THESE

présentée

par

Fabien LEYMONERIE

pour obtenir le diplôme de

DOCTEUR DE L'UNIVERSITE DE SAVOIE

(Arrêté ministériel du 30 Mars 1992)

spécialité

INFORMATIQUE

ASL : un langage et des outils pour les styles architecturaux.

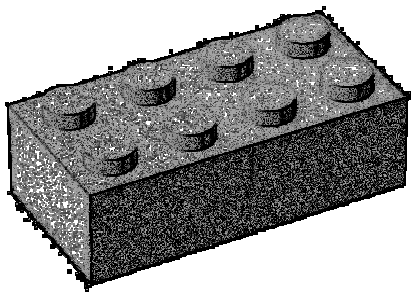
Contribution à la description d'architectures dynamiques.

Soutenue publiquement le 15 décembre 2004 devant le jury composé de :

Noureddine BELKHATIR	Président du jury Rapporteur	Professeur à l'Université de Grenoble II
Mourad OUSSALAH	Rapporteur	Professeur à l'Université de Nantes
Régis DINDELEUX	Examinateur	Directeur associé de Thésame Mécatronique & Management
Flavio OQUENDO	Directeur de thèse	Professeur à l'Université de Savoie
Sorana CÎMPAN	Co-encadrante	Maître de conférences à l'Université de Savoie

Préparée au sein du LISTIC
Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance
Financée par Thésame Mécatronique & Management

A Yolande,



REMERCIEMENTS

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

J'adresse mes plus vifs remerciements à M. Noureddine BELKHATIR, Professeur à l'université Grenoble II et à M. Mourad OUSSALAH Professeur à l'Université de Nantes, pour m'avoir fait l'honneur d'étudier mes travaux de thèse et de les avoir cautionnés en qualité de rapporteurs.

Merci également à M. Régis Dindeleux, Directeur associé de Thésame Mécatronique & Management, pour sa présence dans le jury en tant qu'examinateur et pour la confiance que lui et M. André MONTAUD (Directeur de Thésame) ont placé en moi en m'engageant sur le projet européen ArchWare.

Merci à M. Flavio OQUENDO, Professeur à l'Université de Savoie pour avoir dirigé mes travaux, pour ses conseils et son expérience. Merci notamment pour ses efforts au cours de la rédaction et de la préparation de la soutenance.

Merci, à Mme Sorana CÎMPAN, Maître de Conférences à l'Université de Savoie, pour son encadrement, son soutien, sa complicité et parfois sa patience au cours des nombreuses heures passées ensemble.

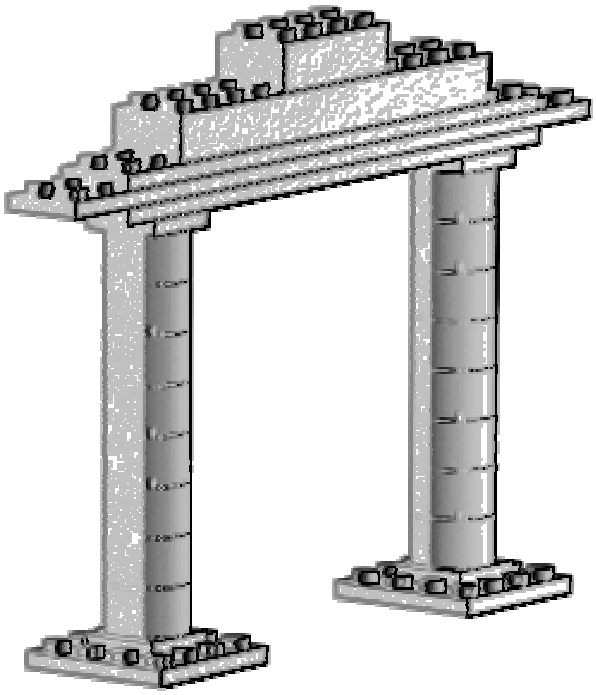
Merci aux différentes personnes avec qui j'ai eu des interactions concernant les travaux de ma thèse pour leurs retours d'expérience.

Merci aux membres du LISTIC pour leur accueil. Merci particulièrement à Lionel, compagnon de galère, au couple Sorana-Hervé, pour leurs multiples conseils, leur soutien moral et les quarts d'heure philosophiques, et à Valérie, pour sa relecture et sa bonne humeur.

Enfin, merci à mes proches pour le soutien et l'aide qu'ils m'ont apportés durant ces trois années.

SOMMAIRE

INDEX DES FIGURES



Chapitre 1

INTRODUCTION

Chapitre 1 - Introduction

L'informatique est présente partout dans notre vie quotidienne. Les applications logicielles étant de plus en plus complexes et nombreuses autour de nous, leur production est de plus en plus assistée et rigoureuse.

Plusieurs générations d'environnements de développement ont vu le jour. Il y a une trentaine d'années, les environnements étaient des boîtes à outils supportant uniquement la programmation ; on utilise le terme "programming in the small" [Li&Ram 85]. Les outils de développement n'étaient pas où peu intégrés, le niveau d'intégration étant lié à l'uniformité des interfaces, à la communication des données et à la complémentarité de leurs fonctions dans le développement du logiciel.

Avec la croissance de la taille et de la complexité des logiciels, ce type de développement n'était plus adapté. Ainsi, il a été remplacé par un développement dit "programming in the large", facilitant le travail d'équipe. C'est avec cette mouvance que la discipline de "programmation" laisse place au "génie logiciel" ; le logiciel prend le pas sur le programme [Hun 81].

Le génie logiciel est la discipline qui traite de la production et de la maintenance des logiciels. Elle a évolué en même temps que les logiciels afin de répondre à leur complexité croissante, leur qualité et leur fiabilité, puis pour améliorer la productivité lors de leur développement.

Dans ce chapitre introductif, nous allons présenter notre problématique, le cadre de notre thèse ainsi que l'organisation de ce manuscrit.

1. Introduction à la problématique

1.1. Besoins des développeurs industriels

La réduction des coûts et des délais tout en améliorant la qualité du produit, font partie des objectifs principaux des industriels. Dans le domaine du logiciel, les industriels visent à permettre ces réductions au stade du développement et au stade de la maintenance, tout en sachant que les systèmes sont de plus en plus grands, complexes et distribués.

Le développement de systèmes complexes demande des approches bien établies qui facilitent la robustesse des produits, l'économie du processus de développement, et une mise rapide sur le marché [DiN&Ros 99]. La phase de conception d'un système prend alors de l'ampleur. La modélisation d'un système permet de raisonner sur ce dernier afin de prévenir les erreurs avant l'implémentation. Elle permet aussi une meilleure compréhension du système et une meilleure organisation de l'équipe de développement. Ainsi, au début des années 90, on assiste à l'établissement d'un domaine de recherche appelé architecture logicielle [Per&Wol 92][Gar&Sha 93].

1.2. Les architectures logicielles

La taille des systèmes ne fait qu’augmenter et le contexte économique est de plus en plus exigeant. Les chercheurs ont démontré l'utilité de formaliser la définition de la structure de haut-niveau des systèmes logiciels et de permettre aux concepteurs d'effectuer des analyses préliminaires sur ces systèmes. De telles analyses ont pour but de découvrir et de résoudre les problèmes de conception dès les premières étapes du développement.
L’architecture logicielle est une discipline récente focalisant sur la structure, le comportement et les propriétés globales d'un logiciel et s’adresse plus particulièrement à la conception de systèmes logiciels de grandes tailles ou de familles de systèmes logiciels.

Les architectures logicielles promeuvent la compréhension, la réutilisation et la prévention afin de garantir de faibles coûts, des délais raisonnables et une meilleure maintenance.

Des Langages de Description d'Architecture (ADL) ont été proposés afin de supporter la description des architectures logicielles. Ils permettent de décrire (à différents degrés de formalité) la structure des systèmes et fournissent différents outils permettant leur réalisation, leurs analyses, leurs simulations,.... S'il s'agit de langages strictement formels, des propriétés sur la structure et le comportement du système peuvent être prouvées. De plus, certains langages autorisent la simulation des systèmes modélisés, afin de vérifier si le comportement obtenu est bien celui recherché, avant l'implémentation de l'architecture. Celle-ci peut, selon le langage utilisé pour la description de l'architecture, générer automatiquement l'application correspondante (éventuellement après plusieurs étapes de raffinement).

La formalisation des architectures promeut la réutilisation (la réutilisation des éléments architecturaux est facilitée) et la compréhension (la description formelle lève toute ambiguïté). Pour aller plus loin et capturer l'expertise de conception pour un domaine particulier, la notion de style architectural est mise en avant.

1.3. Les styles

Un style définit un ensemble de règles générales qui décrivent ou contraignent la structure des architectures et la manière dont les composants interagissent [DiN&Ros 99]. Une architecture peut être définie comme une instance de styles spécifiques.

Les styles permettent de renforcer les bénéfices de l'utilisation des architectures logicielles. Ils fournissent des abstractions en amont de la conception architecturale, formalisant la connaissance et l'expérience propre à un domaine spécifique.

Un style donne :

- une meilleure compréhension,
    - du domaine de conception,
    - des architectures,
- une meilleure réutilisation,
    - le style fournit des éléments et des patrons réutilisables dans la description des architectures,
- une meilleure prévention,
    - un style cadre la conception architecturale à travers un ensemble de contraintes. Ainsi, il est possible d'être averti lorsque l'architecture ne respecte pas un certain nombre de propriétés voulues.

1.4. Le développement des systèmes dynamiques

Les systèmes sont de plus en plus dynamiques (leur topologie change à l'exécution) afin de :

- pouvoir répondre rapidement à de nouveaux besoins,
- pouvoir continuer un traitement critique tout en évoluant,
- alléger les ressources matérielles.

Parmi les systèmes dynamiques, on trouve les systèmes critiques et les systèmes à longue exécution. Les architectures dites dynamiques permettent de planifier à la conception leurs changements topologiques à l'exécution. Par exemple, ces changements sont nécessaires pour que le système soit capable de gérer des défaillances et de basculer sur une architecture plus sûre ; ils sont nécessaires pour pouvoir gérer un environnement sans cesse en évolution, comme c'est le cas pour les systèmes client-serveur dans lesquels les clients se connectent et se déconnectent continuellement.

L'utilité de modéliser la dynamique des systèmes est reconnue [Med 96]. Il est important de pouvoir prévoir l'évolution de l'architecture et de vérifier si cette évolution ne pose pas de problème (il peut par exemple s'agir de problème de ressources matérielles).

Une fois reconnue la nécessité pour certains systèmes d'être dynamiques, une question qui s'impose est de savoir s'il existe une expertise liée à la conception de tels systèmes. Nous avons constaté que plusieurs ADLs offrent la possibilité de représenter le comportement dynamique des systèmes. Cependant, à un niveau plus haut d'abstraction, au niveau des styles architecturaux, il n'y a pas de langage supportant ce genre d'expertise. Cette thèse tend donc d'offrir une réponse à ce constat, par un support pour la représentation des styles architecturaux pour les systèmes dynamiques.

2. Cadre : Projet Européen ArchWare

Les travaux de cette thèse ont été menés dans le cadre du projet européen IST-2001-32360, ArchWare.

L'objectif du projet ArchWare est de répondre à une demande pour les systèmes logiciels. Ceux-ci doivent s'adapter aux besoins d'applications particulières et aux évolutions tout au long de leur vie. ArchWare fournit des solutions consistant en des langages, des modèles de conception et des outils pour l'ingénierie des architectures logicielles évolutives. Les résultats du projet intéressent directement les développeurs de systèmes capables de s'adapter à des demandes fonctionnelles qui évoluent et à des attributs qualité.

Les partenaires du projet sont The Victoria University of Manchester (Angleterre), The University of St Andrews (Ecosse), Engineering Ingegneria Informatica S.p.A. (Italie), CPR - Consorzio Pisa Ricerche (Italie), INRIA Rhônes-Alpes (France), Thésame - Mécatronique et Management (France) et InterUnec - University of Savoie (France).

Dans ce projet, nos travaux consistent essentiellement en deux points. D'abord, fournir un langage de description pour supporter la conception architecturale centrée sur l'utilisation des styles, puis au développement d'un outil d'exploitation des styles, le ASL ToolKit.

3. Organisation du document

Le document est globalement organisé en trois parties.

Une première partie présente deux chapitres. Le premier fournit une étude du domaine et une définition de la problématique de la thèse. Le deuxième est un état de l'art.
Une deuxième partie présente la solution apportée en trois chapitres. Le premier présente ASL, un langage pour la définition des styles architecturaux. Le deuxième porte sur la formalisation d'un style propre à une catégorie d'architectures, les architectures composant-connecteur. Le troisième porte sur un outil d'exploitation des styles architecturaux, l'ASL ToolKit.
La troisième partie porte sur la validation de notre proposition. Cette validation porte sur trois études de cas.
Une conclusion propose une synthèse et un bilan du travail effectué, ainsi qu'un ensemble de perspectives liées à la continuation du travail, aux nouvelles applications et aux nouveaux thèmes de recherche.

Plusieurs annexes sont également fournies en fin de document comme nous pouvons le voir dans la présentation détaillée qui suit.

Chapitre 2

Nous commençons par présenter notre domaine de recherche, les architectures logicielles et plus particulièrement les styles architecturaux. Nous en introduisons le cadre et la terminologie. Puis, nous y définissons la problématique que nous abordons dans cette thèse.

Chapitre 3

Nous présentons un état de l'art des travaux concernant la formalisation des architectures dynamiques et des styles architecturaux. Nous montrons les spécificités de différents systèmes et proposons une classification de ceux-ci.

Chapitre 4

Nous proposons le formalisme ASL, un langage pour la description des architectures et des styles architecturaux. Le chapitre commence par la présentation de l'approche adoptée pour ce formalisme ainsi que les bases formelles sur lequel il repose. Ensuite nous présentons les concepts et la syntaxe sous-jacents : il y est question de constructeurs, de contraintes et d'analyses. Puis, nous proposons une utilisation de ce langage dans le cadre de la conception d'un système ou d'une famille de système.

Chapitre 5

Nous proposons le style Composant-Connecteur et sa formalisation en ASL. Le style Composant-Connecteur est construit sur des concepts propres aux architectures dynamiques. Il définit lui-même une syntaxe pour la description de ces architectures. Nous y présentons aussi un style de base, le Client-Serveur, que nous définissons comme une spécialisation du style Composant-Connecteur.

Chapitre 6

Nous présentons des études de cas validant l'utilisation d'ASL et du style Composant-Connecteur. Chaque cas présente une utilisation différente des styles dans le processus de développement d'une application. Le premier concerne la conception d'un système de gestion des connaissances. Le second concerne la génération d'environnement de développement pour des systèmes de gestion de processus métiers. Le dernier concerne la conception d'une interface homme-machine pour un système de contrôle de sécurité d'un accélérateur de particules.

Chapitre 7

Nous présentons un outil associé au langage ASL : l'ASL ToolKit. Nous y montrons l'architecture de l'outil, les principes de son implémentation, comment il est intégré dans l'environnement ArchWare. Nous montrons son utilisation à travers un scénario.

Chapitre 8

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

Annexe 1

Cette annexe offre une présentation du π-calcul [Mil 89].

Annexe 2

Cette annexe offre une présentation du μ-calcul [Koz 83].

Annexe 3

Nous donnons la BNF du langage ASL.

Annexe 4

Nous donnons les règles de typage du langage ASL.

Annexe 5

Nous donnons l'implémentation des styles de fondation : Client-Server, Pipe-Filter, Data Indirection et Layered style.

Chapitre 2

ETUDE DU DOMAINE

Chapitre 2 - Etude du domaine

Dans la dernière décennie, l'architecture logicielle a émergée comme une notion centrale dans le développement logiciel des gros systèmes complexes. La principale caractéristique des architectures logicielles réside à être des abstractions gérables mais significatives d'un système en développement. Dans le cycle de vie d'un système logiciel, l'architecture logicielle est la première représentation de la structure et du comportement d'un système. La vue statique, structurelle, décrit comment un système est construit par des sous-systèmes interconnectés appelés composants. La vue dynamique, comportementale, spécifie comment ces composants interagissent à l'exécution pour atteindre les objectifs du système. Les architectures logicielles permettent de valider très tôt les choix de conception en considérant les besoins fonctionnels et non fonctionnels. De plus, les architectures logicielles jouent un rôle significatif à travers le cycle de vie d'un développement de logiciel, de l'analyse et de la validation du cahier des charges, à la conception puis jusqu'à l'implémentation et l'exécution [Ber&Inv 03].

Des modèles architecturaux ont été codifiés et réutilisés de manière primaire à travers la transmission informelle d'idiomes architecturaux et de patrons organisationnels - de styles architecturaux [Gar 95][Met&Gra 92][Gar&Sha 93][Gam et al. 94]. Ainsi, on parle de système client-serveur, de tableau noir, de pipeline, d'interpréteur ou de système en couches.

Dans ce chapitre, nous présentons le domaine des architectures logicielles (section 2) et notamment les concepts sur les styles architecturaux (section 3). A la suite de cela, nous mettons en valeur dans le domaine, des points qui nous ont semblés insuffisamment abordés, ce qui nous permet de définir la problématique de cette thèse (section 4).

La section suivante présente un historique, introduisant le domaine architectural.

1. Historique

Parmi les objectifs principaux des industriels on a toujours la réduction des coûts et des délais. Dans le domaine du logiciel, les industriels visent à permettre ces réductions au stade du développement et au stade de la maintenance, tout en sachant que les systèmes sont de plus en plus grands, complexes et distribués.
Depuis le début de l'informatique, les techniques de développement ont évoluées ; les langages de programmation ont suivi cette évolution. On recherche à minimiser le travail nécessaire et à améliorer le travail d'équipe pour en obtenir des logiciels de qualité. Pour cela, on recherche à améliorer la compréhension d'un système par l'ensemble des acteurs, à réutiliser le plus possible les acquis, à éviter les erreurs et à les corriger le plus tôt possible.

La figure suivante schématise l'évolution des concepts au cours des cinquante dernières années. Ce schéma [Gar 03] traduit notamment l'évolution d'une programmation "linéaire" vers une programmation "modulaire".

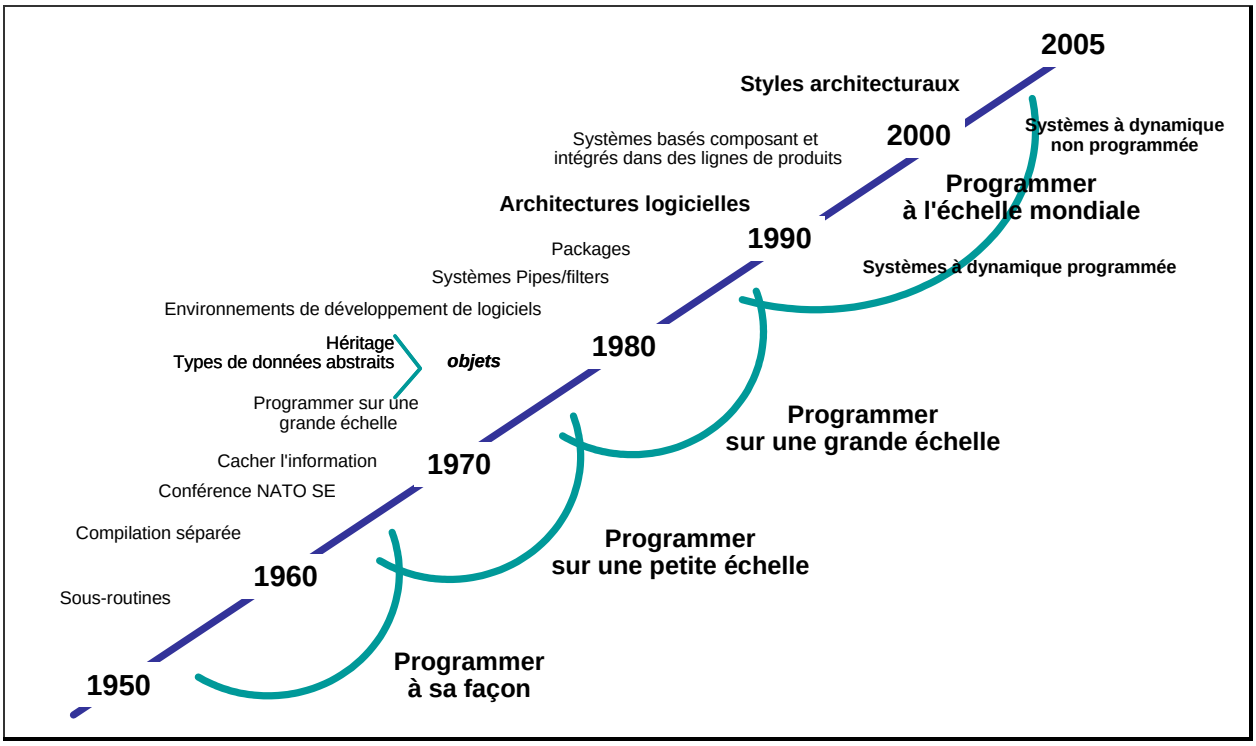


Figure 1 - Evolution du développement de logiciels

Les premiers ordinateurs sont nés à la fin des années 40. L'assembleur est né en 1950 pour donner à l'être humain les premières facilités pour programmer : des symboles textuels remplacent alors le code binaire. Avec l'assembleur vient la notion de sous-routines, premiers pas vers la réutilisation du code. En 1957, naît le premier langage de programmation universel : le FORTRAN (FORmula TRANslator). Dans les années 60, sont nées des techniques pour programmer plus facilement et en équipe. Une application peut alors être programmée sur plusieurs fichiers liés à la compilation.
La recherche dans le domaine de la conception logicielle, attira les chercheurs dans les années 70 [Per&Wol 92]. Elle émergea pour répondre à des problèmes, reconnus dans les années 60, apparaissant lors du développement de systèmes logiciels de grande échelle. La conception apparaît alors comme une activité séparée de l'implémentation, demandant des notations, des techniques et des outils spécifiques.
Dans les années 80, la recherche dans le génie logiciel évolua vers l'intégration de la conception logicielle dans les processus de développement de logiciels et leur gestion. Les années 80 ont aussi vu une grande avancée dans notre capacité à décrire et analyser des systèmes logiciels. On a vu émerger des techniques de description formelle et des notions sophistiquées de typage qui nous permettent de raisonner plus efficacement sur les systèmes logiciels.
L'architecture logicielle apparaît au début des années 90 lorsque des équipes de recherche identifient un domaine émergent et appellent à la recherche dans cette voie [Shaw 90][Per&Wol 92][Gar&Sha 93]. Le terme "architecture" est utilisé en contraste avec "conception" pour évoquer les notions de codification, d'abstraction, de standard, d'entraînement formel (des architectes logiciels), et de style. Plusieurs résultats sont attendus avec l'émergence de l'architecture logicielle comme discipline majeure. Au cours du développement d'un système, sa compréhension diminue tandis que le coût dû aux erreurs augmente. Le but est de préserver la compréhension du système au cours du temps, de détecter et de corriger les erreurs le plus tôt possible. L'architecture doit être un cadre pour la satisfaction du cahier des charges. L'architecture doit être la base technique pour la conception comme la base gestionnaire pour l'estimation des coûts et du processus de gestion. L'architecture doit être une base performante pour la réutilisation. L'architecture doit être la base pour les analyses de dépendance et de cohérence.

2. La conception orientée architecture

L'approche caractéristique dans les disciplines d'ingénierie est de construire des systèmes à partir de solutions connues telles que des modèles de conceptions prouvés et des composants existants [Kog&Cle 95, Sha 90, DIp 89]. Les ingénieurs dans les disciplines mûres ont la volonté d'éviter de manière proactive des problèmes coûteux en évaluant le système avant qu'il soit implémenté [LiuMcG&Epp 87]. Le corollaire en génie logiciel est de modéliser des architectures logicielles.

Un problème critique dans la conception et la construction de tout système logiciel complexe est son architecture, c'est à dire l'organisation des éléments qui composent le système. Une bonne architecture peut aider à garantir que le système va satisfaire les besoins primordiaux dans des domaines tels que la performance, la fiabilité, la portabilité, et l'interopérabilité. Au contraire, une mauvaise architecture peut avoir des conséquences désastreuses sur le système [Gar 00]. C'est pourquoi, au cours de la dernière décennie, les architectures logicielles ont fait l'objet d'une attention croissante, elles sont ainsi devenues un important sous domaine du génie logiciel.

En pratique, une conception architecturale remplie deux rôles principaux [Mon et al 97]. Premièrement, elle fournit un niveau d'abstraction tel que les concepteurs des systèmes logiciels puissent raisonner sur le comportement du système (fonction, performance, confiance, etc). En s'abstrayant des détails de l'implémentation, une bonne architecture offre une vue du système plus lisible et expose les propriétés les plus cruciales pour un développement efficace. C'est souvent le document technique qui permet de déterminer si un système répond au cahier des charges.
Deuxièmement, une architecture sert de "conscience" à un système au cours de son évolution. En caractérisant les assomptions cruciales d'un système, une bonne architecture guide le processus d'amélioration d'un système, indiquant quels aspects du système peuvent être changés sans compromettre l'intégrité du système.

Nous allons voir ce qu'est une architecture logicielle, pour ensuite montrer un processus de développement centré architecture. Puis, nous présenterons les avantages de ce type de conception.

