'un système :

PIM vers PIM : Permettent de raffiner les PIM afin d'améliorer la précision des informations qu'ils contiennent. En UML, de telles transformations peuvent être, par exemple, la création de classes UML à partir d'un ensemble de diagrammes de séquences ou la création de diagrammes d'états à partir de diagrammes de classes. Ces transformations sont omniprésentes dans les outils d'élaboration de modèles PIM. Elles permettent d'accélérer la production des PIM et donc de raccourcir le cycle de développement.

PIM vers PSM : Lorsque le PIM est suffisamment raffiné pour pouvoir être immergé dans une plateforme technique donnée, on peut le convertir en PSM en lui rajoutant des informations spécifiques à la plateforme technique choisie. Cette opération de conversion est une transformation de PIM vers PSM.

PSM vers PSM : Une transformation PIM vers PSM N'est pas toujours suffisante pour permettre la génération de code d'où la nécessité de raffiner davantage le PSM en utilisant des formats intermédiaires. Par exemple, pour générer un code C++, à partir d'un formalisme en UML, un passage d'UML vers SDL (Spécification and Description Language)**[98]**, puis de SDL vers C++ pourrait être utilisé. La transformation PSM à PSM s'effectue lors des phases de déploiement, d'optimisation ou de reconfiguration **[91]**.

PSM vers PIM : Ces transformations sont des opérations de rétro-ingénieries qui s'effectuent dans le but de reconstruire des modèles indépendants des plateformes techniques (PIM) à partir de modèles spécifiques aux plateformes (PSM) techniques.

Génération de code : Dans la pratique, certains travaux font la distinction entre les PSM exécutables (ou code) et les PSM non exécutables, mais la génération de code n'est pas toujours considérée comme une transformation de modèles. La figure 3.5 montre quand même qu'il est possible de passer d'un PSM non exécutable à du code et inversement. Il est important de souligner que le passage du code au PSM est une opération de rétro-ingénierie qui est assez complexe à réaliser.

En somme, il est possible de classer les transformations de modèles possibles dans le MDA dans quatre catégories comme l'indique la figure 3.5.

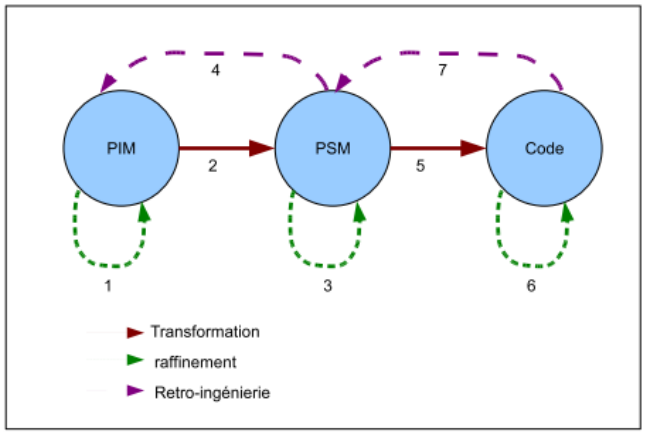


Figure 3.5: Transformations de modèles MDA

- *les transformations (2)* : décrivent le processus de conversion d'un PIM en un PSM;
- *les raffinements (1), (3) et (6)* : introduisent ou suppriment des informations dans un modèle;
- les retro-ingénieries (4) et (7) : convertissent un modèle vers un niveau d'abstraction plus élevé;
- *la génération de code(5)*: transforme un PSM non exécutable en un code exécutable. Nous avons déjà mentionné que la génération de code n'est pas toujours considérée comme une transformation de modèles dans la pratique.

Quel que soit son type (PIM vers PSM, PIM vers PIM, etc.), une transformation de modèles s'apparente toujours à une fonction qui prend en entrée un ensemble de modèles et qui fournit en sortie un ensemble de modèles **[91]**. Selon les différents types de modèle (en entrée ou en sortie), plusieurs types de transformation existent, la section suivante présente ces déférents types de transformation.

3.7.2 Taxonomie des transformations de modèles

Partant de la nature des métamodèles source et cible, on distingue encore selon une classification les transformations dites endogènes et exogènes combinées à des transformations dites verticales et horizontales **[104]**.

Transformations endogènes: Une transformation est dite endogène si les modèles cible et source sont issus du même métamodèle (Figure 3.6). Une transformation endogène permet :

- L'optimisation : la transformation a pour but d'améliorer par exemple la qualité d'exécution (en termes de performance) du modèle, tout en préservant la sémantique du modèle ;
- La restructuration : la transformation entraine un changement de la structure interne du modèle afin d'améliorer la qualité de certaines caractéristiques comme la modularité et la réutilisation, sans modifier le comportement du modèle ;
- La simplification : la transformation va permettre de simplifier la complexité de la syntaxe du modèle.

Transformations exogènes : Une transformation est dite exogène si les modèles cible et source possèdent des métamodèles différents (Figure 3.6).Une transformation exogène permet:

- La migration : la transformation d'un modèle écrit dans un certain langage en un modèle écrit dans un autre langage, tout en gardant le même niveau d'abstraction.
- La synthèse : la transformation d'un modèle de haut niveau, très abstrait (spécification, modèle fonctionnel) vers un modèle de bas niveau plus concret (code généré, exécutable).
- La rétro-ingénierie : il s'agit d'une transformation de synthèse inverse qui permet d'extraire des spécifications de haut niveau d'un modèle de bas niveau.

Transformation verticale Une transformation est dite verticale si elle met en jeu différents niveaux d'abstraction dans la transformation. Le passage de PIM vers PSM est une transformation exogène et verticale alors que le raffinement est une transformation endogène et verticale.

Transformation horizontale Une transformation est dite horizontale lorsque les modèles source et cible impliqués dans la transformation sont au même niveau d'abstraction. La restructuration, la normalisation et l'intégration des patrons sont des exemples de

transformation endogène et horizontale ; tandis que la migration des plates-formes et la fusion de modèles sont des exemples de transformation exogène et horizontale.

La figure 3.6 ci- dessous résume les combinaisons possibles entre transformations de modèles.

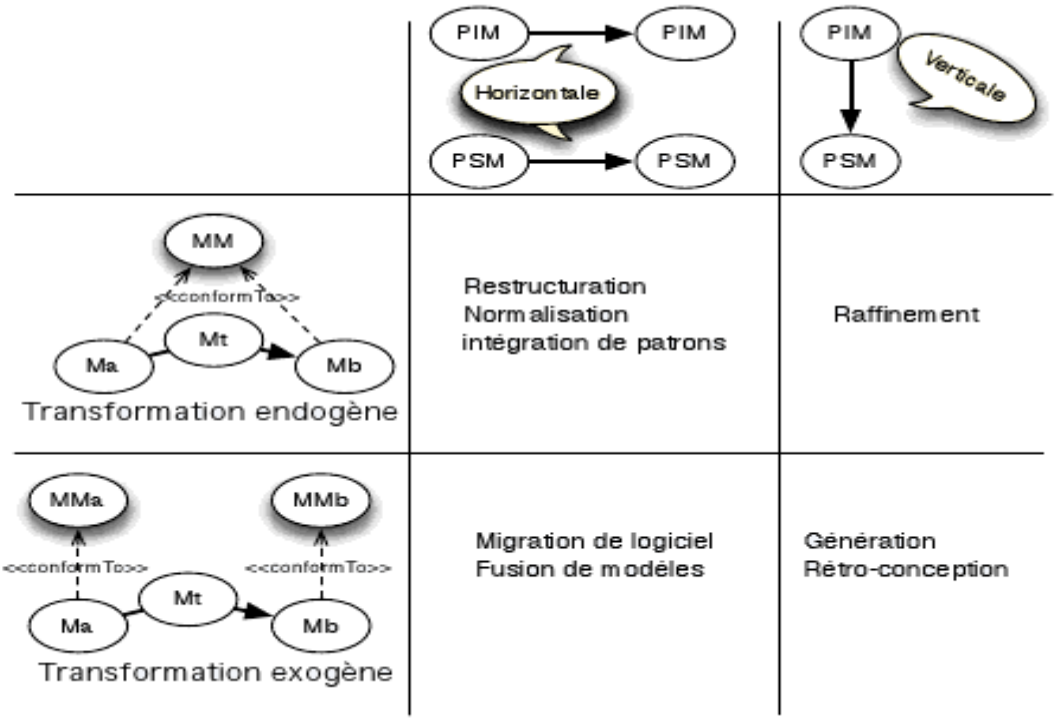


Figure 3.6: Taxonomie des transformations de modèles [104]

3.7.3 Spécification des règles de transformation :

Ce qui différencie les différentes approches permettant l'élaboration des transformations de modèles est la façon dont sont spécifiées les règles de transformations. Dans **[91]** trois approches de transformations sont retenues: l'approche par programmation, l'approche par template et l'approche par modélisation.

L'approche par programmation consiste à utiliser les langages de programmation en général, et plus particulièrement les langages orientés objet. Dans cette approche, la transformation est décrite sous forme d'un programme informatique à l'image de n'importe quelle application informatique. Cette approche reste très utilisée car elle réutilise l'expérience accumulée et l'outillage des langages existants.

L'approche par template consiste à définir des canevas des modèles cibles souhaités. Ces canevas sont des modèles cibles paramétrés ou des modèles template. L'exécution d'une transformation consiste à prendre un modèle template et à remplacer ses paramètres par les

valeurs d'un modèle source. Cette approche par template est implémentée par exemple dans Softeam MDA Modeler[10].

L'approche par modélisation consiste quant à elle à appliquer les concepts de l'ingénierie des modèles aux transformations des modèles elles-mêmes. L'objectif est de modéliser les transformations de modèles et de rendre les modèles de transformation pérennes et productifs, en exprimant leur indépendance vis-à-vis des plates-formes d'exécution. Le standard MOF 2.0 QVT de l'OMG a été élaboré dans ce cadre et a pour but de définir un métamodèle permettant l'élaboration des modèles de transformation de modèles. A titre d'exemple, cette approche a été choisie par le groupe ATLAS à travers son langage de transformation de modèles ATL **[105]**.

3.8 Rapide panorama de l'ingénierie des modèles

Il existe actuellement de multiples langages dans le paysage de l'ingénierie des modèles. Une taxonomie de ces langages peut être organisée autour de trois axes principaux:

- **les langages de métamodélisation :** Ces langages permettent de décrire des modèles, avec par exemple Ecore d'IBM **[96]**, EMOF de l'OMG **[121]**, ou les schémas XML du W3C **[106]**;
- **les langages de contrainte et de requête** avec par exemple OCL issu de l'OMG **[109]** ou XQUERY issu du W3C **[110]**;
- l**es langages de transformation :** On peut grossièrement distinguer quatre catégories d'outils :
  1. Les outils de transformation génériques : Dans cette première catégorie on trouve notamment d'une part les outils de la famille XML **[105]**, comme XSLT **[108]** ou Xquery **[110]** et d'autre part les outils de transformation de graphes (la plupart du temps issus du monde académique) **[107]**. Les outils de cette première catégorie ont l'avantage d'être déjà largement utilisés, ce qui leur a permis d'atteindre un certain niveau de maturité. En revanche, l'expérience montre que ce type de langage est assez mal adapté à des transformations de modèles complexes, car ils ne permettent pas de travailler au niveau de la sémantique des modèles manipulés, pour remédier à ce problème ils sont couplés avec des langages de programmation standards tel que Java ou C#.
  2. Des langages de scripts intégrés à la plupart des ateliers de génie logiciel :

[10] www.objecteering.fr/products_mda_modeler.php

Dans cette seconde catégorie figure une famille d'outils de transformation de modèles proposés par des éditeurs d'ateliers de génie logiciel. Par exemple, l'outil Arcstyler **[112]** de Interactive Objects , l'outil Objecteering **[113]** de Objecteering Software, qui propose OptimalJ **[114]** de Compuware comme un langage de script pour la transformation de modèles ,et bien d'autres encore, y compris dans le monde de l'open source avec des outils comme Fujaba **[115]**.L'intérêt de cette catégorie d'outils de transformation de modèles est d'une part leur relative maturité et d'autre part leur excellente intégration dans l'atelier de génie logiciel qui les héberge. Leur principal inconvénient est que la plupart du temps les langages sont propriétaires sur lesquels il peut être risqué de miser sur le long terme. Ces langages montrent à nouveau leurs limites lorsque les transformations de modèles deviennent complexes et qu'il faut les gérer, les faire évoluer et les maintenir sur de longues périodes.

3. Les outils conçus spécifiquement pour faire de la transformation de modèles: Cette catégorie regroupe les outils prévus pour être plus ou moins intégrable dans les environnements de développement standards, on va trouver par exemple MiaTransformation **[116]** de MiaSoftware ou PathMATE **[117]** de Pathfinder Solutions. On trouvera aussi dans le monde académique de nombreux projets open source s'inscrivant dans cette approche : les outils ATL **[105]** et MTL **[118]** de l'INRIA, AndroMDA **[119]** ou encore QVTEclipse **[120]**.Cette catégorie d'outils est bien adaptée pour faire des transformations de modèle simples mais elle trouve ses limites lorsque les transformations de modèles deviennent complexes surtout pour l'évolution et la maintenance sur de longues périodes.

4. Les outils de métamodélisation au sens propre : La dernière catégorie des outils de transformation de modèles est celle des outils de métamodélisation dans lesquels la transformation de modèles revient à l'exécution d'un métaprogramme. Parmi ces outils, nous pouvons citer Kermeta **[92]** de NRIA, MetaEdit+ **[122]** de MetaCase, EMF/Ecore **[121]** de la fondation Eclipse, et l'outil TOPCASED **[123].**

3.9 Conclusion

Dans ce chapitre, nous avons présenté sommairement le principe et les concepts essentiels et stables de l'IDM. Cette présentation n'est ni exhaustive, ni normative. C'est un tour d'horizon sur les concepts fondamentaux et une vision de notre idée de la démarche MDA dans le processus de construction d'un système.

Nous avons vu que l'approche MDA conçoit l'intégralité du cycle de vie comme un processus de production, de raffinement itératif et d'intégration de modèles. L'utilisation des modèles permet de capitaliser les connaissances et le savoir-faire à différents niveaux d'abstraction qu'elles soient des connaissances du domaine métier ou du domaine technique. Elle couvre ainsi les différentes vues du système. Le processus de développement des systèmes peut alors être vu comme un ensemble de transformations de modèles partiellement ordonné, chaque transformation prenant des modèles en entrée et produisant des modèles en sortie, jusqu'à obtention d'artefacts exécutables.

Ainsi, cette technique d'ingénierie semble être la plus adaptées pour répondre aux problématiques soulevées lors du chapitre précédant dans le domaine de la spécification de l'architecture logicielle. Dans le chapitre suivant, nous allons exposer notre approche basée sur l'approche MDA.

CHAPITRE 4
VERS UNE SPECIFICATION FLEXIBLE D'ARCHITECTURE LOGCIELLE SELON L'APPROCHE IASA

4.1 Introduction

Selon l'étude des langages de spécification d'architecture logicielle que nous avons menée dans le chapitre 2, nous avons identifié certaines lacunes. Les points faibles rencontrés dans les ADLs sont de plusieurs ordres. La majorité des ADLs proposent une notation spécifique, ce qui nécessite un effort de plus pour l'architecte non seulement pour comprendre le langage de spécification mais même pour pouvoir spécifier l'architecture de son système à concevoir, d'une manière qui se rapproche plus à son modèle mental. Cette liberté de spécification n'est pas bien supportée par les ADLs. En effet, ces ADLs permettent difficilement de prendre en compte le modèle mental de l'architecte élaboré dans les premières phases d'un processus de développement de logiciel. Ainsi, il n'est pas possible par exemple d'effectuer des opérations d'interconnexion entre les composants à l'improviste qui peuvent survenir surtout durant les premières phases d'un processus de développement de logiciel. Les ADLs, introduisent eux-mêmes des suppositions spécifiques sur les architectures. Par exemple, certains ADLs sont spécifiques à un domaine et à un style particulier, donc ces suppositions peuvent être incohérentes ou conflictuelles avec celles sous-jacentes au système que l'architecte veut concevoir.

C'est pour tenter de remédier à ces lacunes, nous proposons dans ce travail d'appliquer une solution de l'architecture logicielle dans le cadre de l'approche IASA (Integrated Approach for Software Architecture), en fournissant un riche, extensible et souple ADL nommé X3ADL. Dans ce chapitre, nous présentons les concepts et les techniques développés pour l'approche IASA afin de supporter la spécification directe des modèles mentaux de l'architecte, selon un modèle à un haut niveau d'abstraction. Ensuite il serait question de déterminer comment une spécification architecturale sera transformée en un logiciel exécutable. A ce niveau un processus de transformation est développé, en spécifiant un ensemble de règles de transformation permettant de produire des composants logiciels réalisés dans des langages cibles tel que Java.

4.2 La flexibilité dans le cadre de la spécification architecturale

Pour la résolution d'un même problème la spécification architecturale peut être très différente d'un architecte à un autre. Les différences résident principalement dans l'approche d'élaboration (du global vers le détail ou du détail vers le global), le choix des composants et les techniques utilisées pour spécifier les diverses interconnexions entre composants, leurs sémantiques peuvent avoir plusieurs déclinaisons. Par exemple, un composant peut être un simple opérateur arithmétique ou une application complexe. Il peut avoir une ou plusieurs interfaces à travers lesquelles il peut percevoir son environnement ou communiquer avec d'autres composants. La sémantique de tous ces concepts qui vont être utilisés dépend largement du domaine d'application. Ainsi, la spécification d'un système particulier nécessite une certaine flexibilité où l'architecte peut définir la sémantique qui convient le plus à son domaine d'application.

La flexibilité d'un cadre de spécification architecturale peut être obtenue grâce à l'utilisation des composants avec un très haut niveau d'abstraction, en se détachants des détails techniques des plateformes d'exécutions. En effet, tout domaine d'application (télécom, commerce électronique, etc.) possède des concepts spécifiques. Ces concepts ainsi que leurs relations sont décrits au niveau plus concret du domaine. Celui-ci est utilisé pour préciser ce qu'est un composant logiciel pour un domaine particulier. Etant donné qu'il n'y a pas de définition unique des composants logiciels, ce point nous semble important. De plus, nous avons noté qu'au niveau des systèmes informatiques, une multitude de composants peuvent être exprimés pour des utilisations diverses. Il est important, dans ce contexte, d'accorder la liberté à l'architecte de définir le composant qui convient le plus à ses besoins.

Toutefois les composants interagissent avec leur environnement via des interfaces, et avec toutes les avancées actuelles dans différents domaines, y compris les systèmes embarqués, dans un seul système, on peut avoir un certain nombre de différents types d'interfaces utilisés (ce qui est souvent le cas). Capturer de telles architectures implique une abstraction un certain nombre de types d'interface, sans forcer les architectes à utiliser des styles spécifiques ou types d'interfaces tout au long de leur architecture, en offrant un mécanisme souple qui permet à l'architecte de définir ses propres types d'interface.

Par ailleurs, nous avons vu que la complexité des systèmes leur décomposition en plusieurs vues. Chaque vue peut être définie dans une partie propre. Par la suite, les vues vont être intégrées pour obtenir une architecture complète d'un système particulier. De par cette

manière de définir une architecture, il est possible de disposer d'un support complet, adapté à une application particulière et respectant la séparation des préoccupations. Notons que la séparation des préoccupations permettra de capturer les propriétés de chaque vue de manière plus aisée.

Le premier objectif de ce travail consiste à offrir les moyens aux architectes de définir les composants représentant les vues de leur système pour exprimer les propriétés fonctionnelles et non fonctionnelles de ce système. Au niveau de ces concepts, nous pensons qu'il est important d'utiliser des notations graphiques pour supporter plus aisément les premières idées de l'architecte. En effet, le modèle mental est par essence graphique, c'est ce qu'en témoignent les premières activités utilisant le box and line. Ensuite, il revient à bien définir les contraintes structurelles et comportementales relatives aux entités contenues dans chaque vue à travers un ADL spécifiques à l'architecture qui sera généré à partir de ces vues qui sont intégrés. Les contraintes sont souvent des précisions que l'architecte devra donner à ses actions de spécification d'architecture. Ces informations complémentaires mais minimales qui permettraient de rendre la spécification formelle .C'est en fait la première transformation à réaliser, il s'agit donc de spécifier comment passer d'une spécification très proche de l'informelle à une spécification formelle représenté par X3ADL.

Le deuxième objectif de ce travail consiste à déterminer comment une spécification architecturale sera transformée en un logiciel exécutable. En effet un ADL décrit une spécification de haut niveau d'abstraction qui est par la suite raffinée jusqu'à ce que les composants utilisés soient des composants dits primitifs, ayant au moins une réalisation bien précise dans une technologie d'implémentation particulière.

Pour résoudre cette problématique, il est nécessaire d'adopter un cycle de développement permettant de passer d'une spécification de haut niveau d'abstraction au code exécutable en explicitant les règles de passage d'une phase à une autre. Ceci peut correspondre par exemple, à la définition de règles permettant de générer les modèles à partir du métamodèle ou spécifier des règles permettant le passage des modèles vers le code, etc. Ces règles sont en général appelées règles de transformation et règles de raffinement **[124]**.

Nous proposons dans ce travail d'appliquer une solution de l'architecture logicielle dans le cadre de l'approche IASA (Integrated Approach for Software Architecture). Dans la section qui suit, nous présentons les concepts et les techniques développés pour l'approche IASA afin de supporter plus aisément la spécification directe des modèles mentaux de

l'architecte. Un riche, extensible et souple ADL nommé X3ADL que nous avons développé, permet de spécifier d'une manière plus formelle ces concepts architecturaux de IASA. Ensuite il serait question de déterminer comment une spécification architecturale en X3ADL, sera transformée en un logiciel exécutable. A ce niveau un processus de transformation est développé, en spécifiant un ensemble de règles de transformation permettant de produire des composants logiciels réalisés dans des langages d'implémentation tel que Java.

4.3 Spécification graphique

Une spécification d'architecture logicielle représente les premières idées que se fait un architecte de son système. Cette spécification possède souvent une forme graphique dans laquelle nous retrouvons les rectangles et les liaisons. Parfois au lieu d'utiliser un rectangle pour représenter un composant, l'architecte utilise des figures spécifiques telles qu'un cylindre pour représenter une base de données. Les figures spécifiques renseignent dés leur vue sur la fonctionnalité globale du composant associé. Cette spécification graphique représente souvent une vue très abstraite du système. Elle correspond fortement à ce que nous avions appelé le modèle mental de l'architecte (Figure 4.1). Le modèle mental est par essence graphique. C'est ce qu'en témoignent les premiers comportements de l'architecte. Le document reportant le modèle mental représente un document sur lequel peut aussi intervenir le client pour spécifier de manière encore plus précise ses exigences ou indiquer des choix stratégiques.

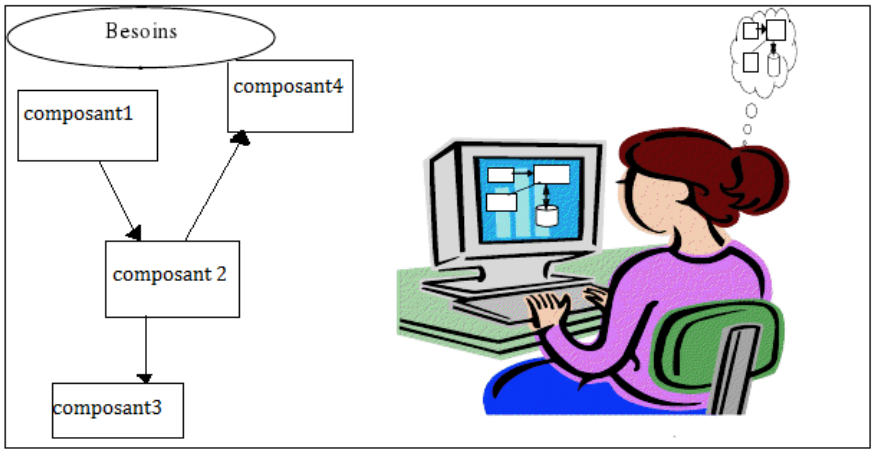


Figure 4.1: Modèle mental de l'architecture d'un logiciel.

Le support direct du modèle mental d'un architecte permet de réduire à l'extrême le gap sémantique séparant le domaine du problème du domaine de la solution dans le processus d'élaboration d'une solution informatique à un problème. La réduction de ce gap sémantique

entre l'espace problème et l'espace solution aura un impact décisif et direct sur les facteurs déterminant de la réussite d'un projet logiciel, tel que l'augmentation de la qualité et de la sûreté du produit logiciel, la réduction des coûts, la réduction des délais de réalisation et la réduction du taux d'erreur **[55]**.

4.4 Vers une approche qui débute par une spécification graphique

Dans notre approche de spécification d'architecture logicielle, nous pensons qu'il est important de rechercher une notation graphique capable de supporter directement ou via une légère transformation le modèle mental de l'architecte, et plus exactement les comportements de l'architecte dans les premières phases de la spécification d'une architecture logicielle. La représentation de l'architecture en termes de «boîtes et flèches» et de figures spécifiques, est le moyen le plus simple pour décrire la structure d'un système. En effet, elle permet d'identifier facilement les éléments importants du système (i.e., les boîtes et figures spécifiques) ainsi que leurs relations (i.e., les flèches) (Figure 4.1).

Une spécification d'architecture logicielle se base sur les concepts fondamentaux suivants : Le concept de composant, les interfaces d'interaction du composant et le concept de connecteurs. Le modèle de composant à déterminer et le modèle de connexion doivent ainsi être assez flexibles pour supporter la spécification directe du modèle mental. Une architecture issue du modèle mental peut être plus abstraite que le PIM (Platform Independent Model) du MDA. Le modèle mental d'architecture est non seulement indépendant de la plateforme, mais peut être indépendant des mécanismes logiciels, notamment ceux dédiés aux interactions.

Notre approche devra donc débuter par la mise en place d'un système capable de capturer la vue graphique du modèle mental. Ce système ne peut être qu'un éditeur graphique. La spécification graphique devra en outre supporter les divers comportements de l'architecte lors de l'élaboration d'une architecture. Elle devra simplifier la spécification et améliorer la lisibilité et de la compréhension du système à concevoir. La spécification graphique devra être, à un moment ou à un autre, vérifiée puis transformée en un programme exécutable. Ainsi le système qui se chargera de la capture du modèle mental n'est pas simplement un éditeur graphique mais un environnement intégré de développement (IDE) doté de capacités de capture de spécification, de vérification de l'architecture, et de transformation d'une architecture en un code exécutable (Figure 4.2).