2.1. Qu'est ce qu'une architecture logicielle?

Plusieurs définitions du terme "architectures logicielles" sont proposées dans la littérature [Boa 95][BasCle&Kaz 99][Gar et al. 92][Per&Wol 92], mais il est largement accepté qu'une architecture soit définie par un ensemble de composants (ex : filtres, objets, bases de données, serveurs, etc.) ainsi que par la description des interactions entre ceux-ci (ex : appels de procédures, envois de messages, émissions d'événements, etc.) [Gar et Sha 93]. Celle-ci permet de spécifier les caractéristiques d'un système en définissant : les types de modules composant le système ; combien de composants de chaque type peut avoir ; les interactions entre les composants [RDT 97] ; les propriétés structurelles et comportementales devant être respectées.
La description architecturale d'un système informatique permet en outre de spécifier :

- sa structure : éléments de traitement et leurs interactions (pas de détail d'implémentation),
- son comportement : fonctionnalités et protocoles de communication, dynamisme,
- ses propriétés globales (exemples : sécurité, vivacité, etc …).

Dans cette section, nous donnons plus d'information sur cela. D'abord, nous donnons plusieurs définitions. Puis, nous comparons les architectures logicielles avec celles d'autres domaines. Nous donnons par la suite de plus amples détails sur ce que définit une architecture, sur ce qu'est une architecture composant-connecteur et les architectures dynamiques.

2.1.1 Plusieurs définitions

Il n'y a pas de définition standard et universellement acceptée pour le terme "architecture logicielle", bien que dans la littérature on en trouve un grand nombre. Le site web de l'institut de génie logiciel (SEI) à Carnegie Mellon répertorie un ensemble d'une centaine de définitions allant des plus classiques aux plus modernes et propose aux visiteurs de rajouter leurs propres définitions [Cle et al. 02]. Ceci est dû à la relative jeunesse du domaine, chaque expert propose sa propre vision, sa propre définition.

La définition proposée par l'IEEE est la suivante : "Une architecture logicielle est l'organisation fondamentale d'un système incarnée dans ses composants, leurs relations avec chacun des autres et avec l'environnement, et les principes guidant sa conception et son évolution."

[Per&Wol 92] propose le modèle suivant:
Architecture logicielle = { Eléments, Forme, Raisonnement }.
Une architecture logicielle est un ensemble d'***éléments*** architecturaux. Ils sont de trois types : les éléments de traitement, les éléments de données et les éléments de connexion. Une ***forme*** architecturale consiste en des propriétés et des relations associées à des poids. Les propriétés sont utilisées pour contraindre les éléments à l'implémentation. Les relations sont utilisées pour contraindre la manière dont les éléments architecturaux interagissent et sont organisés les uns par rapport aux autres. Le poids indique soit l'importance de la propriété ou de la relation, lorsque on a le choix parmi plusieurs alternatives. Le ***raisonnement*** décrit les motivations pour le choix du style architectural, pour le choix des éléments et pour la forme. En d'autres termes, le raisonnement explique la satisfaction des contraintes du système. Ces contraintes sont déterminées par des aspects fonctionnels et non-fonctionnels tels que les aspects économiques, la performance et la confiance.

[BasCle&Kaz 99] propose "L'architecture logicielle d'un programme ou d'un système de calcul est la structure ou les structures du système, ce qui inclut des composants logiciels, les propriétés visibles externes de ces composants, et les relations entre ces composants".

La définition retenue dans le projet ArchWare est la suivante : "Une architecture est un ensemble de constituants architecturaux (des composants et des connecteurs parmi les composants) qui vérifie un style architectural[1] défini" [Oqu et al. 02].

Par la suite, nous allons étudier la notion d'architecture dans d'autres domaines et la comparer avec celle en génie logiciel.

2.1.2 Les architectures dans d'autres domaines

La notion d'architecture tient son origine du domaine du bâtiment, elle a été portée sur d'autres domaines dont l'informatique. Les études suivantes sont tirées de [Per&Wol 92].

Architecture bâtiment

Le domaine classique pour les architectures, celui du bâtiment, fournit des visions des plus intéressantes pour les architectures logicielles. Les points intéressants sont la notion de vues multiples, la notion de style architectural, le rapport entre le style et l'ingénierie, le rapport entre le style et les matériaux.
Un architecte propose différentes vues (ou plans) d'un bâtiment selon ce qu'il veut montrer (vue d'ensemble, vue d'intérieur) et à qui il le montre (client ou constructeur). Par analogie, l'architecte logiciel utilise plusieurs vues d'une architecture logicielle. Une de ces vues est l'implémentation. Cette vue n'est pas suffisante, elle ne permet pas d'abstraire les caractéristiques importantes d'un système ; c'est comme si d'une maison on ne voyait que les cloisons, la tuyauterie, les câblages électriques.
Ainsi, ces nombreux aspects fondamentaux peuvent être transposés au domaine logiciel par analogie ; de multiples vues sont utiles pour la compréhension des différents aspects d'une architecture ; les styles sont une importante forme de codification utilisée de manière descriptive et prescriptive ; les principes techniques et les propriétés des matériaux ont une importance fondamentale dans le développement d'une architecture ou d'un style en particulier.

Architecture informatique matérielle

Les machines en pipeline et les machines multiprocesseurs sont des exemples d'architectures matérielles représentant la configuration des différents composants électroniques.

[1] La définition d'un style architectural dans le cadre du projet ArchWare apparaît dans la section 3.1.1.

Il est intéressant de noter deux caractéristiques. Une architecture matérielle comporte un nombre relativement petit d'éléments de construction. Son accroissement est atteint par réplication de ses éléments de construction.
Ces points contrastent avec les architectures logicielles. Celles-ci peuvent comporter un nombre important d'éléments différents. Leur accroissement est obtenu par l'ajout d'éléments nouveaux.[2] Cependant les architectures logicielles utilisent des configurations similaires (ex. pipeline). De plus le vocabulaire pour décrire les architectures est souvent emprunté au domaine matériel (ex : composant, connecteur, port).

Architecture réseau

Les architectures réseaux sont composées de nœuds et de connexions. Elles proposent un vocabulaire pour les relations que ces deux entités ont en commun. Ainsi, nous avons des réseaux en étoile, des réseaux en anneau et des réseaux en bus. Nous notons donc peu de types d'éléments et de topologies différents.
Les deux points intéressants à propos des architectures réseaux sont :

- il y a deux types d'éléments : les nœuds et les connexions,
- il y a peu de topologies à considérer.

On peut abstraire ceci à un niveau similaire dans l'architecture logicielle. Par exemple, les processus et leurs interactions. Cependant, il y a un nombre de topologies différentes beaucoup plus important.

2.1.3 Que définit une architecture?

Une description architecturale est concernée avant tout par les aspects suivants.

La structure du système

Une architecture caractérise la structure d'un système en termes d'éléments fonctionnels et de leurs interactions. Ainsi, une architecture est conçue comme une configuration de composants interagissant. Il n'est question ni du cahier des charges (par exemple, les relations abstraites entre les éléments dans le domaine du problème), ni des détails d'implémentation (tels des algorithmes ou des structures de données). La description d'une architecture doit être assez simple pour permettre de raisonner sur un système et de pouvoir faire des prévisions. Par conséquent, elle est généralement définie sur plusieurs niveaux hiérarchisés : les éléments qui sont atomiques à un niveau abstrait sont généralement décrits par une architecture détaillée à un niveau plus concret.

De riches abstractions pour les interactions

Les interactions entre les composants architecturaux fournissent un vocabulaire riche pour les concepteurs. Bien que les interactions puissent être aussi simples que des appels de procédure ou des variables de données partagées, elles représentent souvent des formes de communication plus complexes. Les exemples incluent les pipes[3] (avec des conventions pour gérer les fins de fichiers et le blocage), les interactions client-serveur (avec des règles sur l'initialisation, la terminaison, et la gestion des exceptions), les évènements (avec de multiples destinataires), et les protocoles d'accès aux bases de données (avec des protocoles pour l'invocation des transactions).
Les interactions sont définies au sein de ***comportements*** définissant l'ordonnancement des communications d'un élément architectural.

Des propriétés globales

Une architecture décrit typiquement les propriétés globales d'un système. Ainsi, les problèmes auxquels elle répond sont habituellement au niveau système, tel que les temps de traitement, la latence, ou la résilience d'une part du système à une panne. Ces propriétés globales seront référées comme ***attributs*** d'une architecture dans la suite de ce mémoire.

---

[2] Cela dépend toutefois à quel niveau de détail on se situe. A un niveau très abstrait tout est composant ou connecteur.

[3] Nous utiliserons le terme anglais *pipe* qui est plus couramment utilisé que sa traduction *tube*.

Afin de standardiser la description des architectures, des concepts ont été mis en place : les concepts de composants et de connecteurs.

2.1.4 Les architectures composant-connecteur

Très tôt, Shaw et Garlan proposent un modèle d'architecture basé sur trois abstractions : le ***composant***, le ***connecteur*** et la ***configuration*** [Gar&Sha 93]. Il n'existe pas de définition universelle pour décrire un élément architectural (ou constituant architectural), mais il y a un large consensus pour classifier ces éléments en composants et connecteurs.

Un ***composant*** est une *unité de calcul* ou de *stockage de données* dans une architecture. Ces éléments sont les constituants de base. Ils sont indépendants de la structure dans laquelle ils sont placés, et possèdent une interface permettant de communiquer avec cet environnement.

Un ***connecteur*** représente le protocole de communication entre différents composants. Il modélise leurs interactions et spécifie les règles qu'ils doivent respecter durant la communication.

Une ***configuration*** représente un ensemble de composants interconnectés par des connecteurs. L'ensemble des composants d'un système et de leur interaction forme l'architecture du logiciel.

Comme nous le verrons dans le chapitre suivant la plupart des systèmes pour la conception architecturale s'appuient sur les concepts composant-connecteur. Nous donnerons des détails supplémentaires sur les architectures composant-connecteur dans le chapitre 4 dans lequel nous formalisons un style Composant-Connecteur.

2.1.5 Dynamique, évolution et mobilité.

Des systèmes à long temps d'exécution et des systèmes à mission critique ont besoin d'être mis à jour et d'évoluer pendant leur exécution. Comme il ne peuvent cesser leur évolution ils doivent être maintenu en cours d'exécution. Ils peuvent se maintenir par eux-mêmes en gérant leur propres reconfiguration et en intégrant de nouveaux éléments à l'exécution. Ils peuvent aussi permettre leur reconfiguration par des sources extérieures tout en assurant leurs services.

Dynamique

Les architectures dynamiques sont celles dont une partie de leur évolution à l'exécution est programmée à la conception.

> La **dynamique** d'une architecture consiste en sa capacité à pouvoir modifier sa topologie à l'exécution. Il peut s'agir de la création ou de la suppression dynamique d'éléments ou de leur reconfiguration dynamique [Med 96].

En définissant de telles architectures, l'architecte est capable d'exprimer des propriétés dynamiques telles que l'extensibilité, la customisabilité, et l'évolutivibilité des gros systèmes logiciels à un haut niveau d'abstraction. Les manières communes selon lesquelles une architecture peut être modifiée après avoir été construite sont les suivantes [Med 96] :

- l'addition ou la suppression de composants,
- la mise à jour d'un composant (par exemple, le réglage de la performance),
- la reconfiguration de la topologie (par exemple, la reconnexion des composants et des connecteurs).

Des solutions architecturales basées sur la dynamique ont été développées. Ces solutions entraînent des propriétés que les systèmes peuvent requérir telles que la performance, la modifiabilité ou la fiabilité.

La mobilité

Parmi les propriétés intrinsèques à certains systèmes en général distribués, il y a la mobilité.

> La **mobilité architecturale** consiste en la capacité d'envoi et de réception d'éléments architecturaux d'un environnement vers un autre. En d'autres termes, des éléments peuvent se déplacer au sein de l'architecture.

L'évolution d'une architecture

L'évolution est un mécanisme différent de la dynamique.

> L'***évolution*** d'une architecture consiste au changement (non-planifié à la conception) de celle-ci, à l'exécution ou non, par une action extérieure [Mor et al. 04].

Les mécanismes d'évolution permettent de décomposer une architecture, de créer, de remplacer ou de supprimer des éléments, et de la recomposer.

2.2. Processus de conception orientée architecture

La Figure 2 présente le cycle de vie d'une application logicielle. Ce cycle démarre par l'analyse des besoins ; puis continue par la conception (celle-ci est séparée en conception architecturale et en conception détaillée) ; puis vient l'implémentation (codage) et les tests ; les phases suivantes concernent le cycle de vie du produit une fois sur le marché avec l'intégration, la livraison, l'exploitation, la maintenance et finalement son retrait du marché.

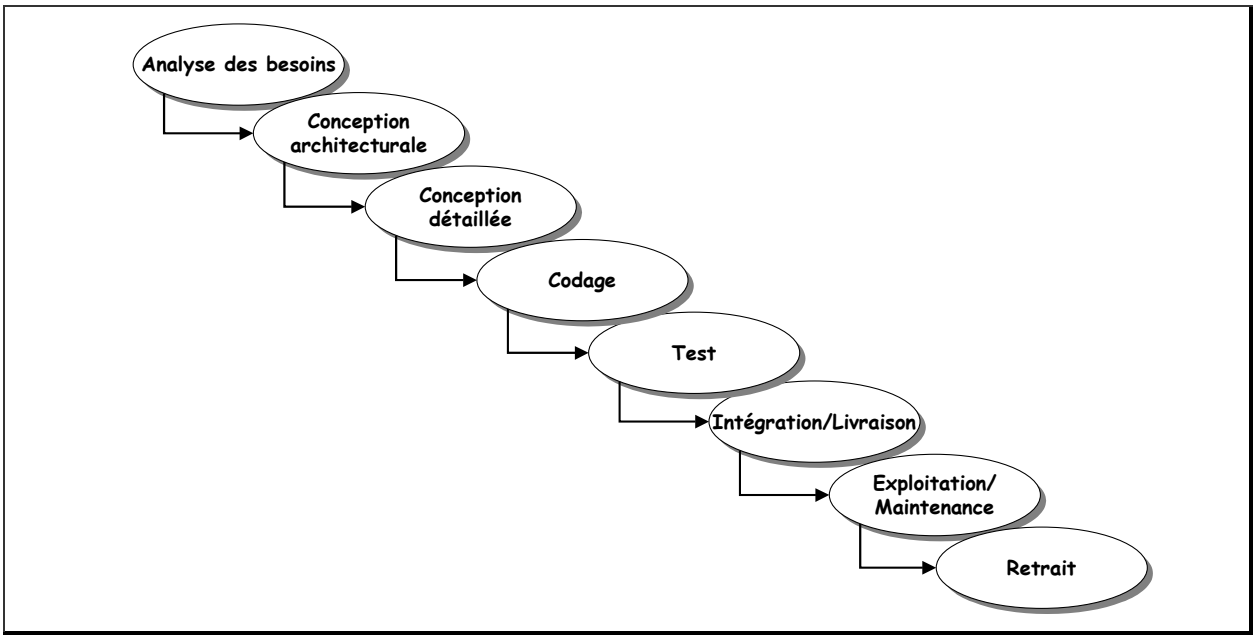


Figure 2 - Cycle de vie d'une application logicielle

Au sein du cycle de vie, les architectures sont essentiellement utilisées dans les phases de conception et de codage. La figure suivante zoome sur ces trois phases et représente le processus de développement centré architecture ainsi que les acteurs qui interviennent directement dans celui-ci : l'architecte, le développeur et l'analyste.

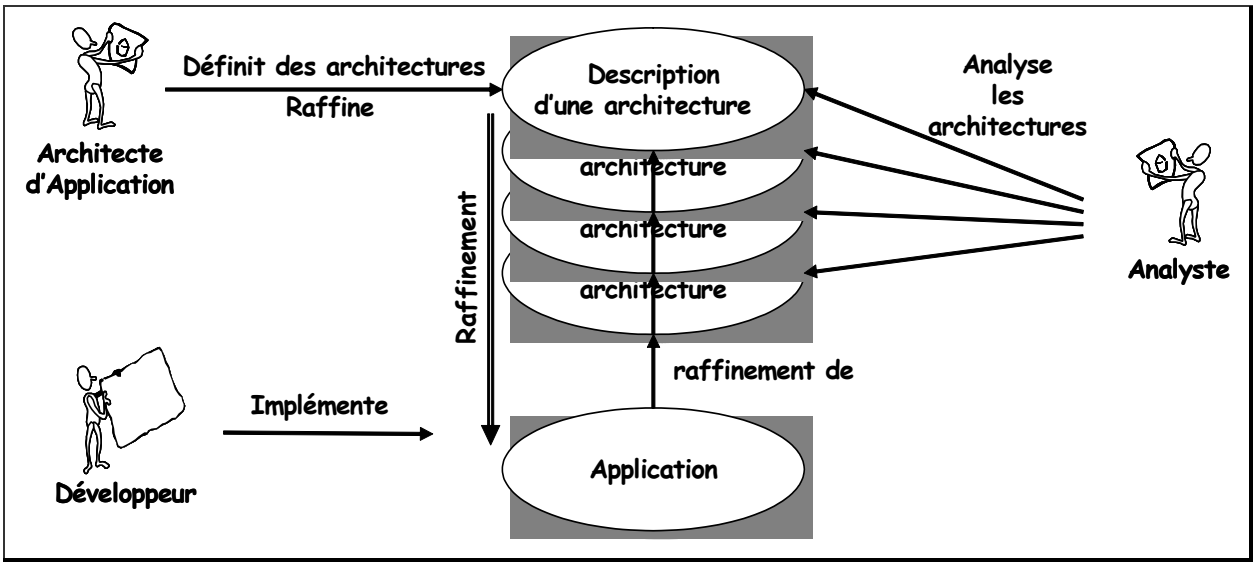


Figure 3 - Processus de développement orienté architecture

L'***architecte*** a pour rôle de concevoir et de décrire l'architecture qui servira de base au développement de l'application. Ce rôle inclut le raffinement de l'architecture. Le raffinement d'une architecture s'effectue jusqu'à ce que l'architecture soit assez détaillée pour pouvoir être implémentée sans ajouter d'informations nouvelles.

L'***analyste*** vérifie que l'architecture répond au cahier des charges et cela pour chaque niveau de raffinement ; il permet de garantir que l'application obtenue respecte les propriétés non-fonctionnelles, structurelles et comportementales définies par l'architecte en accord avec le client et les utilisateurs.

Le ***développeur*** se base sur une architecture pour le guider dans l'implémentation du système, à la fois vis-à-vis de sa propre tâche dans le projet et vis-à-vis de son interaction avec les autres acteurs. Dans certains cas, on peut faire de la génération de code automatique à partir d'une architecture et le développeur n'intervient pratiquement plus.

Dans les sections suivantes, nous allons nous attarder sur différentes phases du cycle de vie.

2.2.1 Description d'architecture

Le développement d'une bonne architecture pour un système complexe est un point critique pour s'assurer qu'il satisfera les objectifs principaux [Gar 03]. Malencontreusement, dans l'industrie encore aujourd'hui, les descriptions sont encore largement basées sur des diagrammes informels (cf. Figure 4) (sous forme de lignes et de boîtes) et sont souvent ambiguës, incomplètes, inconsistantes, et non-analysables ; la description informelle d'architectures induit des difficultés au niveau de son interprétation et possède donc un certain nombre de limitations [Abo et al. 1995].

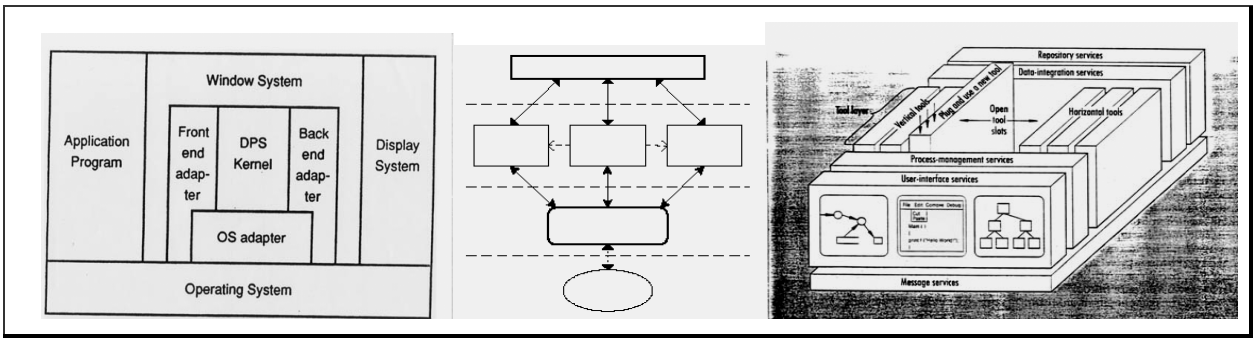


Figure 4 - Diagrammes informels d'architectures

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 »,
- comment le comportement global du système est déterminé par celui de ses composants.

De simples diagrammes comportant des annotations ne sont pas suffisamment expressifs pour spécifier une architecture en exprimant toutes ces informations. C'est pourquoi les concepteurs de logiciels qui utilisent de tels diagrammes les interprètent en fonction du contexte d'utilisation. Par exemple, pour décrire une architecture de type pipe-filter un diagramme utilisera les boîtes pour représenter les filtres (filter) les lignes pour représenter les pipes. Pour un autre système, les « boîtes » pourraient représenter des objets et les « lignes » des appels de méthodes. Toutefois, l'interprétation reste imprécise, il est donc impossible de donner à ces diagrammes des significations non ambiguës. Il est alors difficile de déterminer si l'implémentation du système satisfait sa description architecturale. De même, ce manque de précision empêche l'application d'analyses formelles : il est impossible de raisonner formellement sur les descriptions architecturales des systèmes, ou de faire des comparaisons efficaces entre différentes descriptions architecturales exprimées informellement. Ainsi, afin de définir plus précisément et de mieux comprendre l'architecture des systèmes logiciels, de nombreux travaux ont proposé d'utiliser des notations permettant d'associer des sémantiques formelles.

Dans la dizaine d'années passées plusieurs chercheurs ont développé des langages formels et des outils d'analyses associés pour les architectures logicielles. La description formelle de l'architecture logicielle d'un système complexe a deux raisons d'être. Premièrement, elle rend disponible un document précis décrivant la structure, le comportement et des propriétés du système à tout acteur de la conception, de l'implémentation et de la maintenance du système. Deuxièmement, elle donne la possibilité d'analyser les propriétés du système au niveau architectural. Ceci permet une détection précoce des erreurs, économisant ainsi temps et argent.

Les langages de description d'architectures - ADLs[4] (Architecture Description Languages) - émergent comme le support de notation pour les modèles d'architectures [Kog&Cle 95]. Ils usent de notations textuelles et graphiques pour exprimer l'information architecturale. Ces langages sont souvent accompagnés d'outils de création, de modification, de navigation, de simulation et d'analyse. Il y a une grande variété d'ADLs émergeant de divers groupes industriels et académiques. Le champ couvert par les ADLs est très large, ceci étant dû aux styles qu'ils supportent et aux types d'analyses qu'ils permettent. Chacun a ses particularités et vise des aspects précis de la conception architecturale et il n'y a pas d'ADL qui sied le mieux à toutes les situations possibles. Ils sont constamment étudiés, classés et comparés dans de nombreux articles [Kog&Cle 95][Cle 96][Med&Tay 97][Med&Tay 00][Fux 00][LeyCim&Oqu 02]. Nous étudierons plusieurs ADLs dans un état de l'art constituant le chapitre 3.

[4] Dans ce document, nous utilisons l'acronyme anglophone en raison de sa large adoption dans le milieu académique.

L'utilisation des ADLs comporte de multiples bénéfices : le fait de permettre de décrire formellement l'architecture de systèmes logiciels rend possible l'utilisation de mécanismes pour représenter les architectures, leur analyse, leur raffinement, et la génération automatique du code correspondant.

2.2.2 Analyse d'architecture

L'évaluation des propriétés d'un système au niveau architectural permet de prévenir les erreurs et de diminuer les coûts dûs à celles-ci [Med&Tay 97] ; les analyses ont pour but de supporter l'identification et la solution aux problèmes de conception dès les premières étapes du développement d'un logiciel [Din&Ros 99]. Des méthodes d'analyse d'architecture ont été développées dont une portant sur les architectures : SAAM (Software Architecture Analysis Method) [Kaz et al. 94].
Les types d'analyses applicables dépendent de l'ADL utilisé et de son modèle sémantique. Par exemple, Wright permet de vérifier l'absence d'interblocage ; Unicon supporte l'analyse de planning d'ordonnancement via des attributs non-fonctionnels représentant priorité et niveaux critiques ; SADL permet d'établir la correctitude de deux architectures vis à vis d'un raffinement ; Rapide permet l'analyse des traces d'évènements. Un autre aspect des analyses est la vérification du respect de contraintes architecturales qui peuvent aussi bien porter sur la structure et sur le comportement que sur des propriétés non-fonctionnelles d'une architecture.

2.2.3 Raffinement d'architecture

> Le ***raffinement*** architectural est le passage d'une architecture abstraite vers une architecture plus concrète[5]. Lors du raffinement les propriétés de l'architecture abstraite sont conservées par l'architecture concrète. Cette dernière contient plus d'informations qui sont consistantes avec les informations de l'architecture abstraite. [Bol 04]

Le raffinement de descriptions d'architecture est une tâche complexe dont la correctitude ne peut pas toujours être garantie par des preuves formelles, mais l'usage d'outils adéquats permet d'accroître sa confiance en cela [Med&Tay 97].

2.2.4 Génération de code

Le but ultime de la conception logicielle et de la modélisation est de produire un système exécutable. Un modèle architectural élégant et efficace a peu de valeur s'il n'est pas convertible en application exécutable. Cela peut se faire manuellement ; cas dans lequel de nombreux problèmes de consistance et de traçabilité entre l'architecture et son implémentation peuvent apparaître. Il est préférable d'utiliser des outils de génération de code qui devraient être fournis par chaque ADL [Med&Tay 97].