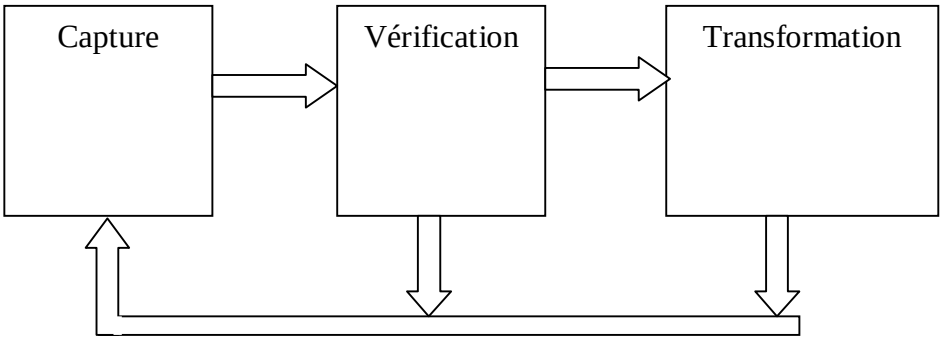


Figure 4.2 : Processus de spécification de l'architecture dans l'IDE

4.5 Les comportements de l'architecte durant la spécification d'une architecture

Dans le processus de recherche des modèles efficaces (composants, interfaces, connecteurs) qui peuvent supporter le modèle mental, et par la suite permettre de le transformer manuellement ou en assisté, il est nécessaire d'essayer de comprendre ce que fait réellement un architecte lors de la spécification informelle de son modèle mental et aussi durant les phases ou une solution commence à devenir plus formelle. Si nous comprenons ces comportements, nous pourrons trouver les modèles qui pourraient accueillir efficacement les éléments de la spécification informelle et les diverses décisions architecturales, et delà nous pourrons assurer une formalisation du modèle mental de l'architecte, même avec un minimum de contrainte que devrait prendre en considération l'architecte. Cette étude aura un impact décisif sur l'IDE qui servira à la capture d'une architecture. A travers cette étude nous essaierons de mettre en évidence les besoins en fonctionnalité d'un architecte lors de la spécification d'une architecture.

4.5.1 L'architecture

L'architecture est en réalité un art qui dépend fondamentalement du raisonnement de l'architecte **[124]**. Ainsi, il est très possible que pour une solution assez simple, deux architectes peuvent avoir des raisonnements totalement différents dans la spécification de l'architecture. Les différences résident principalement dans l'approche d'élaboration (du global vers le détail ou du détail vers le global), le choix des éléments de base (composants) et les techniques utilisées pour spécifier les diverses interconnexions entre ces éléments, leurs sémantiques peuvent avoir plusieurs déclinaisons. L'architecture étant un art, l'IDE doit être capable de supporter toutes les décisions architecturales de l'architecte qui découle de sa vue

de la solution. L'IDE doit permettre un haut degré de liberté dans les opérations de spécification d'une architecture.

Conclusion : L'IDE doit permettre une spécification libre d'architecture dans laquelle il n'est pas nécessaire de retrouver la rigueur imposée par les autres approches d'architecture logicielle qui impose par exemple dés le départ qu'il faut définir les interfaces avant de les lier et qu'il ne faut lier que des interfaces compatibles.

4.5.2 L'architecte

Il est à noter ici que l'architecte est une personne qui maitrise son domaine. Il peut réaliser des raisonnements très abstraits, comme il peut faire usage dans son raisonnement d'élément technologique particulier de son domaine. Il est capable de fixer la structure de son application en choisissant à partir d'un stock de composants certifiés les composants les plus adéquats pour son architecture puis les interconnecter. C'est ce qu'on appellera par la suite par une décision architecturale. Dans le cas où aucun composant du stock ne répond à ses attentes et ne peut supporter sa décision architecturale, l'architecte est capable d'initier la conception d'un composant spécifique et lui fixer les fonctionnalités et propriétés qui doivent le caractériser. La conception du composant selon une approche orientée composant faisant intervenir l'art de définir une architecture est du ressort de l'architecte.

L'architecte peut à une certaine étape du processus de conception arriver à une conclusion que pour diverses raisons, le composant doit être réalisé selon une approche non orientée composant et soumis à des experts dans l'approche désirée (approche objet par exemple). Ce comportement, nous le retrouvons aussi en architecture des ordinateurs, ou à un moment donnée, la réalisation d'un système requiert un composant logique qui lui doit être réalisé en technologie plus fine utilisant des circuits analogiques tels que transistor, diode etc.

Conclusion :

- L'architecte doit disposer d'un stock de composants certifiés, ou composants sur étagère.
- L'architecte utilise une approche orientée composant.
- Le système es défini comme étant un assemblage de composants.
- Un composant peut être réalisé dans une approche qui n'est pas l'approche à composant.

4.5.3 L'esquisse

Lors de la première spécification, l'architecte élabore une esquisse qui n'est pas

vraiment précise. Cette esquisse sera ensuite précisée à l'aide de commentaire que l'architecte portera au niveau des composants et des liaisons (Figure 4.3). Les esquisses sont proposées pour répondre aux exigences fonctionnelles et non fonctionnelles du client. L'architecte avant d'entamer une prochaine étape du processus de développement réalise manuellement ou à l'aide d'outils, une évaluation de ses esquisses pour s'assurer qu'elles répondent aux exigences du client.

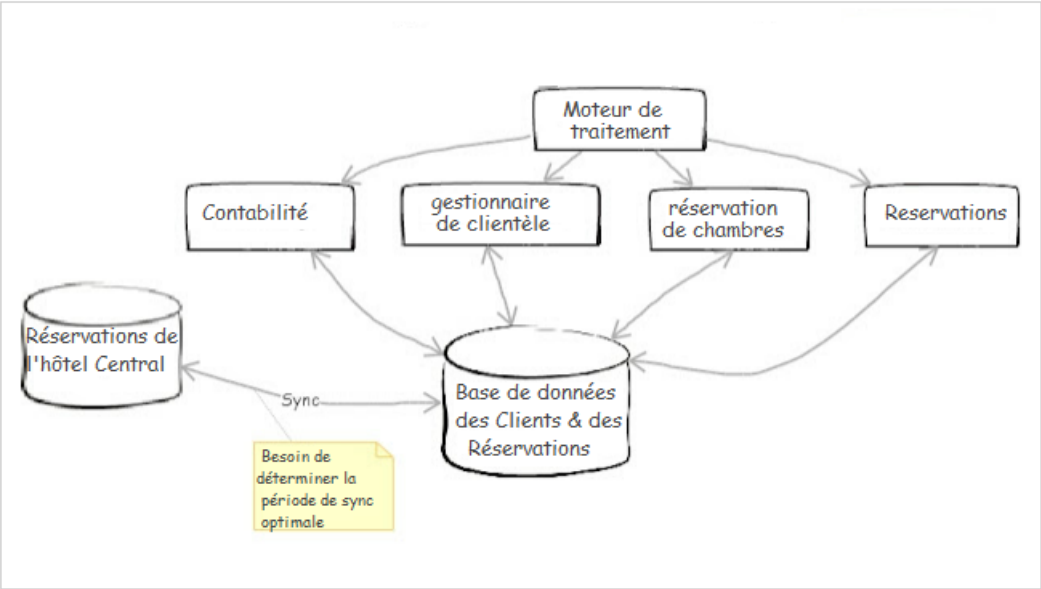


Figure 4.3 : Exemple d'élaboration des esquisses

Conclusion :

- Les spécifications doivent se faire dans le contexte d'un projet.
- Les exigences fonctionnelles et non fonctionnelles sont indiquées dans le contexte d'un projet.
- Un projet peut comporter plusieurs esquisses, chacune tente de répondre aux objectifs fonctionnels et non fonctionnels.
- Une esquisse peut être incomplète.
- Une esquisse est précisée de manière progressive.
- Une esquisse doit être évaluée pour permettre de vérifier si les exigences fonctionnelles et non fonctionnelles ont été atteintes.

Remarque : La précision d'une architecture n'est pas censée être réalisée par une et une seule personne. Il est possible d'imaginer que dans le contexte d'un projet, un architecte réalise une première ébauche et un autre la précise.

Contraintes :

- L'architecte doit obligatoirement définir un projet auquel il doit donner un nom.

- Un projet doit obligatoirement atteindre un état pour être transféré à un autre architecte ou autres membres de l'équipe pour son raffinement. A titre d'exemple, un projet transférable est un projet qu'il est possible d'évaluer.

4.5.4 Les éléments de base

Quels sont les éléments fondamentaux utilisés pour l'esquisse :

- Des boites ayant parfois des formes un peu différentes et représentant des composants.
- Des figures spécifiques représentant des composants.
- Des lignes qui relient les boites et figures.
- Des lignes qui partent de lignes existantes et aboutissent à des composants.
- Des lignes reliant d'autres lignes.
- Des lignes représentant en fait un groupe de lignes. C'est ce qu'on appelle souvent un Bus.

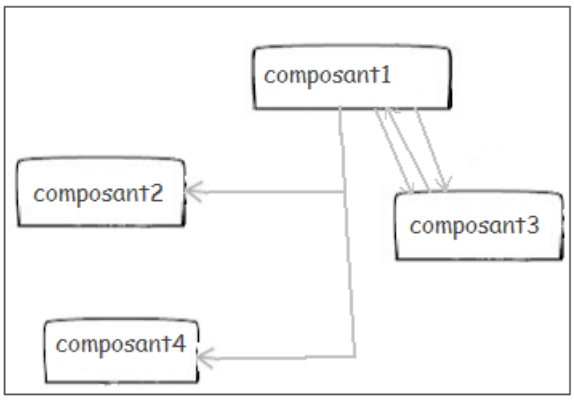


Figure 4.4 : Les éléments de base pour la spécification d'architecture

Conclusion :

- L'IDE doit mettre à la disposition de l'architecte un ensemble de figures de base, pour que l'architecte puisse sélectionner la figure qui lui semble la plus appropriée pour représenter un composant de l'architecture.
- L'IDE doit permettre le tracé de liaison entre les figures, entre les lignes et figures et entre les lignes elles-mêmes.

Contrainte : Une fois placée dans la zone de dessin, une figure ou une connexion doit être dotée d'un nom. Ce nom sera interprété comme un nom de type (de composant ou de connecteur). Un nom d'instance est attribué automatiquement et pourrait être changé par l'architecte. Lorsque la connexion est un bus, le nom d'instance de chaque fil est le nom d'instance du bus, suivi d'un numéro d'ordre (exemple bus01 est le nom d'instance, bus01-02

est le nom d'une connexion d'un bus).

4.5.5 Le tracé de connexion inter-composant

Lors de la réalisation des connexions entre composants, l'architecte opère sans contrainte en liant un premier composant à un deuxième dans une connexion point à point. Cette liaison est tracée sans aucune contrainte.

Cependant, nous remarquons que toujours le tracé débute d'un composant et atteint un autre. A ce niveau cette liaison peut représenter 3 situations précises :

- Le premier composant a besoin d'une fonctionnalité du deuxième.
- Le deuxième composant a besoin d'une fonctionnalité du premier.
- Il y a un équilibre : le 1er a besoin du 2ème et le 2ème a besoin du 1er.

Conclusion : L'interconnexion doit être précisée directement ou indirectement

Contrainte :

- Précision directe : L'architecte doit indiquer le type de cette connexion.
- Client Serveur : Doit préciser qui est le client ou le serveur.
- Egal a Egal : Les deux sont client et serveurs.

4.5.5.1 Anticipation du type de connexion à partir du tracé lui-même

L'anticipation permet de désigner le client et le serveur en se basant sur le point de départ et le point d'arrivée d'une connexion. Par exemple, le composant par lequel débute la connexion peut être considéré come le client.

Ce type d'anticipation peut être un des paramètres de l'IDE à travers lequel il est possible de mettre l'IDE dans un état qui correspond au raisonnement de l'architecte. Nous pouvons dans un premier temps considérer une valeur par défaut (par exemple celui par lequel débute la connexion est un client). Cependant cette valeur peut être modifiée pour spécifier l'inverse et adapter l'IDE au raisonnement de l'architecte.

Cette anticipation n'est pas valable pour une connexion de type égal à égal et qui doit être directement précisée.

4.5.5.2 Anticipation du type de connexion à travers la description de l'interaction :

A un moment ou un autre l'architecte est mené à mettre des descriptions au niveau

des composants ou au niveau des liaisons. Cette description pourrait être un commentaire, une contrainte ou une fonctionnalité. La fonctionnalité décrite au niveau d'une liaison est une interaction entre les composants liés par cette connexion. Cette fonctionnalité doit pouvoir être écrite dans un langage assez simple, proche du langage naturel sans pour autant perdre de sa précision et mener à des ambigüités. Ce langage devra posséder une qualité d'adaptation facile au vocabulaire de l'architecte. Ce sera à partir de cette description d'interaction qu'il sera possible de déterminer le type de l'interaction (client-seveur, égal à égal) et le rôle de chaque intervenant dans l'interaction.

4.5.6 Plusieurs feuilles de schémas pour un même projet

Lors de la spécification d'une architecture complexe, l'architecte utilise soit une grande feuille, capable de reporter la totalité de l'architecture, soit de petite feuille, chacune traitant une partie de l'architecture générale. Dans le cas de l'utilisation de plusieurs pages, les mêmes liaisons portent le même nom.

Conclusion : L'IDE doit permettre la spécification dans une seule vue (feuille) ou dans des vues séparées. Plus encore, l'IDE pourra découper une feuille en petite feuilles selon les indications de frontière entre les feuilles. De ce fait, l'éditeur devra être capable d'assembler de petite feuilles en feuilles plus grandes et inversement.

4.5.7 Multiples Apparitions d'un même composant sur une même feuille ou des feuilles distinctes

Parfois l'architecte fait paraitre dans son architecture plusieurs composants portant le même nom de type. Comme indiqué précédemment ce type de comportement dans l'IDE sera interprété par le fait que ce sont des instances différentes du même type de composant. Un nom d'instance par défaut est associé automatiquement à chaque instance. Cependant dans certaines situations, et pour des raisons de clarté du schéma, l'architecte redessine le même composant à divers endroits d'une grande feuille ou sur des feuilles différentes.

Sur papier l'architecte devra user d'une technique pour dire qu'effectivement c'est le même composant. Parfois, il n'utilise aucune notation et reste seul capable de comprendre le schéma. Parfois, c'est la nature du composant qui tranche pour dire que c'est la même instance qui est citée dans plusieurs endroits.

Conclusion :

- Nécessité de fournir un mécanisme pour indiquer qu'il s'agit d'un même composant si

ce composant est cité à des endroits divers.

- Nécessité pour certains types de composant d'interpréter par défaut que c'est la même instance qui est répétée à divers endroits différents. Dans cette situation nous pouvons dire qu'il y a une instance de base et les autres instances sont en fait des raccourcis ou des liens.

4.5.8 Utilisation de standards:

Dans les première phases de l'élaboration d'une solution logicielle sous la forme d'une architecture, même au niveau des documents accessibles aux clients et sur lesquels il peut intervenir et fixer ses objectifs, il est fort possible que des standard en terme d'infrastructure de communication (FTP, HTTP, RMI, CORBA, etc.) ou de déploiement de composant (Composant EJB, service web)soient choisis comme faisant partie des décisions stratégiques dans le cours de l'étude et le développement d'un produit logiciel. Il est clair que ces standards puissent avoir un impact sur la zone ou ils sont mis en œuvre. A titre d'exemple, si nous relions un composant à un connecteur FTP, le composant doit obligatoirement disposer d'un client FTP), le port de communication du composant est un port FTP qui ne peut comprendre que le protocole FTP. Si l'architecte décide qu'un composant est un service Web, alors toute connexion à ce composant est une connexion standard (SOAP sur HTTP par exemple) et tout composant à ce service web est un client.

4.5.9 Composant abstrait et primitifs dans une spécification

Lors de l'élaboration de l'architecture d'un logiciel, souvent l'architecte use de composant abstrait dont il précise au fur et à mesure ses aspects externes, selon l'avancement du projet. Il peut aussi utiliser des composants qui ont déjà une existence et ne nécessitent pas d'étude ou de réalisation. Ils nécessitent seulement une exploitation correcte. Ces composants, quelque soit leur taille sont appelés des composants primitifs. Un serveur HTTP ou un serveur de base de données sont des composants primitifs. Un additionneur, ou plus exactement un opérateur dans les termes d'un langage de programmation est aussi un composant primitif.

Conclusion : L'esquisse peut donc être un mélange de composants abstraits, qu'il faut réaliser et de composant primitifs.

4.5.10 Raffinement : La conception orientée composant est un processus récursif

Une fois une esquisse réalisée et plus ou moins documentée, l'architecte se concentre sur la réalisation d'un des composants abstraits. Il procède soit par raffinement ou procède à

la conception du composant abstrait dans le contexte d'une autre approche, telle que l'approche objet. S'il procède par une autre approche, le composant sera alors un composant primitif une fois réalisé. S'il procède selon une approche à composant, il définira alors ce composant en utilisant des composant abstraits et primitifs. Le composant est ici un composant composite. Le processus est ainsi un processus orienté composant qui opère par raffinement successif. Le raffinement s'arrête au composant primitif. Nous pouvons dire aussi que ce processus est récursif.

Conclusion :

- L'IDE doit supporter cette forme de développement par raffinement.
- Il devra permettre des allers et des retours entre composants dans le processus de raffinement.
- Il devra en outre permettre de vérifier, et c'est un plus très important pour l'architecte, que les caractéristiques du composite cités lors de l'élaboration de sa vue externe, sont respectées lors de sa réalisation en utilisant d'autres composants ayant des propriétés certifiées (composants primitifs) ou définies (composants abstrait).

4.5.11 Style d'architecture:

Une des décisions stratégiques que peut prendre l'architecte dés le début de son projet concerne l'organisation générale de son architecture. A titre d'exemple il peut dire que son architecture doit avoir une allure d'un pipeline à plusieurs étages, chaque étage réalisant un traitement similaire ou particulier. Les données arrivent sur le 1er étage et le résultat est obtenu au niveau du dernier étage. C'est ce qu'on appelle souvent un style d'architecture. En plus de sa structure un style possède des règles qu'il faudrait respecter.

Conclusion : Une bibliothèque de style devra être un élément important d'un IDE pour l'architecture logicielle.

4.5.12 Nature des connexions : Donnée ou Flot de Contrôle

Lors de l'établissement d'une connexion, l'architecte se trouve en fait devant deux situations possibles : Soit il demande un transfert de donnée d'un composant à un autre, soit il demande la réalisation d'une opération. Ce type de connexion influera sur le type des interfaces. A titre d'exemple, dans une architecture de type pipeline, la liaison entre deux étages est une liaison orientée données. Dans une architecture de type composition par orchestration, la connexion entre le composant d'orchestration et les autres composants est une connexion orientée opération.

Conclusion : Deux types de connexions : Un type orienté donnée et type orientée service (ou transfert du flot de contrôle).

4.5.13 Synchronisation des transferts de flot ou des transferts de données

Parfois l'architecte spécifie un raisonnement orchestré. Un composant orchestrateur demande à un autre la réalisation d'une opération, attend la terminaison et répète ce processus avec le composant lui même et avec d'autres composants. De même pour les données, il fait un transfert et attend que ce transfert soit achevé correctement. Dans ce contexte une seule opération est réalisable par le demandeur et celui-ci reste toujours bloqué en attente d'une réponse. C'est ce qu'on appelle une connexion synchrone.

Dans d'autre situation, le demandeur d'un transfert de donnée ou de la réalisation d'un service lance l'opération et se détache de cette opération pour lancer une autre. Il peut ainsi réaliser des opérations en parallèles. De plus il ne dispose d'aucune information du moment où il aura une repose sur sa demande s'il attend effectivement une réponse. Nous disons que ce type de connexion est asynchrone.

4.5.14 Maintient des données au même état

Dans une connexion orientée données, l'architecte désire que l'état de la connexion soit toujours la même. Si l'information est modifiée au niveau d'un composant, elle doit l'être au niveau de tous les composants qui sont reliés à cette donnée. Deux cas de figurent peuvent corresponde à cette situation.

- La zone de données est dotées de capacité permettant le transfert automatique de l'information vers tous les composants connectés une fois un changement a lieu. chez un des composants, elle est automatiquement mise à jour au niveau du composant. Nous sommes ici en face d'opérations qui sont déclenchées suite à un événement.
- Le connecteur lui-même possède un état qui est diffusé à tous les composants qu'il relie.

4 .5.15 Les données au niveau des ports

Lors du raffinement d'une architecture (ou composant composite), l'architecte peut être en face d'une architecture qui semble être une architecture ordinaire au sens des ADLs. Dans cette architecture nous retrouvons des composants reliés entre eux à travers leurs interfaces. En architecture logicielle, les liaisons sont toujours établies entre interface comme le montre la figure 4.5. Cette figure semble refléter un raisonnement habituel. Les opérations

sont atomiques.

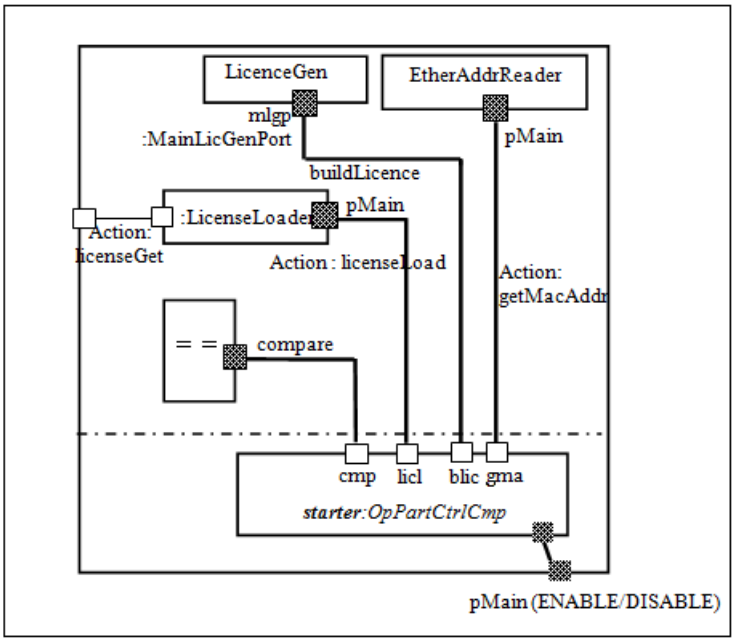


Figure 4.5 : Vue interne du composant *LicenseControllerCmp* utilisant une approche ordinaire d'interconnexion entre interfaces de composants

Le schéma de la figure 4.5 représente un composant qui contrôle la validité ou non de la licence d'exploitation d'un logiciel quelconque. La licence est construite à partir de l'adresse MAC d'une carte réseau Ethernet. Cette adresse est unique au niveau mondial[11] et ne peut pas être retrouvée dans une autre carte réseau Ethernet. Dans cette architecture interne du composant *LicenceControllerCmp*, le composant *main* joue le rôle d'orchestrateur de cette architecture. Il commence par demander au composant *EtherAddrReader* de lui récupérer l'adresse *MAC* de la carte réseau. Une fois récupérée le composant *main* l'envoie au composant *LicenceGen* et lui demande de construire un fichier de licence à partir de l'adresse MAC qui a été récupérée au préalable. Une fois construit ce fichier est retourné au composant *main* qui demande ensuite au composant *LicenceLoader* de récupérer le fichier licence de l'application. Le composant *main* envoie ensuite les deux fichiers de licence au composant de comparaison. Ce dernier réalisera la comparaison des deux fichiers et informera le composant *main* si les deux fichiers sont similaires ou non. Si les deux fichiers sont similaires le composant main renverra à travers son interface *pMain* la valeur constante ENABLE sinon il retournera la valeur constante DISABLE.

Pour la réalisation de la même fonctionnalité (vue interne du composant

[11] C'est l'IEEE qui est l'autorité d'attribution des adresses MAC aux cartes réseau et garantit l'unicité d'une adresse MAC attribuée

LicenceControllerCmp) l'architecte peut avoir le raisonnement de la figure 4.6. Dans ce fichier nous montrons en détail ce que contient chaque interface. Dans la figure 4.6, chaque interface est composée d'une seule opération (une méthode ou fonction). Le petit carré plein ou vide correspond au nom de la méthode et les cercles pleins ou vides correspondent aux paramètres de l'opération. Le raisonnement de la figure 4.6 sort en réalité des habitudes que nous retrouvons dans les ADLs ou les habitudes que nous avons lors de la réalisation de programme. Dans les approches d'architecture logicielles ordinaires, ce type d'interconnexion n'est pas supporté. Il y a un accès à un élément interne d'une opération sans passer par l'opération elle-même. C'est comme si on demandait l'accès à un paramètre d'une fonction sans faire l'appel à la fonction elle-même. Cependant le raisonnement architectural est un raisonnement correct. Ce raisonnement n'est plus centralisé mais plutôt collaboratif et parallèle. Dans ce raisonnement le composant *main* lance simultanément les autres composants. Ensuite, il y a échange entre composants sans intervention du composant *main*. C'est le composant *EtherAdrReader* fournit directement l'adresse MAC au composant *LicenceGen* qui produit le fichier et le soumet au composant de comparaison. Ce dernier fait la comparaison et son résultat est directement mis à la disposition des applications utilisant ce composant.

Conclusion : L'accès individuel aux éléments associés à une opération d'une interface doit être possible.

Remarque : Si nous permettons ce type de comportement dans une spécification d'architecture, il n'est pas judicieux d'appeler interface la partie d'un composant qui interagit avec le monde externe. Nous le retrouvons ce type de raisonnement en architecture des ordinateurs où l'architecte peu tirer un fil d'un port indépendamment des autres fils du port d'interconnexion qui peut être un port standard.

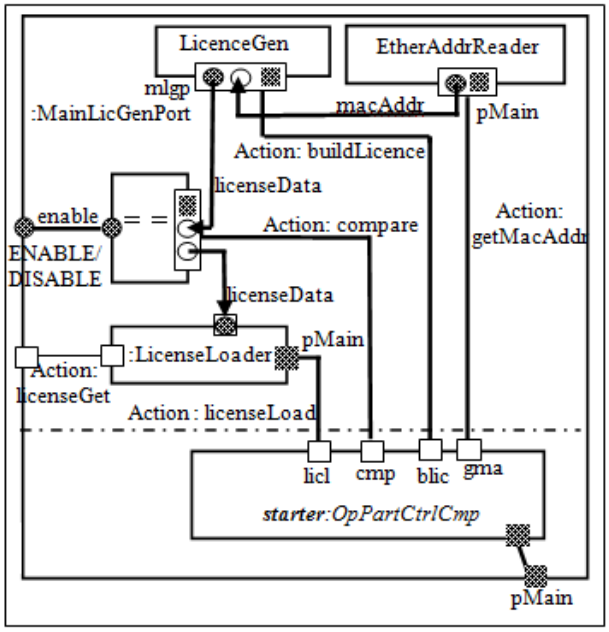


Figure 4.6 : Vue interne de *LicenseController* défini selon un raisonnement dans lequel il y a accès à un élément interne d'une opération

4.5.16 Vue structurée de données

De même que pour le besoin d'accès à un élément d'une interface, il est possible d'avoir la situation suivante dans le contexte d'une connexion orientée donnée. Dans cette dernière en principe ce sera une source de données qui est relié à un ou plusieurs consommateurs de cette donnée. La donnée échangée pourrait être simple, comme elle pourrait être vue comme une données structurée ou semi structurée. Dans ce contexte, il serait possible d'avoir un raisonnement dans lequel une partie de la donnée structurée ou semi structuré représente une source de données pour des consommateurs qui ne sont intéressés qu'à cette partie.

Conclusion : Il est nécessaire de mettre sur pied un mécanisme qui permet de voir une information sous une structure particulière (une forme de casting) pour pouvoir s'intéresser par la suite à une partie bien précise de l'information. A titre d'exemple nous pouvons voir une page HTML comme étant la donnée qui est fournie par une source. Dans ce contexte on pourrait imaginer que certain consommateurs ne s'intéressent qu'aux images de cette sources, d'autres consommateurs sont intéressé par le texte et a d'autres s'intéressent au code en langage JavaScript. Une donnée entière pourrait aussi être vue comme un groupage de bits. C'est le cas par exemple d'une adresse IP. Ainsi certain consommateur ne s'intéresseront qu'aux 3 premiers octets et un autre consommateur s'intéressera au 4ème octet.

4.6 Les notations graphiques de base pour la spécification du modèle mental

Dans cette section nous essayerons de définir les notations graphiques et les actions de manipulation de ces notations qui permettraient de supporter de manière directe et efficace le modèle mental d'un architecte. L'objectif de notre travail étant de rechercher une approche flexible de spécification d'architecture logicielle pour l'approche IASA, nous nous baserons essentiellement sur la notation graphique définie dans IASA. Nous essayerons de voir si la notation actuelle de IASA est assez puissante et suffisante pour être une première cible dans le processus de spécification et de transformation du modèle mental et aussi de supporter les diverses décisions architecturales tout au long du processus de raffinement d'une architecture.

4.6.1 Les éléments fondamentaux

A la base de l'étude précédente, nous voyons clairement que la spécification d'une architecture se base :

- Sur un ensemble d'éléments graphiques fondamentaux.
- Sur un degré de liberté important dans la manipulation de ces éléments, notamment en ce qui concerne l'établissement des connexions entre les diverses formes représentant les composants.

La notation graphique devra adresser les éléments suivant :

- Le composant qui sera représenté par une forme graphique précise, notamment les rectangles.
- Les ports d'interactions du composant. Nous les appelons port pour les distinguer du concept d'interface et pour indiquer aussi qu'un port possède une structure et qu'il est possible de manipuler individuellement les éléments du port, ce qui n'est pas possible avec les interfaces.
- Les connecteurs simples.
- Les bus.

4.6.2 La notation actuelle de IASA

La notation graphique utilisée par IASA est reportée dans la figure 4.7.

Dans IASA, la notation graphique pour les connecteurs n'est pas indiquée. En réalité IASA supporte le concept de connecteur simple et complexe. Le connecteur simple pourrait être représenté par un trait. Le connecteur complexe est en fait un composant de communication dans IASA. Il est représenté comme un composant ordinaire par un rectangle. Le concept de bus est aussi supporté mais aucune notation graphique n'a été définie.

Dans IASA, la notation graphique associée à un composant est un rectangle. IASA possède aussi une notation pour ce qui est appelé un composant partagé. Ce dernier est représenté par un rectangle en pointillé.

IASA supporte deux types de port : Les ports orientée flux de données et les ports orientés actions. Ces derniers sont des ports de services. Un port est un ensemble de point d'accès. Un point d'accès est accessible individuellement. Comme les ports les points d'accès sont soit de points d'accès orientés flux de données (DOAP pou Data Oriented Access point), soit des points d'accès représentant un service (ACTOAP pour Action Oriented Access Point). Les points d'accès ASPOAP sont des points d'accès utilisés lors de la spécification orienté aspect d'une architecture.

Dans IASA un service est un ensemble d'actions. Une action pourrait correspondre à une méthode d'un objet, une fonction ou une procédure. Au sein d'un même port une action peut être associée à un ou plusieurs points d'accès orientés données et plusieurs actions peuvent être associées aux mêmes points d'accès de données.

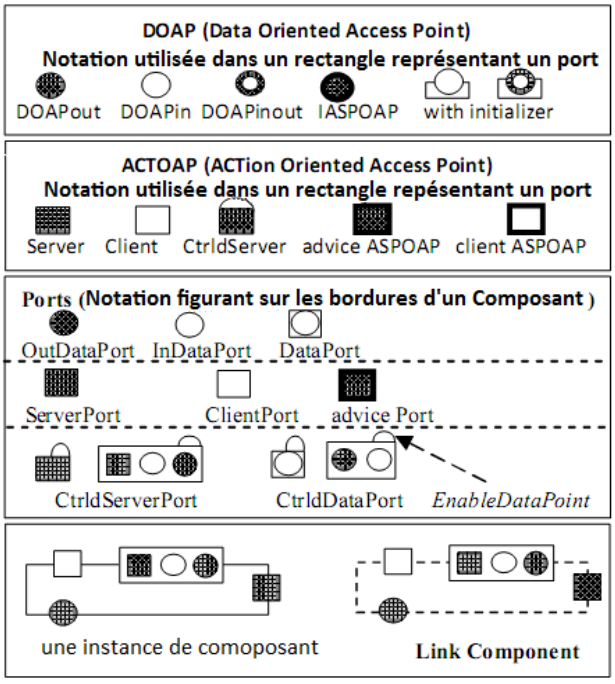


Figure 4.7 : Les principales notations graphiques de l'approche IASA [57]

4.6.3 Notation enrichie pour IASA

La notation IASA se prête bien pour la transformation d'une spécification informelle en une spécification basée sur la notation IASA. Cependant, dans certaines situations citées dans les comportements de l'architecte, cette notation n'est pas suffisante et nécessite un

enrichissement. L'enrichissement concerne les points suivants :

4.6.3.1 Les composants

Dans IASA la forme associée à un composant est unique et prédéfini. L'enrichissement de IASA sur cet aspect consiste à doter l'outil de spécification, ici l'IDE, d'une bibliothèque prédéfinie de formes chacune associée à une fonctionnalité bien précise. Cette bibliothèque doit être extensible. Ainsi parmi les fonctionnalités de l'IDE de IASA, nous devons trouver la gestion des formes associées au composant et même aux autres éléments fondamentaux pour la spécification d'architecture.

4.6.3.2 Connecteurs et bus

IASA utilise la même notation, un trait, pour les connecteurs. Il n y a pas de distinction explicite au niveau de la notation graphique entre un connecteur liant des ports et un connecteur liant des points d'accès. De plus il n y a pas de notation pour bus. Dans notre proposition un connecteur est représenté par un trait simple. Un connecteur ne peut lier que des points d'accès. Un bus est représenté par un trait épais. Un bus interconnecte des ports (Figure 4.8).

Parmi les pratiques à intégrer au niveau de l'IDE, la possibilité de tirer une connexion d'un BUS vers un point d'accès et la possibilité de tirer un BUS ou une partie du BUS vers un port.

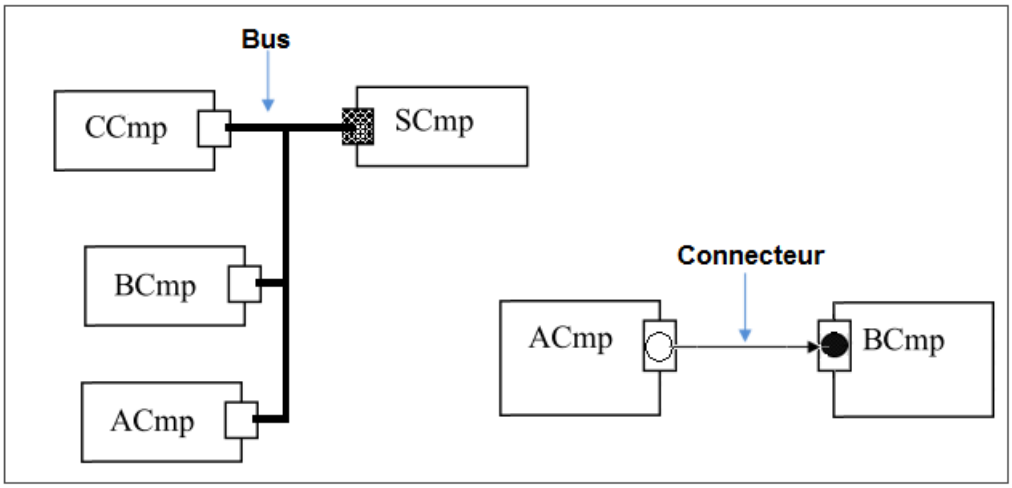


Figure 4.8 Conncetur et bus dans IASA

4.6.3.3 Vue détaillée d'un point d'accès

Un point d'accès de données (DOAP) véhicule une information qui pourrait être vue

sous une structure particulière. La structuration a souvent pour objectif de faire ressortir les parties d'une information pouvant être considérée indépendamment des autres parties. Ainsi un DOAP pourrait être vue comme étant un ensemble de DOAP, chacun orienté vers le support d'une partie de l'information. Dans l'IDE à mettre sur pied pour IASA, il est nécessaire d'offrir les mécanismes permettant de sélectionner une sorte de lunette avec laquelle l'architecte désire voir le détail d'un DOAP. C'est à l'aide de cette lunette que l'architecte pourrait mettre en évidences les DOAP interne d'un DOAP. Un DOAP quelque soit sa profondeur (utilisation de plusieurs lunettes imbriquées) est accessible individuellement.

4.6.3.4 Les points et ports contrôlés

Dans la version actuelle de IASA, un point d'accès contrôlé ou un port contrôlé possède une vue graphique dans laquelle le contrôle est représenté par un cadenas. Ce cadenas correspond en réalité à un point d'accès orienté donnée (DOAP) qui n'est pas visible dans la notation détaillée, car même dans celle-ci il est représenté par un cadenas. De plus ce DOAP visible sous forme d'un CADENAS correspond à un booléen.

Dans l'approche actuelle de IASA, lorsque le connecteur atteint un port, il doit obligatoirement atteindre le DOAP de contrôle, mais visuellement on ne connait pas exactement quel est le fil du bus qui atteint le DOAP de contrôle (le cadenas).

Pour permettre une clarté de la spécification, et permettre à l'approche de spécification de supporter un large éventail de décisions architecturales, donc une grande variété de raisonnements architecturaux, nous proposons une nouvelle manière de voir les ports contrôlés.

Nous maintenons le concept de cadenas associé à un port (ou à un point d'accès) lorsque le port n'est pas vu dans le détail. Lorsque le détail du port est demandé, nous proposons de montrer explicitement le mécanisme de contrôle, représenté par un DOAP dans la version actuelle. Le détail d'un point de données ou d'un port contrôlé montre explicitement le point d'accès utilisé comme point de contrôle. Dans la vue détaillé le cadenas disparait. Avec cette nouvelle vue, il est possible de lier explicitement le point de contrôle à une source de donnée si bien sur le point de contrôle est un DOAP. (Figure 4.9).

Nous proposons aussi d'étendre le mécanisme de contrôle pour atteindre deux objectifs :

- Le point de contrôle n'est plus figé à un DOAP de type Booléen. Le point de contrôle est un DOAP qui peut être associé à n'importe quelle information. Le contrôle sera effectue en se basant sur la valeur mise sur le DOAP. Avec cette facilité, un large éventail de choix est alors ouvert concernant le contrôle de port et de point d'accès.
- Le point de contrôle peut être un ACTOAP. De ce fait le mécanisme de contrôle pourrait être un mécanisme comportemental. Celui-ci pourrait être très simple (une action correspondant à un appel de procédure synchrone ou asynchrone) ou complexe tel qu'un protocole.

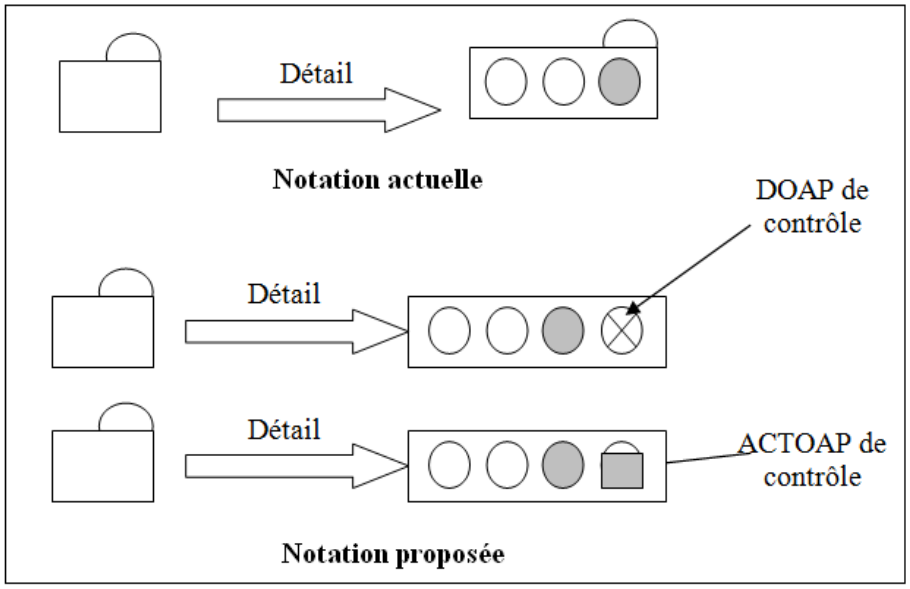


Figure 4.9: Vue détaillée du port contrôlé

4.6.3.5 Les connecteurs complexes

Dans IASA, un connecteur complexe est représenté par un composant de communication. Dans la spécification d'une architecture, ce composant possède la même structure qu'un composant ordinaire. Il est représenté par un rectangle au niveau duquel un certain nombre de ports sont placés. Dans une architecture, présenter un connecteur sous forme d'un composant ordinaire pourrait mener à une ambigüité lors de la lecture de l'architecture. Nous proposons de maintenir la vue composant mais d'introduire une autre notation sous forme d'un trait épais qu'il est nécessaire de distinguer d'un bus ordinaire.

Dans IASA, il y a en réalité un aspect qui caractérise les composants de communication. Cet aspect c'est le positionnement des ports clients et des ports de service. A titre d'exemple dans un composant de communication dédié à l'équilibrage de charges, le flanc gauche du rectangle correspond aux ports serveurs et le flanc droit correspond aux clients. Pour la vue sous forme d'un trait épais, les connexions ne peuvent être injectées ou

extraites à partir que des deux extrémités (Figure 4.10), ce qui n'est pas le cas avec les bus qui acceptent une prise de connexion de n'importe quelle position du tracé. Au niveau de l'IDE il sera ainsi possible de passer d'une vue sous forme de composant à une vue sous forme de trait et vis versa.

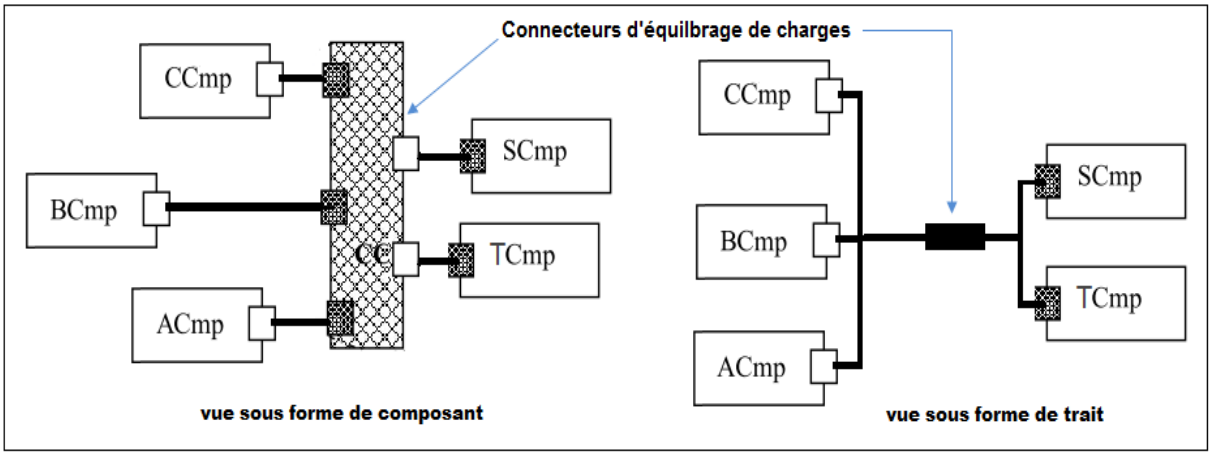


Figure 4.10: Exemple de connecteurs complexes

4.6.4 Nécessité d'une description textuelles accompagnant la notation graphique

La notation graphique nécessite souvent d'être associée et parfois surchargée sur le dessin avec des informations expliquant les calculs effectués par les diverses figures de base (boites et autre formes) et la nature des interactions représentées par les liaisons. Par exemple, une flèche peut aussi bien représenter un partage de données entre deux composants qu'un appel de procédure. En effet, pour être exploitable, une architecture doit clairement définir :

- quels traitements les "boîtes" et leurs annotations représentent,
- quelles relations de contrôle/flux de données sont indiquées par les "lignes" ou "flèches",
- comment le comportement global du système est déterminé par celui de ses composants,
- quelles sont les propriétés structurelles et comportementales qui doivent être conservées tout au long du développement.

4.6.5 Exemples de formalisation des quelques décisions Architecturale

Le vrai défit auquel fait face notre approche est le fait d'assurer un haut degré de liberté dans l'expression des décisions architecturales lors de la spécification d'une architecture. Le haut degré de liberté devrait permettre de capter efficacement les modèles mentaux de l'architecte pour ensuite pouvoir les formaliser automatiquement. Cette dernière

opération correspond en fait à une transformation automatisée ou assistée d'une spécification paraissant informelle en une spécification formelle. Cette automatisation porte essentiellement sur le contrôle des diverses décisions architecturales auxquelles il est nécessaire d'affecter un sens bien précis de manière implicite ou explicite de la part de l'architecte.

Dans ce qui suit nous présentons comment certaines décisions architecturales sont interprétées par l'IDE que nous envisageons de réaliser pour l'approche IASA.

4.6.5.1 Connexion inter composant sans présence de port

Souvent, dans les premières phases d'élaboration d'une solution, un architecte établit des connexions entre les frontières de deux composants représentés par des rectangles ou des formes particulières. Nous remarquons qu'en général, l'architecte ne commence pas par définir complètement les composants avant de les utiliser. Il opère plutôt par un raisonnement trop abstrait qu'il précise au fur et a mesure de l'avancement de son projet. Ainsi les composants qu'il utilise dans les premières phases du processus d'élaboration d'une solution ne comportent pas de ports.

Une fois la connexion tirée, l'architecte lui associe une description décrivant souvent un protocole d'échange entre les composants liés. Ce protocole pourrait être basique (un composant demande à un autre la réalisation d'une opération) ou complexe (un dialogue représenté par la demande de la réalisation d'une suite d'opérations selon les besoins). Cette description est souvent appelée une interaction. Selon la manière avec laquelle la connexion a été tracée et le contenu de l'interaction, l'IDE peut déterminer automatiquement les éléments nécessaires pour que la spécification devienne formelle. Dans notre cas, l'IDE dotera systématiquement les deux composants de port. Selon le sens pris pour l'établissement de la liaison, l'interaction indiquée, l'IDE peut déterminer et proposer le type de port que relie le connecteur établi. Nous présentons dans ce qui suit deux cas selon l'interaction associée au connecteur.

Cas 1: L'interaction entre les deux composants correspond à un transfert de données :

Lors de la spécification d'une interaction l'IDE propose des noms d'actions de base telle que réalise, *execute*, *fait*, *receive*, *send*, *get*, *set* etc. L'ensemble de ces verbes peut être enrichie au fur et a mesure par l'architecte.

La reconnaissance d'une interaction orientée transfert de données se fait par la détection d'action demandant un transfert de donnée telles que « *receive* », « *send* », « *set* »

ou « *get* ». La mise en place d'une telle interaction sera concrétisée par la création de deux ports de données aux niveaux des deux composants à interconnecter. Le premier composant d'où part la liaison, se verra doter d'un port avec une opération de transfert de données « *set* » ou « *send* » et l'autre composant, un port avec une opération de réception de données « *get* » ou « *receive* ».

Le premier composant d'où part la liaison, se verra doter d'un port de données composé d'un DOAP avec un sens « *out* » pour indiquer que le DOAP est une source de données (Figure 4.11). L'autre composant, se verra doter d'un port de données composé d'un DOAP avec un sens « *in* » pour indiquer que le DOAP indique la nécessité de pourvoir le DOAP d'une donnée (Figure 4.11).

Si au lieu d'utiliser une flèche, l'architecte utilise juste un trait, ou change une flèche en un trait, le DOAP qui sera utilisé de part et d'autre est un DOAP « *inout*» (Figure 4.11).

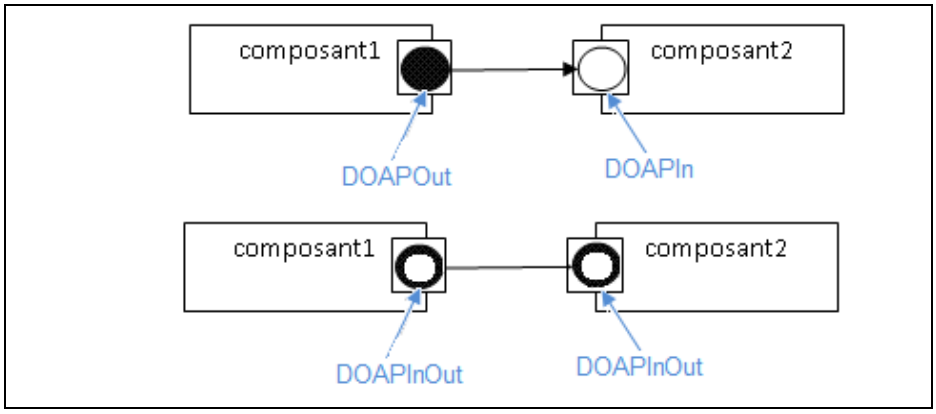


Figure 4.11 : Interaction pour un transfert de données

Cas 2: L'interaction entre les deux composants correspond à un transfert de service :

La reconnaissance d'une interaction orientée service (ou action) se fait par la détection d'action demandant la réalisation d'une opération telles que exécute, réalise, entame etc. Dans ce cas l'interaction sera concrétisée par la création de deux ports de services aux niveaux des deux composants à interconnecter. Ainsi le composant d'où part la liaison se verra doter d'un port client et l'autre d'un port serveur. Le changement du sens du connecteur changera le type de port interconnecté.

Le composant où arrive la liaison sera doté d'un port composé d'un *ServerACTOAP* à travers lequel un service est fourni (Figure 4.12). L'autre composant sera doté d'un port composé d'un ClientACTOAP pour indiquer que le port spécifie un besoin de service (Figure 4.12).

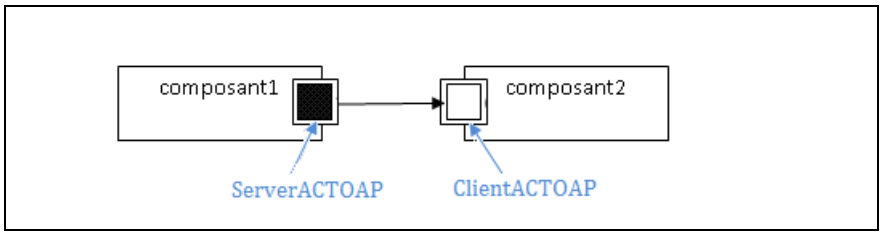


Figure 4.12 : Interaction pour un transfert de service

Pour permettre une transformation automatique et parfois assisté d'une spécification qui est à la base informelle en une spécification formelle utilisant la notation IASA améliorée, l'IDE que nous projetant de mettre sur pied proposera via des fenêtres spécifiques de choisir le sens des connexions (cas 1 ou bien cas 2), le type d'interaction. Il permettra en outre le support du processus de raffinement des composants ou des ports d'interaction. A titre d'exemple , au niveau de transfert de service (cas 2), le port contenant le ServerACTOAP peut être raffiné encore plus d'avantage en lui rajoutant d'autres DOAP nécessaire à la réalisation d'une action (service). Ces DOAP qui sont associés l'ACTOAP dans le port de serveur, peuvent être utilisés indépendamment de l'action supportée par l'ACTOAP associé (Figure 4.13).

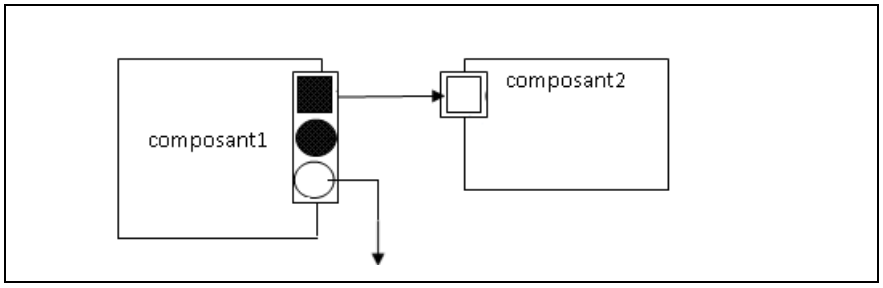


Figure 4.13 : Raffinement au niveau du port d'action

4.6.5.2 Gestion des maladresses causées par les architectes

Quelque soient les qualités de l'architecte, il est judicieux de le guider dans son travail d'élaboration d'une architecture, afin de réduire les sources potentielles pouvant engendrer des erreurs, qu'il serait délicat de tracer par la suite. Une source importante d'erreurs provient des incohérences dans une architecture logicielle. Ces dernières sont fréquentes notamment lors de l'élaboration d'une nouvelle architecture basée en grande partie sur l'élaboration de nouveaux types de composants. La mise en place d'architectures sûres consiste à essayer au maximum de prendre en charge, selon une approche rigoureuse, ces diverses sources d'erreurs. Cette prise en charge se matérialise dans le contexte de notre approche par la nécessité d'automatisation d'une grande partie du processus d'élaboration d'une architecture.