2.2.5 Outils d'exploitations

Pour supporter un processus de développement centré architecture et pour exploiter les descriptions formelles des architectures, des outils et des environnements ont été développés. La plupart des ADLs sont accompagnés par des outils ou des environnements de développement permettant de définir et d'exploiter des descriptions architecturales.

Une équipe de développement en a répertorié les principaux afin de définir une boîte à outils complète [ABL] pour la prise en charge des ADLs. Les outils architecturaux sont principalement des :

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

[5] Dans les approches formelles, le processus de raffinement peut être totalement ou partiellement automatisé.

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

2.3. Qu'apportent les architectures logicielles ?

Les architectures logicielles jouent un rôle important dans au moins six aspects du développement logiciel [Gar 00] présentés ci-dessous :

Compréhension *:* les architectures logicielles rendent plus facile la compréhension du fonctionnement de systèmes complexes en les représentant à un haut niveau d'abstraction.

Réutilisation *:* les descriptions architecturales supportent la réutilisation à de multiples niveaux. Les travaux actuels dans le domaine de la réutilisation se concentrent généralement sur l'utilisation de bibliothèques de composants et de structures dans lesquelles ces composants peuvent être intégrés. Ceci est prouvé par de nombreux travaux existants dans les domaines des DDSA (Domain-Specific Software Architectures), des architectures de référence, ou des patrons de conceptions [Met&Gra 92][Bus et al. 96]. Nous verrons cela plus spécifiquement dans la section suivante sur les styles architecturaux.

Construction *:* une description architecturale fournit un plan de développement en indiquant les composants principaux et les dépendances entre eux.

Evolution *:* l'architecture d'un système peut décrire comment celui-il est censé évoluer. La définition explicite des limites d'évolutions d'un système permet de faciliter sa maintenance et d'estimer plus précisément les coûts des modifications. De plus, les descriptions architecturales distinguent les aspects fonctionnels des composants de la façon dont ces composants interagissent entre eux. Cette séparation permet de modifier facilement les mécanismes de connexion, ce qui favorise l'évolution en termes de performance, d'interopérabilité et de réutilisation.

Analyses *:* les descriptions architecturales fournissent des moyens d'analyse, tels que la vérification de la consistance d'un système [All&Gar 94] [Luc et al. 95], la vérification de la conformité aux contraintes imposées par un style architectural [AboAll&Gar 93], la vérification de la conformité à des attributs qualité [Cle et al. 95], l'analyse de dépendance [StaRic&Wol 98], ainsi que des analyses spécifiques au style à partir desquels les architectures ont été construites [Cog&Szy 93][Mag et al. 95][GarAll&Ock 94].

Gestion *:* l'expérience a montré que la définition précise d'une architecture logicielle est un facteur clé dans la réussite d'un processus de développement logiciel. L'évaluation d'une architecture mène à une meilleure compréhension des besoins, des stratégies d'implémentation, et des risques potentiels [Boe et al. 94].

De plus, l'utilisation d'une conception orientée architecture comporte des avantages pour toutes les personnes intervenant dans le cycle de vie d'un projet logiciel (cf. Figure 2) :

- pour le **client** : l'architecture permet une estimation du budget, de la faisabilité et du temps de développement. Elle fournit aussi un support pour établir des scénarios d'utilisation, des simulations, pour donner une idée du futur système et prendre des décisions de modifications avant le développement,
- pour le **chef de projet** : elle fournit un support pour la traçabilité des besoins, l'évaluation du système et le suivi des progrès,
- pour le **développeur** : l'architecture offre une ligne de conduite et les spécifications nécessaires pour l'implémentation de nouveaux composants. De plus, elle offre les garanties pour l'interopérabilité avec les systèmes existants,
- pour le **mainteneur** : l'architecture est un guide au cours de l'évolution de l'architecture.

3. La conception orientée style

Les concepteurs de systèmes logiciels reconnaissent de plus en plus l'importance d'exploiter les connaissances de conception dans la mise en place de nouveaux systèmes [Mon et al. 97]. Comme dans n'importe quelle activité de conception, une question centrale est : "comment réutiliser les expériences passées pour produire de meilleurs modèles ?". Une première réutilisation est réalisée à travers l'usage des composants et des connecteurs plusieurs fois dans une même ou plusieurs architectures. Cependant, ces éléments architecturaux sont souvent très spécifiques ce qui limite leur réutilisation. De plus, il est souhaitable de capitaliser certaines caractéristiques architecturales de ces éléments sous forme d'expertise de conception. Ceci a été réalisé de manière informelle à travers la codification et la réutilisation de modèles primaires ou idiomes architecturaux. Ainsi, on parle de système client-serveur, de tableau noir, de pipeline, d'interpréteur ou de système en couches.
Dans le domaine du bâtiment, les architectures sont classifiées selon des styles architecturaux. Ainsi, on parle par exemple de style gothique et de style roman [Per&Wol 92]. La notion de style est très utile, à la fois d'un point de vue descriptif et prescriptif. Un style définit une codification particulière pour les éléments architecturaux et leurs arrangements. Un style limite les types d'éléments architecturaux et la manière dont ils sont arrangés. Il est très important de noter les relations entre les principes technologiques et un style architectural. On ne construit pas un préfabriqué de la même manière et avec les mêmes outils qu'on construit une église romane. Enfin, la relation entre un style architectural et les matériaux et d'une importance critique. Les matériaux possèdent certaines propriétés exploitées pour fournir un style particulier.

Cette notion de style architectural a été portée au domaine logiciel pour définir différentes classes d'architectures.

3.1. Qu'est-ce qu'un style architectural ?

Les principes d'organisation architecturale pour les systèmes logiciels sont appelés styles architecturaux. Les styles architecturaux [Sha&Gar 96] sont des patrons organisationnels qui ont été développés alors que les concepteurs reconnaissaient la valeur de principes organisationnels et des structures spécifiques pour certaines classes de logiciels.
Un style formalise la connaissance et l'expérience dans un domaine logiciel spécifique. Le but principal des styles architecturaux est de simplifier la conception des logiciels et la réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système [Mon et al. 97]. L'objectif est de capitaliser et de codifier les principes et les expériences pour spécifier, analyser, planifier et monitorer la construction de systèmes logiciels complexes avec un haut niveau d'efficacité et de confiance. Comme exemple de styles architecturaux il y a les systèmes client-serveurs, les organisations pipe-filter ou encore les architectures en couches. Ces différents styles apportent des solutions architecturales à des problèmes définis en termes d'attributs qualité comme la performance, la modifiabilité ou encore la robustesse d'un système. Par exemple, on utilise le style Pipe-Filter lorsqu'on veut privilégier la réutilisation et que la performance n'est pas une priorité [Kle et al. 99]. [Kle&Kaz 99] propose une étude sur l'association des styles avec des attributs qualité à travers des ABAS (Attribute-Based Architectural Style) et fournit une librairie de ces ABAS.

> Un ***attribut qualité***[6] [Bar et al. 95] est une caractéristique (ou une propriété) d'une architecture logicielle (ou d'un système logiciel) permettant d'évaluer la qualité de cette dernière. Parmi ces caractéristiques, on trouve entre autres la fiabilité, la performance, la modifiabilité ou la portabilité. Ces caractéristiques peuvent être raffinées en sous-caractéristiques.

Un style architectural définit une famille des systèmes logiciels ayant un vocabulaire commun pour désigner les composants, des caractéristiques topologiques et comportementales communes, et un ensemble de contraintes sur les interactions parmi les composants. Un style architectural est moins contraignant et moins complet qu'une architecture spécifique. Il spécifie uniquement les contraintes les plus importantes, au niveau par exemple de la structure, du comportement, de l'utilisation des

[6] Défini d'après la norme ISO-9126.

ressources des composants et des connecteurs dans un système [Abd 96]. D'une façon générale, les styles architecturaux permettent à un développeur de réutiliser l'expérience concentrée de tous les concepteurs qui ont précédemment fait face à des problèmes similaires [Kle&Kaz 99].

En outre, l'utilisation des styles architecturaux comporte des intérêts précis :

- elle promeut la réutilisation au niveau de la conception (et non seulement au niveau du code, comme c'est le cas de la plupart des techniques de réutilisation) [Mon&Gar 96],
- elle peut donc aussi mener à une réutilisation significative de code,
- elle permet de normaliser une famille d'architectures, ce qui améliore la compréhension de l'organisation d'un système,
- elle permet l'utilisation d'analyses spécifiques au style concerné [Cia&Mas 96].

3.1.1 Plusieurs définitions

A l'instar des architectures, il existe de nombreuses définitions pour les styles architecturaux. Cette section en présente une liste.

[Sha 90] Un style architectural logiciel est un *ensemble de règles de conception* qui identifie les *sortes de composants et de connecteurs* qui peuvent être utilisées dans la composition d'un système ou d'un sous-système selon des *contraintes locales ou globales* sur la manière dont la composition est faite.

[Per&Wol 92] Un style architectural est ce qui *abstrait les éléments et les aspects formels de plusieurs architectures spécifiques*. Il n'y a pas de séparation claire entre ce qu'est une architecture est ce qu'est un style. Un style architectural est moins contraignant et moins complet qu'une architecture spécifique. Il y a un continuum.

[AboAll&Gar 93] Un style architectural *caractérise une famille de systèmes* qui ont les mêmes propriétés structurelles et sémantiques.

[Sha&Gar 96] Les styles architecturaux sont des *patrons et des idiomes organisationnels récurrents*.

[Tay et al. 96] Un style architectural est une *abstraction de composition récurrente* et de *caractéristiques d'interaction* dans un ensemble d'architectures.

[Mon et al. 97] Un style architectural fournit un *langage spécialisé* pour une classe spécifique de systèmes.

[Med&Tay 97] Les styles sont des idiomes clés pour la conception qui permettent l'exploitation de patrons structurels et d'évolution et facilitent la réutilisation de composants, de connecteurs et de processus.

[Fux 00] Un style est un *ensemble de propriétés* partagées par un ensemble de configurations qui sont membres du style.

[Arc 02] Dans le cadre du projet ArchWare la définition d'un style architectural est la suivante : un style architectural définit une abstraction particulière de composants et des propriétés qui doivent être satisfaites par les constituants architecturaux.

La définition que nous retiendrons pour nos travaux et la suivante :

> Un ***style architectural*** est un ensemble de propriétés architecturales (structurelles, comportementales et non-fonctionnelles) communes à une famille d'architectures.

3.1.2 Styles courants

Afin de donner une idée plus concrète de ce qu'est un style, nous vous proposons une étude succincte de quelques styles existants. De plus, il nous semble aussi intéressant de présenter une

étude de ces styles, car d'un point de vue pratique et chronologique, c'est la reconnaissance et l'adoption de ces styles qui ont mené à l'émergence de travaux sur la formalisation des styles.

Plusieurs styles architecturaux ont été répertoriés parmi ceux qui ont été les plus fréquemment utilisés [Gar&Sha 93][Lan 02][Sha&Cle 97][Kog&Cle 95]. Un certain nombre de styles sont énumérés ci-dessous :

- systèmes événementiels : systèmes où les invocations sont implicites,
- systèmes pipe-filter[7]: ce sont des systèmes de transformateurs de flux connectés,
- systèmes orientés objet : ils définissent des types de données abstraits et des mécanismes d'héritage,
- systèmes en couches : ce sont des couches structurées définissant soigneusement des interfaces et des restrictions sur les invocations inter-couches,
- processus communicants : systèmes où des processus communiquent par des envois, synchrones ou asynchrones, de messages ; ils incluent les systèmes client-serveur et peer-to-peer,
- programme principal – sous routines : c'est la décomposition fonctionnelle traditionnelle,
- systèmes de données indirectionnelles : ils définissent un dépôt de données central qui est consulté par des requêtes conduites ; ce style est aussi appelé tableau noir (en anglais: blackboard),
- …

Les exemples de styles cités jusqu'à maintenant apportent des solutions à des problèmes récurrents. Ce sont des styles offrant des solutions visant à atteindre des objectifs en termes d'attributs qualité. Nous pouvons trouver des études de certains de ces styles vis à vis des attributs qualité qu'ils promeuvent dans [Kle&Kaz 99].

Ces styles ne sont pas suffisamment précis et contraignants pour pouvoir servir de guide de développement de logiciels spécifiques à un domaine d'application. C'est pourquoi de nombreux architectes ont défini des architectures et des styles propres à leur secteur d'activité, adaptés à leurs propres besoins (DSSA : Domain-Specific Software Architectures). Ces styles spécifiques sont généralement basés sur un ou plusieurs styles plus communs, tels que ceux que nous avons énumérés.

Dans les sous-sections suivantes, nous donnons des détails sur les styles Pipe-Filter, Client-Serveur, BlackBoard, C2.

Pipe-Filter

Dans le style Pipe-Filter le centre d'intérêt est les flux de données dans le système [BasCle&Kaz 99][Sha&Cle 96b][Sha&Gar 96][Wal 98]. Il y a un certain nombre de composants où la production d'un composant forme l'entrée du prochain. Un exemple typique est l'utilisation des pipes d'Unix. La Figure 5 montre les unités de traitement comme des boîtes et les flux de données comme des flèches. Sous sa forme plus pure, les différents composants sont complètement séparés (ils ne partagent aucune donnée ou état), et peuvent commencer leur traitement dès qu'ils ont une entrée.

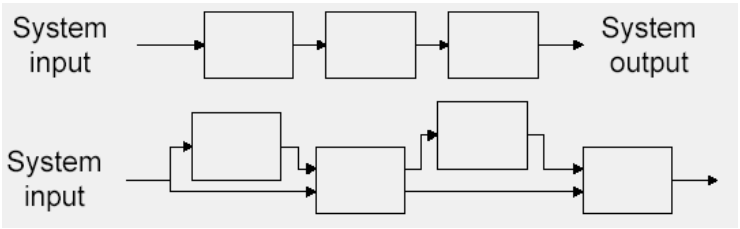


Figure 5 - Deux systèmes Pipe-Filter : un simple et un plus compliqué

[7] La traduction française est tube-filtre mais la dénomination anglaise est celle qui est toujours utilisée.

Ce style convient à un système analysant et formatant du texte ou des données, mais moins pour un système interactif.

Blackboard

Une architecture en tableau noir (ou avec dépôt) appelle l'attention sur les données dans le système [BasCle&Kaz 99][Sha&Cle 96b][Sha&Gar 96][Wal 98]. Il y a un magasin central de données, le tableau noir, et des agents écrivant et lisant des données. Les agents peuvent être implicitement appelés quand les données changent, ou explicitement par une action externe telle qu'une commande utilisateur. Une architecture en tableau noir est décrite sur la Figure 6, où le magasin central de données est représenté par un rectangle, les agents par des rectangles arrondis, et les flèches dénotent des demandes de lecture ou d'écriture de données.

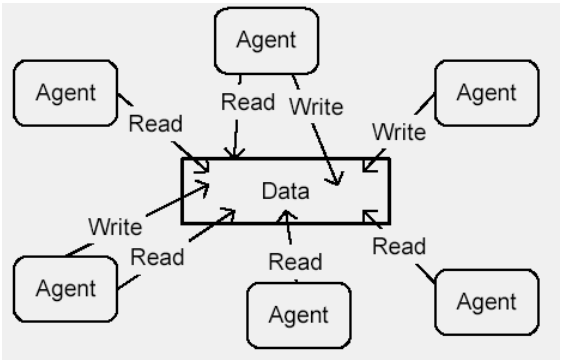


Figure 6 - Une architecture en tableau noir (blackboard) ou à dépôt

Une base de données peut facilement être décrite en utilisant le style tableau noir, où le tableau noir lui-même est naturellement la base de données. Les exemples des agents sont des applications client, des déclencheurs de base de données (petites applications se déclenchant automatiquement lorsque les données changent), et des outils d'administration.

Client-Serveur

Dans le style Client-Serveur le centre d'intérêt sont les différents services auxquels un ensemble de clients peut faire appel [BasCle&Kaz 99][Sha&Cle 96b][Sha&Gar 96][Wal 98]. Ce style est particulièrement adapté quand le matériel est organisé en un ensemble d'ordinateurs (par exemple postes de travail personnels) et une ressource centrale telle qu'un système de fichiers ou une base de données (cf.Figure 7).

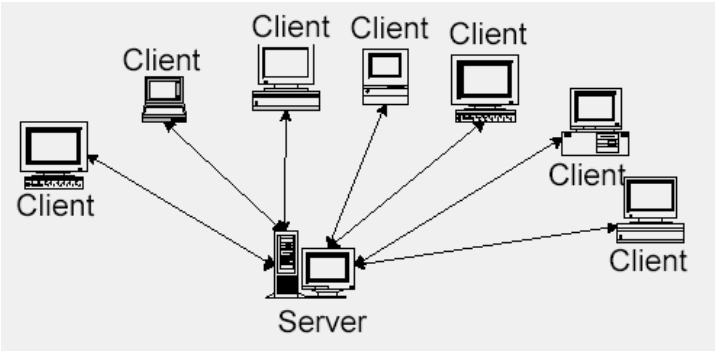


Figure 7 - Vue d'un système client-serveur

Dans un système client-serveur, il peut y avoir plusieurs clients dans un ordinateur, et même le serveur peut fonctionner sur le même ordinateur. Ce style permet notamment de décrire un système de base de données multi-utilisateur à un niveau différent d'abstraction que le style tableau noir.

Le style C2

Le style C2 (ou Chiron-2) [Tay et al. 96][MedOre&Tay 97][Med et al. 96] est un style architectural basé composant qui supporte la réutilisation de "gros grains" (composants) et un système de composition flexible. Ce style est initialement dédié pour supporter la conception d'interfaces graphiques, constatant que, dans ce domaine, le support pour la réutilisation était limité à la création de boîte à outils ("widgets"). De plus, il supporte la conception d'applications distribuées et concurrentes. Bien que l'objectif initial soit des applications impliquant des interfaces graphiques, le style offre des possibilités intéressantes pour une plus large utilisation.
Les composants qu'il définit interagissent au moyen de messages asynchrones de requêtes et de notifications. Ils sont construits de manière indépendante de leur environnement : chaque composant ne contient pas d'information a priori sur les autres composants.

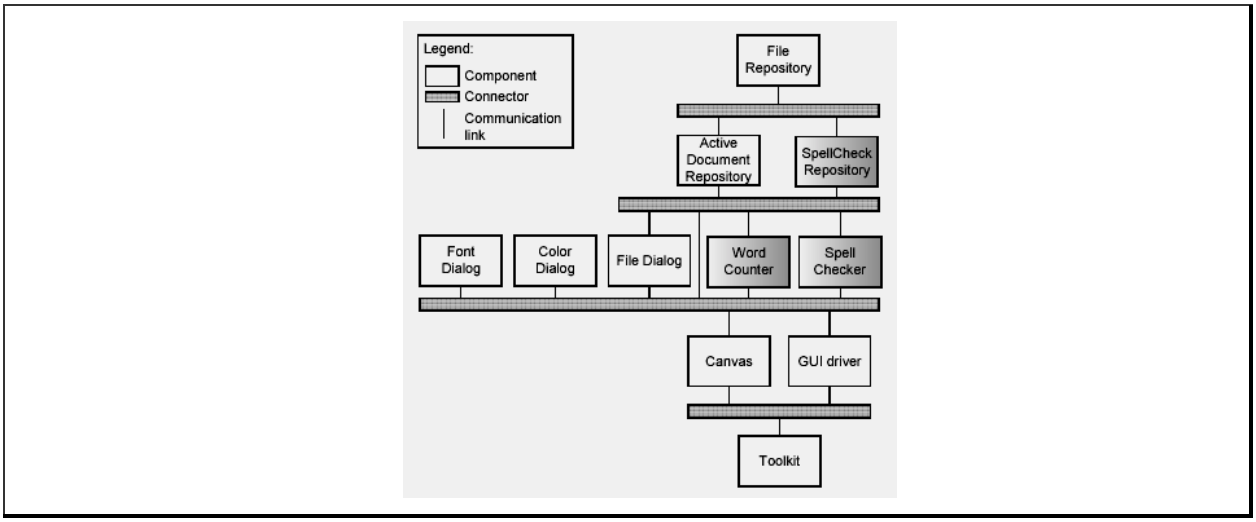


Figure 8 - Architecture dans le style C2 pour une application de traitement de texte

Le style C2 étant très générique, il a servi de base conceptuelle au langage C2SADEL [MedRos&Tay 99].

3.1.3 Que définit un style?

Un style représente une famille de systèmes connexes [Gar 01]. D'après [Gar 95] un style architectural fournit typiquement :

- Un **vocabulaire** pour spécifier les types d'éléments de construction : Filtre, Pipe, Client, Serveur, …
- Des **règles de configuration**, ou **contraintes**, qui déterminent comment les éléments architecturaux peuvent être composés. Une contrainte peut interdire un cycle dans le style Pipe-Filter. Elles peuvent définir aussi des patrons spécifiques (décomposition pipeline d'un compilateur).
- Une **interprétation sémantique** par laquelle les compositions des éléments architecturaux ont une signification bien définie.
- Des **analyses** qui peuvent être appliquées sur les systèmes construits dans ce style.

Ceci est un point de vue souvent repris à travers la littérature. Cependant, nous pensons que le terme vocabulaire est un terme mal choisi pour désigner un support à la construction architectural en termes d'éléments architecturaux car il fait penser à un aspect syntaxique. De plus, nous pensons que ce vocabulaire pourrait désigner aussi des configurations particulières entre les éléments. Quoi qu'il en soit, nous fournissons plus de détails dans les paragraphes suivants.

Le vocabulaire

Derrière ce concept de vocabulaire, il y a la définition d'éléments de construction. Ces éléments de construction peuvent être très abstraits, comme ceux que nous avons déjà donnés en exemple,

mais ils peuvent être aussi très spécifiques. Ainsi, un style peut définir une librairie d'éléments architecturaux (en principe hiérarchisés) ayant des comportements spécifiques. Ceci rejoint le point de vue selon lequel la limite entre un style et une architecture n'est pas toujours très franche.

Les contraintes

Dans nos travaux, nous différencions les contraintes en trois classes principales : les contraintes ***topologiques***, les contraintes ***comportementales*** et les contraintes d'***attributs*** :

- les contraintes ***topologiques*** précisent les éléments de construction utilisables, leur nombre d'occurrences possibles au sein de l'architecture, ainsi que les règles de configuration contraignant leurs interactions. Par exemple, "une architecture client-serveur contient au moins un serveur" est une contrainte portant sur la structure.
- les contraintes ***comportementales*** concernent l'évolution temporelle de l'architecture et le comportement des éléments architecturaux. Concernant l'évolution temporelle de l'architecture, un style peut spécifier comment l'architecture peut évoluer dynamiquement – quelle sorte d'élément peut apparaître ou disparaître – et quelles contraintes les architectes doivent respecter en modifiant une architecture à la volée. Les styles peuvent aussi spécifier des propriétés de mobilité propres à des éléments architecturaux, et des propriétés comportementales (exemple : pas d'interblocage) – ces propriétés peuvent être celles du comportement d'une configuration entière ou seulement d'un élément architectural.
- les contraintes ***d'attributs*** concernent les aspects non structurels et non fonctionnels d'une architecture. Les attributs apportent des informations sur les éléments architecturaux. Ils peuvent être contraints sur leurs types, leur nom et leur gamme de valeurs. Les styles peuvent spécifier comment les valeurs des attributs sont liées avec d'autres aspects architecturaux (la structure et le comportement).

Les analyses

Un style offre un support pour raisonner sur les architectures à travers un ensemble d'analyses architecturales.

> Une ***analyse architecturale*** est une étude ou un traitement automatisé d'une architecture pour en déterminer ou quantifier une ou plusieurs caractéristiques.

Les caractéristiques identifiées peuvent être de différentes sortes. Il peut s'agir d'une vérification ou d'une mesure. Une analyse peut vérifier la satisfaction d'une propriété par l'architecture ; par exemple, une analyse permet de vérifier que la structure de l'architecture est cyclique. Une analyse peut retourner une mesure ; par exemple, l'analyse retourne le nombre d'éléments d'une architecture.

Les analyses sont associées aux styles car leur capacité à évaluer telle ou telle caractéristique d'une architecture dépend fortement de la façon dont cette dernière est construite. En d'autres termes, elles sont dépendantes d'un style. Il y a différents moyens de quantifier une caractéristique en fonction des propriétés topologiques, comportementales ou d'attributs d'une architecture. Ainsi, la performance d'un système s'évalue différemment selon la nature de son architecture. S'il s'agit d'une architecture Client-Serveur on évaluera la capacité de connexion du serveur. S'il s'agit d'une architecture Pipe-Filter, l'évaluation se basera sur le temps de traitement des données par les filtres. Un autre exemple figurant dans [Kle&Kaz 99] propose deux façons de quantifier la modifiabilité d'une architecture suivant qu'elle suit un style en couche ou un style tableau noir.

3.1.4 Concepts relatifs aux styles

Dans cette section nous allons nous attarder brièvement sur des domaines du génie logiciel où les concepts proposés (dans leur forme ou leur sémantique) sont proches des styles architecturaux. Ainsi, nous allons présenter les patrons architecturaux ainsi que les lignes et familles de produits.

Patrons

Les travaux concernant les styles architecturaux sont étroitement liés à ceux effectués dans le domaine des patrons de conception (ou *design patterns*) [Sha&Cle 96a]. Les concepteurs de

logiciels utilisent fréquemment des patrons informels pour décrire les architectures de leurs systèmes [Sha 94]. L'idée de base derrière les patrons est que des idiomes communs se retrouvent répétitivement dans les modèles logiciels et ces patrons devraient être explicites, codifiés et appliqués de manière appropriée à des problèmes similaires [Mon et al. 97]. [Sha 95] a identifié plusieurs patrons de haut niveau (ex : architecture en couches, programme principal / sous-routines, invocation implicite, …) guidant la conception de la structure générale de systèmes logiciels. La description de chacun de ces patrons[8] inclut des informations concernant le problème qu'il résout, son contexte d'utilisation, sa solution (c'est à dire le modèle proposé par le patron), le diagramme correspondant, les variantes possibles, ainsi que des exemples d'utilisation. Les informations sont exprimées de façon narrative et se concentrent particulièrement sur les motivations de l'utilisation du patron.