L'automatisation porte essentiellement sur le contrôle des diverses décisions de l’architecte et sur la possibilité de mettre le processus dans des situations sûres à partir desquels, des actions de conception pourraient être entamées de manière plus saine dans son travail d’élaboration d’une architecture.

Ces erreurs ou maladresses peuvent apparaitre durant le processus de conception. Elles sont provoquées par des décisions de conception qui remettent en cause un type déjà instancié. Les erreurs internes à un composant (i.e. suppression de connexion interne, suppression de composants non attaché aux ports externe par connecteur de délégation) sont des erreurs non exceptionnelles. Elles sont locales à un composant et n'influent pas sur le type du composant. Les exceptions de conception sont principalement dues à la modification de types de composant, de connecteurs ou ports déjà instanciés. La modification de types ne concerne que les types définis dans un projet. La modification de port et connecteur se fait par l'ajout ou la suppression de point d'accès, la modification des actions et la modification des contextes d'actions associés. La décision de modification de types instanciés entraînent l'apparition d'erreurs dans la conception et mènent à l'instabilité de plusieurs types de composants qui auraient été stables à un moment donné du processus de conception. Lorsque un tel état exceptionnel survient, il devient nécessaire d’arrêter le processus de conception pour le redémarrer à partir d’une situation sure. Dans notre approche, la décision de conception menant à l'erreur peut être acceptée. Dans ce cas il est nécessaire d'arrêter le processus courant et entamer un processus de correction d'erreurs. A titre d’exemple, la modification d’un port d’un composant, déjà instancier par l'ajout ou la suppression de point d'accès entraine à une erreur de conception, l'outil signale cependant cette erreur à l’architecte. Pour être capable d'introduire la modification sur le modèle, chaque architecte opère selon une approche locale liée à son bon sens (i.e; reprise du processus dés le début, déterminer et éliminer les facteurs bloquant la modification etc.). Dans notre exemple de modification d’un port déjà instancié, l’architecte sera amené soit à modifier tous les ports du même type que le port modifié, soit à changer le type du port concerné ou bien éliminer carrément l’opération de modification.

4.7 Spécification architecturale dans IASA

IASA est une approche orientée aspect d'Architecture logicielle. Elle vise à généraliser l'approche architecture logicielle à la conception de logiciels de tailles diverses, grâce à un modèle de composant, de connecteur et un ADL très flexible capable de supporter les modèles mentaux d'un architecte après une première transformation informel vers le formel. Dans cette section, nous allons présenter les concepts et les techniques développés pour l'approche IASA afin de supporter la spécification directe du modèle mental de l'architecte, élaboré dans les premières phases d'un processus de développement de logiciel.

Le langage de l'approche IASA 3ADL (Architecture, Aspect and Action Description Language) a été défini pour l'approche IASA **[57].** Cependant la forme actuelle de ce langage ne couvre que des parties bien précises de la spécification d'architecture. Dans sa réalité, la définition complète de 3ADL n'a pas encore été achevée. Un des objectifs de notre travail de recherche est de proposer une forme définitive pour ce langage. La tendance actuelle en ADL consiste à utiliser XML plutôt que de définir une syntaxe bien précise pour un langage qui sera par la suite compilé ou interprété. La réalisation en XML d'un langage de description d'architecture logicielle semble être efficace, car après la définition des concepts que le langage doit supporter, il n'est pas nécessaire d'écrire un compilateur. C'est pour ces raisons que nous avons choisi une forme basée sur XML pour 3ADL. Nous avons appelé ce langage X3ADL (eXtensible Architecture, Aspect and Action Description Language). X3ADL tel que nous l'avons défini, peut être fortement extensible, flexible et adaptable afin de servir aux expérimentations au fur et à mesure de l'avancée des recherches dans le cadre du projet IASA. Le langage x3ADL rejoint la famille des langages de description d'architecture basés sur le langage XML, tel que XArch **[111]**, xADL **[112]** et xAcme **[113]**. Une spécification X3ADL sera par la suite l'entrée d'un processus de transformation de modèle qui nous mènerait à une vue exécutable du système ou logiciel décrit. Nous rappellerons dans ce qui suit les modèles fondamentaux de IASA avant de procéder à la description du processus de transformation.

4.7.1 Le modèle de composant de IASA

Le concept de composant est utilisé pour représenter n'importe quel élément rentrant dans la définition fonctionnelle d'une application. Cela veut dire que toute fonctionnalité faisant partie de la logique d'une application est explicitement prise en charge par un composant.

Dans IASA, nous distinguons deux types de composants : les composants primitifs et les composants composites. La structure interne d'un composant primitif est inaccessible. Celle d'un composant composite possède une organisation bien précise. Elle est composée de trois parties. Une première partie, appelée partie opérative (représenté par le type *OperativePart*), comprend les composants réalisant les fonctionnalités pures de l'architecture. La deuxième partie, appelée partie contrôle (représenté par le type *ControlPart*), contient des composants qui réalisent les opérations de contrôle global sur les autres composants des deux autres parties. La troisième partie, appelée partie aspect (représenté par le type *OptionPart*), est optionnelle, elle est dédiée à contenir les aspects techniques d'une application.

L'instanciation d'un composant est réalisée dans le contexte du concept d'enveloppe. Une enveloppe permet d'isoler l'instance pure d'un composant de son environnement d'exploitation en fournissant à ce dernier les éléments nécessaires à l'exploitation de l'instance. L'enveloppe est en réalité l'endroit où seront solutionnés les divers problèmes liés au déploiement de l'instance du composant et à la spécification de topologies très variées, notamment celle mettant en œuvre directement les points d'accès de port.

La description en X3ADL d'un composant composite est hiérarchisé en plusieurs niveaux, le premier niveau est illustré dans la figure 4.14 .La balise <Component> possède les attributs *name* qui donne le nom du composant et *deployedAs* qui renseigne sur le cas de déploiement du composant.

```
- <Component name="" deployedAs="">
    <!-- DEFINITION DU COMPOSANT  -->
  - <Ports>
      <!-- DEFINITION DES PORTS  -->
    </Ports>
  - <Connectors>
      <!-- DEFINITION DES CONNECTEURS  -->
    </Connectors>
  - <OperativePart>
      <!--  DEFINITION DE LA PARTIE OPERATIVE  -->
    </OperativePart>
  - <ControlPart>
      <!--  DEFINITION DE LA PARTIE CONTROLE  -->
    </ControlPart>
  - < OptionPart>
      <!--  DEFINITION DE LA PARTIE ASPECT     -->
    </ OptionPart>

  - <Properties>
      <!-- SPECIFICATION DES PROPRIETES  -->
    </Properties>
</Component>
```

Figure 4.14: Les balises du langage x3ADL

4.7.2 Les points d’accès

Le point d'accès est le plus petit élément manipulable dans une architecture. Dans le modèle de composant IASA, les points d’accès représentent les concepts de base échangés entre deux ports de composants. En effet, toute information, quelque soit sa nature arrive ou part d’un composant à travers un point d’accès. Les concepts supportés par les points d’accès se réduisent à l’échange de données et au transfert du flux de contrôle. Ainsi, un paramètre d'une méthode peut être associé à un point d'accès vu qu'il véhicule une information.

Les points d’accès sont tous représentés par le type de base *AcessPoint*. Un point d’accès possède un mode de communication bien précis. Il est doté de propriétés générales aux points d’accès. Il est aussi associé à un ensemble d’opérations de base notamment celles destinées à des opérations réflexives **[58]**. Un point d’accès peut être marqué ou non marqué (*marked*, *unmarked*). Un point d’accès marqué est un point d’accès correctement connecté. Cette caractéristique est très utile dans le processus de validation d’une architecture et elle est exploitée dans un processus de conception du général vers le particulier avec validation progressive de l’architecture durant les différentes phases de conception, sans nécessité que les composants utilisés soient effectivement réalisés. Un point d’accès est destiné soit au transfert de données (Data Oriented Access Point (DOAP)) soit à émettre ou recevoir des actions (Action Oriented Access Point (ACTOAP)) (Figure 4.15). Un ACTOAP indique la présence d’un service qui peut être initié à partir de ce point. Il est dédié aux transferts de flux de contrôle (invocation d’une action, reprise du flux de contrôle par une action).

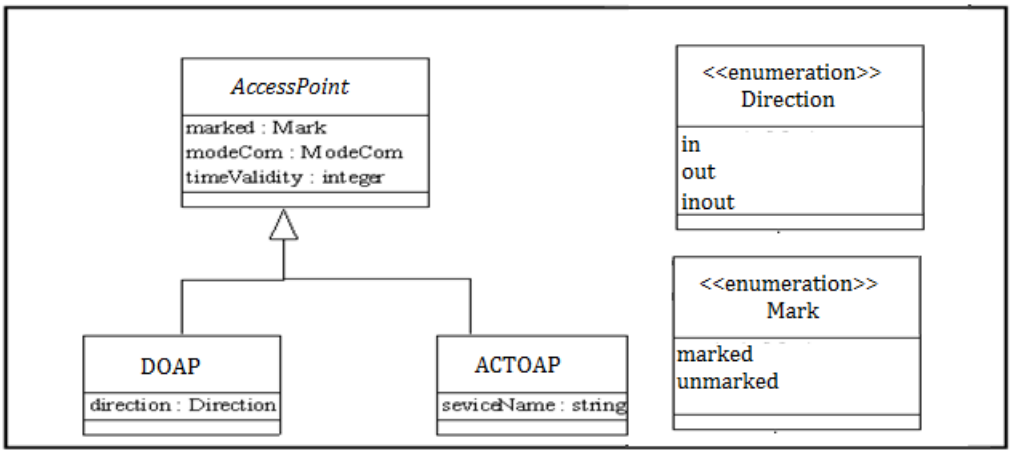


Figure 4.15 : Extrait du métamodèle des points d'accès

4.7.2.1 Les points d’accès aux données (DOAP)

Un DOAP est utilisé pour spécifier un transfert explicite de données. L’architecte peut

utiliser un ensemble de DOAP prédéfini ou définir un DOAP qui lui est spécifique. Les DOAP prédéfinis englobent les types de données primitifs que nous retrouvons dans les divers langages de programmation (entier, réel, caractère, booléen) ainsi que des types spécifiques à l'approche IASA. A titre d'exemple, IntDOAP, ByteDOAP, CharDOAP et BooleanDOAP sont successivement associés aux types primitifs entier, byte, caractère et booléen.

Un point d'accès de type DOAP est doté d'attributs indiquant le sens des opérations de transfert de données (Figure 4.16). Trois valeurs sont possibles pour spécifier le sens des données : *in*, *out* et *inout*. L'attribut *in* indique la nécessité de pourvoir le point d'accès d'une donnée. L'attribut *out* indique que le point d'accès est une source de données. L'attribut *inout* spécifie que la donnée peut être transférée dans les deux directions.

La connexion de DOAP suit les règles suivantes : Un DOAPin (respectivement DOAPout) ne peut se connecter qu'à un DOAPout (respectivement DOAPin) ou DOAPinout. Un DOAPinout est connectable à un DOAPin, DOAPout et DOAPinout. Lorsqu'un point d'accès est correctement connecté, il est alors marqué (positionnement de l'attribut de marquage du point d'accès à la valeur marked).

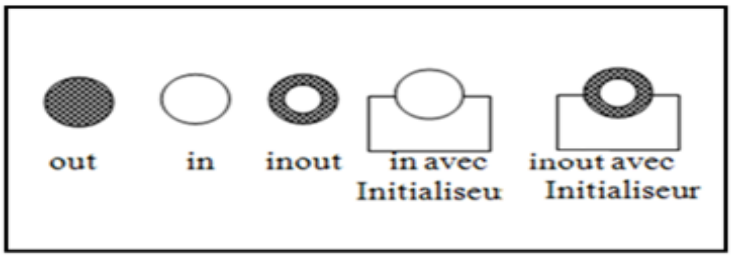


Figure 4.16 : Représentation graphique des DOAP

4.7.2.2 Les points d'accès de services (ACTOAP)

Un point d'accès de type ACTOAP indique qu'un service peut être initié à partir de ce point. Dans la version actuelle du modèle IASA, nous considérons qu'un ACTOAP correspond uniquement à un seul service et un service permet de réaliser plusieurs actions. Généralement le nombre d'actions que peut prendre en charge un service, est assez restreint et concerne un aspect bien précis d'un domaine d'application. C'est cette idée qui est à la base de la définition de la notion de contexte d'action au niveau du langage 3ADL. Vis-à-vis d'un service, un point d'accès de type ACTOAP ne peut être que fournisseur ou client. Un ACTOAP à travers lequel un service est fourni est représenté par le type spécifique ServerACTOAP (ACTOAPS). Lorsqu'un ACTOAP spécifie un besoin de service, il est alors

représenté par le type spécifique ClientACTOAP (ACTOAPC) (Figure 4.17).

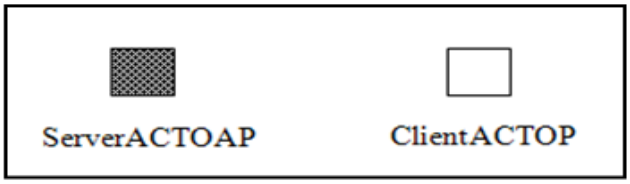


Figure 4.17: Représentation graphique des ACTOAP

Dans X3ADL, les points d'accès sont représentés par la balise *AccessPoint* comme illustré dans la figure 4.18. La balise *AccessPoint* comporte les attributs suivant :

- L'attribut *type* pour le type du point d'accès.
- L'attribut *name* pour le nom de l'instance.
- L'attribut *direction* qui peut prendre les valeurs IN, OUT ou INOUT pour le sens de la communication.
- L'attribut *modeInteraction* pour la indiquer la synchronisation (**SYNCHRONE** si le point d'accès est synchrone **ASYNCHRONE** sinon).
- L'attribut *time* pour le temps de validité d'une action ou d'une donnée (il prend la valeur 0 lorsque le temps de validité est infini).

```
- <accesspoints>
    <accesspoint name="pAlarmOut" type=" STRINGDOAP " direction ="OUT" modeInteraction="SYNCHRONE" time ="0" />
    <accesspoint name="Enable" type=" INTDOAP " direction ="IN" modeInteraction="SYNCHRONE" time ="0" />
    <accesspoint name="pMain" type=" ServerACTOAP" modeInteraction="SYNCHRONE" time ="0" />
  </accesspoints>
```

Figure 4.18: Exemple de la spécification des points d'accès en X3ADL

4.7.3 Les ports

Un port est un regroupement de points d'accès, étroitement associés dans le contexte de la réalisation d'un objectif commun. Un port possède un attribut renseignant sur le nom de l'instance. Le port représente un espace de nom pour les points d'accès. Chaque point d'accès est identifié de manière unique dans le contexte d'un port (Figure 4.19). Le composant représente un espace de nom pour un port. Ce dernier est identifié de manière unique dans un composant.

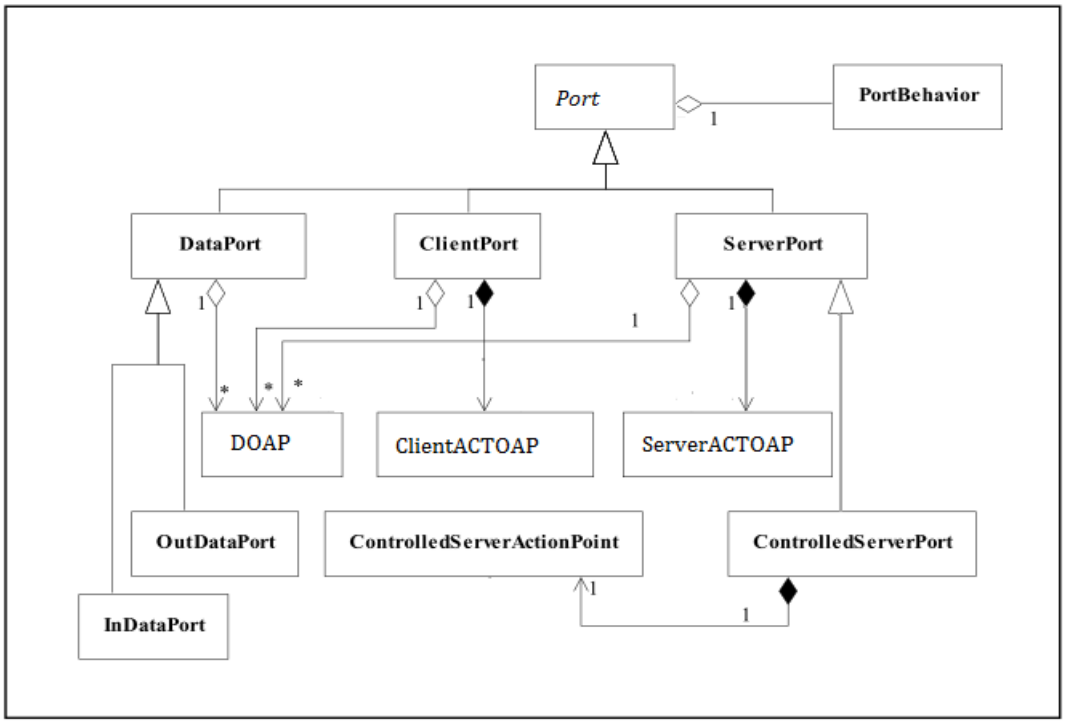


Figure 4.19: Extrait du métamodèle du port de IASA

Les ports modélisent la vue externe d'un composant. C'est seulement à travers les ports qu'un composant est manipulable. Les ports divulguent les ressources (services et données) requises et fournies d'un composant ainsi que les aspects comportementaux du composant, observables sur ces ports. Les comportements sont décrits dans le langage d'action 3ADL **[58]** qui est une des composantes essentielles de l'approche IASA. Un port est une entité à part, pouvant être ajouté ou supprimé à un composant. Il peut en outre être modifié par l'ajout ou la suppression de point d'accès, notamment les points d'accès de données.

Dans sa version actuelle, l'approche IASA utilise plusieurs types de ports que nous présentons ci-dessous (Figure 4.19).Il est à remarquer qu'il est possible de définir d'autres types de ports spécifiques aux concepteurs.

- Les ports de données (DataPort) qui ne peuvent contenir que des DOAP.
- Les ports clients (ClientPort) : contiennent au moins un ClientACTOAP et optionnellement un ou plusieurs DOAP.
- Les ports de service (ServerPort) : contiennent au moins un ServerACTOAP et optionnellement un ou plusieurs DOAP.
- Les ports contrôlés : C'est des ports de service qui contiennent un ServerACTOAP contrôlé au lieu d'un ServerACTOAP et/ou une EnableDOAP.

- Les ports orientés aspects : Un port orienté aspect ou *AdvicePort* est destiné à permettre l'accès aux aspects techniques que fournit un composant aspect.

La balise Ports est illustrée à travers la figure 4.20 A. Cette balise est une descendante directe de la balise Component. Elle englobe un ensemble de balises port. Chaque balise port définit un nouveau type de port en indiquant ses points d'accès (Figure 4.20 (B))

A

```
- <ports>
  + <port name="pFtp" type="FTPClientPort">
  + <port name="pSQL" type="SQLClientPort">
  + <port name="pPrinter" type="PrintSpooler">
  </ports>
```

B

```
- <port name="pMain" type="MainPort">
  - <accesspoints>
      <accesspoint name="pAlarmOut" type=" STRINGDOAP " direction ="OUT" modeInteraction="SYNCHRONE" time ="0" />
      <accesspoint name="Enable" type=" INTDOAP " direction ="IN" modeInteraction="SYNCHRONE" time ="0" />
      <accesspoint name="pMain" type=" ServerACTOAP " modeInteraction="SYNCHRONE" time ="0" />
    </accesspoints>
  </port >
```

Figure 4.20: Exemple de spécification de port en X3ADL

4.7.4 Les connecteurs

Lors de la définition des connexions entre les ports de composants, l'architecte spécifie parfois une interaction et parfois indique que la connexion doit être réalisée par une technologie bien connue tels qu'un protocole de communication ou un appel distant de procédure dans le contexte d'une infrastructure de communication (i.e. Bus CORBA, RMI).

Afin de pouvoir spécifier librement diverses topologies, IASA utilise un seul modèle de connecteur de base qui est le connecteur direct représenté par un appel de procédure.

La description des connecteurs en X3ADL est illustrée dans la figure ci-dessous. La balise *Connectors* regroupe un ensemble de balises *connector*, dans lesquelles les différents types de connecteurs sont définis (Figure 4.21 A). Chaque connecteur est identifié par son nom et est défini par ses rôles, son architecture et son comportement (respectivement les balises *Roles*, *ConnectorArchitecture* et *Behavior*) (Figure 4.21 B).

A

```
- <Connectors>
  - <connector>
      <!-- Definition du type de connecteur 1  -->
    </connector>
  - <connector>
      <!-- Definition du type de connecteur 2  -->
    </connector>
  - <connector>
      <!-- Definition du type de connecteur n  -->
    </connector>
  </Connectors>
```

B

```
- <connector name="">
  - <Roles>
      <!-- Declaration des roles    -->
    </Roles>
  - <ConnectorArchitecture>
      <!-- Declaration du type de connecteur    -->
    </ConnectorArchitecture>
  - <Behavior>
      <!-- Definition du comportement  -->
    </Behavior>
  </connector>
```

Figure 4.21: description des connecteurs en X3ADL

4.7.5 L'enveloppe

La vue externe de tout composant est formée d'une enveloppe. L'enveloppe IASA n'est pas manipulée directement par le concepteur. Elle est générée quand un type de composant est instancié. Son premier objectif est d'isoler l'instance du composant du monde extérieur. L'enveloppe est également utilisé pour spécifier la plan de déploiement, pour permettre la spécification de connexions impliquant des éléments structurels du port et de gérer l'injection et de suppression des conseils (advices) fournis par les composants aspect. L'enveloppe est le lieu où les opérations de tissage de code sont effectuées. Chaque port d'une instance de composant enveloppé est associé à un port défini dans l'enveloppe. Les ports de l'enveloppe représentent l'interface réelle de l'instance du composant avec le monde extérieur. Les ports d'une enveloppe représentent les endroits où résident les ressources nécessaires pour atteindre les objectifs de l'enveloppe (injection aspect, le tissage d'aspect, manipulation de l'élément structurel du port).

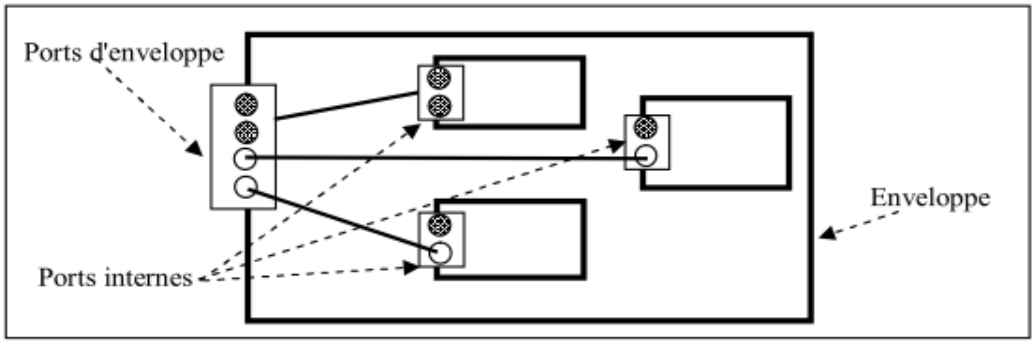


Figure 4.22: Enveloppe IASA

4.7.6 Organisation de la vue interne du composite

La vue interne est organisée en trois parties (Figure 4.23): Deux parties obligatoires (la partie opérative et la partie contrôle) et une partie optionnelle (la partie option appelée aussi la partie aspect).