Les styles architecturaux sont proches des patrons de deux manières ; un style architectural peut être vu comme un langage de patrons ou comme une sorte de patron.
Premièrement, un style architectural peut être perçu comme un langage fournissant un vocabulaire et un cadre avec lesquels un architecte peut construire des patrons utiles répondant à des problèmes spécifiques [Mon et al 97].
Deuxièmement, pour un style donné, il peut exister un ensemble d'utilisations idiomatiques. Ces idiomes sont comme des micro architectures, ou patrons architecturaux, conçus pour un style spécifique. Un style offre généralement des guides pour la construction des architectures propres à un domaine spécifique. Les patrons résolvent des problèmes plus petits et plus spécifiques pour un style donné (ou peut être pour plusieurs styles).

De ceci, nous retiendrons la définition suivante :

> Un ***patron architectural*** définit un modèle topologique. En d'autres termes, il exprime comment les éléments architecturaux sont disposés les uns par rapports aux autres.

Un style peut ainsi définir un ou plusieurs patrons architecturaux. Dans les styles de base tels que le style Pipe-Filter, le patron architectural est le cœur du style.

Familles et Lignes de produits

Les méthodologies de conception logicielle sont généralement construites pour supporter le développement d'applications individuelles. La définition de méthodes permettant le développement de lignes de produits, ou familles d'applications logicielles apparentées, est le résultat de motivations économiques et pratiques : il est trop coûteux de construire tous les membres possibles d'une famille ; il est plus rentable de construire des composants communs et de développer les membres de la famille à partir de ceux-ci [LoH&Bat 01]. En effet, les entreprises devant introduire de nouveaux produits, ou ajouter de nouvelles fonctionnalités à des produits existants en respectant de très courts délais, ne peuvent se permettre de les re-développer un par un.

Nous différencions les familles de produits des lignes de produits.

> Une ***ligne de produits*** est un ensemble de produits différents de par leur utilisation mais qui partagent des composantes en commun [DiN&Fug 96]. Une ***ligne de produits*** logiciels est un ensemble de systèmes logiciels partageant des fonctionnalités communes qui satisfont les besoins spécifiques d'un segment de marché particulier, et qui sont développés à partir d'éléments de base communs en suivant un processus prédéfini [Cle et al. 02].

Chaque produit de la ligne de produits est formé en prenant des composants de la base d'éléments communs, en les adaptant si nécessaire via des mécanismes de variation prédéfinis (paramétrage, héritage, …), en ajoutant les nouveaux composants qui peuvent être nécessaires, et en les assemblant conformément aux règles de l'architecture de la ligne de produit [Ber et al 03].

[8] La plupart des patrons sont orientés sur le développement objet [Gam et al. 95][Pre 95].

> Une ***famille de produits*** est un ensemble de produits qui ont la même utilité mais qui présente des variations [DiN&Fug 96]. Une ***famille de produits*** logiciels est un ensemble de systèmes logiciels dont la raison d'être est la même.

Plus concrètement, il s'agit de plusieurs versions du même produit. Ces versions peuvent dépendre de la plate-forme d'exécution, des utilisateurs ciblés (professionnels, particuliers, enfants, …), d'évolutions différentes du produit.

Les outils logiciels jouent un rôle important dans la mise en place des lignes de produits. Un sondage effectué par [Coh 02] auprès d'un échantillon d'entreprises développant des lignes de produits indique que ceux-ci sont principalement utilisés aux niveaux de la gestion des besoins ([Req], [Doo], …), de la modélisation de la conception ([Rat], [Rha], …) et de la gestion de configuration ([ClC], [CCC], …). En outre, ce sondage met en évidence la nécessité de définir une « architecture » de ligne de produits. Une architecture de ligne de produits (PLA - Product-Line Architecture) est un plan pour créer une famille d'applications apparentées [Bat 98]. Le succès d'une ligne de produits implique qu'elle soit bâtie sur une architecture qui supporte les variations découvertes dans la compréhension des domaines relatifs à la ligne de produits. L'architecture ne doit pas être simplement un « plan » de construction des produits, mais doit aussi inclure un mécanisme pour organiser le développement logiciel. De nombreuses méthodologies ont été mises en place pour créer des architectures de lignes de produits (ex : [Ame et al. 00], [Bat&Ger 97], [Bos 99], [DeB&Sch 99], [Gom et al. 94] et [Wei et Lai 99]).
[VdH 02] souligne cet aspect et traite dans ses travaux de l'utilisation d'ADLs pour la représentation d'architectures de lignes de produits. [Per 98] propose l'utilisation des architectures pour la modélisation de lignes de produits. Il propose plusieurs méthodes dont modéliser une ligne de produits par un style architectural. De même, [Lal 99] souligne les similarités entre les concepts de lignes de produits et des styles architecturaux.

3.2. Processus de conception orientée style

Les styles architecturaux sont des outils d'un très haut niveau d'abstraction. Comme nous l'avons déjà mentionné, les premiers styles ou styles de bases ont émergé naturellement de l'expérience du développement logiciel et en particulier de la conception architecturale. Ils sont utilisés très tôt dans le processus de développement d'un système logiciel, au début de la conception architecturale.

Les styles architecturaux ont longtemps été définis et utilisés comme des concepts, des guides de conception informels. Puisqu'un style architectural représente une famille entière de systèmes logiciels, il est désirable de formaliser le concept de style architectural pour à la fois avoir une définition précise de la famille de systèmes et pour étudier les propriétés architecturales communes à tous les systèmes de la famille [BerDon&Cia 02]. Les travaux menés depuis lors ont souligné l'intérêt de formaliser les styles [AboAll&Gar 95], et de nombreux langages dédiés à cette tâche ont été développés. Les ADLs devraient permettre la définition des styles architecturaux [DiN&Ros 99]. De plus ils devraient fournir un mécanisme pour exploiter un style dans la définition d'une architecture. Il est aussi utile de pouvoir définir des sous-styles.

La formalisation d'un style architectural est compliquée par la présence de deux degrés de liberté au sein de l'ensemble des instances du style :

- la variabilité du comportement interne des composants,
- la variabilité des topologies formées par les composants.

La plupart des ADLs ne sont pas satisfaisants pour la description des styles [DiN&Ros 99]. Ils introduisent eux-mêmes des suppositions spécifiques sur les architectures. Ces suppositions peuvent être incohérentes ou conflictuelles avec celles sous-jacentes au système. Par exemple, des langages ne permettent pas à deux ports d'être connectés directement à un même troisième. De nombreux ADLs sont spécifiques à un domaine et à un style en particulier. Par exemple, C2SADEL [MedRos&Tay 99] est basé sur le style C2 [Tay et al. 96] et MetaH [Ves 92][Ves 94] est un langage spécifique aux architectures des systèmes multiprocesseurs temps réel pour l'aviation.

Les styles sont présents tout au long du cycle de développement centré architecture, depuis la spécification des besoins jusqu'à l'exécution du système (cf. Figure 9). Les styles offrent un *cadre* et un *support* à la conception architecturale, mais un style en particulier peut promouvoir un aspect plutôt que l'autre suivant sa définition.

Le schéma ci-dessous montre que les styles peuvent être utilisés pour formaliser les spécifications d'une famille (ou d'une ligne) de produits à partir d'un cahier des charges, pour apporter un support à la description avec des notations spécifiques, pour apporter un support analytique ou pour apporter un support à la conception avec des styles répondant à des problèmes spécifiques.

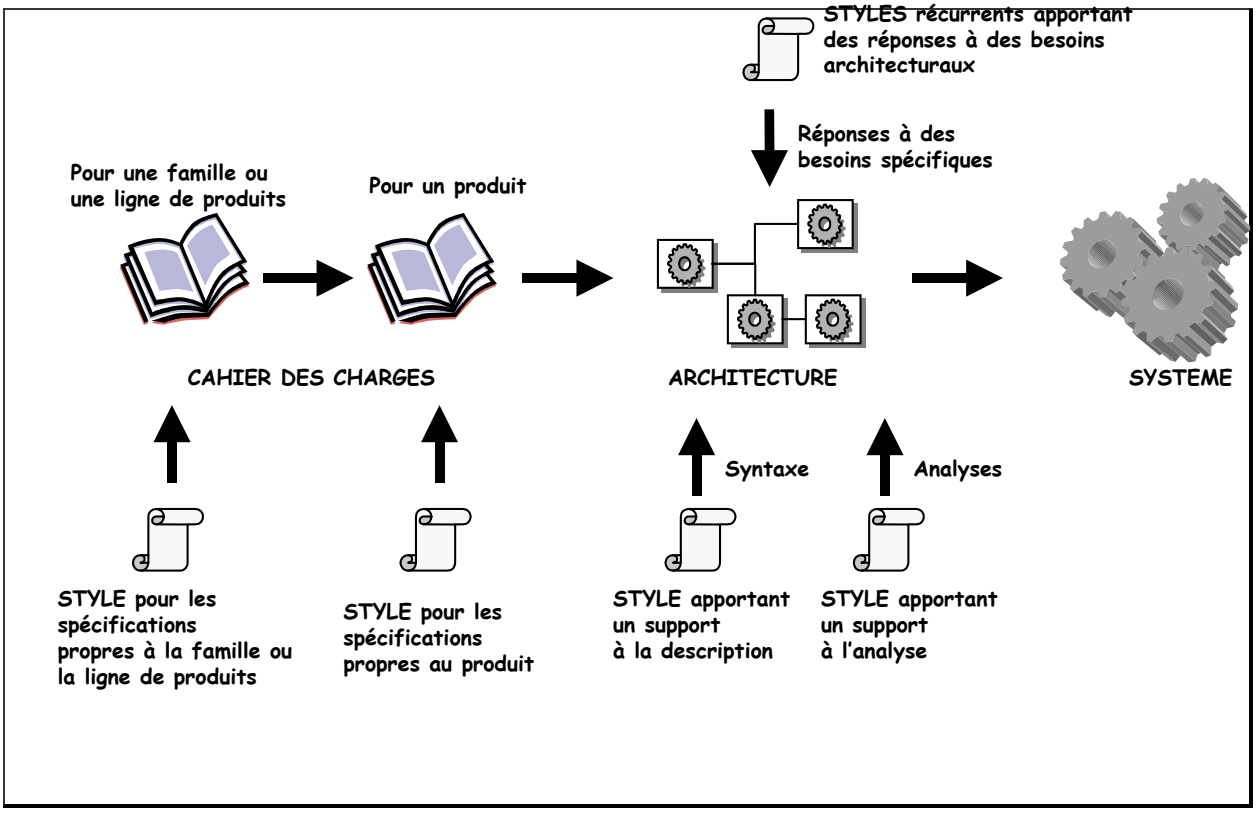


Figure 9 - Les styles dans un processus de développement centré architecture

La Figure 10 présente un processus de développement basé sur l'utilisation des styles architecturaux. Ce schéma (inspiré de [Arc 02]) reprend et étend le processus de développement centré architecture représenté par la Figure 3.

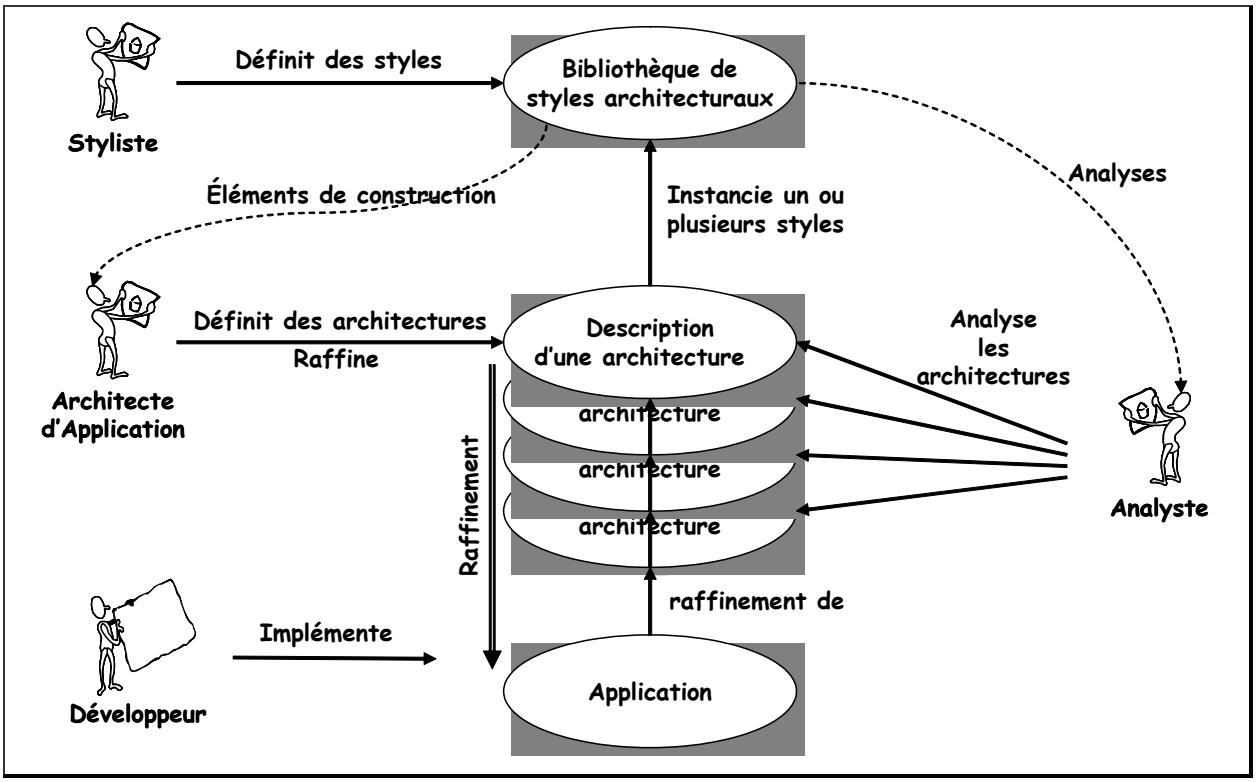


Figure 10 - Processus de développement architectural orienté style

Dans le processus que nous proposons, nous suggérons que les styles soient formalisés par un acteur spécifique, le ***styliste*** logiciel.

> Un ***styliste*** logiciel décrit soit des styles de bases, soit des styles spécifiques au développement d'un système ou à une famille de systèmes.

Dans le premier cas, les styles de bases sont définis à partir de descriptions informelles déjà existantes, qui ressortent des expériences et des connaissances dans le domaine logiciel.
Dans le deuxième cas, le styliste se base sur le cahier des charges pour fournir en premier lieu un style pour cadrer la conception. Le style est générique pour une famille de systèmes. Le styliste le spécialise alors pour chaque système en fonction du cahier des charges associé. Le style est d'abord défini comme un ensemble de contraintes. Puis, il est agrémenté d'un support à la conception architecturale sous forme d'éléments de construction, de patrons, et d'analyses. De plus, de nombreuses informations connexes peuvent faire partie d'une description de style comme :

- une documentation précisant quand et comment utiliser un style,
- une notation spécifique à la description des architectures suivant le style,
- une visualisation propre à la description graphique de ces architectures.

L'ensemble des styles formalisés constitue une base à l'attention des acteurs du développement.

L'***architecte*** se laisse guider par les styles imposants une feuille de route au développement et il peut utiliser d'autres styles choisis en fonction des propriétés que son architecture doit satisfaire. Les architectures construites sont des instances des styles.

La suite du développement est identique au processus précédemment étudié : le développeur raffine petit à petit l'architecture de base en architectures de plus en plus concrètes jusqu'à l'obtention de l'application finale.

Toutefois, l'utilisation des styles ne s'arrête pas là. En effet, les styles mettent à disposition de l'***analyste*** un nombre d'analyses spécifiques aux architectures construites suivant ces styles.

D'autre part, les styles architecturaux restent utiles après l'implémentation du système. En effet, ils permettent lors de la maintenance et des mises à jour de cadrer l'évolution de l'architecture du système.

Les ADLs proposant la formalisation des styles fournissent généralement des mécanismes d'instanciation et d'héritage.

> On dit qu'une architecture ***satisfait un style*** lorsqu'elle vérifie l'ensemble des contraintes définies par celui-ci.

> L'***instanciation d'un style*** est la définition d'une architecture telle que celle-ci satisfasse ce style.

L'instanciation permet de réutiliser les éléments définis au niveau d'un style pour définir une architecture satisfaisant un style.

Au lieu de définir une architecture suivant un style, on peut aussi définir un style suivant un autre style, dans ce cas on parle d'héritage.

> L'***héritage d'un style*** est la définition d'une relation entre deux styles tel que l'un satisfait l'autre.

Un style représente une famille d'architectures. Plus le style est abstrait, moins il est contraignant et plus la famille qu'il représente est grande [Per&Wol 92]. Un style peut être concret jusqu'à ne représenter qu'une architecture. On peut ainsi organiser les styles par leur niveau d'abstraction. Lorsqu'un style représente un sous-ensemble d'une famille d'architectures représentée par un style plus abstrait, on parle de sous-style. Ainsi, un sous-style hérite des contraintes définies par son parent et ne peuxt qu'en définir de nouvelles plus contraignantes.

> Un ***sous-style*** est un style qui en satisfait un autre.

Un sous-style permet la réutilisation de descriptions architecturales déjà existantes [DiN&Ros 99]. De plus, une organisation hiérarchique des styles guide l'architecte dans la sélection du paradigme le plus approprié pour une application et un domaine spécifique.

3.3. Qu’apportent les styles architecturaux ?

Un style architectural caractérise une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques [AboAll&Gar 93] (exemples : pipe-filter, client-serveur). Le but principal des styles architecturaux est de simplifier la conception des logiciels et la réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système [Mon et al. 97]. Un style architectural est moins contraignant et moins complet qu’une architecture spécifique. Il spécifie uniquement les contraintes les plus importantes, au niveau par exemple de la structure, du comportement, de l’utilisation des ressources des composants et des connecteurs dans un système [Abd 96]. D’une façon générale, les styles architecturaux permettent à un développeur de réutiliser l’expérience concentrée de tous les concepteurs qui ont précédemment fait face à des problèmes similaires [Kle&Kaz 99].

L’utilisation des styles architecturaux comporte des intérêts pratiques précis [Gar 95] :

- elle promeut la réutilisation au niveau de la conception (et non seulement au niveau du code, comme c’est le cas de la plupart des techniques de réutilisation) [Mon&Gar 96] : des solutions avec des propriétés bien comprises peuvent être ré-appliquées à de nouveaux problèmes en toute confiance.
- elle peut mener à une réutilisation significative de code : souvent les aspects invariants d'un style architectural se prêtent au partage des implémentations.
- elle permet de normaliser une famille d’architectures, ce qui améliore la compréhension de l’organisation d’un système. Il est plus simple aux autres de comprendre l'organisation d'un

système si des structures conventionnelles sont utilisées. Par exemple, la caractérisation d'un système comme une organisation en couche renvoie une forte image des types d'éléments qui collaborent dans les différents niveaux du système.

- elle permet l'utilisation d'analyses spécifiques au style concerné [Cia&Mas 96]. Contraignant l'espace de conception, un style architectural permet souvent des analyses spécifiques aux architectures construites dans le style. Par exemple, les analyses possibles pour les systèmes pipe-filter portent sur l'ordonnancement, la latence et l'absence d'interblocage.

4. Conclusion

Dans ce chapitre, nous avons introduit le domaine des architectures logicielles en mettant en avant les styles architecturaux.

Une architecture logicielle est une abstraction d'un système logiciel définissant un ensemble de composants, la description des interactions entre ceux-ci et un ensemble d'attributs permettant de dénoter les qualités du système. De plus, nous avons notamment souligné que la dynamique des systèmes et modélisable au niveau architectural.

Un style architectural est un support à la construction architecturale défini notamment en termes de contraintes, d'éléments de construction et de patrons. C'est aussi un cadre à la conception, il délimite par un ensemble de contraintes un espace de définition architecturale ; il représente ainsi une famille d'architectures.

A partir de cette étude, nous avons constaté l'importance de l'introduction des styles architecturaux dans un développement centré architecture.

Dans le chapitre suivant, nous présentons un état de l'art permettant de prendre connaissance des travaux effectués autour de ce thème. Puis, nous faisons ressortir du domaine notre problématique de recherche.

Chapitre 3

ETAT DE L'ART

Chapitre 3 - Etat de l'Art

1. Introduction

Depuis l'émergence du domaine des architectures logicielles, il y a une quinzaine d'années, de nombreux travaux ont été entrepris pour l'utilisation formelle des architectures, avec tout d'abord une forte concentration aux Etats-Unis puis partout dans le monde.

Parmi ces travaux, une variété de langages pour la conception architecturale a été créée pour fournir les architectes logiciels avec des notations pour spécifier des modèles architecturaux et pour pouvoir raisonner dessus [Mon et al. 97]. Les Langages de Description d'Architecture (ADL) et leurs outils soutiennent le développement centré architecture [Med&Tay 97].

Dans ce chapitre, nous présentons un état de l'art sur les outils formels utilisés pour la conception architecturale en correspondance avec la problématique que nous avons définie. Ainsi, nous nous plaçons dans un contexte de conception de systèmes dynamiques et d'utilisation des styles architecturaux. Nous étudierons les langages de description d'architecture (section 2) et les environnements de conception d'architectures (section 3). Nous établirons ensuite un bilan des systèmes présentés selon un ensemble de critères (section 4). Puis, nous verrons pourquoi les travaux étudiés ne couvrent pas la problématique adressée dans cette thèse (conclusion).

2. Les langages de description d'architectures

Parmi les outils destinés au domaine architectural, ceux qui entrent en première ligne sont les langages de description d'architectures. Ils permettent de décrire formellement, sans ambiguïté, les architectures logicielles.

Avant de présenter ces langages ainsi que leur classification, la prochaine section introduit, à travers quelques considérations, les langages supportant la dynamique et les styles.

2.1. Considérations

A travers ces considérations, nous voulons tout d'abord préciser que plusieurs travaux sur les architectures sont fondés sur des algèbres de processus[9]. Parmi celles-ci, il y a les algèbres de processus telles que CSP [Hoa 85] et le π-Calcul [Mil 89]. Les algèbres de processus [Mil 89, Hoa 85] sont des algèbres qui supportent la description compositionnelle de systèmes distribués et concurrents, ainsi que la vérification formelle de leurs propriétés. Le μ-Calcul [Koz 83] est un calcul pour exprimer les propriétés des systèmes à transitions étiquetées (LTS - Labelled Transition System) par l'utilisation d'opérateurs de points fixes minimum et maximum. CHAM [Ber&Bou 92] (Chimical Abstract Machine - Machine Chimique Abstraite) est un formalisme dans lequel les systèmes logiciels sont vus comme des solutions chimiques dont les réactions sont contrôlées par des règles d'état explicites. [Inv&Wol 95] ont exploré une approche pour spécifier et analyser formellement des architectures logicielles décrites en CHAM. Z [Spi 89] est un langage de spécification pour la description de modèles formels. [AboAll&Gar 95] ont élaboré une méthode structurée pour définir les styles architecturaux en Z.

Nous structurons la présentation des ADLs autour de leur capacité à représenter la dynamique du système et les styles architecturaux.
Lors de nos travaux, nous avons établi une classification de ces langages selon plusieurs critères. Nous avons choisi de présenter cette classification après la présentation des ADLs et des environnements sous forme de bilan récapitulatif.

[9] Nous présentons le π-Calcul et le μ-Calcul en annexe de ce document. Comme nous le verrons par la suite, ces algèbres sont les fondements formels des travaux à partir desquels nous construisons notre solution.

Concernant la dynamique, il y a deux tendances, celle qui consiste à la contraindre et celle qui consiste à la prévoir. Dans le premier cas, on donne un cadre à la dynamique à travers un certain nombre de contraintes. On peut spécifier les éléments qui sont dynamiques et combien d'occurrences peuvent être créées dynamiquement. Dans le second cas, on spécifie explicitement le comportement dynamique. On exprime, par exemple, quand les éléments sont dynamiquement créés.

Concernant les styles, il est tout d'abord à noter que chaque ADL supporte au moins un style, celui-ci représentant la sémantique et les mécanismes de l'ADL [LeyCim&Oqu 02]. Ce style est plus ou moins générique selon le domaine de l'ADL, par exemple : META-H [Ves 92] est un ADL spécifique aux architectures des systèmes multiprocesseurs temps réel pour l'aviation [Bin et al. 96] ; ADAGE [Cog&Szy 93] : ce langage permet la description de structures architecturales pour le guidage et la navigation dans le domaine de l'aviation ; ACME [GarMon&Wil 97] et π-SPACE [Cha 02] ne sont pas spécifiques à un domaine, leur style propre est plus générique. Afin d'augmenter le niveau de réutilisation et de s'adapter à n'importe quel domaine, quelques ADLs permettent aux utilisateurs de formaliser leurs propres styles. Les styles sont généralement considérés comme des supports en termes d'éléments réutilisables et/ou comme des ensembles de contraintes. Mais, les ADLs permettent de spécifier d'autres informations au sein des formalisations de styles afin de pouvoir les exploiter. Ces informations peuvent être : des analyses d'architectures, des règles de traduction pour la génération automatique de code, une syntaxe spécifique au style, une visualisation des éléments architecturaux, des outils spécifiques ou de la documentation.