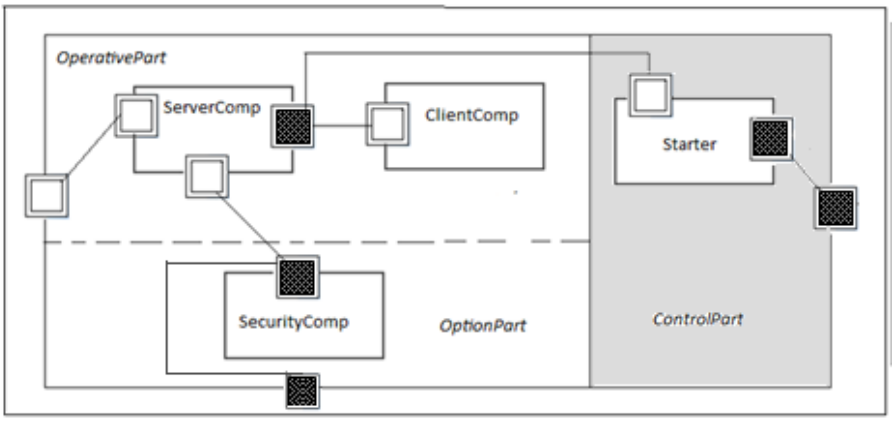


Figure 4.23 : Vue interne d'un composant composite

4.7.6.1 La partie opérative

Elle contient les composants représentants par leurs interconnexions la logique générale du composite à un moment bien précis. Elle est représentée par le type *OperativePart*.

```
- <operativepart>
  - <components>
      <component name="x25" type="X25CM_Core" />
      <component name="lc" type="LicenseController" />
    </components>
```

Figure 4.24: Exemple de description de la partie opérative en X3ADL

4.7.6.2 La partie contrôle

La partie contrôle représentée par le type *ControlPart*, réalise les diverses opérations de contrôle sur les composants de la partie opérative, telles que la gestion du flux de contrôle des divers services (arrêt, lancement en séquence ou en parallèle), le contrôle de l'évolution structurelle, la gestion des exceptions, l'exportation des états du composant et la génération de log. La partie contrôle est composée d'au moins un composant, appelé le contrôleur (OpPartController). Ce dernier est un composant dit comportemental. Sa spécification est faite complètement en langage d'action 3ADL.

```
- <controlpart>
  - <components>
    - <composant name="starter" type= " OpPartController">
      - < Behavior >
          < !-- Definition du comportement du controleur -- >
        </ Behavior >
      </composant >
    </components>
</controlpart>
```

Figure 4.25 : Exemple de description de la partie contrôle en X3ADL

4.7.6.3 La partie aspect

Cette partie est optionnelle, elle est représentée par le type *OptionPart*. Elle est dédiée à contenir les aspects techniques d'une application. Ces aspects techniques, une fois instanciés au niveau de cette partie, seront injectés au niveau des composants de la partie opérative. C'est cette partie qui permet de faire de IASA une approche Architecture logicielle orientée aspect. La spécification de la partie aspect est illustrée dans la Figure 4.26. La balise optionpart est composée de la balise *Advice* indique comment les aspects, sont injectés en indiquant le type de l'advice (*before*, *after* et *around*). La clause *PointCut* précise l'endroit où les *advices* sont insérés.

```
- <optionpart>
 - <pointcuts>
    <joinpoint name="log" aspect="lc.pEnable.OutData1.send" />
  </pointcuts>
 - <advices>
    <advice aspect="logCmp .ServerAction1" type="after" joinpoint="log" />
  </advices>
</optionpart>
```

Figure 4.26 Exemple de description de la partie aspect en X3ADL

4.7.7 Déploiement des composants

Le modèle de composant de l'approche IASA permet une haute flexibilité dans la spécification des propriétés de déploiement des composants. Ces propriétés sont utilisées pour produire le code, et éventuellement les descripteurs de déploiement effectif nécessaires. Ces codes et descripteurs sont utilisés pour charger les composants dans leurs environnements d'exécution. Le déploiement adressé dans IASA, ne concerne pas la méthode de chargement et de lancement des composants dans leurs environnements mais la production du code et des descripteurs qui seront utilisés par les outils de déploiement effectif des composants dans

leurs environnements d'exécution. La spécification du déploiement consiste à définir pour un type de composant des cas de déploiement et des plans de déploiement. Les cas de déploiement renseigne sur les formes réelles que le composant pourra avoir une fois en exécution (i.e. tâche, processus). Le plan de déploiement indique comment un cas de déploiement est appliqué (i.e. tâche dans un processus fonctionnant sur la machine locale).

Le cas de déploiement spécifie l'état du composant à l'exécution (i.e. tâche principale, web service, processus, etc.). L'approche IASA a identifié plusieurs cas de déploiement direct parmi lesquels nous citons à titre d'illustration: PROCESS, THREAD, APPLET, SERVLET, EJB, et JAVASCRIPT. Il est clair que cette liste est extensible pour la prise en charge de cas de déploiement divers. Chaque cas de déploiement doit être spécifié avec son environnement. A titre d'exemple, pour le cas PROCESS, il est nécessaire d'indiquer le type du système d'exploitation et l'adresse Internet de la machine.

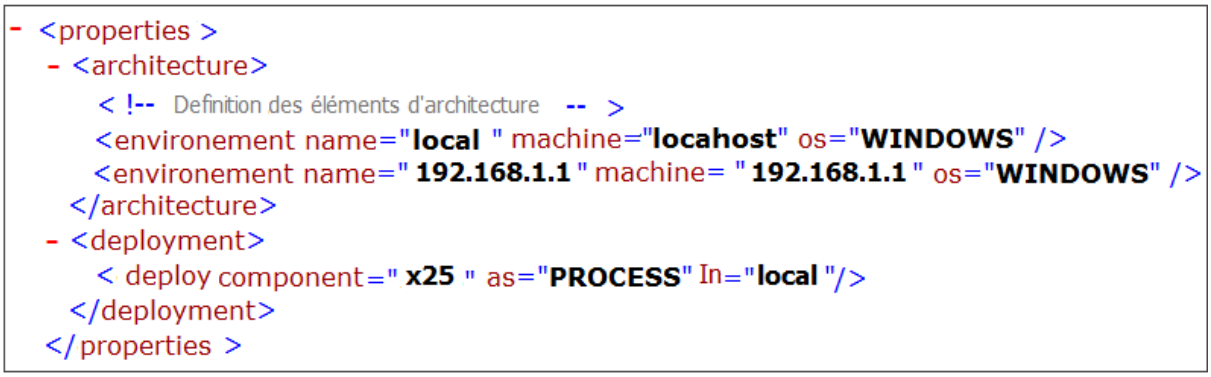


Figure 4.27: Description du déploiement en X3ADL

4.8 Le Processus de transformation pour IASA

L'approche IASA permet la spécification libre de topologies de composants à un haut niveau d'abstraction et ceci grâce au concept de port qui offre des facilités pour la spécification quasi libre et sans contraintes dans l'optique d'atteindre des objectifs particuliers. Vis-à-vis de l'approche MDA, le modèle de composant et port permettent de spécifier des architectures qui sont encore plus abstraites que le PIM (Platform Independent Model) du MDA. Notre approche permet de produire, un modèle d'architecture non seulement indépendant de la plateforme, mais indépendant des mécanismes logiciels, notamment ceux dédiés aux interactions.

Le processus de transformation d'une spécification x3ADL vers une spécification dans une technologie d'implémentation bien précise prend son départ de la spécification graphique.

Celle-ci est transformée de manière automatique ou assistée (l'architecte doit fournir des informations complémentaires permettant une prise de décision dans l'étape interprétation des décisions architecturales) en une description X3ADL. Le processus de transformation comprend ainsi 4 étapes :

- La formalisation des décisions architecturales : obtention d'une description X3ADL à partir d'une spécification libre d'architecture logicielle similaire à la spécification informelle est avec un minimum de contrainte.
- Le tissage des aspects : Cette opération a lieu lorsque la conception orientée aspect est mise en œuvre. C'est une transformation X3ADL en X3ADL. Dans la forme générée, les clauses relatives à la spécification orientée aspects sont résolues et aucune clause orientée aspect ne se trouve dans le X3ADL généré par cette étape.
- La normalisation de l'architecture.
- La production de la vue d'implémentation.

4.8.1 La formalisation des décisions architecturales

Comme introduit auparavant, une décision architecturale concerne le choix des composant, la manière de les interconnecter pour former la topologie que l'architecte possède en tête, la manière de spécifier les divers aspects comportementaux, notamment les interactions, le comportement des composant, les conditions de connexion aux ports imposées par les composants. La transformation à ce niveau concerne les actions architecturales qui semblent être du domaine de l'informel, comme par exemple décider de relier deux composants en tirant un trait épais (bus) ou simple (connecteur) liant les pourtour (frontière) des formes géométriques représentant les composants ou relier la frontière d'un composant à un connecteur ou bus.

Comme introduit au préalable, les règles de transformation appliquées dépendent de plusieurs facteurs notamment :

- Les éléments à interconnecter.
- Les standards de communication utilisés.
- La façon dont la décision architecturale est reportée : Sens du connecteur, description des interactions.
- Un choix explicite de l'architecte.

Ce processus de transformation se fait en temps réel, c'est-à-dire qu'il est réalisé au fur et à mesure des actions de l'architecte au niveau de l'éditeur graphique, et que le résultat de la

transformation est immédiatement accessible à l'architecte. Ainsi à chaque action l'IDE réalise une transformation selon les conditions dans laquelle est spécifiée la décision architecturale. Dans ces conditions l'architecte peut parfois intervenir et ajuster la transformation selon des exigences. Dans certaines situations l'architecte n'a aucun pouvoir de changer le résultat de la transformation. A titre d'exemple si nous avons deux composant, un représentant un serveur FTP et l'autre un composant ordinaire. Les décisions architecturales d'établissement d'une connexion en partant du serveur FTP vers le composant et d'établissement d'une connexion en partant du composant vers le serveur FTP, auront le même résultat. Au niveau du serveur c'est un port de service standard, qui doit être prédéfini dans l'éditeur, et au niveau du composant c'est un port client FTP.

Lorsque l'établissement d'une connexion démarre d'un composant (de sa frontière et non pas d'un port existant) et atteint un autre composant, le point de départ du connecteur sera doté d'un port client, et le point d'arrivée d'un port de service. L'architecte peut changer la nature d'un port. Cependant, il ne peut pas imposer des constructions qui ne sont pas supportée par IASA et en architecture logicielle en générale. Ainsi s'il change le port client en port serveur, l'autre port se transforme automatiquement en port client.

Si l'architecte tire un fil d'un connecteur vers un composant, ce sera un port client qui sera crée par défaut au niveau du composant, vu qu'en règle général nous avons plusieurs client pour un serveur. Si dans cette situation l'architecte indique que le port est un port de service, l'IDE proposera automatiquement un ou plusieurs connecteurs complexes tels qu'un commutateur simple ou un équilibreur de charge. Le connecteur complexe, qui est un composant de communication, sera automatiquement ajouté à l'architecture sous forme d'un fil spécial. Dans ce type de connecteurs, IASA impose une organisation des ports client et serveurs. Les clients arrivent sur une face et les serveurs sont sur l'autre face.

4.8.2 Le tissage des aspects

Le tissage des aspects se base principalement sur le concept d'enveloppe. C'est au niveau de celle-ci que les types de ports de composant sont transformés par ajout de nouveau point d'accès appelé ASPOAP (Aspect Oriented Access Point). La transformation de ces ports entraîne la création de nouveau type interne de ports. Le résultat de cette phase est une description X3ADL dans laquelle les opérations d'injection d'aspects sont totalement résolues. Les figures 4.28 et 4.29 illustrent l'injection de l'aspect *log* (l'advice est fourni par le composant aspect logCmp) après la pointcut log. La figure 4.28 montre l'architecture

d'origine.

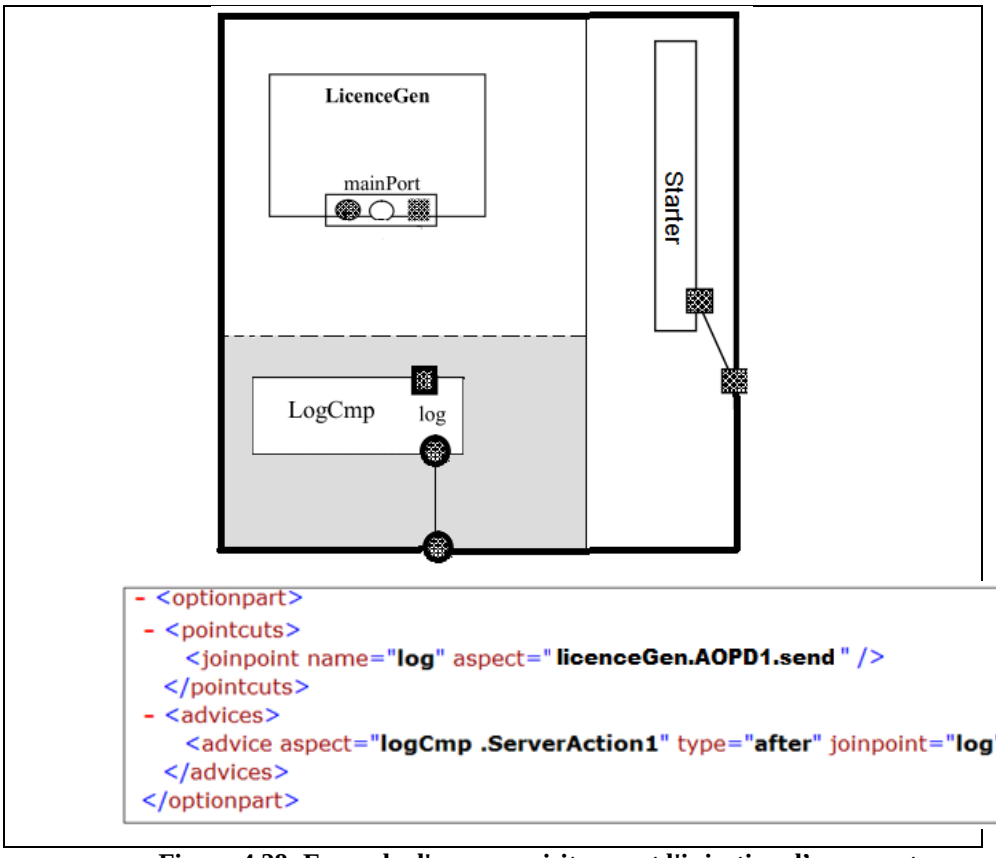


Figure 4.28: Exemple d'un compoisite avant l'injection d'un aspect

Le résultat de l'opération d'injection de l'aspect *log* est une architecture tissée (Figure 4.29) caractérisé par:

- L'introduction d'un ClientASPOAP *log* dans le port *mainPort* ;
- L'établissement d'une connexion aspect entre le port contenant le ClientASPOAP client *log* et le port de l'advice logCMP.

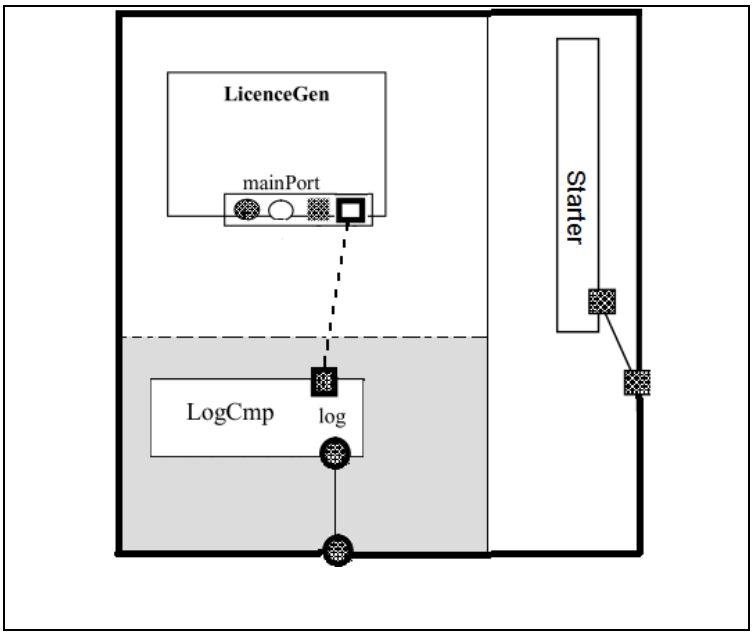


Figure 4.29 : Exemple d'un composite après le tissage d'un aspect

4.8.3 La normalisation

La phase de normalisation nous permet de déterminer un modèle de niveau PIM de MDA. Cette normalisation transforme une description x3ADL en une description basée sur les concepts ordinaires de port et d'interface. Nous rappelons que le port IASA, contrairement aux ports basés sur le concept d'interface, permet la manipulation individuelle de tout élément structurel ou comportemental le définissant. Ceci n'est pas le cas dans les autres approches d'architecture logicielle où le port, appelé aussi interface, est un concept atomique, ne permettant pas la manipulation de ses éléments constitutifs. Une architecture construite avec la liberté de manipulation des points d'accès (DOAP notamment) n'est pas une architecture ordinaire. Sa transformation en une architecture ordinaire permettrait par la suite d'obtenir le code exécutable en utilisant les outils et approches de transformation existantes si elles répondent correctement à nos besoins. A titre d'exemple, un diagramme utilisant les composants et les connecteurs UML2.0 est une architecture ordinaire. Une description ARchJava est aussi une architecture ordinaire.

4.8.3.1 Les concepts de base du processus normalisation

Le processus de normalisation permet d'avoir une description d'architecture dite ordinaire qui est considérée comme un PIM selon l'approche MDA. Dans la description PIM obtenu, les ports sont modélisés par interface. Le résultat de cette étape peut être une description dans un ADL modélisant les ports par interfaces comme dans le cas d'ArchJava, ou une description en UML2.0 centrée sur un diagramme de composant. Ce processus de

normalisation n'est rien d'autre qu'une transformation exogène (raffinement), qui prend en entrée le modèle source qui est une description x3ADL basée sur les ports de IASA comme présenté précédemment, et en sortie, fourni un modèle cible qui correspond à une description X3ADL basée sur les concepts ordinaires de port et d'interface.

Pour transformer une conception selon l'approche intégrée dans un langage cible, il est nécessaire de définir les modèles source et cible, nous considérons donc le modèle source X3ADL et le modèle cible X3ADL normalisé (à base d'interfaces ordinaires), ces modèles sont décrits à l'aide d'une DTD XML (voir Annexe A et Annexe B) qui sert à définir la syntaxe selon laquelle les modèles vont être exprimés, cette DTD représente le métamodèle.

4.8.3.2 Règles de transformation

La phase de normalisation est la plus importante dans le processus de génération de la vue implémentation. Elle met en œuvre un ensemble de règles de transformation pour produire une spécification en X3ADL d'une architecture régulière à base d'interfaces ordinaires. En effet, un défi, lors de l'utilisation des approches de transformation de modèles, est de définir les règles de transformation qui guident le développeur du logiciel à travers le processus. Dans le contexte de notre processus de normalisation, nous présentons dans ce qui suit les règles de transformation générales qui ne dépendent d'aucune technologie d'implémentation:

- Un port peut correspondre à une ou plusieurs interfaces selon la topologie dans laquelle un port est engagé.
- Un port peut aussi correspondre à plusieurs ports ArchJava.
- Dans le cas de topologies simples, un port correspond à une interface UML, un port ArchJava ou une interface Java.
- Une interface est composée de plusieurs méthodes comme ceci est le cas en UML.
- Un ACTOAP correspond à une ou plusieurs méthodes implémentant le service associé à ce point. Dans le modèle de port, le service est pris en charge par une ou plusieurs méthodes.
- Une action est réalisée par une seule méthode. Une méthode peut prendre en charge plusieurs actions. Ce seront alors les paramètres qui distingueront l'action associée à la méthode.
- Un DOAP peut être utilisé avec une action dans le cas ou il correspond à un ACTOAP. Dans ce cas il correspond à un paramètre de la méthode associée à l'action.

Un type de retour de méthode est considéré comme un paramètre ordinaire et peut correspondre à un DOAP.

- Un DOAP peut être associé à un paramètre d'une méthode (comme nous venons de le voir), à une méthode de transport spécifique ou aux deux.
- Lorsque le DOAP est interconnecté individuellement, il est représenté par un port, il doit être pourvu de transporteur spécifique représenté par les accesseurs *get* ou *set* .La méthode de transport associé à un DOAP sera une méthode d'envoi (un accesseur set) si le DOAP possède la direction in et une méthode de réception (accesseur *get*) si le DOAP possède une direction out. Si la direction est inout, le DOAP sera associé simultanément aux deux accesseurs *set* et *get*.

Ainsi la phase de normalisation s'achève par la détermination de la topologie finale de l'application. A titre d'exemple, le résultat de la phase de normalisation du composant *calcCmp* est présenté dans la figure 4.30, est une spécification à base d'interfaces ordinaires en X3ADL est présentée dans la figure 4.31. Le port du composant *calcCmp* est composé d'une interface, qui correspond à deux méthodes (*add* et *sqr*). Les DAOP x1, r et r sont utilisés dans le contexte des actions du ACTOAP, ils vont correspondre donc à des paramètres des fonctions associées à ces actions (*add* et *sqr*). Le DAOP x2 est utilisé individuellement donc il est représenté par un port, il est pourvu d'un transporteur représenté par l'accesseurs *get.*

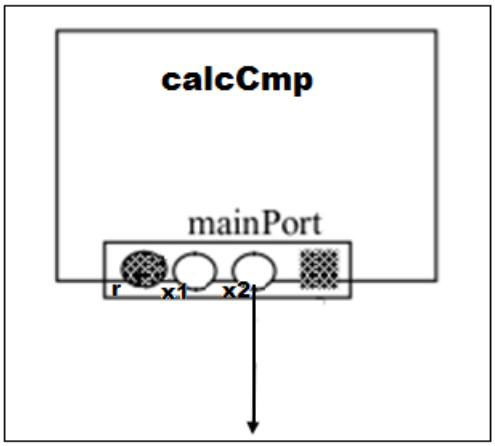


Figure 4.30 : composant calcCmp

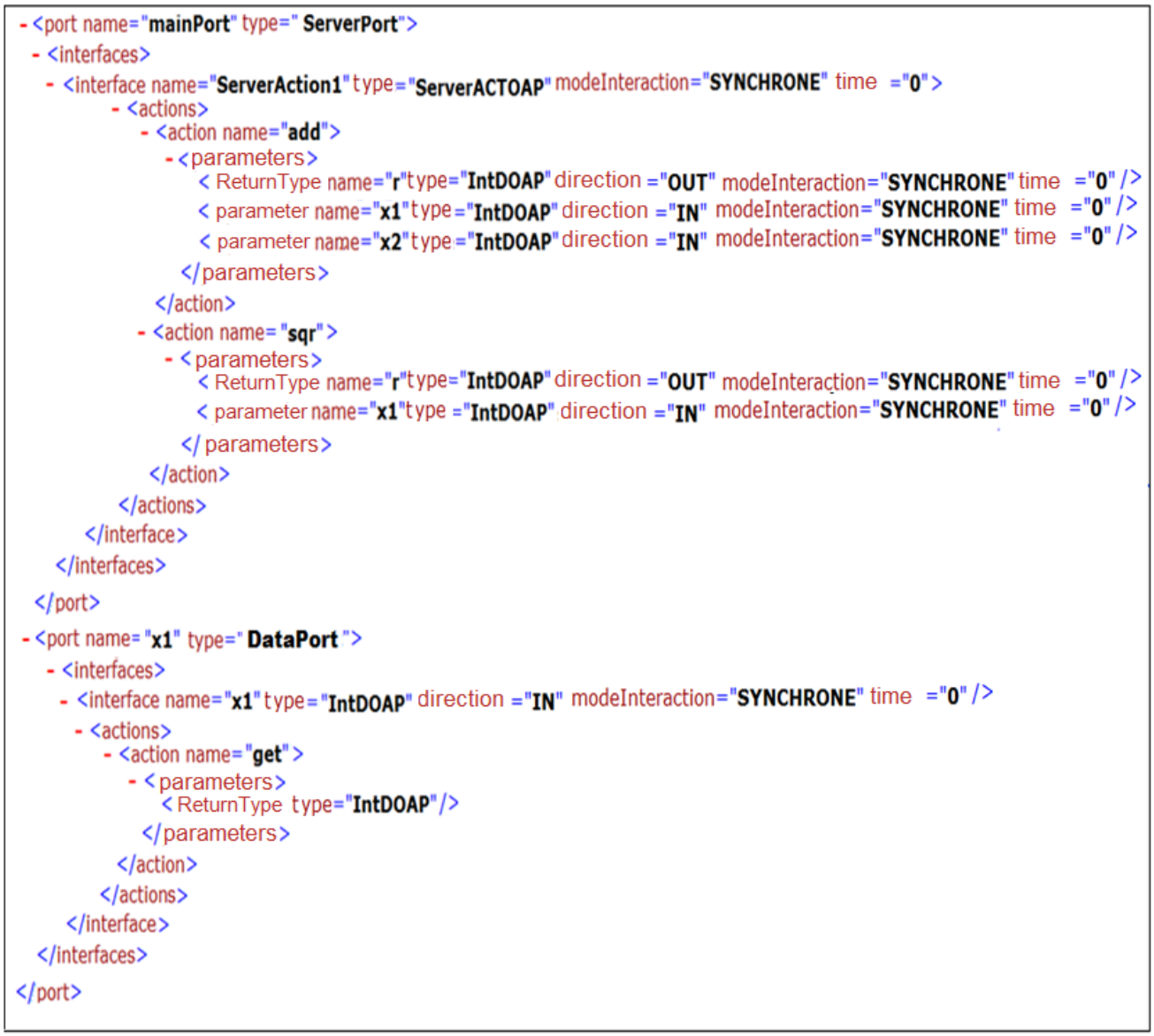


Figure 4.31: Exemple de description d'une description à base d'interfaces ordinaires en X3ADL

Les règles que nous avons définies sont modélisées et implémentées dans des descriptions XML. Ainsi la description d'architecture à base d'interfaces ordinaires est automatiquement générée en appliquant ces règles de transformation.

4.8.4 Production de la vue implémentation

L'objectif principal de cette phase est la production de la vue d'implémentation. En d'autres termes, il s'agit de la projection de l'architecture régulière produite par la phase de normalisation dans une technologie d'implémentation choisie. A ce niveau, nous devons spécifier les règles de transformation spécifiques à une technologie d'implémentation. Dans le cadre de la validation de notre approche, nous choisissons le langage d'implémentation Java, vu qu'actuellement l'approche IASA se base fondamentalement sur la technologie Java. Dans

ce cas, les règles de transformation spécifiques sont celles qui permettent de générer des composants écrits en Java. Nous rappelons qu'un composant IASA est toujours instancié dans une enveloppe. Ce concept d'enveloppe permettra plus de flexibilité dans la transformation. Le processus de transformation est en réalité la création des enveloppes dont l'objectif principal est l'interception de toute information ou action. La flexibilité de IASA permet d'adapter facilement la logique de n'importe quel domaine à partir du moment où la logique de ce domaine est décrite en spécifiant l'enveloppe. Ainsi à titre d'exemple, si un composant est déployé en tant que tâche, l'enveloppe sera une classe qui étend la classe *Thread* de Java. Toutes les informations de communications entre les différents Thread seront interceptées par l'enveloppe et aiguillées vers l'instance enveloppée.

Les règles de transformation mise en œuvre durant cette phase de génération de la vue d'implémentation se résument aux points suivants :

- Chaque composant est représenté par des POJO (Plain Old Java Object), il est instancié dans une enveloppe.
- Chaque enveloppe est représentée par un POJO.
- L'enveloppe doit implémenter la même interface que le composant qu'elle enveloppe. L'implémentation au niveau de l'enveloppe aura un sens d'interception de tout appel entrant ou sortant du composant enveloppé. L'enveloppe est aussi le lieu ou sont hébergés les rôles de connecteurs.
- Le constructeur de l'enveloppe doit instancier les POJO représentant le composant enveloppé.
- L'enveloppe du contrôleur doit disposer de toutes les références des autres enveloppes afin de réaliser la composition. Cette enveloppe contient la fonction Main.
- Les interfaces des ports sont projetées directement en Interfaces Java.
- Chaque méthode implémentant un service est projetée en une fonction Java.