Nous nous concentrons sur deux classes d'ADL : ceux capables de formaliser des architectures dynamiques et/ou de formaliser des styles architecturaux. Voici la liste des ADLs que nous avons étudié : ACME [GarMon&Wil 00], ARMANI [Mon 01], WRIGHT [All 97], π-SPACE [Cha 02], σπ-SPACE [LeyCim&Oqu 01], ARCHWARE ADL [Cim at al. 02], AML [wil 99], DARWIN [Mag et al. 95], RAPIDE [RDT 97], CSADEL [MedRos&Tay 96], UNICON [Sha et al. 95].

Dans la présentation des ADLs, nous considérons pour chacun d'entre eux les aspects suivants :

- le contexte dans lequel il a été développé,
- les concepts sur les architectures, la dynamique et les styles architecturaux,
- les mécanismes qu'il propose,
- des exemples,
- l'environnement de développement associé,
- les avantages et les inconvénients.

2.2. ACME et ARMANI

L'origine du développement d'ACME [GarMon&Wil 96][GarMon&Wil 00] est venue de la volonté de vouloir combiner l'utilisation des outils associés à des formalismes différents. C'est un langage générique pour la description des architectures, conçu pour faciliter l'échange de descriptions architecturales entre différents ADLs et outils. Il supporte la description d'architectures composant-connecteur et la description de familles d'architectures. Dynamic ACME [Wil 01] est une extension permettant de décrire la dynamique des architectures. ARMANI [Mon 01] est une extension d'ACME conçue pour capturer les expertises de conception d'architectures logicielles et pour spécifier des modèles d'architectures logicielles ; en d'autres termes, ARMANI supporte la description des styles architecturaux.

Concepts sur les architectures

ACME fournit une ontologie architecturale consistant en sept éléments de conception. D'abord, il y a les briques de construction : les composants (***Component***) et les connecteurs (***Connector***). Ceux-ci sont interfacés par des ***Ports*** dans le cas d'un composant et des ***Rôles*** dans le cas d'un connecteur. Les composants et les connecteurs sont interconnectés par des attachements (***Attachments***) au sein d'un système (***System***). Ensuite, un mécanisme appelé ***Représentation*** permet la décomposition hiérarchique d'un élément architectural en un sous-système. Associé à ce mécanisme un autre appelé ***Rep-map*** permet de faire correspondre la représentation interne d'un système avec l'interface externe d'un composant ou d'un connecteur.

Ces sept classes d'éléments de conception sont suffisantes pour définir la structure d'une architecture comme un graphe hiérarchique de composants et de connecteurs. Afin de s'accommoder avec une large variété d'informations auxiliaires, ACME supporte une annotation des éléments de conception avec des listes de propriétés (***Properties***). Chaque propriété a un nom, un type optionnel et une valeur.

L'exemple ci-dessous montre la formalisation d'un système Client-Serveur avec un composant *client* et un composant *server* reliés par un connecteur *rpc*. Dans cet exemple, les propriétés sont utilisées de plusieurs manières. Elles font référence à des descriptions de style Client-Serveur définies dans d'autres systèmes (Aesop et Unicon). Elles font référence à des implémentations (par exemple : "CODE-LIB/client.c"). Elles font référence à un comportement défini en WRIGHT pour le connecteur *rpc*. Elles apportent des informations supplémentaires à l'architecture comme le type de communication : synchrone ou non.

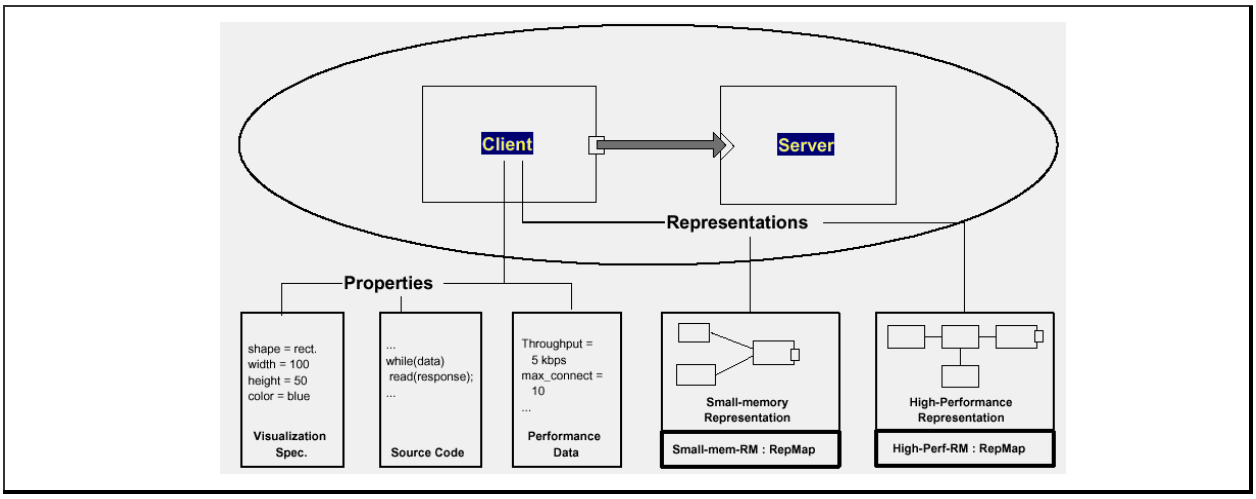


Figure 11 - Diagramme d'un système Client-Serveur simple

```
System simple_cs =
{     Component client = {
      Port send-request;
      Properties { Aesop-style : style-id = client-server;
            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 }
}
```

Concepts sur la dynamique

Dynamic ACME étend ACME pour supporter la dynamique. On y différencie deux classes d'éléments architecturaux : les éléments *fermés* (qui sont statiques) et les éléments *ouverts* (qui peuvent évoluer). Ainsi, un élément décrit en ACME est implicitement fermé.
On explicite avec le modificateur ***open*** qu'un élément peut évoluer. Associé à l'élément, on peut de plus indiquer, via une cardinalité, le minimum et le maximum de ses occurrences.
Les occurrences d'un élément *E* sont différenciées par des numéros : *E[1]* est la première occurrence *E* ; *E[*]* représente toutes les occurrences de *E*. Pour une meilleure lisibilité, l'opérateur ***Let*** permet de donner un nom à une occurrence : par exemple, ***Let*** *FirstE=E[1]*, nomme *FirstE* la première occurrence de *E*.

Dynamic ACME permet de spécifier la sémantique de la dynamique. Cette sémantique est décrite à travers des contraintes exprimées par des prédicats. Un opérateur, ***id***, traduit l'identification d'un élément. L'idée principale est que les propriétés et les relations entre les éléments ne sont valables pour un élément dynamique que si celui-ci est identifié.
Par exemple, l'attachement dynamique de deux éléments qui s'exprime de la manière suivante :

```
Attachments { I.Out to F.In }
```

ce traduit par :

```
id I and id F => attached-to(I.Out, F.In).
```

```
open System Fusion ={
     open 0.. Connector R = {
          Roles { In; Out}
          Properties { Delay:integer}}
     Let RI =R[1]
     open 1.. Connector B = {Roles {Listen; Talk}
          Properties { Delay:integer}}
     open Component S = {
          Port Broadcast Properties { Protocol = Serial };
          Properties { DataRate: rational }}
     open 0.. Component A= {
          Port In Properties { Protocol= Serial }}
          Port Out Properties { Delay:integer}}
Let AI =A[1]
Group P = { members { I, PI, F, AI, RI }}
Component I = {Ports {0.. In, Out}; Properties { Delay:integer}}
Component PI = {Ports {In, Out, Synch}}
Connector F = {Roles {In, Out}
Properties { Delay:integer}}
Attachments{ I.Out to F.In;
PI.In to F.Out;
S.Broadcast to B[*].Listen;
A[*].In to cor B.Talk;
cor A.Out to R[*].In;
I. cor In to R[*].Out }
Constraints: Correlate B whenever A
Correlate A whenever R
Correlate In whenever R
(exists j :: id R[j] and j > 1) => not id AI
id RI => id AI
id B[1]
Tmax > delay(Fusion)}
```

Concepts de styles

ACME permet de décrire des *familles* et des *types* d'éléments. Ce sont des mécanismes permettant de définir des éléments puis de les réutiliser et de les étendre. Ces travaux ont été prolongés dans une extension d'ACME, ARMANI qui offre un meilleur support pour la définition des styles notamment par la possibilité de définir des contraintes.
Dans ARMANI, les styles architecturaux capturent l'expertise de conception et sont, pour cela, constitués des instances des cinq types d'entités suivantes :

- Les Types d'éléments sont des prédicats décrivant la structure fondamentale et les contraintes du vocabulaire de conception pour les éléments architecturaux (composants, connecteurs, ports et rôles). En d'autres termes, un type est la définition réutilisable d'un élément.
- Les Types de propriétés sont des prédicats décrivant le type et la structure des propriétés.
- Les Invariants de conception capturent les contraintes devant être respectées à la conception. Les invariants sont exprimés dans un langage de prédicat spécifique. Ce dernier permet de spécifier des invariants concernant les éléments structurels et les propriétés. L'exemple suivant montre comment contraindre la valeur d'une propriété.

```
Property buffersize : int;
Invariant buffersize >= 0;
```

- Les Heuristiques de conception capturent des conseils pour une conception efficace. Les heuristiques sont exprimées dans le langage de prédicat d'ARMANI.
- Les Analyses de conception sont des fonctions traitées sur les architectures instanciées. Ces analyses peuvent être spécifiées dans le langage de prédicat d'ARMANI ou comme des fonctions externes liées à l'environnement.

Les styles sont instanciés pas des systèmes. Ces derniers :

- doivent respecter les invariants exprimés au niveau du style,
- héritent d'une structure minimale si elle est définie dans le style,
- doivent contenir des éléments instanciant des types définis au niveau du style,
- peuvent instancier plusieurs styles (seule la structure minimale d'un seul style est explicitement héritée dans ce cas).

ARMANI permet des relations d'héritage entre les styles. Un sous-style peut étendre un style existant (voire plusieurs styles existants) pour réutiliser les types et les règles de conception. Un sous-style est l'union des types, des règles de conception, des analyses et de la structure définis à la fois dans la définition de ses styles parents et dans sa propre définition. Il ne peut pas redéfinir des types et des règles existantes dans ses styles parents, il peut cependant définir de nouveaux types étendant des types définis dans les styles parents.

L'exemple suivant montre la définition d'un nouveau style appelé *sub* et étendant le style *super*. Il définit un nouveau type de composant et un nouvel invariant. *foo* représente un prédicat qui doit être vérifié pour tous les composants d'une architecture suivant le style *sub*.

```
Style super = { ... };
Style sub extends super with {
     Component type new-component = { ... };
     Invariant forall x in self.components • foo(x);
};
```

2.3. WRIGHT

WRIGHT [AboAll&Gar 93][All&Gar 94][All 97][All&Gar 97] fournit des bases formelles pour spécifier les interactions entre les composants (via des connecteurs). Cet ADL permet la définition des comportements au moyen d'une algèbre de processus, le CSP [Hoa 85]. Les extensions de WRIGHT [All 97][AllDou&Gar 97][AllDou&Gar 98] permettent d'une part, la définition des styles

architecturaux d'une manière similaire à ARMANI, et d'autre part, la description du dynamisme des architectures.

Concepts sur les architectures

Les concepts de WRIGHT sont extrêmement proches de ceux d'ACME. Il y a les composants, les connecteurs, les ports et les rôles. Un ensemble d'éléments interconnectés est appelé une ***configuration***.

La spécification en WRIGHT de l'exemple du client-serveur est donnée ci-dessous.

```
Configuration ClientServer
     Component Client
          Port Access = [Protocol d'interaction]
          Computation = [Requiert un service et effectue un traitement]
     Component Server …
     Connector Link …
     Instances
          C : Client ;
          L : Link ;
          S : Server
     Attachments
          C.request as L.reply ;
          S.reply as L.request
End Configuration
```

Dans la configuration, sont déclarées les instances *C* et *S* des composants de type *Client* et *Server*, ainsi que l'instance *L* du connecteur de type *Link*. Puis, dans la partie ***Attachments***, sont déclarés les attachements entre les instances de composants et les instances de connecteurs. Dans l'exemple, le port *request* de l'instance *C* est attaché au rôle *reply* de l'instance *L* et le port *reply* de l'instance *S* est attaché au rôle *request* de l'instance *L*.

Concepts sur la dynamique

Dynamic WRIGHT [AllDou&Gar 97][AllDou&Gar 98] étend WRIGHT et supporte des manipulations pour les changements dynamiques de la topologie d'une architecture. Pour cela, il définit un *configurateur* (***Configurator***) décrivant les différentes évolutions possibles de l'architecture. Dynamic WRIGHT centralise la gestion et le contrôle de la dynamique dans le configurateur. La solution adoptée met en œuvre des événements de contrôle. Ces événements de contrôle sont reçus directement par le configurateur et permettent le déclenchement d'actions de reconfiguration comme : ***new***, ***del***, ***attach*** et ***detach***.

Pour mettre en évidence les propriétés dynamiques de Dynamic WRIGHT, complétons l'exemple du client-serveur. Considérons que le système possède deux serveurs, un serveur primaire qu'il est préférable d'utiliser, et un secondaire qui sert de serveur de secours quand le premier tombe en panne. Initialement, le connecteur relie le client au serveur primaire. Dès que le serveur primaire tombe en panne la connexion est interrompue. Elle est ensuite rétablie vers le serveur secondaire jusqu'à la restauration du serveur primaire. Pour simuler cette situation, des modifications sont apportées dans la description des composants et connecteurs : des événements de contrôle sont introduits dans leur alphabet. Ainsi, le serveur primaire signale sa panne à l'aide de l'événement de contrôle *control.down* et son rétablissement avec *control.up*.

Le configurateur est spécifié ci-dessous.

```
Configurator ClientServer
     new.C : Client
          -> new.Primary : FlackyServer
          -> new.Secondary : SlowServer
          -> new.L : FaultTolerantLink
```

```
            -> attach.C.request.to.L.reply
            -> attach.Primary.reply.to.L.request -> WaitForDown
Where
      WaitForDown = Primary.control.serverDown
            -> detach.Primary.reply.from.L.request
            -> attach.Secondary.reply.from.L.request
            -> WaitForUp
             §
      WaitForUp = Secondary.control.serverUp
            -> detach.Secondary.reply.from.L.request
            -> attach.Primary.reply.from.L.request
            -> WaitForDown
             §
```

Dans l'exemple, les instances intervenant dans le changement de la configuration sont déclarées ainsi que les attachements initiaux. Le port request de l'instance *C* est attaché au rôle *reply* de l'instance L et le port *reply* de l'instance *Primary* (représentant le serveur primaire) est attaché au rôle *request* de l'instance *L*. Puis le configurateur se met dans l'attente d'un événement de contrôle, comme décrit dans la partie par *WaitForDown*. Dans cette partie, sont spécifiées deux situations possibles : le ***where*** système peut fonctionner correctement et se terminer avec succès (§), ou une erreur peut apparaître avant la terminaison de l'application (réception de l'événement de contrôle *serverDown* de l'instance *Primary*). Si le serveur primaire est détaché du connecteur *detach.Primary.reply.from.L.request* et que le second est attaché à sa place *attach.Secondary.reply.from.L.request*, la nouvelle configuration est alors établie. Au rétablissement du primaire, *WaitForUp* sera activé, modifiant de nouveau la configuration de manière similaire ; les rôles entre les deux serveurs sont alors inversés.

Dynamic WRIGHT propose aussi des actions ***new*** et ***delete*** permettant la création et la suppression de composants et connecteurs de manière dynamique. Or Dynamic WRIGHT est basé sur CSP qui ne peut décrire que des configurations statiques de processus. Pour permettre la création et la suppression de composants, Dynamic WRIGHT restreint les systèmes à ceux qui ont un ensemble fini de configurations possibles. De plus, il renomme les événements avec la configuration quand ils apparaissent. Chaque version C_p d'un composant transformé commence par un événement $C_p.go.p_1.Cn_1.r_1 \ldots r_n.Cn_n.r_n$ sélectionnant la version C_p où chacun de ces ports p_i ($i \in [1..n]$) est attaché au rôle r_i de Cn_i. Le configurateur est transformé dans un processus CSP où les actions ***new***, ***del***, ***attach*** et ***detach*** sont compilées dans des événements $C_p.go\ldots$ qui sélectionnent la propre sélection du composant à chaque étape de reconfiguration.

Concepts de styles

En plus de permettre de décrire et d'analyser des configurations de système, WRIGHT permet au concepteur de décrire et analyser les familles de systèmes, ou des styles architecturaux. En formalisant un style, l'architecte peut analyser toute une famille de systèmes et réduit ainsi l'effort de production de ces différents systèmes. Un style en WRIGHT a deux parties : le vocabulaire et les contraintes sur des configurations. Le vocabulaire définit des types de composant et de connecteur. Les contraintes définissent les prédicats qui doivent être vrais pour n'importe quelle configuration suivant le style.

Par exemple, on peut considérer une famille d'architectures dont toutes possèdent un connecteur appelé *Link*. Nous pouvons définir un style pour définir cette famille comme nous le montrons ci-dessous. Ce style fournit le connecteur *Link*, de sorte qu'il puisse être employé dans toutes les configurations suivant le style, ainsi qu'une interface générique *LInterface*. En outre, le style indique qu'il n'y aura qu'une forme d'interaction, et qu'elle passera par l'intermédiaire d'un connecteur *Link*. Puisqu'il n'y a aucune contrainte placée sur des composants, une configuration suivant le style est libre pour employer n'importe quelle collection de composants qui peuvent participer au connecteur *Link*. Le nombre et le traitement des composants peuvent changer.

```
Style CS
     Interface Type LInterface = [Interaction of one simulation]
     Connector  Link (nsims : 1..)
          Role Model1nsims = LInterface
          Glue = [Data travels from one Model to another]
     Constraints
          ∃ L : Connectors|{L}=Connectors ∧ Type(L) Link
End Style.
```

Un style est instancié par une configuration, dont la définition est simplifiée car elle hérite des définitions par le style.

```
Configuration ClientServer
     Style CS
     Component Client …
     Component Server …
     Instances  …
     Attachments …
End ClientServer.
```

2.4. π-SPACE et σπ-SPACE

π-SPACE [Cha&Oqu 01][Cha 02] est un ADL pour la description des architectures dynamiques à composants. Il est formellement fondé sur le π-calcul dont il retire des propriétés pour décrire les systèmes dynamiques. Ce langage est étendu par σπ-SPACE [LeyCim&Oqu 01] pour la description des styles architecturaux.

Concepts sur les architectures

Les architectures sont représentées par des configurations de ***composites***, de ***composants***, et de ***connecteurs***. Un composite est un élément composé et définissant une configuration.
Les composants et les connecteurs sont composés :

- de ports qui définissent l'interface de communication. Ils sont eux mêmes définis comme un ensemble de canaux (points d'interaction élémentaires),
- d'un comportement définissant les interactions de l'élément. Un comportement est défini sur les bases d'une description de processus en π-calcul,
- d'opérations (pour les composants) qui définissent les traitements internes.

L'exemple suivant est la description d'un élément composite représentant un système client-serveur.

```
compose Simple {
     C :Client ||
     L:Link ||
     S:Server
where
     attach C@p@request to L@c@cReply,
     attach C@p@reply to L@c@cRequest,
     attach S@p@reply to L@s@sRequest,
     attach S@p@request to L@s@sReply
}
```

Des instances des types *Client*, *Link* et *Server* sont définies. Ces instances sont interconnectées par leurs ports et plus précisément par leurs canaux. Ainsi, le canal *request* du port *p* du composant *C* (C@p@request) est attaché au canal *cReply* du port *c* du composant *L* (L@c@cReply).

Concepts sur la dynamique

π-SPACE permet de formaliser la dynamique d'une architecture. Lors d'une description, on peut définir quels éléments sont dynamiques, leurs noms sont alors suffixés du caractère ***π***. Dans l'exemple suivant, le client est déclaré comme dynamique. Cela signifie qu'il peut y avoir plusieurs occurrences créées dynamiquement. La clause ***whenever*** permet de spécifier des réactions à des évènements. Ici il est spécifié que lorsqu'un nouveau client est créé, celui-ci est attaché au connecteur et que ce dernier évolue (un nouveau port est créé) pour permettre cela.

```
compose Simple {
     C π:Client ||
     L:Link ||
     S:Server
where
     attach S@p@reply to L@s@sRequest,
     attach S@p@request to L@s@sReply
whenever
     new Cπ ⇒ new L@cπ,
     new L@cπ ⇒ attach Cπ@p@request to L@cπ@cReply,
     new L@cπ ⇒ attach Cπ@p@reply to L@cπ@cRequest;
}
```

π-SPACE permet de définir des règles de dynamique, afin de préciser les changements topologiques dus à la création dynamique d'un nouvel élément. La dynamique peut être déclenchée par un composant. Pour cela, des ports spécifiques sont définis : *attachementEvolutionPort*, *ComponentEvolutionPort* et *EvolutionPort*. Ces ports permettent d'évoquer des reconfigurations, pour cela ils sont sollicités via le mot-clé ***evolvable***.
L'exemple ci-dessous définit le comportement d'un composant. Ce comportement engage deux ports, un (*pC*) qui permet la communication avec un autre élément, et un qui permet de faire évoluer la topologie (*changementAttachement*). Le comportement commence par la réception d'un message via le port *pC*, puis continue par le déclenchement d'un événement pour une évolution via le port *changementAttachement*.

```
define behaviour component type ExempleComportement
     [pC:PortReception [canalReception:[] ],
     changementAttachement:AttachmentEvolutionPort] {
ExempleComportement[pC, changementAttachement] =
     ( pC@canalReception () •
     evolvable(changementAttachement) •
     ~Boucle[pC] ),
Boucle[pC] = ( ( pC@ canalReception () • ~Boucle[pC] ) + $ )
}
```

Cet événement est traité au niveau de la composition du système dans la clause ***whenever***. A la réception de cet événement, le composant *C1* est détaché du composant *C2* puis attaché au composant *C3*.

```
compose ExempleComposition
{
C1 : ExempleComposant ||
C2 : Component2 ||
C3 : Composite3
where
     attach C1@pC2@canalReception to C2@pC1@canalEmission
whenever
     evolvable(C1@changementAttachement) ⇒
          unattach C1@pC@canalReception to C2@pC1@canalEmission
     unattach C1@pC@canalReception to C2@pC1@canalEmission ⇒
          attach C1@pC@canalReception to C3@pC1@canalEmission ;
```

```
}
```

Concepts sur les styles

Comme nous l'avons mentionné, σπ-SPACE [LeyCim&Oqu 01] étend π-SPACE avec des concepts spécifiques aux styles. Dans σπ-SPACE, un style permet de contraindre un type d'élément. Un style peut être défini pour des composites, des composants, des connecteurs, des ports, des comportements et des opérations.
La définition d'un style est un ensemble de contraintes qui représentent les caractéristiques d'une famille de types. On y différencie les contraintes topologiques des contraintes comportementales.

Un style contient une liste d'éléments de construction. Ces éléments possèdent des noms génériques qui sont spécialisés lorsqu'une occurrence de l'élément est utilisée dans une architecture. De plus, les éléments de construction peuvent être associés à une cardinalité définissant une contrainte sur le nombre d'occurrences dans l'architecture.
Dans l'exemple suivant, le style *Filter* est un style de composant. Il définit deux ports de type *In* et *Out*. Leurs occurrences porteront des noms commençant respectivement par ***in***_ et par ***out***_. De plus, les composants suivant le style *Filter* doivent contenir au moins un port "in_*inname*" et un port "out_*outname*".

```
define component style Filter
     port in_inname[1..*] : In ||
     port out_outname[1..*] : Out;
```

Enfin il est possible de spécifier les éléments qui peuvent être dynamiques. Ici, le symbole ***π*** indique qu'un élément peut être dynamique.

```
define composite style Pipe_filter
     filter_fnameπ [2..*] in style Filter
```

Il est possible de définir des règles de configuration dans un style de composite, en décrivant les attachements possibles. Dans l'exemple suivant, il est spécifié que 2 filtres peuvent être attachés à 1 ou plusieurs pipes.

```
define composite style Pipe_filter
     filter_fname [2..*] in style Filter
     pipe_pname [1..*] in style Pipe
     where
          filter_fname [2..2]        can be attached to pipe_pname [1..*];
```

Concernant les contraintes comportementales, σπ-SPACE permet de définir des comportements génériques ou encore des patrons comportementaux.
On peut, par exemple, spécifier qu'un serveur reçoit parallèlement des requêtes qu'il traite et envoie des réponses :

```
specification {
     Serveur [port_any] =
          %<|>
          port_name(requete)●
          traitement[in(requete),out(reponse)]●
          port_name<reponse>
%}
```

Les styles permettent de vérifier que des descriptions de types présentent certaines caractéristiques. On utilise le mot-clé ***in style*** pour spécifier qu'un type doit satisfaire un style. Le type doit alors respecter les contraintes imposées par le style et il ne peut utiliser dans sa description que les éléments de construction fournis par le style.

```
define component type AFilter in style Filter ...
```

Un style peut hériter d'un autre ; on utilise le mot-clé ***extends***. Le sous-style peut seulement ajouter de nouvelles contraintes à celles héritées de son parent.

```
define component style UnixFilter extends Filter …
```

2.5. ARCHWARE ADL

ArchWare ADL [Oqu et al. 02][Cim et al. 02] fournit la structure de base et les constructions comportementales pour décrire les architectures logicielles évolutives. C'est un langage formel de spécification conçu pour être exécutable et pour supporter l'analyse et le raffinement automatique des architectures. ArchWare ADL est un langage formel défini comme une extension spécifique à un domaine du π-calcul typé d'ordre supérieur. ArchWare ADL est associé à un langage d'analyse, ArchWare AAL (Langage pour les Analyses des Architectures). Celui-ci est défini comme une extension spécifique à un domaine du μ-calcul.

Concepts sur les architectures

En ArchWare ADL, les éléments architecturaux sont définis en terme de comportements (***behaviour***). Un comportement est défini par un ensemble d'actions ordonnancées qui spécifie à la fois le traitement interne de l'élément architectural (actions internes) et les interactions avec son environnement (actions de communication).
Un élément architectural communique avec les autres par une interface représentée par un ensemble de ***connexions***. Les éléments architecturaux communiquent en transitant des données par ces connexions. Pour interagir, les éléments architecturaux sont mis en relation par un mécanisme de composition et un mécanisme de liaison, appelé ***unification*** (c'est une substitution au sens du π-calcul). Lorsque plusieurs éléments sont composés, ils peuvent interagir lorsque leurs connexions sont liées.
Afin de réutiliser des définitions de comportement, ArchWare ADL fournit un mécanisme de réutilisation appelé ***abstraction***. L'abstraction permet d'encapsuler des définitions paramétrables de comportement. On obtient un comportement d'une abstraction par l'*application* de cette dernière, en fournissant éventuellement une liste de paramètres.

Dans l'exemple suivant, un serveur (*server*) et un client (*client*) sont définis. Un système *OneClientOneServer* est défini comme la composition des instances de ces éléments. Le client est défini comme une abstraction *récursive*. Le comportement du client consiste à envoyer un appel, attendre une réponse, puis effectuer un traitement inobservable (***unobservable***) en fonction de cette dernière, puis à recommencer de nouveau depuis le début.

```
recursive value client is abstraction ();{
     via call send; via wait receive; unobservable; client()};
recursive value server is abstraction ();{
     via request receive; unobservable; via reply send; server()}
```

Le système aussi est défini comme une abstraction, mais son comportement est la composition (la mise en concurrence) d'un comportement de type client, *c,* et d'un comportement de type server, *s.* Pour que ces comportements, représentant les éléments architecturaux du système, soient en relation, des connexions sont définies (par exemple : *wait* et *reply*) et unifiées pour permettre la communication (*reply* ***unifies*** *wait*).

```
value oneClientOneServer is abstraction ();{
     value call, wait, request, reply is connection();
     compose { c is client() where {reply unifies wait}
     and       s is server() where {call unifies request}
     }
}
```

Concepts sur la dynamique

ArchWare ADL fournit des mécanismes qui promeuvent la définition d'architectures dynamiques. D'une part, l'abstraction est un mécanisme permettant la création dynamique. En effet, une abstraction est appliquée à la volée, ainsi plusieurs éléments peuvent être créés dynamiquement à partir d'une même définition. Cela permet l'introduction de nouveaux éléments architecturaux. D'autre part, ArchWare ADL hérite des propriétés du π-calcul d'ordre supérieur et promeut ainsi la mobilité. C'est à dire que des connexions ainsi que des comportements peuvent transiter par d'autres connexions. Cela permet le changement dynamique de la topologie de l'architecture.

Nous verrons que nous utilisons ce langage dans nos travaux. Nous utilisons le langage ArchWare AAL qui permet la description de propriétés architecturales sur des architectures décrites en ArchWare ADL. Ainsi, nous donnerons plus de détails sur ces langages dans le prochain chapitre.

2.6. AML

AML (Architectural Meta-Language) [Wile 1999] est un métalangage pour la spécification de la sémantique des ADLs. Il permet notamment de spécifier des structures et d'en contraindre leur évolution. AML est basé sur trois concepts :

- element : l'élément de base,
- relationship : pour décrire des relations entre les éléments,
- kind : pour spécifier des types ou des styles d'éléments.

Concepts sur les architectures

Dans AML, les architectures logicielles sont représentées comme un ensemble d'éléments (*element*) liés par des relations topologiques (*relationship*) soigneusement décrites et contraintes.
Le mot clé ***element*** permet de déclarer des instances. Par exemple, on déclare le système global et ses constituants.

```
element ClientServer, Client, Server
```

La structure est définie par des assertions via le mot clé ***assume***. Dans l'exemple suivant, on s'appuie sur la relation *has-part* pour spécifier que *Client* et *Server* sont des constituants de *ClientServer*.

```
assume has-part(ClientServer,Client) ∧ has-part(ClientServer,Server)
```

La relation *has-part* est définie comme suit :

```
relationship has-part(b,a)
     assume ~ exists p : has-part(p,a) ^ p <> b
```

Concepts sur la dynamique

AML ne permet pas de spécifier la dynamique d'une architecture. Mais il permet de définir des contraintes à travers des assertions temporelles utilisant les quantificateurs ***sometimes*** et ***always*** et en vérifiant l'existence d'un élément avec le quantificateur existentiel ***id***.

AML permet de spécifier comment l'architecture réagit lorsque sa structure évolue. Dans l'exemple suivant, on spécifie que :

- si l'élément Helper est défini alors l'élément Main.Instruments est défini,
- si l'élément Controller est en mode d'urgence (Emergency) alors l'élément Main.Instruments est défini et l'élément Main.Surfaces.Secondary est supprimé.

```
assume id Helper => id Main.Instruments
assume mode(Controller) = Emergency => id Main.Instruments
assume mode(Controller)= Emergency  => ~ id Main.Surfaces.Secondary;
```

L'exemple suivant utilise le quantificateur ***sometimes*** pour exprimer que l'élément *Helper* doit être au moins défini une fois au cours de l'évolution de l'architecture.

```
assume sometimes id Helper;
```

Concepts sur les styles

Dans AML, les styles sont décrits autour du concept ***kind***. Les kinds jouent le rôle des styles. Ils sont associés à des relations pour imposer des contraintes. Deux éléments vérifient une relation si les assertions définies par celle-ci sont validées. Par exemple, la relation suivante définit l'appartenance d'un élément *a* à un élément *b*. Sa définition est la suivante : il n'existe pas d'élément *p* tel que *a* appartienne à *p* et *p* est différent de *b*.

```
relationship has-part(b,a)
     assume ~ exists p : has-part(p,a) ^ p <> b
```

Dans l'exemple suivant, il est spécifié qu'un composant (*component*) doit être composé de zéro ou un élément *top*, de zéro ou un élément *bottom*, et rien d'autre.

```
kind component
     has-part ! {    0..1 element top;
                     0..1 element bottom};
```

Un style est instancié de la manière suivante. On spécifie le kind puis le nom de l'instance. La définition de l'instance doit respecter les relations définies au niveau du kind. Nous pouvons noter que *element* est le kind de base pour la définition des éléments. Ainsi, la définition suivante permet d'instancier le kind *component*, le nom de l'instance étant *mycomponent*.

```
component mycomponent
```

AML permet à une instance de suivre plusieurs kinds (*component1* et *component2*).

```
component1 mycomponent
component2 mycomponent
```

On peut restreindre à un seul style avec le ***!***.

```
component1! mycomponent
```

AML permet l'héritage de style. *kind* est considéré comme le style de base ; ainsi il représente tout le domaine architectural. Pour définir un nouveau style, on doit spécifier son nom à la suite de celui de son style parent. Le sous-style hérite des éléments et des relations de son parent. Dans sa description, on peut introduire de nouveaux kinds représentant les briques de construction utilisables, d'autres relations, et étendre les relations du style parent. Dans le premier cas, ci-dessous, *component* hérite de *kind*, dans le deuxième cas *pipe* hérite de *component*.

```
kind component …
component pipe …
```

2.7. DARWIN

DARWIN [Mag et al. 95] est un langage de configuration défini pour supporter l'analyse des transmissions de messages dans les systèmes parallèles et distribués. Ainsi Darwin permet de modéliser des architectures de systèmes hautement distribués, dont le dynamisme est guidé par des soutiens strictement formels.
Il ne présente pas de mécanisme propre à la formalisation des styles.

Concepts sur les architectures

Les principales abstractions gérées par Darwin sont les composants. Darwin les décrit en termes de services qu'ils fournissent et qu'ils requièrent pour leur permettre de communiquer avec d'autres composants. Deux types de composants existent : les *primitifs* et les *composites*. Un composant primitif est une brique de base, tandis que le composant composite est constitué d'autres composants qui sont connectés entre eux explicitement par des liens (*bind*) qui relient leurs ports.

La figure suivante est le diagramme d'un système Client-Serveur simple.

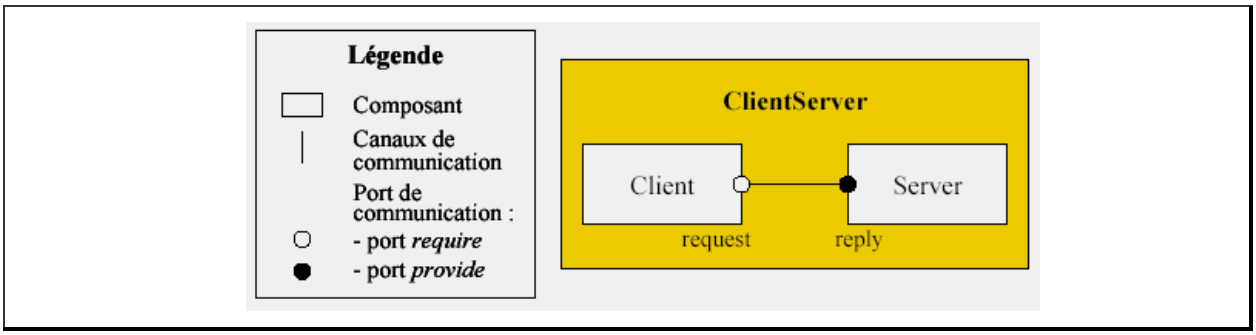


Figure 12 - Diagramme représentant un système Client-Serveur en Darwin

La spécification en Darwin de ce système est le suivant :

```
component ClientServer{
      inst client : Client ;
      inst server : Server ;
      bind client.request -- server.reply ;
}
```

Le composite englobant le système du client-serveur est un composant ne possédant pas d'interface. L'opérateur *inst* permet de déclarer les instances *client* et *server* des composants *Client* et *Server*. L'opérateur *bind* relie, à l'aide du constructeur --, un port requis en partie gauche avec un port fourni en partie droite.

Concepts sur la dynamique

Darwin permet de spécifier des composants dynamiques, c'est-à-dire de créer des composants en cours d'exécution du système. Le système n'est alors plus considéré comme étant fixe par rapport à sa phase de conception. Il fournit deux principaux mécanismes pour la description de structures dynamiques : l'*instanciation paresseuse* et l'*instanciation dynamique*.

L'instanciation paresseuse permet à un composant fournissant un service d'être instancié au moment où un autre composant souhaite accéder à ce service.

```
component ClientServer{
      inst client : Client ;
      inst server : dyn Server ;
      bind client.request --server.reply
}
```

L'instance *client* est créée à l'instanciation du composant composite *ClientServer*. L'instance *server*, quant à elle, est déclarée dynamique et ne sera créée que lorsque le *client* la sollicitera. L'instanciation paresseuse permet donc de décrire la configuration potentielle à l'exécution en déclarant des instances potentiellement nécessaires. Ce mécanisme ne permet d'instancier qu'un seul composant par clause d'interconnexion, contrairement à l'instanciation dynamique.

L'instanciation dynamique permet à un composant d'instancier un ou plusieurs autres composants.

```
component ClientServer{
     inst server : Server ;
     bind server.creation -- dyn Client
}
```

Dans cet exemple, l'instance *server* est créée à l'instanciation du composant composite *ClientServer*. Le client n'est pas déclaré, mais un des services du server est relié à la création dynamique d'un Client(*server.creation--dyn Client*). Ainsi, le composant server peut déclencher la création d'un ou plusieurs clients. Ici, les instances créées ne sont pas référencées, ainsi les composants dynamiques ne sont pas accessibles par les autres composants. En d'autres termes, les composants qui sont créés statiquement ne peuvent pas déclencher une communication avec ceux qui sont instanciés dynamiquement, seul l'inverse est possible.

2.8. RAPIDE

RAPIDE [Luc et al. 95] [Luc&Ver 95][RDT 97] est un langage orienté objet basé sur des événements concurrents. Il est spécifiquement conçu pour prototyper des architectures dans des systèmes distribués. Il permet de décrire les interfaces des composants et les communications entre celles-ci ainsi que les modèles événementiels. Ainsi, il autorise la simulation et l'analyse du comportement des architectures d'un système.
Il ne présente pas de mécanisme propre à la formalisation des styles.

Concepts sur les architectures

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

La spécification en RAPIDE de l'exemple du client-serveur est donnée ci-dessous.

```
architecture ClientServer is
     c : Client ;
     s : Server ;
     service : Service ;
     connect
          c.request(service) to s.reply(service) ;
end ClientServer
```

L'architecture *ClientServer* contient la déclaration des instances *c* et *s* des composants *Client* et *Server*, ainsi que la donnée *service* transmise durant leur communication. Toutes les instances sont représentées par des variables. L'architecture contient également les règles d'interconnexion entre les composants. Une règle est composée d'une partie droite et d'une partie gauche. La partie gauche contient l'expression d'événements qui doit être vérifiée avant que les événements contenus dans la partie droite soient déclenchés. Dans l'exemple quand *c* génère un événement *request* alors *s* observe un événement *reply* avec la même donnée *service*.

Concepts sur la dynamique

RAPIDE est capable de modéliser l'architecture de systèmes dynamiques dans lesquels le nombre de composants peut varier quand le système est exécuté. Rappelons que toutes les instances sont représentées par des variables. Ainsi, pour transcrire la dynamique, RAPIDE introduit deux types de variables particulières : les *placeholder* et les *iterator* auxquelles sont associées des règles de création. Le nom des variables *placeholder* débute toujours par *?*, elles désignent un objet qui est

susceptible d'être présent. Le nom des variables *iterator* débute toujours par *!*, elles désignent une conjonction d'instances d'un certain type. Les règles de création définissent quand, pendant l'exécution, les composants d'un type doivent être créés ou détruits. Prenons l'exemple du client-serveur ci-dessous :

```
architecture ClientServer is
     ?c : Client ;
     !s : Server ;
     ?service : Service ;
     connect
          ?c.request(?service) => !s.reply(?service) ;
end ClientServer
```

Dans l'exemple *?c* : *Client* est une variable *placeholder* et fait référence à une instance de type *Client*. *!s :Server* est une variable *iterator* et désigne toutes les instances de *Server* présentes dans l'application. Ces déclarations de variables définissent un objet devant être présent dans l'architecture ou un ensemble d'objets (borné ou non) de tel ou tel type. Il n'est pas nécessaire de définir les instances de composants présents, car cela peut se faire dynamiquement au fur et à mesure de l'exécution.
La partie connexion contient les règles de connexion et les règles de création. Les règles de création définissent les conditions sur les événements qui guident la création de nouveaux composants. Dans l'exemple, la règle de création spécifie l'existence d'un client *?c* qui émet avec n'importe quelle donnée de type *Service*, alors que la partie droite de la règle est exécutée. Celle-ci spécifie que tous les serveurs *!s* du système reçoivent la même donnée *?service*. Ces règles peuvent être conditionnées par une clause *where*.
RAPIDE supporte uniquement des manipulations dynamiques contraintes ou tous les changements d'exécutions doivent être connus à priori. Il permet donc de créer des composants dynamiques et possède des mécanismes permettant leur gestion. Par contre, dans la littérature, aucun opérateur de création, suppression, migration de composants ou modification d'interconnexions n'est disponible.

2.9. C2SADEL

C2SADEL (Software Architecture Description and Evolution Language) [Med 96][MedTay&Whi 96][Med et al. 96][MedRos&Tay 99] est un ADL dont l'objectif est de décrire des architectures de systèmes hautement distribués, évolutifs, et dynamiques. Les systèmes qu'il souhaite spécifier sont complexes, multi-langages, multi-plates-formes et ont une longue exécution. Pour pouvoir être économiquement viables, ces systèmes, dont la maintenance est estimée à soixante pour cent du coût total de développement, doivent être évolutifs. C'est pourquoi C2SADEL propose un langage permettant le changement de topologie d'une architecture.

Concepts architecturaux

C2SADEL s'appuie sur les règles du style C2 [Tay et al. 96] pour la spécification des architectures. Les connecteurs sont modélisés explicitement ; les composants ne sont attachés qu'à deux connecteurs (un dessus fournissant l'entrée, un dessous récupérant la sortie) ; les connecteurs peuvent être attachés à de multiples composants et connecteurs.

L'*interface* des composants est définie par un *top_port* ou/et un *bottom_port*. Par la sortie d'un *top_port* le composant émet des *messages* et les réceptionne par son entrée. Ces messages sont typés. Or, pour que les interfaces puissent manipuler n'importe quels messages, ces derniers sont placés dans des *enveloppes* cachant leur type. Un composant possède un comportement qui lui est associé. Ce comportement possède un invariant (spécifiant des propriétés qui doivent être vraies durant l'ensemble des états du composant) et un ensemble d'opérations pouvant être fourni ou requis.
Les interfaces des connecteurs sont génériques. En effet les connecteurs sont indifférents des types de données qu'ils manipulent puisque celles-ci sont placées dans des enveloppes masquant leur type. Leur principale tâche est donc de coordonner la communication entre les composants. Une des caractéristiques de C2SADEL est qu'une interface d'un connecteur est déterminée par les

interfaces (potentiellement dynamiques) des composants qui communiquent à travers elle. C2SADEL base les types de connecteurs sur les protocoles d'interactions.
La connexion des composants, via les connecteurs, est explicitée dans la configuration de l'*architecture*. Ces attachements sont explicites.

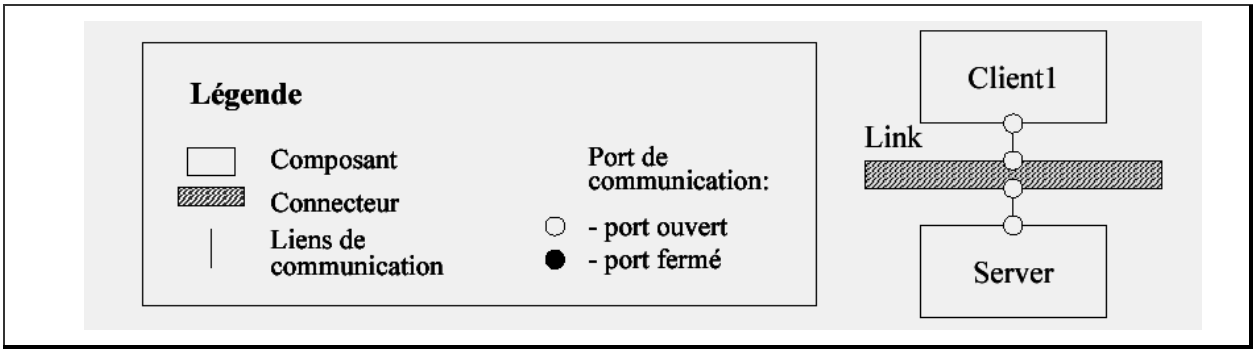


Figure 13 - Diagramme d'architecture client-server dans le style C2

La spécification en C2SADEL du système représenté par le diagramme précédent est la suivante :

```
architecture ClientServer is {
components {
     Client ;
     Server ;
}
connectors {
     Link ;
}
topology {
     connector Link connections {
          top_port {
               Client filter no_filtering;
          }
          bottom_port {
               Server filter no_filtering;
}    }    }    }
```

L'architecture possède deux composants *Client* et *Server*, et un connecteur *Link*. La topologie de cette architecture est définie par la connexion au connecteur *Link* des deux composants. Le composant *Client* est connecté au connecteur par son *top_port* et le *server* par son *bottom_port*.

Concepts sur la dynamique

C2SADEL supporte les manipulations dynamiques. Pour cela le langage spécifie un ensemble d'opérations pour l'insertion, la suppression, et la re-connexion des éléments de l'architecture pendant l'exécution du système : *addComponent*, *removeComponent*, *weld* et *unweld*. C2SADEL exploite les connecteurs flexibles pouvant supporter le dynamisme.

Prenons l'exemple du client-serveur auquel nous souhaitons ajouter un client de manière dynamique.

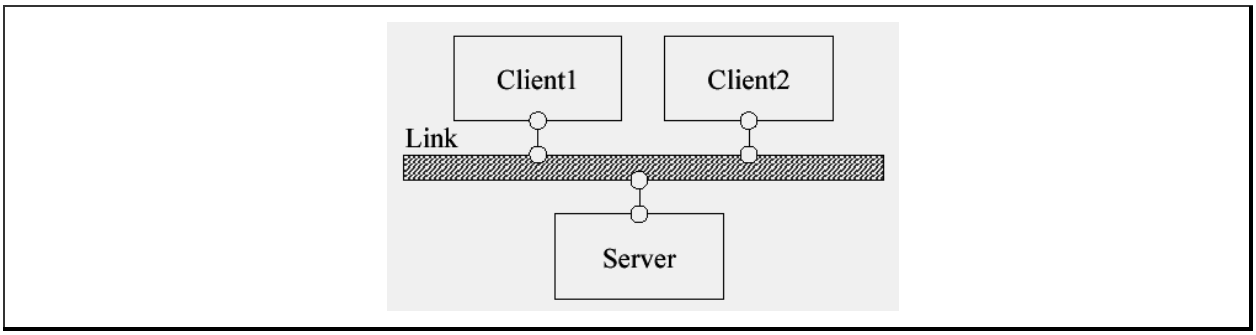


Figure 14 - Diagramme d'architecture client-serveur avec ajout d'un client

Pour effectuer cet ajout, nous utilisons l'opérateur *addComponent* de la manière suivante :

```
ClientServer.addComponent (Client2);
ClientServer.weld (Link, Client2);
```

Le composant *Client2* est ajouté à l'architecture *ClientServer* grâce à l'opérateur ***addComponent***. Puis il est connecté au connecteur *Link* grâce à l'opérateur ***weld***. Ces deux opérations sont effectuées après que le système ait été construit.

De même que C2SADEL permet l'ajout de composants, il propose leur suppression. C2SADEL permet aussi de détacher un composant de l'architecture comme spécifié ci-dessous. Dans l'exemple ci-dessous, le composant *Client1* est détaché de *Link* (***unweld***), puis il est supprimé de l'architecture grâce à l'opérateur ***removeComponent***.

```
ClientServer.unweld (Link, Client1);
ClientServer.removeComponent (Client1);
```

Par rapport aux aspects dynamiques définis dans l'introduction, C2SADEL permet de spécifier les attachements et les composants dynamiques. Par contre, ces changements doivent être planifiés avant la définition de l'architecture. De plus, dans la littérature, ni l'origine de la création ou de la suppression de ces composants, ni les moyens de gestion de ces composants, une fois créés, ne sont spécifiés.

2.10. UNICON

UNICON [Sha et al. 95] est un système conçu pour la compilation de description architecturale et l'implémentation de systèmes à partir de la description de leur architecture. Il supporte et génère du code pour une large variété de styles architecturaux (prédéfinis). Il possède un compilateur de haut niveau qui supporte l'utilisation de types de composants et de connecteurs hétérogènes. UNICON-2 [DeL96] est une extension pour la prise en charge de la définition de nouveaux styles architecturaux.

Concepts sur les architectures

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

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

La définition suivante est celle d'un composant appelé *Client*. Il définit un point de communication *request* qui transite des chaînes de caractères. Son implémentation est donnée dans la ressource "client.rc".

```
component Client
interface is
     type Process
     player request is RPCDef
          signature(“string”)
     end request
end interface
implementation is
     variant client in “client.c”
          impltype(Source)
     end client
end implementation
end Client
```

La définition suivante est celle d'une architecture appelée *ClientServer* est définie comme un composant. Elle est implémentée comme une configuration d'élément dont un élément de type *Client*. Pour chacun des éléments, on précise sur quel processeur celui-ci tournera.

```
component ClientServer
interface is
     type General
end interface
implementation is
     uses client interface Client
          processor (“proc1”)
          entrypoint (client)
     end client
     uses server interface Server
          processor (“proc2”)
          entrypoint (server)
     end server
     establish rpc with
          client.request AS appelant
          server.reply AS appelé
     end rpc
end implementation
end ClientServer
```

Concepts sur les styles

Dans UNICON, un style est représenté par un ***duty***. Un duty décrit l'information qui devrait être fournie pour un type donné de ***player***, d'***interface***, de ***role***, de ***protocol***, ou de configuration. Un duty inclut:

- une clause requires définissant une liste de patrons qui doivent être vérifiés,
- une clause optionnelle provides définissant une topologie minimale,
- une clause optionnelle closes définissant une liste de patrons qui ne doivent pas être vérifiés.

L'exemple suivant décrit le style *ClientServer* sous la forme d'un duty pour une configuration. Il spécifie qu'il faut au moins un composant *Client* et un composant *Server* dans une architecture client-serveur (l'opérateur + signifie qu'il faut au moins un élément). Il fournit aussi une propriété par défaut qui donne une information sur la taille des représentations visuelles des composants (sous forme d'icône).

```
implementation duty ClientServer
requires
     ( Client interface $ ) +
     ( Server interface $ ) +
provides
     gui-icon-size = (50, 50)
end ClientServer
```

Un style est instancié par un type. Dans l'exemple suivant *myClientServer* instancie *ClientServer*.

```
component myClientServer
ClientServer implementation
     uses client1 interface Client …
     uses client2 interface Client …
     uses server interface Server …
     …
```

L'héritage est aussi possible. Dans l'exemple suivant, le style *TwoClientsOneServer* hérite du style *ClientServer*. Pour ça, on utilise le mot-clé ***includes***.

```
implementation duty TwoClientsOneServer
includes ClientServer
…
```

Qu'il s'agisse d'un sous-style ou d'une instance de style, ceux-ci intègrent la partie ***provides*** du style et doivent vérifier les contraintes (sous forme de patrons) des parties ***requires*** et ***closes***.