Dans ce qui suit, nous allons présenter la description en java du composant primitif *Calccmp* de la figure 4.30.

Les deux ports du composant sont décrits dans la portion de code ci-dessous :

```
package iasa.port;
public interface mainPort extends IASAServerPort{
	public int add(int x1,int x2);
	public int sqr(int x1);
}
```

```
package iasa.port;
public interface x1 extends DataPort{
	public int get();
}
```

La description du composant Calccmp est la suivante :

```
package iasa.component.primitive;
import iasa.port.*
public class Calccmp implements mainPort,x1{
	public int add(int x1,int x2){
		return x1+x1;
	}
	public int sqr(int x1){
		return x1*x1;
	}
	public int get(){
		return x1;
	}
}
```

Le composant *Calccmp* est déployé autant que tâche dans ce cas l'enveloppe du composant *Calccmp* hérite de la classe Thread :

```
package iasa.impl.enveloppe;
import iasa.component.primitive.*
public class CalccmpImpl extends Thread{

	Calccmp calc;

	public CalccmpImpl(){
	    start();
	}
	public void run() {
	     calc =new Calccmp();
	}
	public int add(int x1,int x2){
	     return calc.add(x1,x2);
	}
	public int sqr(int x1){
	     return calc.sqr(x1);
	}
}
```

4.9 Conclusion

Nous avons présenté dans ce chapitre notre contribution, à savoir, la proposition d'un ensemble de notations, concepts et techniques développés dans le cadre de l'approche IASA dans le but de supporter la spécification directe des modèles mentaux de l'architecte. Cette flexibilité est obtenue grâce au modèle de port de l'approche IASA, qui permet de manipuler librement ses composants constructifs, nous avons exploité cette caractéristique afin de prendre en charge les comportements informels de l'architecte durant les premières phases du processus d'élaboration d'un logiciel. Cette prise en charge se fait par l'automatisation d'une grande partie du processus d'élaboration d'une architecture. L'automatisation porte essentiellement sur le contrôle des diverses décisions de l'architecte en les projetant dans un langage de spécification X3ADL que nous avons développé.

Nous avons ensuite présenté notre processus de transformation d'une spécification x3ADL vers une spécification dans une technologie d'implémentation, en intégrant les différentes étapes de l'approche MDA, nécessaires pour l'obtention du code exécutable.

CHAPITRE 5
IASASTUDIO : UN IDE POUR LA SPECIFICATION FLEXIBLE D'ARCHITECTURE LOGCIELLE SELON L'APPROCHE IASA

5.1 Introduction

Nous avons vu lors du précédent chapitre que les principaux besoins liés à la spécification flexible de l'architecture logicielle consistent à représenter le modèle mental élaboré par un architecte dans les premières phases d'un processus de développement de logiciel. Dans ce chapitre afin de concrétiser l'ensemble des propositions autour du modèle proposé dans le chapitre précédent. Nous allons proposer une réalisation, dans le contexte d'un IDE qui permet la spécification graphique d'architecture logicielle selon l'approche IASA. Cet IDE que nous avons appelé IASASTUDIO va permettre d'accueillir les éléments du modèle mental de l'architecte et les diverses décisions architecturales. Ces éléments seront par la suite transformés via une série de transformation en un ADL qui lui-même sera transformé en code exécutable.

Nous commencerons tout d'abord par présenter l'architecture générale de cet outil, ensuite nous nous intéresserons aux aspects conception et réalisation. Enfin, nous allons dérouler un exemple complet de spécification d'architecture. En suivant la démarche proposée dans le chapitre précédent, jusqu'à l'obtention du code exécutable.

5.2 Présentation de l'environnement IASASTUDIO

IASASTUDIO est l'environnement de développement qui génère des composants composites selon l'approche IASA. IASASTUDIO permet de concevoir les applications à un haut niveau d'abstraction, afin d'accueillir les éléments du modèle mental de l'architecte et générer automatiquement le code x3ADL correspondant à l'architecture spécifiée.

Un projet IASASTDIO correspond à un composant composite. La fenêtre principale d'un projet IASASTUDIO (Figure 5.1) est composé de plusieurs zones ayants la charge de faciliter la spécification d'une architecture logicielle. Les zones principales sont :

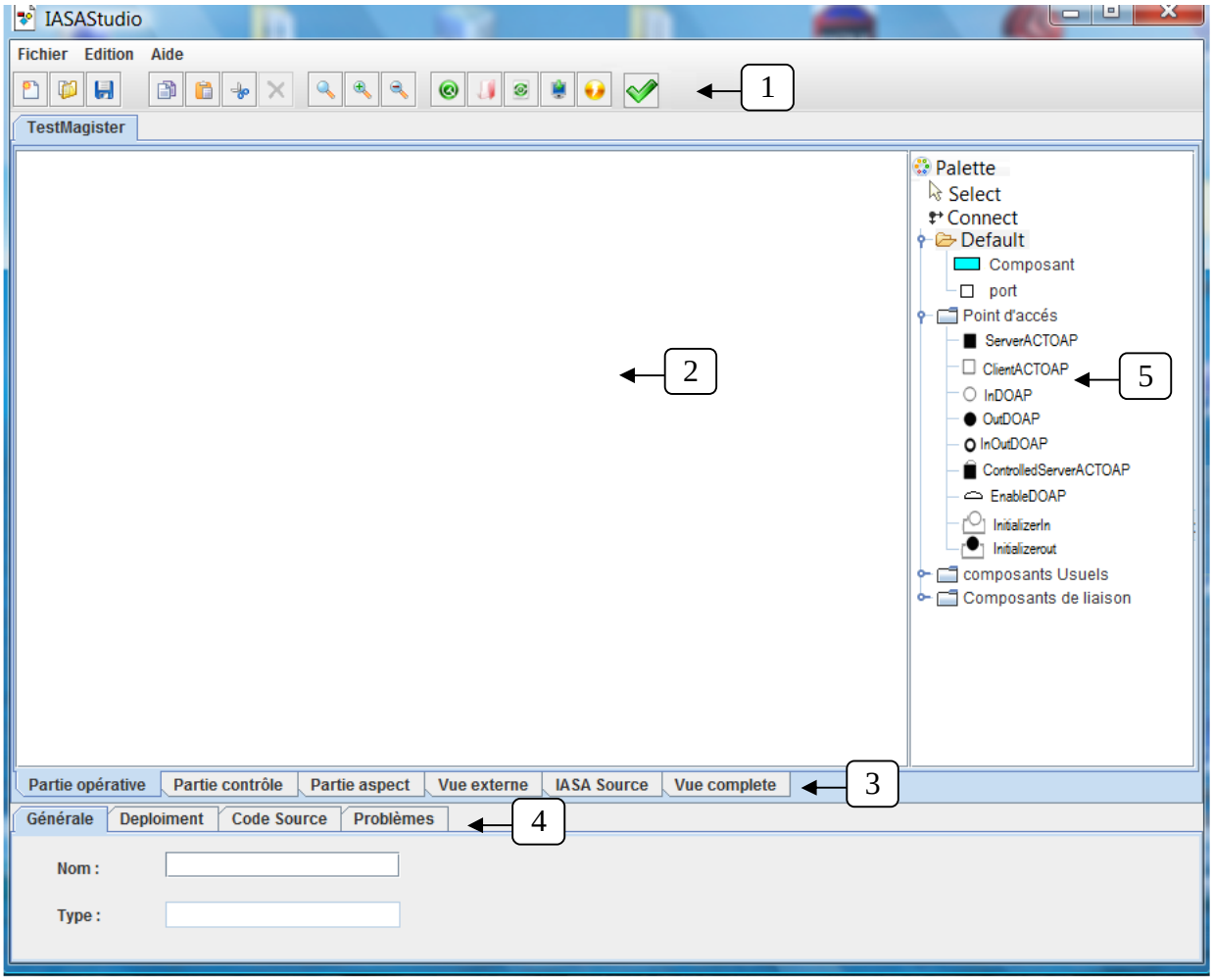
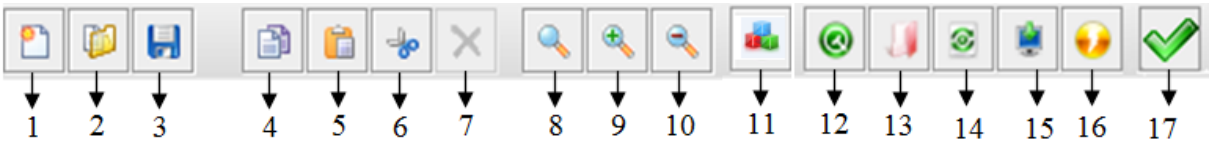


Figure 5.1: la Fenêtre principale de IASAStudio.

- **<u>Zone 1 :</u>** représente la barre d'outils de IASASTUDIO (Figure 5.1), elle est constituée de l'ensemble des boutons suivant :
  - ✓ Bouton 1: pour la création d'un nouveau projet.
  - ✓ Bouton 2: bouton pour ouvrir un projet.
  - ✓ Bouton 3 : pour sauvegarder le projet en cours.
  - ✓ Boutons 4, 5, 6: pour permettre les opérations de copie coller des éléments sélectionnés.
  - ✓ Bouton 7: permet de supprimer l'élément sélectionné.
  - ✓ Boutons 8, 9, 10: pour zoomer sur les éléments architecturaux.
  - ✓ Bouton 11 : permet d'importer des composants de la bibliothèque de composants.
  - ✓ Bouton 12: permet de spécifier le type de l'élément sélectionné.
  - ✓ Bouton 13: permet spécifier le contexte d'action.
  - ✓ Bouton 14: permet spécifier le comportement d'un port.

- ✓ Bouton 15: permet la spécification du déploiement des instances des composants.
- ✓ Bouton 16: permet la spécification de l'interconnexion d'un connecteur.
- ✓ Bouton 17 : pour la production du code exécutable.



- **Zone 2 :** cette zone correspond à la zone de dessin où sont disposés les éléments de l'architecture.
- **Zone 3 :** cette zone est composée de l'ensemble des onglets suivants :
  - ✓ Partie opérative : affiche les composants qui forment la partie opérative du projet (composant composite).
  - ✓ Partie contrôle : affiche les composants de la partie contrôle du projet (composant composite).
  - ✓ Partie Aspect : affiche les composants aspect du composant composite.
  - ✓ Vue externe : affiche la vue externe du projet (composant composite).
  - ✓ Vue complète : affiche la vue complète du composant composite.
  - ✓ Description X3ADL : permet de visualiser la spécification logicielle sous forme textuelle (en X3ADL).
- **Zone 4 :** cette zone contient l'ensemble des onglets suivants :
  - ✓ Propriétés Générales : affiche les noms des éléments d'architecture ainsi que le type.
  - ✓ Propriétés du déploiement : affiche l'état de déploiement du composant,
  - ✓ Connexion : affiche la connexion interne établie entre le composant hardware et le composant software.
  - ✓ Problèmes : signale les problèmes qui peuvent exister dans le projet.
  - ✓ Code X3ADL : permet de visualiser le code X3ADL de l'élément sélectionné.
- **Zone 5 :** correspond à la palette d'outils de spécification IASA, elle fournit une galerie de figures qui permet la spécification graphique des éléments de base de notre approche en faisant glisser les formes dans la zone de dessin.

5.3 Représentation du modèle de composant dans IASASTUDIO

La création d'un nouveau composant composite dans IASASTUDIO correspond à la création d'un projet IASASTUDIO dont le nom est celui du composant composite. Chaque projet comporte des onglets permettant de considérer le composant composite sous diverses perspectives. Ainsi, il est possible de voir ensemble les trois parties de la vue interne du modèle de composant (partie opérative, partie contrôle et partie aspect), sa vue externe et aussi, de voir individuellement chaque partie.

Chaque projet IASASTUDIO, sauvegarde la spécification graphique de ces éléments constructifs (composant, port, point d'accès) dans un document XML afin de pouvoir consulter, modifier, ou importer ces éléments pour les réutiliser par exemple dans d'autres projets. Chaque élément d'architecture est caractérisé par une notation graphique, comme présentée ci-dessous :

Les composants :

Dans IASASTUDIO les composants primitifs sont représentés par un rectangle ayant une broche pour chacun des ports qui sont associés à eux, avec le nom du composant (nom d'instance) qui centre ce rectangle (Figure 5.2). Pour distinguer les composants de chaque partie du composite nous avons précisé une couleur spécifique à chaque partie : les composants de la partie opérative sont de couleur bleue, les composants de la partie contrôle sont de couleur verte et enfin les composants de la partie aspect sont de couleur orange.

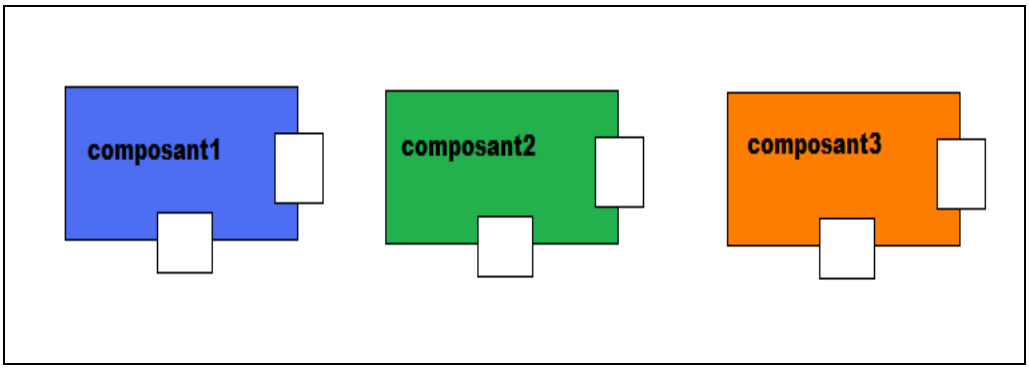


Figure 5.2: Représentation graphique du composant.

De plus, l'IDE IASASTUDIO propose une bibliothèque prédéfinie de formes chacune associée à une fonctionnalité bien précise. Cette bibliothèque peut être extensible (Figure 5.3). Ainsi parmi les fonctionnalités de l'IDE, nous pouvons trouver la gestion des formes associées au composant et même aux autres éléments fondamentaux pour la spécification d'architecture (Figure 5.3).

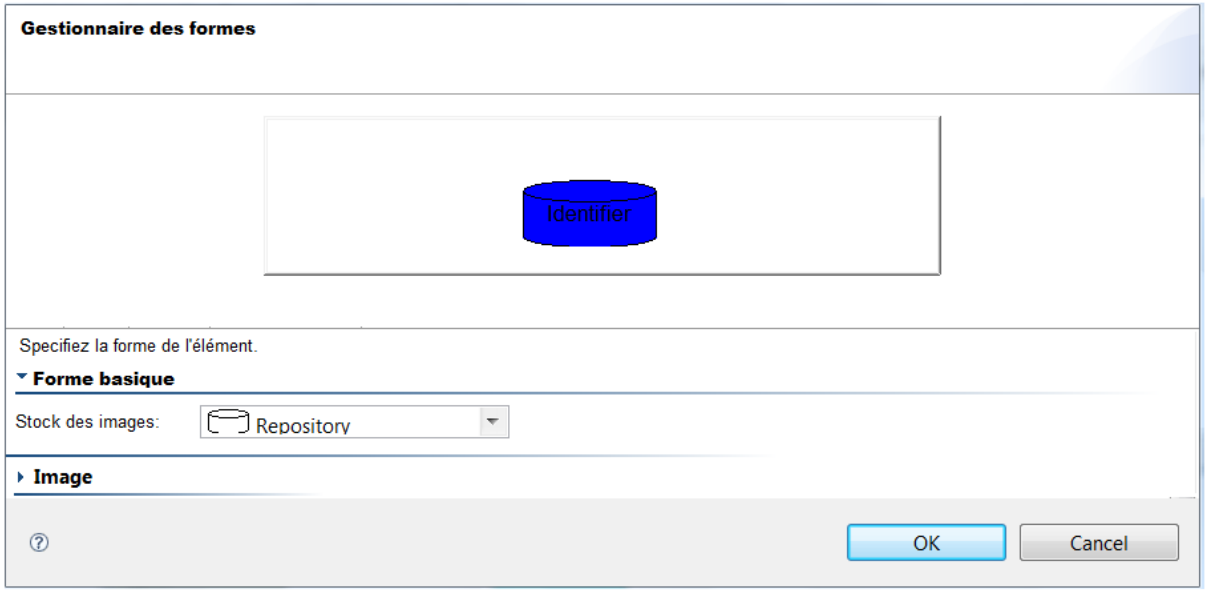


Figure 5.3: bibliothèque de formes

Les propriétés spécifiques aux composants (déploiement, type etc.) peuvent être spécifiées en utilisant le menu contextuel (Figure 5.4).

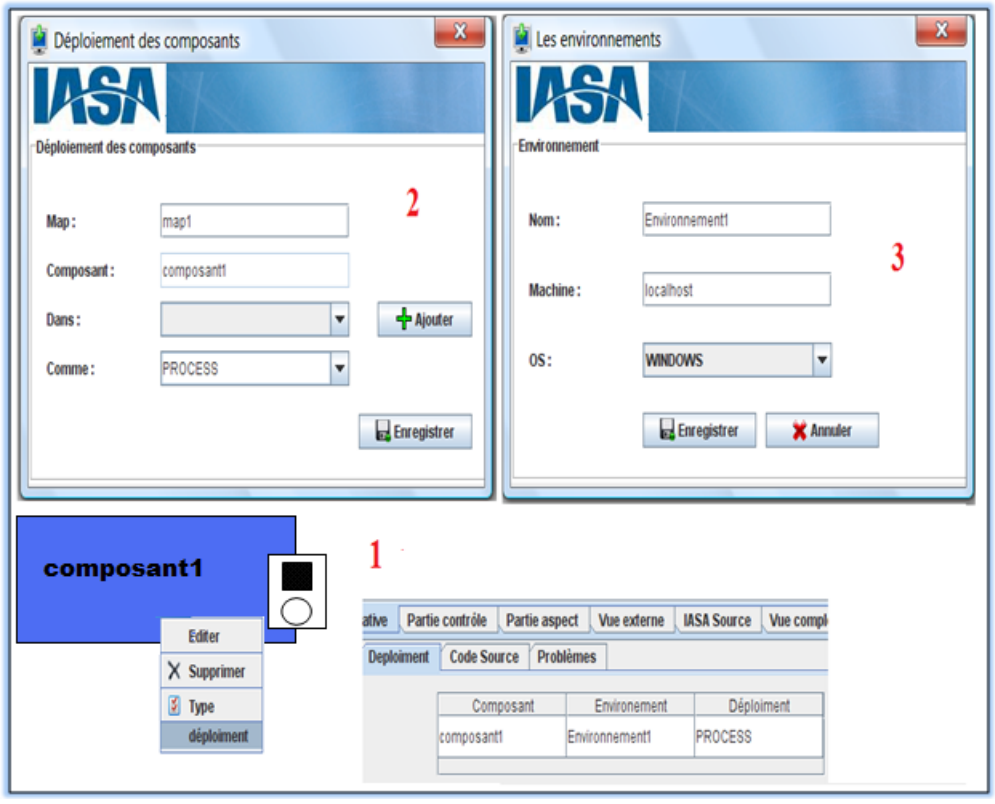


Figure 5.4: Spécification des proprités du déploiement

Les points d'accès :

Chaque point d'accès possède sa propre représentation graphique. Les DAOP sont représentés par des cercles et les ACTOAP ont la forme d'un carré (Figure 5.5).

Figure 5.5: Représentation graphique des points d'accès

Les propriétés des points d'accès peuvent être remplis à l'aide de la fenêtre des propriétés telles que présentées dans la figure 5.6.

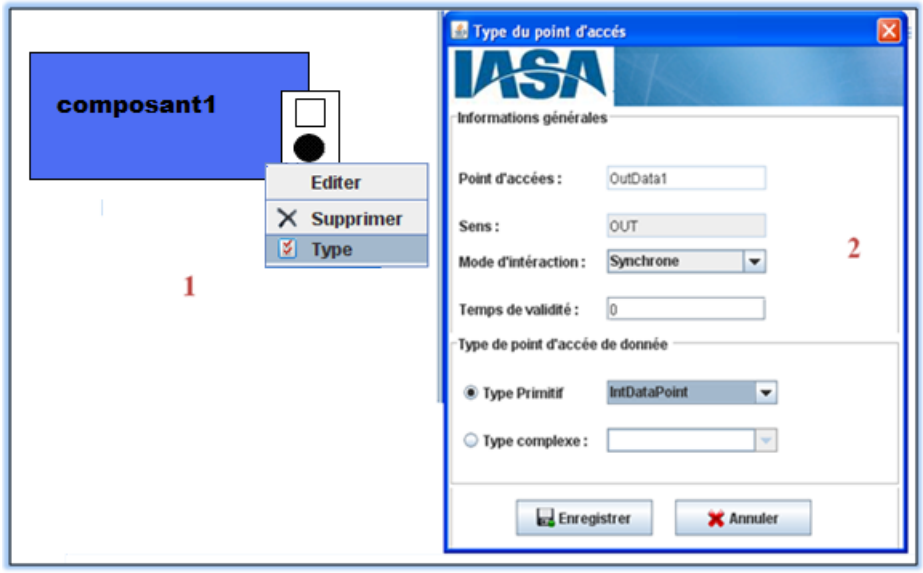


Figure 5.6: Spécification des propriétés des points d'accès

Les ports :

Les ports possèdent une forme rectangulaire blanche. Comme le nombre de ports n'est pas fixé ils peuvent être ajoutés autant que nécessaire aux composants. Le port est un regroupement de point d'accès donc ces point d'accès sont ajoutés autant que nécessaire au port. Dans IASASTUDIO, nous pouvons basculer facilement d'une vue détaillée du port qui laisse apparaitre ces éléments constructifs à une vue moins détaillée. L'utilisation de la fenêtre de propriétés permet de définir le type du port ainsi que les différentes services (actions) associés au port, puis leurs arguments et leurs propriétés peuvent être définies en utilisant la fenêtre des actions, telles que présentées dans les étapes 3, 4 et 5 de la figure 5.7.

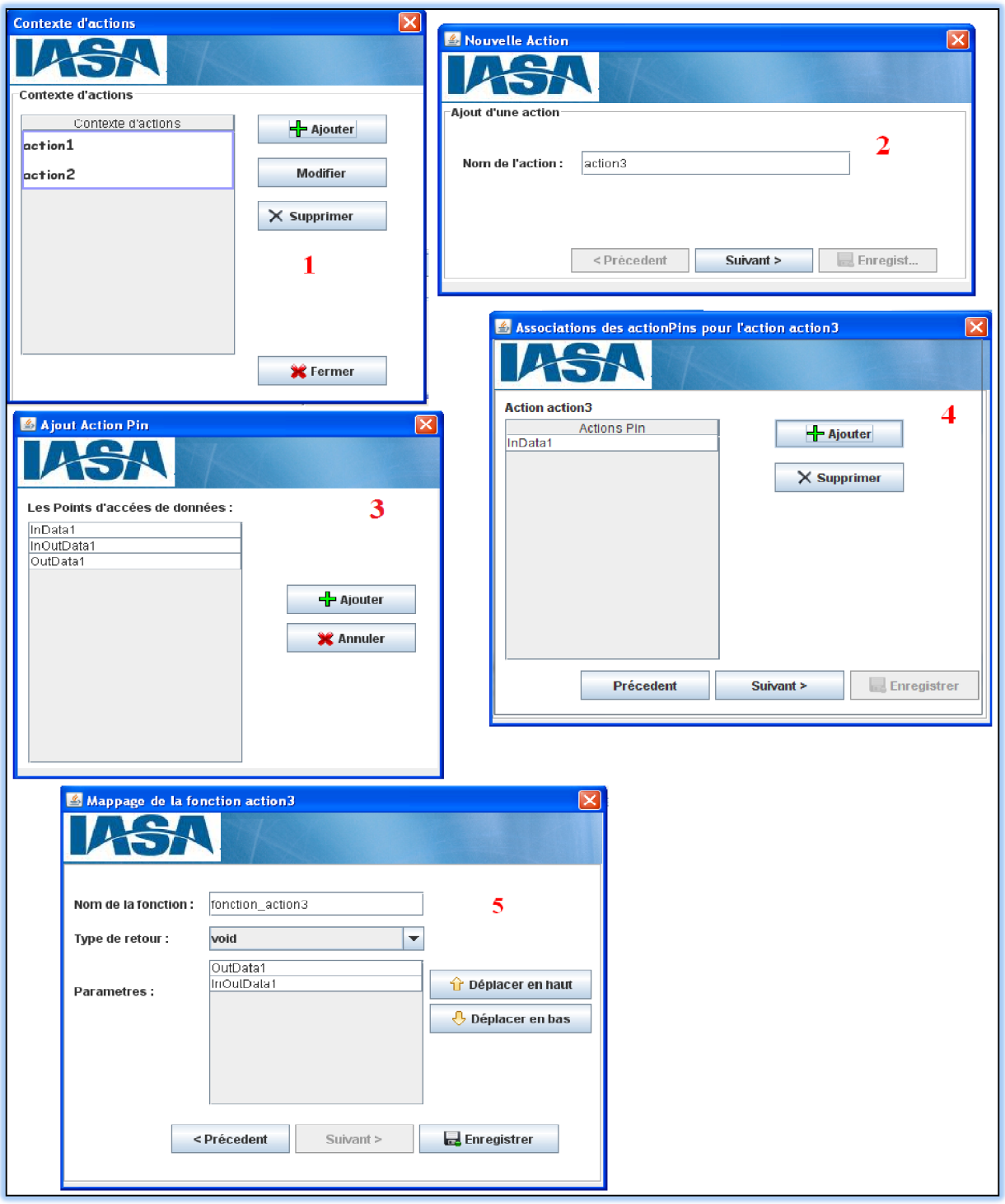


Figure 5.7: Définition des services d'un port

Les connecteurs :

Les connecteurs sont représentés par un trait qui relie les points d'accès. Les connecteurs aspects sont représentés par un trait bleu discontinu. Un bus qui interconnecte des ports est représenté par un trait épais. Parmi les pratiques que nous avons intégrer au niveau de IASASTUDIO est la possibilité de tirer une connexion d'un bus vers un point

d’accès et la possibilité de tirer un bus ou une partie du bus vers un port. Concernant les connecteurs complexes, ils sont représentés par un composant de communication. Ce composant possède la même structure qu’un composant ordinaire. Il est représenté par un rectangle au niveau duquel un certain nombre de ports sont placés. Au niveau de l’IDE il sera ainsi possible de passer d’une vue sous forme de composant à une vue sous forme de trait et vis versa.

Le vrai défit auquel a fait face notre approche est le fait de gérer la cohérence de l’architecture spécifie par l’architecte, tout en laissant une certaine flexibilité d’exprimer ses besoins. L’architecte doit être guidé durant son travail d’élaboration de l’architecture, en le sollicitant à fournir les détails de son architecture de manière progressive. Cette prise en charge se matérialise dans le contexte de notre approche par l'automatisation et le contrôle des diverses décisions de l’architecte à l’aide des messages interactifs (boites de dialogue) avec l’architecte. Ces messages permettent de réduire les sources potentielles pouvant engendrer des erreurs (fenêtres d’avertissement) qui peuvent apparaitre durant le processus de conception (Figure 5.8), ou bien pour permettre l’affectation de sens bien précis à des comportements que peut avoir l’architecte, l'IDE peut donc déterminer automatiquement les éléments nécessaires pour que la spécification spécifiées par l’architecte devienne formelle, cet aspect concerne les fenêtres de spécification et de raffinement (Figure 5.7 , Figure 5.9).

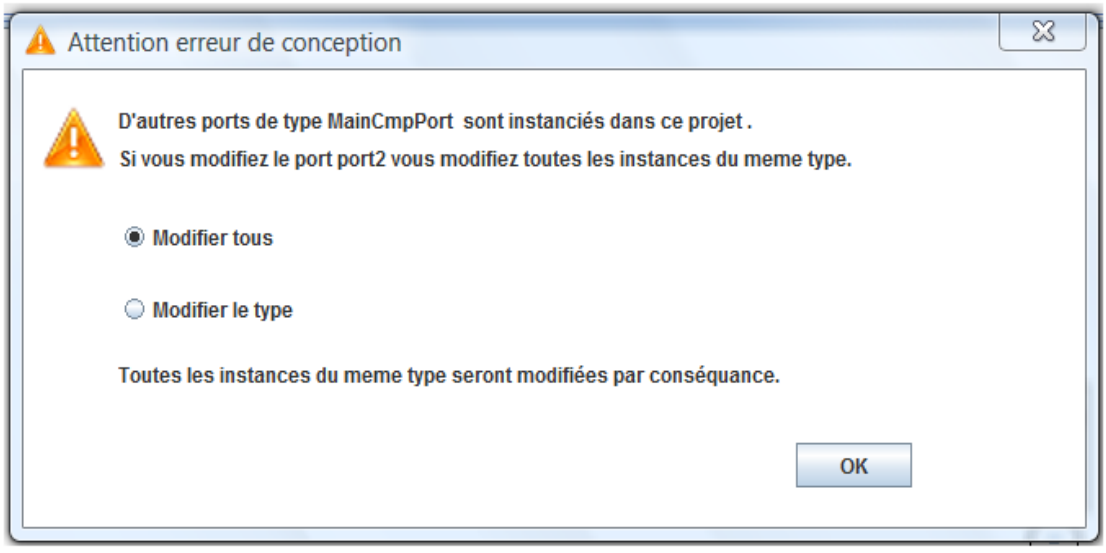


Figure 5.8: Boite de dialogue de gestion des états exceptionnels

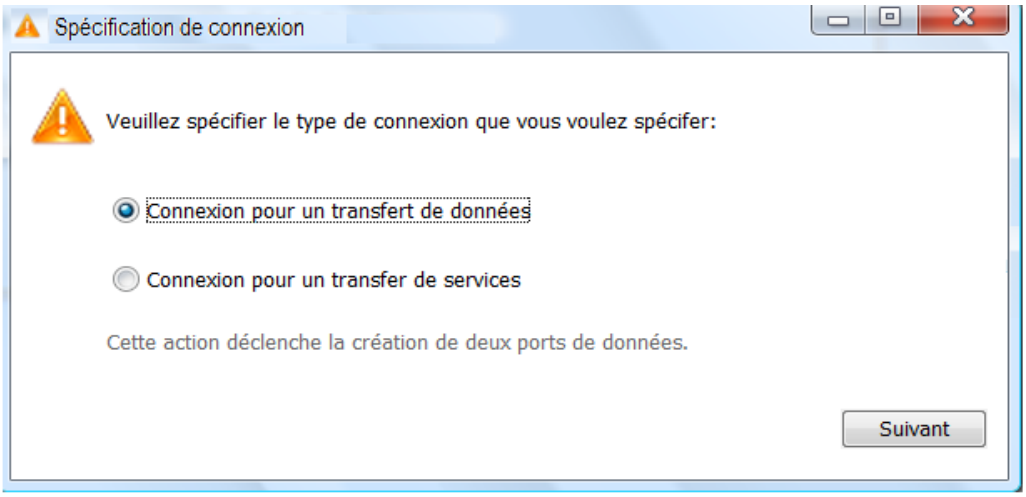


Figure 5.9: Boite de dialogue de spécification de connexion

5.4 Choix techniques d'implémentation

Pour l'implémentation de notre IDE, nous avons utilisé Java comme langage de programmation. Quand on met en avant la portabilité et la sûreté du langage, Java se révèle un environnement de choix qui se révèle très efficace. Ce langage a également l'avantage d'être maintenant très répandu, et de disposer de très bons outils de développement. Par ailleurs, le développement en Java suscite l'intérêt d'un nombre croissant de chercheurs.

Concernant les transformations de modèles, de nombreux langages sont à ce jour, disponibles. Dans le cadre de notre approche et dans le but d'automatiser les transformations abordées dans le chapitre précédent, notre choix s'est porté vers le langage XML couplé avec le langage Java. Un tel choix d'un langage généraliste (de programmation) pour écrire les transformations au lieu d'un langage conçus spécifiquement pour faire de la transformation (tel que ATL), est justifié par le fait que les langages de programmation tel que Java ont l'avantage d'être déjà largement utilisés, ce qui leur a permis d'atteindre un certain niveau de maturité. Contrairement au langage dédié à la transformation qui nécessite un effort et une bonne maitrise du langage afin de réaliser les transformations.

5.5 Exemple d'élaboration d'architecture logicielle dans IASASTUDIO

Notre travail a été évalué dans le cadre de la réalisation d'un composant composite le contrôleur de licence. Dans ce qui suit nous allons détailler pas à pas la réalisation de ce composite dans IASASTUDIO, en mettant l'accent sur le processus de transformation dans IASASTUDIO pour obtenir le du code java du composite.

5.5.1 Le composite LicenseControleur dans IASASTUDIO

Le composite *LicenseControleur* (Figure 5.10) est un lanceur de logiciel qui commence par vérifier si la licence d'utilisation est valide. Une licence est bâtie à partir de l'adresse MAC d'un adaptateur de réseau Ethernet et est comparée au fichier licence fourni avec le logiciel.

Le composant de licence est ainsi composé, en plus du composant de la partie contrôle le Starter, des composants suivants: le lecteur d'adresse MAC (*EtherAddrReader*), le générateur d'information de licence (*LicenseGen*), le chargeur des informations de licence à partir d'un fichier (*LicenseLoader*), et le comparateur des informations de licence (*ComparatorCmp*) représenté par le symbole = =. Nous remarquons dans cet exemple comment les points d'accès sont exploités dans la spécification de ce composant.

Le port client du composant *starter* et le port de service du composant *EtherAddrReader* sont liés par un connecteur de port. Le port client du composant OpPartController et le port de service du composant *EtherAddrReader* correspondent à une méthode avec un seul paramètre. Les DOAP de part et d'autre du connecteur correspondent à ce paramètre. Ce dernier pourrait être une valeur de retour de méthode ou un paramètre ordinaire. Le composant starter doit obtenir la *LicenseData* du composant *LicenseGen*, puis le *LicenseData* du composant *LicenseLoader* avant de les soumettre au composant *ComparatorCmp*.

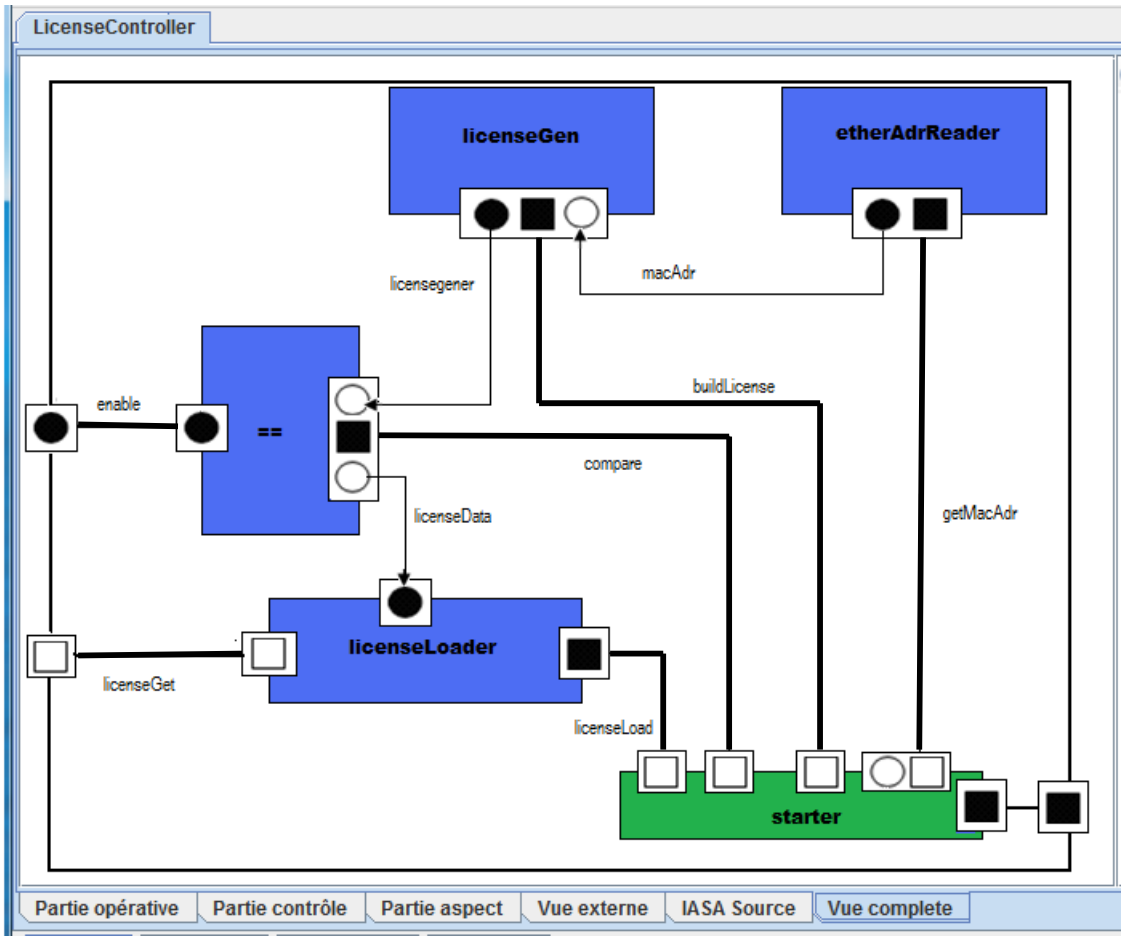


Figure 5.11: le composite LicenseControlleur réalisé dans IASASTUDIO

Le DOAP du port de service du composant EtherAddrReader est directement connecté à un DOAP du port du composant LicenseGen. Les DOAP vont correspondre dans le processus de normalisation à un service de transport représenté par des accesseurs. Le type exact de l'accesseur dépendra du sens du connecteur. La flèche du connecteur est toujours associée au DOAP contenant le service de transport. Ainsi, pour le connecteur entre EtherAddrReader et LicenseGen le service est un accesseur set qui doit être déclenché par le fournisseur de la donnée. Ce type de connexion apparaît aussi entre le port de données du composant opérateur de comparaison (= =) et le composant LicenseReport. La direction du connecteur indique que le service de transport est localisé au niveau du composant LicencReport et le point de données indique qu'il est l'origine de la donnée. Dans ce cas le service de transport est implémenté par un accesseur get. Le lancement de l'application se fait à travers un port de contrôle. Ainsi le lancement effectif n'aura lieu que si le port reçoit une information d'autorisation de fonctionnement. Cette information est le résultat de la du composant ComparatorCmp qui correspond à la valeur prédéfinie ENABLE pour lancer effectivement l'application.

Une fois la spécification graphique du composite est réalisée, sa spécification x3ADL est instanciée et le fichier x3ADL correspondant est enregistré dans le projet. Ainsi, la spécification logicielle peut être alors visualisée dans l'onglet Description x3ADL.

5.5.2 Le tissage d'aspect

Nous allons dans cet exemple injecter un aspect de journalisation (log) dans le composant ComparatorCmp (Figure 5.11) :

```
- <optionpart>
 - <pointcuts>
     <joinpoint name="log" aspect=" ComparatorCmp.compare " />
   </pointcuts>
 - <advices>
     <advice aspect="logCmp .ServerAction1" type="after" joinpoint="log" />
   </advices>
</optionpart>
```

Cette injection de l'aspect *log* (l'advice est fourni par le composant aspect logCmp après la pointcut log) entraîne la création d'un ClientASPOAP *log* dans le port *serveur* du composant ComparatorCmp ainsi que l'établissement d'une connexion aspect entre le port contenant le ClientASPOAP client *log* et le port de l'advice logCmp.

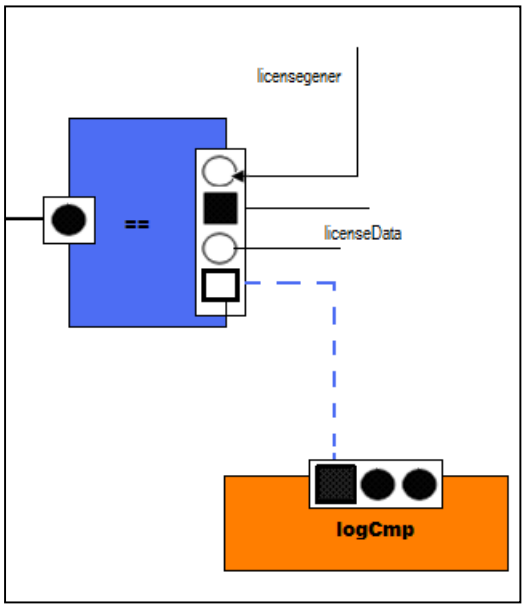


Figure 5.11: Tissage d'aspect de journalisation dans IASASTUDIO

5.5.3 Normalisation du composant LicenseControlleur

Dans le processus de transformation de la description architecturale du composant composite le contrôleur de licences, nous avons utilisé un ensemble de règles de transformation que nous avons détaillé dans le chapitre précédent, afin de produire une

description x3ADL basée sur les concepts ordinaires de port et d'interface. Nous présentons dans ce qui suit les plus importantes de ces transformations.

Chaque action d'un ACTOAP correspond à une méthode. A titre d'exemple les actions *compare* (sur le port serveur du composant *ComparatorCmp*), *buildLicense* (sur le port *pMain* du composant *LicenseGen*), *getMacAdd* (sur le port du composant *EtherAddrReader*) correspondent chacune à une méthode.

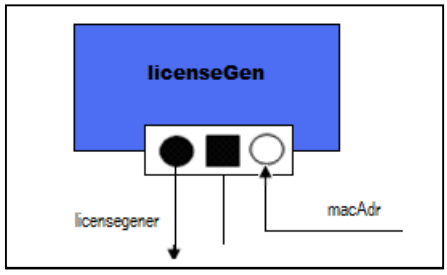


Figure 5.12: Accès individuel au DOAP

Concernant l'accès individuel aux DOAP d'un port selon la méthode de transformation du modèle de port selon notre approche : Lorsqu'un DOAP est utilisé de manière individuelle, ce DOAP sera représenté par un port. Ce port correspondants au DOAP est doté d'un accesseur *get* ou *set*. La détermination de la dépendra du sens du point d'accès comme ceci a été introduit au chapitre 4. Lorsque le sens du DOAP est *in* (*out*) la méthode correspond à l'accesseur *get* (*set*). Ainsi l'action de transfert peut être enclenchée par l'un ou l'autre port.

Si nous considérons le schéma de la figure 5.12, le processus de transformation génère donc trois ports:

- Un port correspondant à l'action buildLicense
- Un port pour le point d'accès macAddr
- Un port pour le point d'accès licenseData

Pour les ports qui correspondent aux points d'accès, le style de nommage consiste à utiliser le nom absolu du point d'accès.

5.5.4 Génération du code java

Une fois la description X3ADL en termes de ports et d'interfaces correspondant au composite est réalisée, la transformation vers le code Java est alors immédiate. Afin de développer ces modèles de génération de code, un ensemble de templates a été identifié pour

générer le code java pour chacun des concepts de IASA, selon les règles de transformation définis dans le chapitre 4.

C'est ainsi que les DOAP sont représentés par une classe Java ordinaire, conçue selon le style introduit au chapitre 4. La classe doit fournir un attribut qui contiendra la valeur du DOAP, les accesseurs et les constructeurs. Les éléments essentiels de la classe DOAP sont illustrés dans les morceaux de code suivants.

```
package iasa.ports;

public abstract class DOAP {
	Port portContainer;
	Object valeur;
	boolean syncModeCom;//mode de communication
	int timeValidity; //-1 = ne pas de vérifier le temps de validité
	final int IN = 1, OUT = 2, INOUT = 3;
	int dir; // IN, OUT, INOUT
	boolean isIn(){return dir == IN;}
	boolean isOut(){return dir == OUT;}
	boolean isInOut(){return dir == INOUT;}
	boolean accessValidity; // true si timeValidiy est différent de 0
	boolean isSynchrone(){return syncModeCom;}
	boolean isTimeOut(){ return timeValidity == 0; }
	Object getValeur(){return valeur;}
	void setValeur(Object valeur){this.valeur= valeur;}
}
```

Si nous considérons le composant LicenseGen, le code java qui lui correspondant est décrit par la classe LicenseGen.

```
package iasa.component.primitive;
import iasa.port.*

public class LicenseGen implements buildLicense_Mainport,
		licenseData_Mainport, macAddr_Mainport {

	StringDOAP macAddr, licenseData, buildLicense;

	@Override
	public String getMacAddr (){
		// TODO Auto-generated method stub
		return macAdrr.getValue();
	}
	Override
	public void setLicenseData (String licenseData){
		// TODO Auto-generated method stub
		this.licenseData.setValue(licenseData);
	}
	Override
	public String buildLicense(String licenseData, String macAddr){
		// TODO Auto-generated method stub
		return buildLicense.getValue();
	}
}
```

Le code des ports buildLicense_MainPort, macAddr_MainPort et licenseData_MainPort est décrit dans la portion de code ci-dessous :

```
package iasa.port;

public interface macAddr_mainPort extends DataPort{// Sens est IN

      public String getMacAddr ();
}
public interface licenseData _mainPort extends DataPort{// Sens est OUT

      public void setLicenseData (String licenseData);
}
public interface buildLicense _mainPort extends ServerPort{

      public String buildLicense(String licenseData, String macAddr);
}
```

Enfin, l'enveloppe dans la quelle est instancié et déployé le composant LicenseGen autant que tâche est décrite par la classe LicenseGenImpl, cette dernière hérite de la classe Thread :

```
package iasa.impl.enveloppe;
import iasa.component.primitive.*

public class LicenseGenImpl extends Thread{

      LicenseGen licenegen;

      public buildLicenseImpl(){
            start();
      }
      public void run() {
            licenegen =new Calccmp();
         }
      public String getMacAddr(){
            return licenegen.getMacAddr ();
      }
      public void setLicenseData(String licenseData){
            Licenegen.setLicenseData (licenseData);
      }
      public String buildLicense(String licenseData, String macAddr){
            return licenegen.buildLicense.(licenseData, macAddr);
      }
}
```

5.6 Conclusion

Dans ce chapitre nous avons présenté notre contribution dans l'outil IASASTUDIO. Nous avons organisé cet IDE en plusieurs vues, et nous avons développé un module important, qui consiste en la prise en charge de la génération du code Java. Notre contribution vise à simplifier le travail de l'architecte lors de la définition de l'architecture logicielle de son système, en mettant à sa disposition un support complet des diverses décisions architecturales tout au long du processus de raffinement d'une architecture jusqu'à la génération automatique des fichiers de codes qui sont enregistrés dans l'emplacement du projet.

L'idée de base est d'assister l'architecte au fur et a mesure de l'avancement de son projet, de façon à s'approcher, petit à petit, du système final, ce qui consiste à ajouter de plus en plus de détails à l'architecture abstraite d'une façon automatique ou assistée jusqu'à ce que l'architecture soit assez détaillée pour pouvoir être implémentée sans ajouter d'informations nouvelles en explicitant les règles de passage d'une phase à une autre.

CONCLUSION

Dans ce travail de recherche, nous avons présenté une nouvelle approche de spécification d'architecture logicielle capable de supporter la spécification directe des modèles mentaux d'architecture élaborés par les architectes dans les premières phases d'un processus de développement de logiciel. Le modèle mental dépend beaucoup des comportements, de la manière d'élaboration d'un architecte. Il dépend beaucoup de l'art de l'élaboration d'une solution basée sur une décomposition architecturale.

En effet, La résolution d'un même problème par la spécification architecturale peut être très différente d'un architecte à un autre. Les différences résident principalement dans l'approche d'élaboration, le choix des composant et les techniques utilisées pour spécifier les diverses interconnexion entre composants, leurs sémantiques peuvent avoir plusieurs déclinaisons.

Après l'étude des différents langages de spécification d'architecture logicielle et des modèles de composant existants, nous avons fait ressortir plusieurs limites qui font face à une spécification flexible des architectures logicielles, en empêchant de prendre en compte le modèle mental de l'architecte.

Afin de surmonter ces problèmes, nous avons proposé dans ce mémoire une approche de spécification d'architecture logicielle, capable de prendre plus aisément le modèle mental des architectes. Ce dernier est souvent spécifié sous forme de figures (le plus souvent des rectangles) et des liaisons entre ces figures. L'approche proposée est capable d'accepter ces diverses formes d'expressions d'une architecture ainsi que les diverses décisions et comportements architecturales et de transformer cette spécification hautement informelle en une spécification formelle. La formalisation se fait dans le cadre de l'approche IASA (Integrated Approach for Software Architecture) qui se distingue des autres approches non seulement par son modèle de composant et de connecteur qui est indépendant de la plateforme et de tout mécanismes logiciels, notamment ceux dédiés aux interactions mais aussi elle propose un modèle de port qui permet de manipuler librement ses composants constructifs qui n'est pas le cas avec les autres approches. C'est à travers ces caractéristiques ainsi que l'enrichissement des notations de IASA que nous avons développé, que nous pouvons spécifier aisément les différents comportements que peut avoir l'architecte durant les premières phases du processus d'élaboration d'un logiciel.

A ce niveau, la conception de systèmes à l'aide de X3ADL que nous avons développé pour notre approche peut être considérée comme une approche de conception à la MDA (Model Driven Architecture) où les structures des systèmes logiciels sont modélisées sous la forme de descriptions d'architecture. L'approche MDA nous a permis d'appliquer les techniques de transformation de modèle afin d'obtenir un logiciel exécutable à partir d'une spécification architecturale en spécifiant les différentes étapes et règles de transformation.

Le fait que notre approche intègre un processus de raffinement et de transformation selon MDA, nous permet de tirer des avantages à savoir un gain de productivité grâce à la spécification des règles de transformation servant à la génération de code, et une détection rapide des erreurs au moment de la conception ce qui rend plus facile la maintenance du système.

Pour expérimenter et valider notre démarche, nous avons développé un IDE de spécification graphique d'architecture logicielle, appelé IASASTUDIO. Cet outil permet d'accueillir facilement les éléments du modèle mental de l'architecte et de générer en temps réel la description x3ADL. Cette description x3ADL est transformée via une série de transformation en code exécutable.

Le travail de recherche que nous venons de réaliser ouvre un nombre important de perspectives, notamment dans le domaine de raffinement de l'architecture logicielle à cause de la complexité de la conversion de l'architecture abstraite à l'architecture spécifique. Les travaux qui sérés menés pour cette perspective autour du processus de transformation selon MDA, concerne la définition de toutes les règles de correspondances et de transformation vers d'autres plateformes cibles afin d'obtenir le code exécutable autres que le code Java.

Enfin, nous tenons à préciser que pour l'instant notre travail s'est concrétisé par une communication acceptée pour présentation à une conférence internationale :"Towards a flexible model driven software architecture", Second International Conference on Computer Science, Engineering and Applications, CCSEA-2012 , India.

REFERENCES

1. R. N. Taylor, N. Medvidovic et E. M. Dashofy, “Software Architecture : Foundations, Theory, and Practice”,John Wiley & Sons, 2009.
2. J. N. Buxton et B. Randell, éditeurs. Software Engineering Techniques : Report of a Conference Sponsored by the NATO Science Committee. Rome, Italie, 27-31 octobre 1969, Brussels, Scientific Affairs Division, NATO, 1970.
3. P. Kruchten, H. Obbink et J. Stafford. The Past, Present, and Future for Software Architecture. IEEE Software, 23(2):22–30. IEEE Computer Society Press, 2006.
4. P. Kruchten. The Rational Unified Process : an Introduction. AddisonWesley, 2003.
5. M. Shaw et P. Clements. The Golden Age of Software Architecture. IEEE Software, 23(2):31–39. IEEE Computer Society Press, 2006.
6. Software Engineering Institute. Community Software Architecture Definitions. http://www.sei.cmu.edu/architecture/start/community.cfm, 2010.
7. Software Engineering INSTITUTE. How do you define software architecture? http://www.sei.cmu.educ/architecture/definitions.html.
8. Sylvain Chardigny , « Extraction d’une architecture logicielle à base de composants depuis un système orienté objet Une approche par exploration », THÈSE DE DOCTORAT ,Université de Nantes, 2010
9. Adel Smeda, Mourad Oussalah, and Tahar Khammaci. A multi-paradigm approach to describe complex software system, WSEAS Transactions on Computers, 3(4) :936–941, October 2003.
10. A. SMEDA, « Contribution à l’élaboration d’une métamodélisation de description d’architecture logicielle », Thèse de Doctorat, Université de Nantes, 2006.
11. Cyril Carrez, « Contrats Comportementaux pour Composants », Phd thesis, l’école Nationale Supérieure des Télécommunications, Spécialité : Informatique et Réseaux, 2003.
12. Elisabetta Di Nitto and David Rosenblum, “Exploiting ADLs to specify architectural styles induced by middleware infrastructures”. In ICSE’99: Proceeding sof the 21st international conference on Software engineering, pages13–22, Lo sAlamitos, CA, USA, 1999. IEEE Computer Society Press.
13. Bass Len, Clements Paul, and Kazman Rick, “Software Architecture in Practice”, Addison-Wesley, second edition edition, April 2003.
14. Jölle Coutaz and Laurence Nigay, « Architecture logicielle conceptuelle des systèmes interactifs ». *Analyse et conception de l’IHM*. Hermés, chapitre 7 :207– 246, 2001.