3. Environnements pour les styles et les familles de produits

Etant donné que la conception d'architectures et de styles architecturaux est devenue une discipline importante de l'ingénierie logicielle, il a été nécessaire de développer des outils et des environnements permettant la description, l'analyse et l'exploitation d'architectures. La plupart des ADLs sont associés à des outils et des environnements de développement permettant de définir et d'exploiter des descriptions architecturales.

Dans cette section, nous donnons une vue d'ensemble succincte sur les outils architecturaux. Puis, nous présentons AESOP un environnement un peu spécifique, très lié aux styles architecturaux. Enfin, nous présentons l'environnement de conception du projet ArchWare avec lequel nos travaux sont liés.

3.1. Outils architecturaux

Une équipe de développement a répertorié les principaux outils afin de définir une boîte à outils complète ([ABL]) pour la prise en charge des ADLs. Les paragraphes suivants présentent une partie des informations que cette équipe a mis à disposition.
Les outils architecturaux sont principalement des :

- éditeurs et visualiseurs graphiques et textuels,
- outils d'analyse statique,
- outils d'analyse dynamique, utilisés pour l'exécution d'architectures (simulation et supervision),
- outils d'implémentation : interfaces ou générateurs de code,
- outils permettant l'évolution d'architectures,
- outils de génération de documentation,
- outils de « traduction » de descriptions architecturales entre différents ADLs.

Ainsi, RAPIDE propose des outils permettant :

- l'édition graphique et textuelle (outil RapArch),

- la compilation et l'exécution des descriptions architecturales,
- l'animation de l'exécution de l'architecture dans un environnement graphique temps réel (outil Raptor),
- la vérification de contraintes.

La boite à outils d'UNICON comporte un compilateur et un générateur de code pour l'infrastructure de communication des architectures écrites en UNICON.

Plusieurs outils et librairies ont été développés pour prendre en charge la conception architecturale en ACME, parmi ceux-ci :

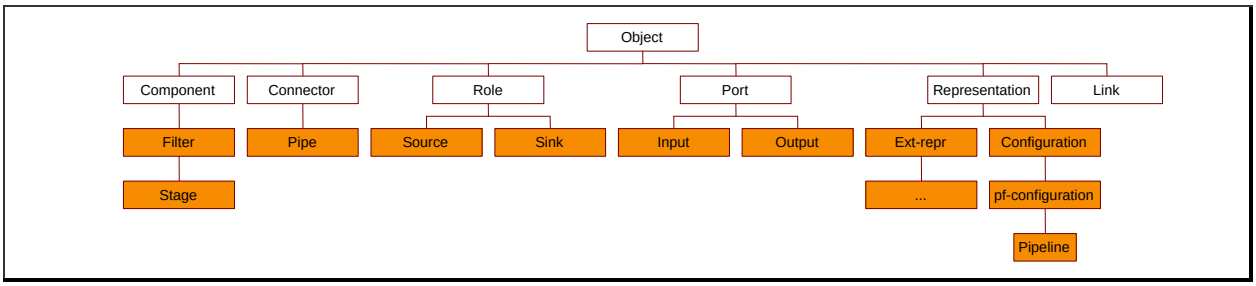
- une librairie (ACMELib, disponible en Java et en C++) de classes réutilisables pour représenter et manipuler les architectures ACME,
- un environnement de développement incluant un éditeur graphique et un vérificateur de règles, ACMEStudio. Cet environnement permet en outre d'éditer et de créer des styles définis notamment avec ARMANI,
- un outil de génération de documentation au format HTML,
- un outil permettant de manipuler, d'analyser et de générer des descriptions d'architectures avec PowerPoint. Cet éditeur permet lui aussi la spécification de styles et des propriétés associées,
- un outil d'analyse d'architectures, ACME PowerPoint Analysis Package, qui communique avec la description PowerPoint ce qui permet l'analyse dynamique de l'architecture créée,
- un outil d'analyse dynamique, ACME PowerPoint Dynamic Analysis Package, permettant la supervision d'architectures décrites en Dynamic ACME.

3.2. AESOP

La plupart des environnements de développement ont été créés pour construire des architectures à partir de styles spécifiques. Ces environnements fournissent des outils servant de support à des concepts particuliers de conception architecturale et aux méthodes de développement associées. Par exemple, des environnements ont été développés pour mettre en œuvre des architectures basées sur les flux de données [Mak 92], la conception orientée objet [Rum et al. 91], les systèmes de contrôle [Bin&Ves 93], ou encore l'intégration réactive [Fro 89]. Malheureusement, ces environnements spécifiques sont construits les uns indépendamment des autres. Une telle approche a un coût élevé, car chaque environnement de développement architectural ne supporte qu'un seul style.

Afin de palier ce problème, l'environnement AESOP [GarAll&Ock 94] a été conçu pour la génération, basé sur des styles d'environnements de développement architectural spécifique à un domaine. Les environnements qu'il génère permettent de modéliser visuellement des architectures dans un style composant-connecteur. Les concepts architecturaux adoptés par AESOP sont ceux qui ont servis de bases pour WRIGHT.

AESOP s'appuie sur un modèle objet pour décrire les styles. Concrètement, les styles sont spécifiés en utilisant java. Les styles sont représentés comme des sous-classes.



Les contraintes de constructions sont spécifiées à travers la définition de méthodes.

L'exemple suivant définit la méthode *attach* de la classe *pf_source* (méthode pour une source dans le style Pipe-Filter). Cette méthode est appelée lors de l'attachement d'une source à un port. Si le type du port n'est pas sous-type de *PF_WRITE_TYPE*.

```
fam_bool pf_source::attach(fam_port p){
 if (!fam_is_subtype(p.fam_type(),PF_WRITE_TYPE))
 {
   return false;
 }
 else
 {
    return fam_port::attach(p);
 }
```

Les instances d'un style sont directement créées dans l'environnement spécifique au style, comme le montre la figure ci-dessous. On peut y voir la modélisation graphique d'une architecture dans un environnement spécifique au style Pipe-Filter. Cet environnement propose une bibliothèque de composants, de connecteurs et de ports.

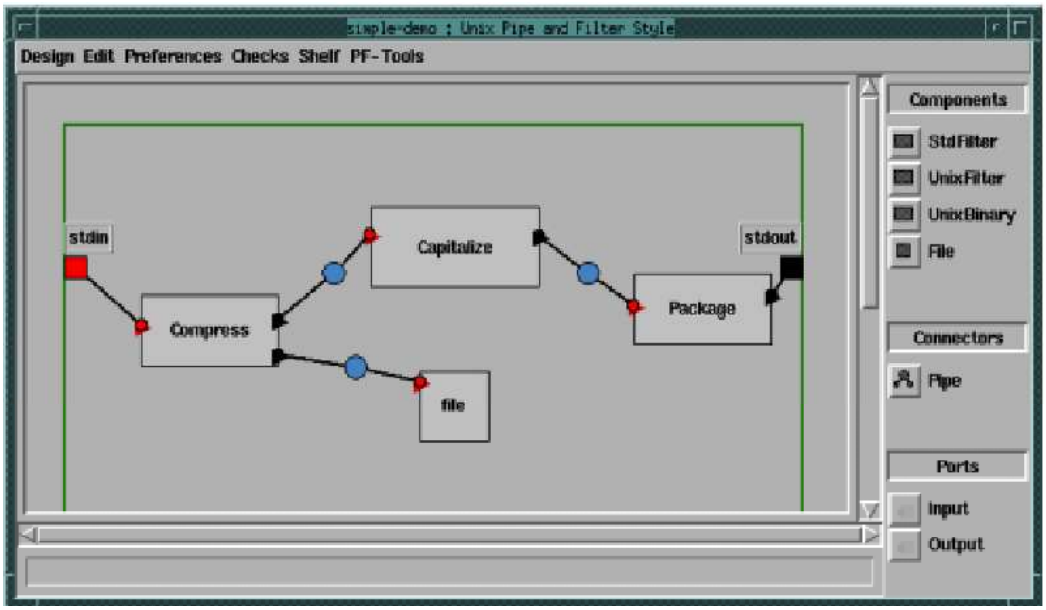


Figure 15 - Modélisation graphique d'une architecture

Un style peut hériter d'un autre en suivant les règles du modèle objet. Une sous-classe doit fournir des méthodes pouvant introduire des sources additionnelles d'erreurs. En effet, un sous-style ne peut que rajouter des contraintes ; ajouter une source d'erreur revient à ajouter une contrainte.

3.3. ArchWare Environment

Le projet ArchWare [Arc 02] fournit un environnement pour supporter le développement centré architecture dans le cadre des systèmes évolutifs. Les systèmes logiciels évolutifs sont ceux qui sont capables de changements à travers un cycle de vie prolongée avec un impact réduit pour les coûts, la planification et un impact contrôlé de la qualité.

L'environnement ArchWare supporte plusieurs formalismes:

- ArchWare Architecture Description Language (ADL) [Oqu et al. 02][Cim et al. 02]; langage pour la description des architectures,
- ArchWare Architecture Analysis Language (AAL) [All et al. 02]; langage pour l'analyse des architectures,

- ArchWare Architecture Refinement Language (ARL) [Oqu 03b]; langage pour le raffinement des architectures,
- ArchWare Architecture Exchange Language (AXL) [Ver&Oqu 03]; langage d'échange entre les outils d'exploitation des architectures.

L'environnement ArchWare fournit plusieurs outils génériques pour la conception architecturale. Tous ces outils sont intégrés dans un "framework" permettant d'exécuter des modèles de processus. Ces outils incluent :

- un compilateur et un moteur d'exécution d'architectures ;
- un éditeur graphique (basé sur UML) et un éditeur textuel permettant la définition et la manipulation d'architectures, le Visual Modeller [Occ&Zav 03],
- un animateur graphique, l'Animator [Azz et al.03],
- des outils de vérification de propriétés, le Theorem Prover [LeBAll&Oqu 03], le Model Checker [Ber et al. 04] et le Model Specific Evaluator [Ley et al.03],
- un raffineur [Meg&Oqu 04],
- un générateur de code, le Synthetisor [Bal et al. 02].

Nous donnerons plus d'informations relatives à l'intégration des outils ArchWare dans le chapitre 6.

4. Récapitulatif

Dans cette section, nous présentons un récapitulatif sur l'état de l'art fait dans les sections précédentes. Nous allons notamment présenter une classification et une comparaison des ADLs. La classification et la comparaison des ADLs est une tâche difficile tant ceux-ci peuvent être différents dans leurs buts et dans leurs concepts. Ceci est repris en partie du document [LeyCim&Oqu 02].

Dans une première section, nous présenterons les critères selon lesquels nous classifions les ADLs. Puis, nous présentons un récapitulatif de l'état de l'art, avec, notamment, des tableaux permettant de distinguer les capacités des ADLs les uns par rapport aux autres. Enfin, nous présentons des schémas permettant de les positionner visuellement les uns par rapport aux autres.

4.1. Présentation des critères

- Concepts architecturaux :
  - les éléments et leurs interfaces,
  - la configuration des éléments : les interconnexions et les encapsulations (composition) des éléments,
  - le comportement,
  - les attributs : les informations non-fonctionnelles,
  - la dynamique et en particulier la mobilité.

- Concepts sur les styles :
  - le vocabulaire : définition des éléments de construction,
  - les contraintes topologiques, comportementales et d'attributs,
  - l'instanciation d'un style,
  - l'héritage de style.

4.2. Concepts architecturaux

La plupart des ADLs permettent la description des connecteurs mais leur associent une sémantique trop forte, par exemple ARMANI ne permet pas un *rôle* d'être attaché à plusieurs *ports* ; deux connecteurs ne peuvent être reliés dans π-SPACE.

Ce paragraphe résume les différentes notions associées aux principaux ADLs dans le cadre de la définition d'architectures. L'objectif est de présenter ce qu'est une architecture, et de quoi elle est composée, pour chacun de ces ADLs. Il est à noter que pour la plupart des ADLs étudiés, l'architecture est considérée comme une *configuration* de *composants* (représentants des

fonctionnalités ou des données du systèmes) et de *connecteurs* (représentant les protocoles de communication et permettant la connexion entre les composants). Les tableaux indiquent par ailleurs si les ADLs permettent la description de comportements et d'attributs, et s'ils gèrent la dynamique et la mobilité des éléments architecturaux au sein de l'architecture.

Parmi les ADLs capables de modéliser les architectures, nous avons répertorié et étudié ACME [GarMon&Wil 00], ARMANI [Mon 01], WRIGHT [All 97], π-SPACE [Cha 02], σπ-SPACE [LeyCim&Oqu 01], ARCHWARE ADL [Cim at al. 02], AML [wil 99], DARWIN [Mag et al. 95], RAPIDE [RDT 97], CSADEL [MedRos&Tay 96], UNICON [Sha et al. 95].
Parmi ces langages, certains sont seulement concernés par des aspects structuraux comme C2SADEL, DARWIN ou ACME. Ainsi, l'évolution d'une architecture étant fortement liée au comportement de celle-ci, elle n'est traitée que partiellement. En conséquence on ne peut pas gérer des éléments dynamiquement créés. Dans le cas de RAPIDE, de nouveaux éléments architecturaux peuvent être créés comme on créerait de nouveaux objets dans un langage orienté objet. Une conséquence est qu'on ne peut en général pas savoir quelles topologies seront créées pendant l'exécution d'une architecture. D'autres ADLs sont fondés sur des algèbres de processus et supportent les aspects comportementaux: π-SPACE et ARCHWARE ADL sont fondés sur le π-calcul [Mil 89] et Dynamic WRIGHT est fondé sur CSP. Dynamic WRIGHT supporte seulement la reconfiguration dynamique (attachement et détachement). π-SPACE supporte en plus la création dynamique d'éléments et la composition et la décomposition dynamique des éléments. ArchWare ADL va plus loin que le précédent en permettant l'évolution à la volée et la mobilité des éléments.

ADLs	ACME / ARMANI	C2SADEL	AML	DARWIN	ArchWare ADL
Architecture	*system*	*architecture*	*element*	*component*	*abstraction*
Elément de base (englobant la fonctionnalité)	*component*	*component*	*element*	*component*	*abstraction*
- interface	*port*	*port*	Un type (*kind*) de port peut être défini	*port*	*connection*
Elément de connexion	*connector*	*connector*	Un type (*kind*) de connecteur peut être spécifié		*connection*
- interface	*role*	*port*	Un type (*kind*) peut être défini		*connection*
Attachement	*attachment*	*connections*	relation *attached-to*	*bind*	*connection*
Comportement			Quantificateurs temporels : *always, sometimes*		*behaviour* basé sur le π- calcul typé d'ordre supérieur
Attributs	Annotation		Prédicats logiques		*location* (valeur de stockage)
Implémentation	Annotation	*representation*			*wrappers*
Composition	*system / representation*	*topology*	Crée une relation « *has-part* »		*compose* (opérateur de composition)
Dynamicité	Dynamic ACME utilise *open* pour déclarer les éléments pouvant être instanciés dynamiquement	Méthodes de création, de suppression, d'attachement et de détachement.	Contraintes sur la dynamique exprimées sous forme de prédicats.	Instanciation paresseuse et instanciation dynamique - *dyn* Pas de mécanisme de gestion	Mécanisme d'application d'abstraction qui permet d'instancier à l'exécution.
Mobilité					Les connexions, les abstractions, et les comportements peuvent transiter via les connexions.

ADLs	π-SPACE/σπ-SPACE	RAPIDE	UNICON	WRIGHT
Architecture	*composite*	*architecture*	*component*	*system*
Elément de base (englobant la fonctionnalité)	*component*	*component*	*component*	*component*
- interface	*port*	*interface*	*player*	*port*
Elément de connexion	*connector*	*connections*	*connector*	*connector*

- interface	*port*		*role*	*role*
Attachement	*attach…to…*	*connections*	*connect*	*attachment*
Comportement	*behavior* avec π-calcul	behavioural constraints		*computation (comp), glue(conn)* avec CSP
Attributs			Annotation	
Implémentation	*operation*		*primitive implementation*	
Composition	*composite*	*architecture*	*composite implementation*	*configuration*
Dynamicité	Tous les éléments sont instanciables dynamiquement	RAPIDE permet la définition de règles décrivant des connections dynamiques		Dynamic WRIGHT, permet de simuler la dynamicité part l'ajout d'évènements *control* dans les composants
Mobilité				

Tableau 1 - Caractéristiques des ADLs concernant la définition d'architectures

4.3. Concepts sur les styles

Selon [Gar 01], un style architectural définit typiquement un vocabulaire pour des types d'éléments de conception et des règles pour la composition des instances de types. Ils définissent un ensemble de règles de configuration, ou des contraintes topologiques, qui déterminent les compositions autorisées de ces éléments, et donnent l'interprétation sémantique de ces compositions. Un modèle définit également les analyses qui peuvent être exécutées sur des systèmes établis selon le style [Gar 95]. Parmi les ADLs capables de modéliser les styles architecturaux, nous avons répertorié et étudié ACME [GarMon&Wil 00], ARMANI [Mon 01], WRIGHT [All 97], σπ-SPACE [LeyCim&Oqu 01], AML [wil 99], UNICON-2 [DeL 96]. Nous incluons aussi AESOP [GarAll&Ock 94] dans cette étude bien qu'il ne s'agisse pas d'un langage.
Au travers des ADLs précédemment cités, les styles sont formalisés de façons différentes, comme *types génériques* (AML, σπ-SPACE, UNICON, ARMANI), comme *ensemble de classes* (AESOP) dans une approche orientée objet, comme *constructeurs* (ACME), ou encore comme *ensemble de types* (ACME, ARMANI). Tous ces langages permettent la définition d'un vocabulaire de style, c'est à dire la définition de briques architecturales. Elles sont habituellement définies comme des types de composant et de connecteur, ou comme des styles. Certains de ces ADLs permettent la définition d'analyses (ARMANI). Cependant, l'intérêt principal concerne la formalisation de contraintes de style. La plupart de ces ADLs supportent principalement l'expression des contraintes structurales. Certains, comme ARMANI et UNICON, permettent l'expression des contraintes sur des "propriétés" (semblables aux attributs de classe elles spécifient des informations non-fonctionnelles, comme la latence et le temps de processus). Aucun ne supporte l'expression de contraintes comportementales au niveau des styles.
Les contraintes sont exprimées de différentes manières, par des méthodes de classe comme avec AESOP, par des cardinalités avec AML. Mais la manière la plus appropriée semble être la logique des prédicats employée notamment par WRIGHT et ARMANI, et la logique du μ-calcul employée dans ARCHWARE AAL (langage d'expression de propriétés associé à ARCHWARE ADL). Alors que la logique des prédicats ne permet à ces langages de ne définir que des contraintes sur des aspects statiques, les mécanismes propres au μ-calcul permettent de définir des contraintes sur les états qu'un système peut avoir à l'exécution et donc sur son comportement.

Les tableaux 3 et 4 présentent les mécanismes de définition des styles propres à chacun des ADLs, ainsi que les contraintes pouvant être exprimées. Ils indiquent par ailleurs si ces langages gèrent la dynamique et la mobilité des éléments et s'ils proposent des outils d'exploitation.

ADLs	AESOP	ACME	ARMANI

Style	*style* = classe	*family* = liste de *type* + structure requise	*style* = liste de *type* + structure requise + contraintes + analyses
Vocabulaire	Ensemble de classes	Ensemble de *types*	Ensemble de *types*
Contraintes	Exprimées par des méthodes de classe	Inclure structure requise à l'instanciation	*invariants* et *heuristics* 10(prédicats)
- Topologiques	Supportées par des méthodes de classe	template (constructeurs)	*invariants* et *heuristics*
- Comportementales			
- D'attributs			*invariants* et *heuristics*
Instanciation	Architecture crée avec l'environnement	Par des systèmes (system)	Par des systèmes (*system*)
Héritage	Seulement ajouter des causes d'erreur	Ajout de nouveaux types et extension de la structure requise – héritage multiple possible	Ajout de relations respectant les relations existantes
Dynamicité		Dynamic ACME	
Mobilité			
Outils d'exploitation	Génération d'environnements de développement, visualisation	Librairies, environnement de développement, analyses, …	Vérificateur de contraintes, environnement de développement

ADLs	**UNICON-2**	**AML**	**σπ-SPACE**
Style	*duty* de *player, role, interface, protocole,* ou *configuration*	*kind* = élément générique	*style* de *port, component, connector, composite*
Vocabulaire		Ensemble de *kinds*	Ensemble de *styles*
Contraintes	3 clauses : *requires, provides* et *closes*	Exprimées par des relations (*relationship*)	Mécanismes propres à chaque type de contrainte
- Topologiques	3 clauses : *requires, provides* et *closes*	- Cardinalités dans *relationship* (« *has-part* » et « *attached-to* »)	- cardinalités - attachements possibles
- Comportementales		Quantificateurs temporels	Spécification générique du comportement
- D'attributs	3 clauses : *requires, provides* et *closes*		
Instanciation	Par un *type*	Par un element	Par un *type*
Héritage	Clause *includes*		Ajout de contraintes
Dynamicité			Tous les éléments sont instanciables dynamiquement
Mobilité			
Outils d'exploitation	Compilateur , Génération automatique de code pour l'implémentation		

Tableau 2 - Caractéristiques des ADLs concernant la définition de styles

4.4. Comparaison des ADLs

Dans cette section, nous comparons les ADLs en les disposant les uns par rapports aux autres dans des espaces à 3 dimensions où chaque axe représente une caractéristique.

4.4.1 Comparaison sur divers points d'intérêt.

Dans cette partie nous comparons les ADLs sur des points qui sont essentiels (Figure 16). Nous ne tenons pas compte des aspects concernant les styles. Il s'agit de savoir si l'ADL gère bien la dynamique, s'il est accompagné d'outils d'exploitation et s'il permet la génération automatique de code exécutable.

[10] Exprimés par des prédicats, les *invariants* sont des contraintes strictes et les *heuristics* sont des propriétés qu'il est conseillé de suivre.

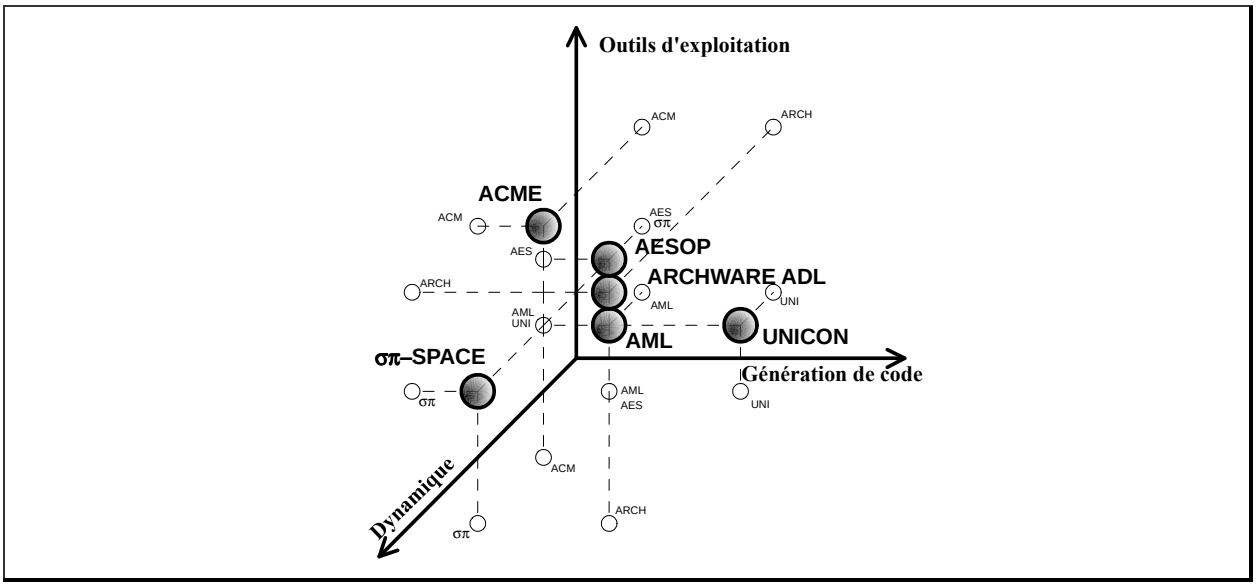


Figure 16 - Positionnement des ADLs pour l'exploitation des styles

4.4.2 Formalisation des contraintes

Comme cela a été expliqué, l'utilisation de styles architecturaux est l'aspect essentiel du développement architectural permettant la construction de familles d'applications logicielles.

Le style architectural devant être développé doit contraindre la construction des architectures. Les contraintes à formaliser sont de plusieurs types :

- contraintes concernant la structure,
- contraintes à appliquer aux valeurs des attributs,
- contraintes concernant le comportement.

L'objet de cette section est de positionner les ADLs en fonction de leur capacité à formaliser ces trois types de contraintes

Concernant les contraintes structurelles, tous les ADLs sont capables de les formaliser, mais ARCHWARE ADL (par l'ARCHWARE AAL), ARMANI et WRIGHT offre un support puissant par l'utilisation d'une logique des prédicats pour l'expression des contraintes. Ce même support est utilisé pour la description des contraintes d'attributs. Concernant le comportement, seuls σπ-SPACE, AML et ARCHWARE ADL sont adaptés. σπ-SPACE et ARCHWARE ADL offrent un meilleur support grâce à l'utilisation d'une algèbre de processus, le π-calcul et la logique du μ-calcul. La Figure 17 permet de situer les ADLs les uns par rapport aux autres dans un espace tridimensionnel où les axes sont associés aux trois types de contraintes.