15. Sylvain Chardigny, « Extraction d'une architecture logicielle à base de composants depuis un système orienté objet Une approche par exploration », Thèse de doctorat, université de Nantes, 2010

16. [Nassima Sadou-Harireche, « Evolution Structurelle dans les Architectures Logicielles à base de Composants », Thèse de doctorat, Université de Nantes ,2008

17. R. Koschke, "Atomic Architectural Component Recovery for Program Understanding and Evolution ", Thèse de doctorat, Université de Stuttgart, 2000.

18. Daniel Le Métayer, "Describing software architecture styles using graph grammars", IEEE Transactionson Software Engineering, 24(7) :521–533, 1998.

19. Reiko Heckel, Alexey Cherchago, and Marc Lohmann, "A formal approach to service specification and matching based on graph transformation", Electronic Notes in Theoretical Computer Science, 105 :37–49, 2004.

20. Paul Ziemann, Karsten Hölscher, and Martin Gogolla, "From UML models to graph transformation systems", Electronic Notes in Theoretical Computer Science, 127(4):17–33,2005.

21. Gerald H. Hilderink, "Graphical modelling language for specifying concurrency based on CSP", IEE Proceedings-Software,150(2) :108–120, 2003.

22. Gwen Salaun, Michel Allemand, and Christian Attiogbe "A method to combine any process algebra with an algebraic specification language : the p-calculus example", In COMPSAC'02 : Proceedings of the26th International Computer Software and Applications Conference on Prolonging Software Life : Development and Redevelopment, pages 385–392, Washington, DC, USA, 2002. IEEE Computer Society.

23. Virginia C. Carneiro De Paula, George R. Ribeiro-Justo, and P.R.F.Cunha, "Specifying and verifying reconfigurable software architectures", In PDSE, pages 21–31, 2000.

24. Etienne Roblet, Khalil Drira,and Michel Diaz, "Formal designand development of a corba-based application for cooperative HTML group editing support", Journal of Systems and Software,60(2) :113–127, 2002.

25. P. Clements et M. Shaw, "The Golden Age of Software Architecture" Revisited, IEEE Software, 26(4):70–72, doi : 10.1109/MS.2009.83.. IEEE Computer Society Press, 2009.

26. Julien Mercadal « Approche langage au développement logiciel : application au domaine des systèmes d'informatique ubiquitaire »,thèse de doctorat, université de Bordeaux, 2011

27. Olivier Barais, Anne Françoise Le Meur, Laurence Duchien, and JuliaLawall, « Software Architecture Evolution », inria-00371226,version 1, Mars 2009

28. Jonathan Aldrich, Craig Chambers, and David Notkin , "ArchJava: Connecting Software Architecture to Implementation",In ICSE'02, , 2002, Orlando, Florida, USA.

29. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, “Documenting Software Architectures : Views and Beyond”, (2nd ed.). Addison Wesley, 2010.

30. Mohamed HADJ KACEM, « Modélisation des applications distribuées à architecture dynamique: Conception et Validation », Thèse de doctorat, Université de Toulouse, 2008

31. P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton Jr. N degrees of separation : Multidimensional separation of concerns. In International Conference on Software Engineering, pages 107–119, 1999. 2, 15, 26, 128

32. R. Allen, “A Formal Approach to Software Architecture”, PhD thesis, Carnegie Mellon, School of Computer Science, janvier 1997. Issue das CMU Technical Report CMU-CS-97-144.

33. C. A. R. Hoare, “ Communicating Sequential Processes”, Prentice-Hall, 1985.

34. C.A.R.Hoare, “An axiomatic basis for computer programming”, Commun .ACM,12(10):576–580, octobre 1969. 21, 33

35. Olivier Barais, « Construire et Maitriser l'Evolution d'une Architecture Logicielle a base de Composants », Thèse de doctorat, Université de Lille, 2005

36. N. Medvidovic and R.N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages”, IEEE Transactions on Software Engineering, Volume 26 Issue 1, doi>10.1109/32.825767, January 2000

37. Guide to the Rapide 1.0 Language Reference Manuals, Rapide Design Team Program Analysis and Verification Group Computer Systems Lab Stanford University, Juillet 1997.

38. Jeff Magee, Naranker Dulay, and Jeff Kramer, « A constructive development environment for parallel and distributed programs”, In IWCCS’94 : Proceedings of the IEEE Work shop on Configurable Distributed Systems,North Falmouth, Massachusetts, USA, Mars 1994.

39. Jeff Magee, Naranker Dulay, and Jeffrey Kramer, « Structuring parallel and distributed programs”, IEEE Software Engineering Journal, 8(2) :73–82, Mars 1993.

40. Projet ACCORD, etat de l’art sur les langages de description d’architecture (ADLs). Technical Report Livrable1.1-2, RNTL, France, Juin 2002.

41. Magee J, Dulay N, Kramer J A , « Constructive Development Environment for Parallel and Distributed Programs », in Proc. of the IEEE Intn'l Workshop on Configurable Distributed Systems (IWCCS'94), Pittsburgh PA, USA, Mars 1994.

42. D. Garlan, R. Monroe, D. Wile, « ACME : An Architecture Description Interchange Language », Computer Science Department, Carnegie Mellon University, Pittsburg et USC/information science institute, Novembre 1997.

43. D. Garlan, R. Monroe, D. Wile, « ACME : Architectural Description of Component-Based Systems », Computer Science Department, Carnegie Mellon University, Pittsburg et USC/information science institute, 1997.

44. Jonathan Aldrich, Craig Chambers, et David Notkin, “Architectural reasoning in Archjava”, In ECOOP ’02: Proceedings of the 16th European Conference on Object-Oriented Programming, pages 334–367, London, UK, 2002. Springer-Verlag.

45. Jonathan Aldrich, Craig Chambers, et David Notkin, “ArchJava: Connecting Software Architecture to Implementation”, In ICSE, pages 187–197. ACM, 2002.

46. D. Bennouar, T. Khammaci, et A. Henni, « The Design of a Complex Software System with ArchJava », Journal of Computer Science, 2(11):807–814, september 2006. Science Publications.

47. Luc FABRESSE,« Du découplage à l’assemblage non-anticipé de composants Conception et mise en œuvre du langage à composants SCL », thèse de doctorat, Université de Montpellier II ,2007

48. Thais Batista, Christina Chavez, Alessandro Garcia, Uira Kulesza, Claudio Sant'Anna, and Carlos Lucena, « Aspectual connectors: Supporting the seamless integration of aspects and adls”, In ACM SIGSoft XX Brazilian Symposium on Software Engineering (SBES'06), Florianopolis, Brazil, 2006.

49. M. Graiet, M.T. Bhiri, J.P. Giraudin, and N. Belkhatir, « Architecture des systèmes avec la norme UML2.0 et l’ADL Wright », In INFORSID, pages 927–942, June 2006.

50. Eric LE PORS, « Interprétation sémantique des exigences pour l’enrichissement de la traçabilité et pour l’amélioration des architectures de systèmes complexes », thèse de doctorat, Université de Bretagne,2010.

51. E. Bruneton, T.Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani, « An open component model and its support in java, In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors,CBSE,volume 3054 of Lecture Notes in Computer Science,pages7–22.Springer,2004. ISBN:3-540-21998-6.

52. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma et J.-B. Stefani, « The fractal component model and its support in java », Software Practice and Experience, special issue on Experiences with Auto-Adaptive and Reconfigurable Systems, Vol. 36, p. 11–12, 2006.

53. L. Seinturier, N. Pessemier, L. Duchien et T. Coupaye, « A component model engineered with components and aspects », in Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE’06), Vol. 4063 in Lecture Notes in Computer Science, p. 139–153, Springer, juin 2006.

54. J.-P. Fassino, J.-B. Stefani et G. Muller, « Think : A software framework for componentbased operating system kernels », in USENIX 2002 Annual Conference, may 2002.

55. C. Escoffier et D. Donsez, « Fracnet ». http ://www-adele.imag.fr/fractnet/, jan 2008

56. E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, « A highly-extensible, xml-based architecture description language », In WICSA ’01 : Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA’01), page 103, Washington, DC, USA, 2001. IEEE Computer Society. ISBN : 0-76951360-3.

57. P.H. Feiler, B. Lewis, and S. Vestal, « The SAE Avionics Architecture Description Language (AADL) Standard : A Basis for Model-Based Architecture-Driven Embedded Systems Engineering », In RTAS 2003 Workshop on Model-Driven Embedded Systems, 2003.

58. P.H. Feiler, D.P. Gluch, and J.J. Hudak, "The Architecture Analysis & Design Language (AADL):An Introduction", Technical report, CMU/SEI2006-TN-011, February 2006.

59. J. Hugues, B. Zalila, L. Pautet, and F. Kordon, « From the prototype to the final embedded system using the Ocarina AADL tool suite », ACM Transactions on Embedded Computing Systems (TECS) , ACM Press, New York, USA, 7(4) :42 :2–42 :25, July 2008.

60. Nassima SADOU-HARIRECHE , « Evolution Structurelle dans les Architectures Logicielles à base de Composants », Thèse de doctorat, Université de Nantes, 2008.

61. T. Bolusset, « B-Space: Raffinement de description architicturales en machines abstraites de la méthode formelle B », these de doctorat, Université de Savoie, 2004.

62. Object Management Group OMG. Unified Modeling Language (UML) specification, mars 2003. Version1.5. 57.

63. Object Management Group OMG. Unified Modeling Language : Superstructure, août 2003. Version 2.0. 31, 57.

64. Sunghwan Roh, Kyungrae Kim, and Taewoong Jeon, "Architecture modeling language based on UML2.0", In APSEC'04 : Proceedings of the11th Asia-Pacific Software Engineering Conference, pages 663–669. IEEE Computer Society,2004.

65. ThomasWeigert, DavidGarlan, JohnKnapman, BirgerMoller-Pedersen, andBran Selic, "Modeling of architectures with UML (Panel)", In UML'00 : Proceedings of the 3rd International Conference on Unified Modeling Language. Advancing the Standard, York, UK, volume 1939 of Lecture Notes in Computer Science, pages 556–569.Springer,2000.

66. Jorge Enrique Pérez-Martînez and Almudena Sierra-Alonso, "UML1.4 versus UML 2.0 as languages to describe software architectures », In EWSA'04 : Proceedings of1st European Workshop Software Architecture", EWSA 2004, St Andrews, UK, May 21-22, volume 3047 of Lecture Notes in Computer Science, pages 88–102. Springer,2004.

67. David Garlan, "Software architecture : a roadmap", In ICSE'00 : Proceedings of the Conference on The Future of Software Engineering, pages 91–101, New York, USA, 2000. ACMPress.

68. David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek," Reconciling the needs of architectural description with object-modeling notations", Science of Computer Programming, 44(1) :23–49, 2002.

69. Christine Hofmeister and Robert L. Nord, "From software architecture to implementation with UML", In COMPSAC'01: Proceedings of the 25th Annual International Computer Software and Applications Conference on Invigorating

Software Development, pages 113–114, Washington, DC, USA, October 2001. IEEE Computer Society.

70. Mohamed Mancona Kandé and Alfred Strohmeier, “Towards a UML profile for software architecture descriptions”, In UML’00 : Proceedings of the3rd International Conference on Unified Modeling Language. Advancing the Standard, York, UK,volume1939 of LNCS,pages 513–527. Springer,2000.

71. R.K. Pandey, “Architecture Description Languages (ADLs) vs UML: A Review”, ACM SIGSOFT Software Engineering Notes, Volume 35 Number 3, May 2010

72. D. Bennouar, « Une approche intégrée pour l’architecture logicielle », thèse en vue de l'obtention de diplôme docteur d'état en informatique, ESI El Harrache , 2009.

73. D. Bennouar, T. Khammaci and A. Henni, "Modeling The Component’s Interaction Point In The IASA Approach", The Mediterranean Journal of Computer and Networks, Vol. 4, N° 4, UK, 2008.

74. Djamal BENNOUAR et Abderrezak HENNI, « A Review of an Aspect Oriented Architecture Description Language», The Meditaranen Journal of Computers and Networks, Vol 6, N° 1, pp 15-22, 2010, © 2010 SoftMotor Ltd., UK.

75. D. Bennouar and A. Henni, « SEAL: An aspect oriented ADL » , Conference ACIT, Sanaa, Yemen, 2009.

76. Ilhem Khlif , Mohamed Hadj Kacem and khalil Drira, « Une approche de description multi-échelles et multi points de vue pour les architectures logicielles dynamiques », hal-00675654, version 1 – 2,Mar 2012.

77. E. M. Dashofy, A. V. D. Hoek and R. N. Taylor, "An infrastructure for the rapid development of XML-based architecture description languages", Proceedings of the 24th International Conference on Software Engineering, pp. 266- 276, 2002.

78. xADL 2.0, http://www.isr.uci.edu/projects/xarchuci/index.html

79. B. Schmerl, "xAcme: CMU Acme Extensions to xArch", 2001. http://www.cs.cmu.edu/~acme/pub/xAcme/guide.pdf

80. Projet FAROS, Etat d e l’art sur la contractualisation et la composition, Août 2006

81. B. Combemale, Ingénierie Dirigée par les Modèles (IDM)État de l’art, hal-00371565, version 1, 2009.

82. R. DEGUIL, « Mapping entre un référentiel d’exigences et un modèle de maturité :application à l’industrie pharamceutique »,Université Toulouse, 2008.

83. D.Di Ruscio, I. Malavolta, H.Muccini, P.Pelliccione, A. Pierantonio, “Developing next generation ADLs through MDE techniques”, ICSE ’10, 2010.

84. R. Weinreicha , G. Buchgeher, “Towards supporting the software architecture life cycle”, The Journal of Systems and Software, doi:10.1016/j.jss.2011.05.036, 2011.

85. R. Eramo, I. Malavolta ,H. Muccini , P. Pelliccione , A. Pierantonio , “A model-driven approach to automate the propagation of changes among Architecture Description Languages”, Softw Syst Model,DOI 10.1007/s10270-010-0170-z, 2012.

86. J. Castroa, M. Lucenab, C. Silvac, F. Alencara, E. Santosa, J. Pimentela, “Changing attitudes towards the generation of architectural models”, The Journal of Systems and Software, doi:10.1016/j.jss.2011.05.047, 2011.

87. OMG. MDA guide version 1.0.1, juin 2003, http://www.omg.org/docs/omg/03-06-01.pdf

88. Ludovic MENET, « Formalisation d’une approche d’Ingénierie Dirigée par les Modèles », Thèse doctorat, Université de PARIS VIII, 2010.

89. Jean Bézivin and O.Gerbé, “Towards a precise definition on the omg/mda framework, In ASE’01, 2001.

90. Guy TURCHA NY, « La théorie des systèmes et systémiques: Vue d’ensemble et définitions », http://www.prof-turchany.eu/culture/La_theorie_des_systemes.pdf , 2008.

91. Xavier Blanc, MDA en action, Groupe Eyrolles, Paris, 2005.

92. IRISA, Kermeta-breathe life into your metamodels, http://www.kermeta.org/ , 2008.

93. OMG. Meta object facility (mof) core specification, http://www.omg.org/docs/formal/06-01-01.pdf, 2006.

94. OMG. Omg’s meta object facility, http://www.omg.org/mof , 2003.

95. Object Management Group OMG. MOF QVT Final Adopted Specification, 2005, Version 2.0.

96. Ed Merks Raymond, Ellersick Frank Budinsky,David Steinberg and Timothy J.Grose, Eclipse Modeling Framework. Hermes - Lavoisier, 2006.

97. Hubert Kadima, Conception orienté objet guidée par les modèles, Dunod, 2005.

98. K.K. Sandhu, “Specification and description language (SDL) ”, In Formal Methods and Notations Applicable to Telecommunications, IEE Tutorial Colloquium , 1992.

99. J. Bézivin, M. Blay, M. Bouzhegoub, J. Estublier, J.M. Favre, S. Gérard, and J.M. Jezequel. Rapport de Synthèse de l’AS CNRS sur le MDA (Model Driven Architecture). Rapport CNRS, janvier 2005.

100. OMG. Uml 2 metamodel. ptc/04-10-05 (UML 2.0 Superstructure FTF Rose model containing the UML 2 metamodel), 2004.

101. T. Dinh, O. Gerbé, and H. Sahraoui, « Un métamétamodèle pour la gestion de modèles », In IDM 06 Actes des 2èmes Journées sur l’Ingénierie Dirigée par les Modèles, Lille, France, 2006.

102. OMG. OMG Unified Modeling Language (OMG UML). http://www.omg.org/doc/formal/07-11-04.pdf, 2007.

103. Jean Bézivin. Ecole d’été d’informatique cea edf inria , cours de jean bézivin, ingénierie des modèles logiciels, http://www.aristote.asso.fr/Presentations/CEAEDF2003/Cours/JeanBezivin/IndexJeanBezivin.html, 2008.

104. B. Combemale, « Approche de métamodélsation pour la simulation et la vérification de modèle. Application à l'ingénierie des procédés », Thèse de doctorat, Université de IRIT ENSEEIHT, 11, 2008

105. ATLAS group LINA &INRIA. ATL: Atlas Transformation Language. ATL User Manual, - version 0.7, Nantes, 2006.

106. W3C.XML Schema Part 0: Primer Second EditionW3C Recommendation, novembre 2004. Version 1.1.

107. J. Estublier, J-M. Favre, J. Bézivin, L. Duchien, R. Marvie, S. Gérard, B. Baudry M. Bouzhegoub, J-M. Jézéquel, M. Blay, and M. Riveil. Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles. Rapport de synthèse 1.1.2, CNRS, janvier 2005.

108. W3C. XSL transformations (XSLT) version 2.0, avril 2005.

109. OMG. UML1.5Object Constraint Language Specification, mars 2003. Version 1.5.

110. W3C. XQuery 1.0 : An XML Query Language Version 2.0, novembre 2005.

111. Mónica Pinto, Lidia Fuentes, Luis Fernández ,"Deriving detailed design models from an aspect-oriented ADL using MDD", The Journal of Systems and Software, doi:10.1016/j.jss.2011.05.026, 2011.

112. Richard Hubert, Arcstyler: the architectural ide for mda, http://www.iosoftware.com/.

113. Objecteering Software. Objecteering. http ://www.objecteering.com/.

114. Compuware. Optimalj. http ://www.compuware.com/products/optimalj/.

115. Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf, "Tool integration at the métamodèle level within the fujaba tool suite", International Journal on Software Tools for Technology Transfer(STTT), 6(3) :203–218, August 2004.

116. Mia-Software. Mia-transformation. http://www.mia-software.com/.

117. Pathfinder Solutions. Pathmate : Transformation engine. http://www.pathfindermda.com/.

118. D. Vojtisek and J.-M. Jézéquel, "MTL and Umlaut NG - engine and framework for model transformation" ERCIMNews 58, 58, juillet 2004.

119. AndroMDA. Andromda. http://www.andromda.org/.

120. Laurence Tratt. Qvteclipse. http://qvtp.org/downloads/qvtp-eclipse/.

121. EMF. The eclipse modeling framework project home page. http://www.eclipse.org/emf.

122. Juha-Pekka Tolvanen and Matti Rossi, "Metaedit+ : defining and using domainspecific modeling languages and code generators", In OOPSLA '03 :

Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages,andapplications, pages 92–93, New York, NY, USA, 2003. ACM Press.

123. TOPCASED, Guide méthodologique pour les transformations de modèles, Rapport de recherche, version 0.1, 18 Novembre 2008, IRIT/MACAO , http://www.topcased.org

124. Dalila Guessoum, Djamal Bennouar, “Towards A Flexible Model Driven Software Architecture”, In CCSEA-2012 ,Delhi India, 2012.

APPENDICE A : DTD PROPOSEE POUR X3ADL

```
< ! ELEMENT  Component (Ports,Connectors, Operativepart, Controlpart,Optionpart, Properties)  >
< ! ELEMENT  Operativepart  (components)  >
< ! ELEMENT  Controlpart    (components)  >
< ! ELEMENT  Optionpart     (components, pointcuts ,advices  >
< ! ELEMENT  Properties     (architecture, deployment)  >
< ! ELEMENT  components   (component)* >
< ! ELEMENT  connectors   (connetcor)* >
< ! ELEMENT  component   (ports) >
< !ATTLIST    Component
                        name            string       #REQUIRED
                        deployedAs      string       #REQUIRED>
< !ATTLIST    component
                          name    string       #REQUIRED
                          type    string       #REQUIRED>
< ! ELEMENT  ports(port)*>
< ! ELEMENT  Port  (accesspoints,behavior)>
< !ATTLIST    port
                       name       string        #REQUIRED
                       type       string        #REQUIRED>
< ! ELEMENT  behavior (rule)*>
< ! ELEMENT  rule  EMPTY>
< !ATTLIST    rule
                      name             string      #REQUIRED
                      precondition     string      #REQUIRED
                      postcondition    string      #REQUIRED
                      pattern          string      #REQUIRED
                      fail             string      #REQUIRED >
< ! ELEMENT  accesspoints (accesspoint)*>
< ! ELEMENT  accesspoint Empty>
< !ATTLIST    AccessPoint
                       name             string                              #REQUIRED
                       type             string                              #REQUIRED
                      direction         string                              #REQUIRED
                      modeInteraction  ("Synchrone"|"ASynchrone")           #REQUIRED
                      time             int                                 #REQUIRED>
< ! ELEMENT  connector  (actioncontext ,interaction)>
```

```
< !ATTLIST    connector
                        name          string          #REQUIRED
                        source   string          #REQUIRED
                        target    string          #REQUIRED >
< ! ELEMENT  interaction EMPTY>
< !ATTLIST    interaction
                      name                string          #REQUIRED
                      sequenceAction    string          #REQUIRED >

< ! ELEMENT  pointcuts  (joinpoint)*>
< ! ELEMENT  joinpoint  EMPTY >
< !ATTLIST     joinpoint
                        name          string                  #REQUIRED
                        aspect       string                  #REQUIRED >
< ! ELEMENT  advices  (advice)*>
< ! ELEMENT  advice EMPTY >
< !ATTLIST     advice
                        aspect       string                  #REQUIRED
                        type          string                  #REQUIRED
                        joinpoint    string                  #REQUIRED >
< ! ELEMENT  architecture  (environment)*>
< ! ELEMENT   environment  EMPTY>
< !ATTLIST     environment
                        name          string                  #REQUIRED
                        machine              string                  #REQUIRED
                        os                   (“windows”|”Linux”)   #REQUIRED >
< ! ELEMENT  deployment  (case)*>
< ! ELEMENT   case  EMPTY>
< !ATTLIST     case
                          compenent      string                                  #REQUIRED
                          as    (“processus “|”MAINTHREAD”|”APPLET”
                                |”SERVLET”|”EJB”|”JAVASCRIPT”)     #REQUIRED
                          in      string                                  #REQUIRED>
```

ANNEXE B : DTD PROPOSEE POUR X3ADL NORMALISE

```
< ! ELEMENT  Component (Ports, Connectors, Operativepart, Controlpart, Properties)   >
< ! ELEMENT  Operativepart  (components)   >
< ! ELEMENT  Controlpart     (components)   >
< ! ELEMENT  Properties      (architecture, deployment)   >
< ! ELEMENT  components   (component)* >
< ! ELEMENT  connectors   (connectcor)* >
< ! ELEMENT  component    (ports) >
< !ATTLIST     Component
                        name            string        #REQUIRED
                        deployedAs      string        #REQUIRED>
< !ATTLIST     component
                          name     string       #REQUIRED
                          type     string       #REQUIRED>
< ! ELEMENT  ports (port)*>
< ! ELEMENT  Port  (interfaces , behavior)>
< !ATTLIST     port
                       name       string         #REQUIRED
                       type       string         #REQUIRED>
< ! ELEMENT  behavior (rule)*>
< ! ELEMENT  rule  EMPTY>
< !ATTLIST     rule
                      name             string       #REQUIRED
                      precondition     string       #REQUIRED
                      postcondition    string       #REQUIRED
                      pattern          string       #REQUIRED
                      fail             string       #REQUIRED >
< ! ELEMENT  interfaces (interface)*>
< ! ELEMENT  interface (actions)>
< ! ELEMENT  actions (action)*>
< !ATTLIST     interface
                       name             string                              #REQUIRED
                       type             string                              #REQUIRED
                      direction         string                              #REQUIRED
                      modeInteraction   (“Synchrone”|”ASynchrone”)          #REQUIRED
                      time              int                                 #REQUIRED>
< ! ELEMENT  action (parameters)>
```

```
< ! ELEMENT  parameters (parameters*,ReturnType)*>
< !ATTLIST    action
                        name     string     #REQUIRED >
< !ATTLIST    parameters
                        name       string      #REQUIRED
                        type       string      #REQUIRED >
< ! ELEMENT  connector  (actioncontext ,interaction)>
< !ATTLIST    connector
                        name       string       #REQUIRED
                        source  string       #REQUIRED
                        target   string       #REQUIRED >
< ! ELEMENT  interaction EMPTY>
< !ATTLIST    interaction
                      name              string       #REQUIRED
                      sequenceAction    string       #REQUIRED >

< ! ELEMENT  architecture  (environment)*>
< ! ELEMENT   environment  EMPTY>
< !ATTLIST     environment
                        name       string                 #REQUIRED
                        machine            string                 #REQUIRED
                        os                 (“windows”|”Linux”)  #REQUIRED >
< ! ELEMENT  deployment  (case)*>
< ! ELEMENT   case  EMPTY>
< !ATTLIST     case
                          compenent     string                                  #REQUIRED
                          as   (“processus “|”MAINTHREAD”|”APPLET”
                                |”SERVLET”|”EJB”|”JAVASCRIPT”)    #REQUIRED
                          in     string                                  #REQUIRED>
```