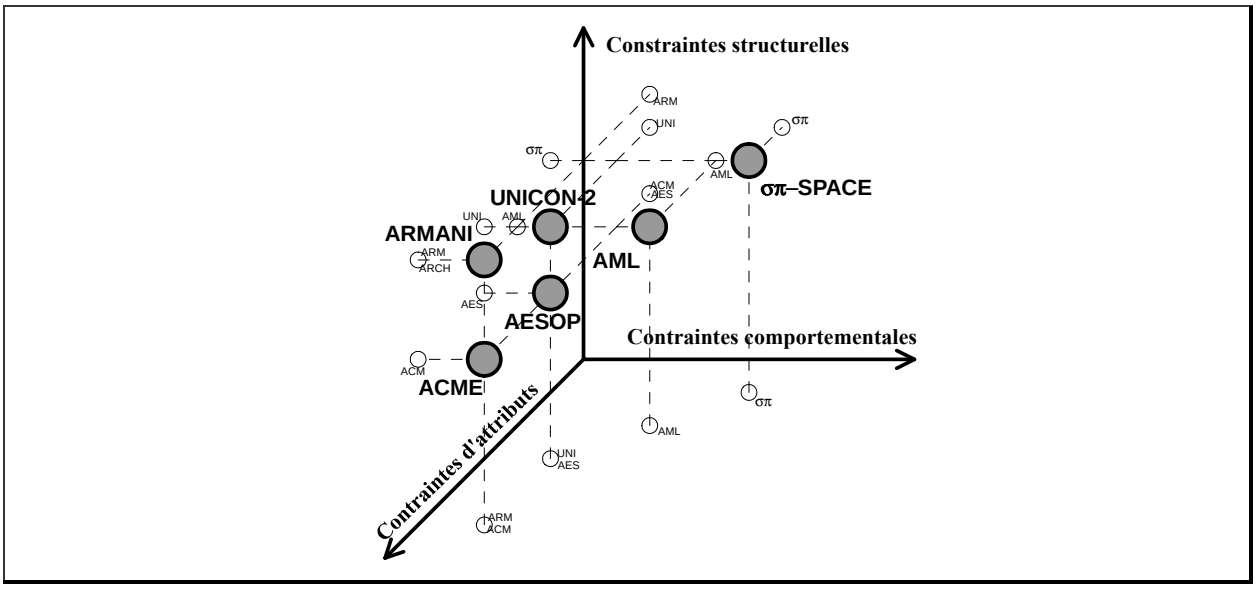


Figure 17 - Positionnement des ADLs pour la gestion des contraintes

4.4.3 Exploitation des styles

On peut exploiter de plusieurs façons un style formalisé. La manière de formaliser les styles est plus ou moins adaptée à chaque type d'utilisation. Un style peut être utilisé pour ***générer un environnement*** spécifique, comme c'est le but d'AESOP. Un style peut permettre de ***générer un langage*** ; AML semble être le plus adapté pour cela. Un style peut servir de ***bibliothèque***, contenant des éléments plus ou moins génériques et réutilisables. Enfin, un style peut être utilisé comme un ***patron structurel*** associé à un attribut qualité, comme le laisse supposer la méthode informelle sur les « Attributes-Based Architectural Styles » (ABAS) [Klein et Kazman 1999]. Dans ce cas l'architecte construit son architecture en choisissant les styles suivant les qualités requises par les différentes parties du système à modéliser. Ce dernier point n'est pas pris en compte par les ADLs étudiés.

La Figure 18 positionne les différents ADLs en fonction des utilisations des styles qu'ils permettent : la réutilisation (bibliothèques d'éléments et de patrons), la génération d'environnement et la génération d'ADL.

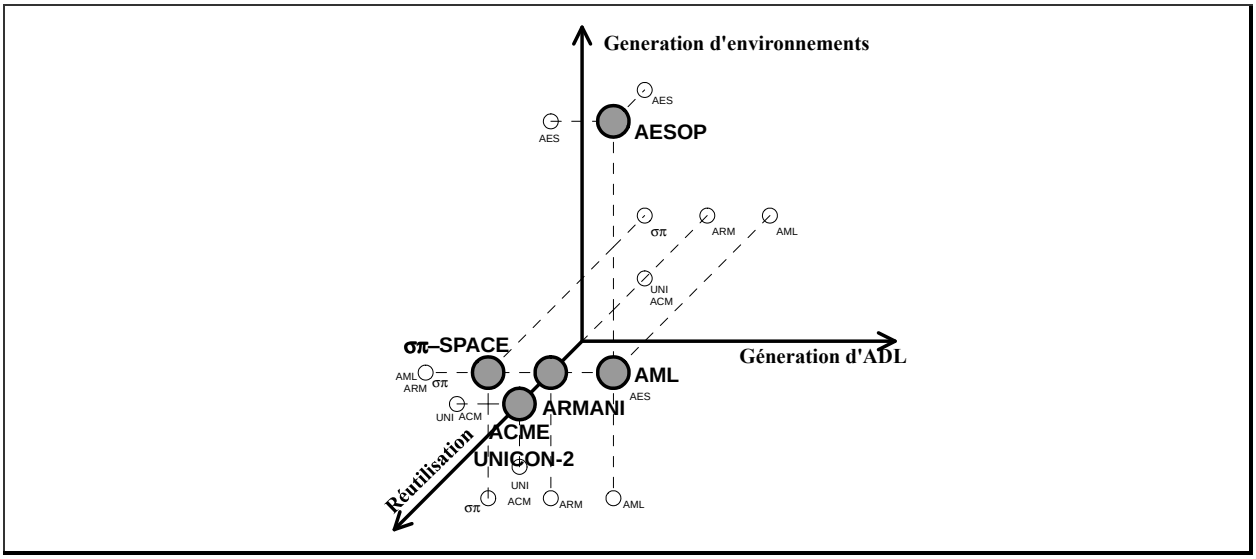


Figure 18 - Positionnement des ADLs pour l'exploitation des styles

5. Conclusion

Dans le domaine industriel, il existe une classe de systèmes logiciels qui ont des propriétés dynamiques. Ces systèmes ont la capacité de changer au cours du temps afin de répondre à différentes problématiques. Par exemple, ils peuvent permettre de :

- pouvoir répondre rapidement à de nouveaux besoins,
- pouvoir continuer un traitement critique tout en évoluant,
- alléger les ressources matérielles.

Parmi ces systèmes, on trouve les systèmes à longue durée d'exécution, les systèmes basés sur les NTIC (Nouvelles Technologies de l'Information et de Communication), ou encore les systèmes à mission critique qui ne peuvent être arrêtés en cours d'exécution pour être modifiés.

La dynamique des systèmes est définie lors du développement. Elle est essentielle pour certains systèmes et l'étudier dès la phase de conception permettrait de bénéficier des avantages liés à l'utilisation des architectures et des styles architecturaux. L'utilité de modéliser la dynamique est reconnue comme l'explique [Med 96]. Les architectures dynamiques permettent de planifier les changements structuraux d'un système qui auront lieu lors de l'exécution. Il est important de pouvoir dès la conception analyser les changements possibles d'un système afin d'éviter des configurations non souhaitables. De plus, l'utilisation des styles architecturaux permettrait de capturer l'expertise concernant la dynamique.

Ainsi, des langages de description d'architectures et de styles architecturaux supportant les aspects dynamiques seraient souhaitables pour le développement des systèmes dynamiques.

Nous pensons que des travaux ayant pour objectifs ceux présentés n'ont pas été entrepris ou ne sont pas complets. En effet, après un état de l'art essentiellement centré sur l'étude des ADLs nous avons relevé que les langages existants permettent seulement une définition des styles architecturaux tenant compte des aspects statiques. Ainsi, il y a un manque de langage formel pour la définition et l'utilisation des styles architecturaux dans le cadre du développement des systèmes dynamiques.

Pour résumer cet état de l'art focalisant sur la description des architectures dynamiques et des styles architecturaux, nous pouvons dire que les langages les plus aboutis dans ces deux directions sont WRIGHT, ACME/ARMANI et π-SPACE/σπ-SPACE. Mais ces langages sont encore limités sur plusieurs points concernant la conception architecturale pour les systèmes dynamiques.

D'une part, ils sont limités pour la description des architectures dynamiques. WRIGHT (basé sur CSP) ne permet que de proposer qu'un ensemble fini de configurations possibles à l'exécution. ACME et ARMANI ne prennent pas en compte la définition des comportements architecturaux. Ainsi, il est impossible de spécifier comment une architecture évolue dans le temps. Concernant Dynamic ACME, les mécanismes qu'il propose permettent seulement de fixer des limitations aux changements, ce qui les rapprochent des mécanismes de styles. Concernant, π-SPACE/σπ-SPACE, la seule restriction réside dans l'incapacité de définir la mobilité.

D'autre part, ils sont limités pour la description des styles. WRIGHT et ACME/ARMANI ne permettent d'exprimer que des contraintes statiques, portant soit sur la topologie soit sur les attributs. Ils ne permettent pas de spécifier des contraintes sur le comportement et sur la dynamique. Quant à la formalisation des styles avec σπ-SPACE, elle est très rigide, fournissant des mécanismes ad hoc pour la spécification des contraintes.

Cependant, nous avons relevé qu'ARCHWARE ADL est relativement abouti en ce qui concerne l'expression des comportements et de la dynamique. De plus, il est associé à d'autres langages, dont ARCHWARE AAL qui permet l'expression de propriétés architecturales aussi bien structurelles que comportementales. Nous constatons qu'ils fournissent une bonne base et qu'ils permettraient de franchir le pas vers la formalisation des styles pour les systèmes dynamiques.

Ainsi, dans le chapitre suivant nous proposons un langage pour la description des styles pour les systèmes dynamiques.

Chapitre 4

DE LA DEFINITION A L'UTILISATION DES STYLES ARCHITECTURAUX

Chapitre 4 - De la définition à l'utilisation des styles architecturaux

L'objet de ce chapitre est ASL, un langage de description architecturale centré sur les styles architecturaux. Nous présentons ce langage et expliquons ses mécanismes. Nous montrons que dans le cadre des systèmes dynamiques, il permet la formalisation des styles architecturaux et, est associé à des outils pour le développement logiciel centré style.

Dans les chapitres précédents, nous avons souligné l'importance de formaliser les styles architecturaux. La formalisation des styles architecturaux répond à deux nécessités dans un processus de développement de logiciels centré sur les styles. La première nécessité est de garantir une manière uniforme de décrire les styles. Tous les acteurs peuvent interpréter une description de la même manière si des notations sont clairement définies et sont les mêmes pour toutes les descriptions. La deuxième nécessité est d'aller plus loin dans l'exploitation des styles à l'aide de logiciels.

Parmi les Langages de Description d'Architecture - ADL[11] - étudiés, quelques-uns adressent la formalisation des styles. Toutefois, nous avons observé qu'il réside un manque de formalisation concernant les styles architecturaux dédiés aux systèmes dynamiques.

Dans ce chapitre, nous proposons un langage centré sur les styles architecturaux. Il permet à la fois la description d'architectures selon un style et la description des styles. Dans la mise en œuvre de ce langage nous focalisons notamment sur plusieurs objectifs :

- permettre la formalisation de styles pour les systèmes dynamiques,
- définir un support à la conception architecturale pour un domaine spécifique,
- promouvoir la réutilisation et la compréhension.

Ce chapitre est structuré en parties dédiées respectivement :

- à l'approche adoptée pour définir le langage ASL,
- à la définition des styles en ASL,
- à l'implémentation du langage ASL et de son outil associé, et
- à l'utilisation du langage.

1. Approche pour la mise en œuvre du langage

Nos travaux s'inscrivent dans l'apport d'outils pour intégrer l'usage des styles dans un processus de développement centré sur les architectures (cf. Figure 19). Les styles offrent un support à la conception architecturale et promeuvent à la fois la réutilisation et la compréhension. Nos travaux s'intègrent dans le projet ArchWare qui offre un large éventail d'outils propres au développement de logiciels centrés architecture [Arc 02].

[11] Nous utilisons l'acronyme anglo-saxon compte tenu de sa plus large utilisation dans la communauté.

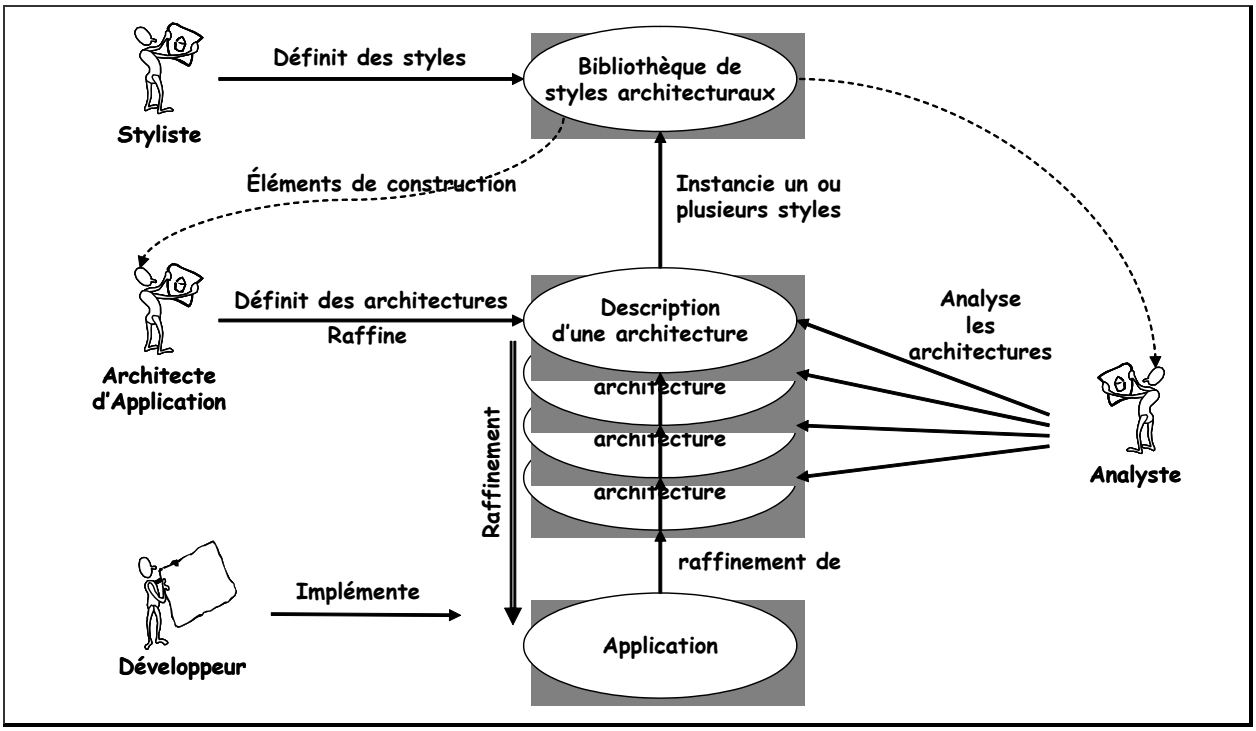


Figure 19 - Processus de développement centré style

Notre objectif est de développer un langage pour la formalisation des styles architecturaux et des architectures. Un outil est développé parallèlement à ce langage, pour fournir aux architectes un support à l'utilisation des styles. Il permet d'*instancier* des architectures à partir d'un style, de vérifier la *satisfaction d'une architecture à un style* et d'utiliser les *analyses* qu'un style fournit.

> L'***instanciation*** est un mécanisme qui consiste en la génération automatique d'une architecture à partir de la définition d'un style.

> La ***satisfaction*** est la relation existante entre une architecture et un style, lorsque celle-ci respecte les contraintes de ce dernier.

> Les ***analyses*** sont des traitements sur les architectures qui permettent d'en déterminer certaines caractéristiques. Un style fournit des analyses spécifiques aux architectures qui le satisfont.

Afin de décrire un style, nous devons pouvoir décrire des éléments architecturaux, des contraintes et des analyses. Ainsi, nous basons nos travaux sur un ensemble d'outils formels nous permettant de répondre aux besoins suivants :

- description des aspects structurels et comportementaux d'une architecture,
- description des contraintes sur des propriétés architecturales,
- définition des analyses.

Nos travaux se basent sur des outils de description d'architectures et de propriétés architecturales. Ceux-ci sont fondés sur des outils formels tel que le π-calcul [Mil 99] et le μ-calcul [Koz 83] (cf. annexes 1 et 2).

Dans ce chapitre, nous allons exprimer à plusieurs reprises les syntaxes de différents langages. Par souci de lisibilité, nous exprimons celles-ci d'une manière informelle. La notation BNF (cf. annexe 3) ne sera utilisée que dans des cas bien précis.

Dans cette introduction de l'approche, nous allons présenter des éléments liés à la formalisation et introduire deux langages sur lesquels nous nous basons, et qui sont fournis dans le cadre du projet ArchWare.

1.1. Formalisation de styles architecturaux

Notre objectif est de décrire formellement des styles architecturaux et d'apporter un support formel à la conception d'architecture en regard d'un style.

Après une étude du domaine (cf. chapitre 2) sur les architectures logicielles et les styles architecturaux, nous proposons la définition suivante.

> Un ***style architectural*** est un ensemble de propriétés architecturales (structurelles, comportementales et non-fonctionnelles) définissant une famille d'architectures partageant des caractéristiques communes.

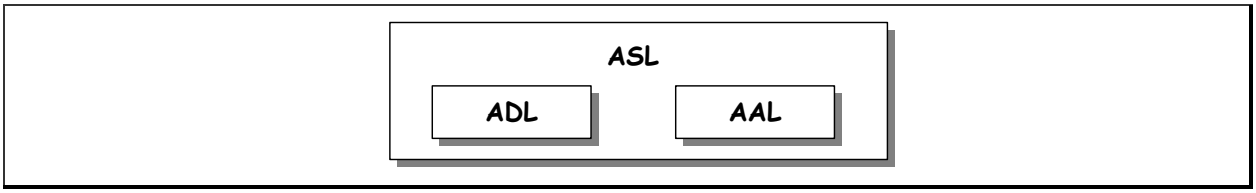
Une architecture qui suit un style offre la garantie de posséder les qualités intrinsèques au style. Ainsi, il est nécessaire de vérifier qu'elle satisfait les propriétés architecturales décrivant le style. Afin d'automatiser cette vérification il est nécessaire de fournir un formalisme permettant la description de propriétés architecturales. Ce formalisme doit être associé à un formalisme de description d'architecture partageant les mêmes concepts architecturaux.

De ce fait, nous basons nos travaux sur deux langages issus du projet ArchWare. Le premier est un langage de description d'architecture, ArchWare ADL. Le second est un langage de description de propriétés architecturales, AAL.

Bien des aspects autres que celui des propriétés architecturales gravitent autour de la notion de style. Nous avons vu cela dans l'étude du domaine. Un style est associé à un vocabulaire, à une bibliothèque d'éléments réutilisables et à des analyses. Tout cela constitue un support à la description architecturale dont nous voulons permettre la formalisation. Nous nous basons toujours sur l'ADL et l'AAL pour définir ce support.

Ainsi, dans notre approche, nous ciblons trois axes. Le premier concerne l'expression des contraintes. Ces contraintes permettent de délimiter un espace de conception qui caractérise la famille d'architectures, de vérifier la satisfaction d'une architecture à un style et de révéler les écarts au style. Le deuxième axe concerne un support à la description architecturale. Dans une optique de réutilisation et de compréhension, nous apportons des mécanismes pour réutiliser des concepts et pour fournir une syntaxe adaptée. Ainsi, nous permettons entre autre de définir des éléments architecturaux réutilisables dans la construction d'architectures propres à un style. Enfin, le troisième axe concerne un support à l'analyse architecturale. A travers un style, nous permettons de définir des analyses spécifiques aux architectures suivant le style.

Nous construisons ASL comme une couche sur les langages ADL et AAL. ASL permet de mettre en relation des éléments et des propriétés décrites dans ces deux langages. ASL permet d'encapsuler plusieurs descriptions au sein d'une seule afin de fournir tous les éléments relatifs à un style.



Nous verrons plus loin que l'ADL et l'AAL ont été étendus pour faciliter leur intégration dans ASL.

1.2. Les langages ArchWare

Le projet ArchWare fournit un ensemble de langages pour la conception architecturale :

- ArchWare Architecture Description Language (ADL) [Oqu et al. 02][Cim et al. 02] est un langage pour la description d'architectures dynamiques,
- ArchWare Architecture Analysis Language (AAL) [All et al. 02] est un langage pour la description de propriétés architecturales,
- ArchWare Architecture Refinement Language (ARL) [Oqu 03b] est un langage pour la description de raffinements d'architectures,
- ArchWare Architecture Exchange Language (AXL) [Ver&Oqu 03] est un langage d'échange pour la communication entre les outils de l'environnement ArchWare.

Dans notre approche, nous nous appuyons sur les résultats du projet ArchWare et, notamment, sur l'ADL et l'AAL. Dans cette section, nous étudions les différents concepts sous-jacents à ces deux langages. Nous explicitons aussi en quoi le choix de ces langages est judicieux.

1.2.1 Description du langage ArchWare ADL

ArchWare ADL [Oqu et al. 02][Cim et al. 02] est un langage formel conçu pour supporter la spécification exécutable d'architectures logicielles évolutives. Il est fondé formellement sur le π-calcul [Mil 99] (cf. annexe 1) typé d'ordre supérieur[12], un calcul pour les systèmes mobiles et communicants ; ArchWare ADL est défini comme une extension du π-calcul typé d'ordre supérieur pour un domaine spécifique : c'est une extension bien-formée pour définir un calcul d'éléments architecturaux mobiles et communicants.

Dans cette section nous présentons ArchWare ADL à travers ses concepts et ses mécanismes pour la description des architectures dynamiques.

Vue d'ensemble

> Une ***architecture*** est un ensemble d'éléments, appelés éléments architecturaux, qui sont reliés par des liens de communication.

En ArchWare ADL (cf. Figure 20), les éléments architecturaux sont définis en terme de comportements (***behaviour***). Un comportement est défini par un ensemble d'actions ordonnancées qui spécifie à la fois le traitement interne de l'élément architectural (actions internes) et les interactions avec son environnement (actions de communication).

Un élément architectural communique avec les autres par une interface représentée par un ensemble de ***connexions***. Les éléments architecturaux communiquent en transitant des données par ces connexions. Pour interagir, les éléments architecturaux sont mis en relations par un mécanisme de composition et un mécanisme de liaison, appelé ***unification*** (c'est une substitution au sens du π-calcul).
Lorsque plusieurs éléments sont composés, ils peuvent interagir lorsque leurs connexions sont liées.

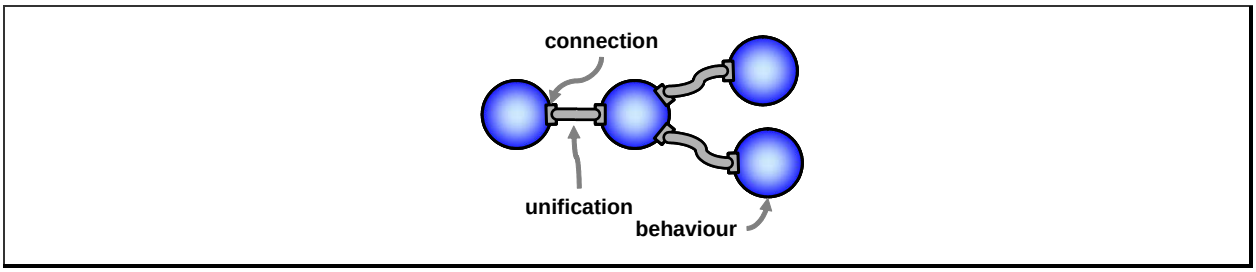


Figure 20 - Diagramme informel d'une architecture en ArchWare ADL

[12] Dans cette version du π-calcul les variables sont typées et des processus peuvent transiter via les canaux.

Un élément architectural peut être défini comme une composition d'autres éléments. Ainsi, un comportement peut être défini comme un ensemble d'autres comportements interconnectés. C'est le cas d'une architecture.

> ArchWare ADL permet la description ***d'architectures dynamiques***. D'abord, les éléments architecturaux peuvent être composés ou décomposés à la volée. Ensuite, des éléments architecturaux et des connexions peuvent être créés dynamiquement. Enfin, des éléments architecturaux et des connexions peuvent transiter comme des données à travers les connexions.

Dans les paragraphes suivants, nous donnons plus de détails sur la description des comportements et sur les données définissables et leur manipulation.

Comportements

Tout comportement est construit autour d'un comportement prédéfini, **done**, le comportement inactif, avec des actions et des opérateurs spécifiques. Ces actions et ces opérateurs sont similaires à ceux du π-calcul. D'ailleurs, un comportement est l'équivalent d'un processus en π-calcul.

Il existe trois types d'actions possibles dans un comportement. Elles sont :

- l'action d'envoi : **via** *connexion* **send** *données*
- l'action de réception : **via** *connexion* **receive** *données* **:** *type de données*
- l'action inobservable : **inobservable**

Notons que l'action inobservable représente une action qui est interne et qui ne participe pas à la communication : elle est inobservable depuis l'environnement du comportement.

Un comportement est défini en utilisant ces actions et les opérateurs suivants.

L'opérateur de succession : **;**

- *action* **;** *comportement* définit un comportement par préfixation d'une action à un comportement.

L'expression suivante est la définition du comportement b. Celui-ci est défini à partir du comportement ***done*** auquel est préfixée une action. Celle-ci est l'envoi du message "Hello". Autrement dit, à l'exécution de b, celui-ci communique (s'il a un correspondant) un message "Hello" à la suite duquel il se comporte comme ***done***, c'est-à-dire qu'il ne fait plus rien.

```
value b is behaviour { via a send "Hello" ; done }
```

Par raccourci d'écriture, ***done*** peut être omis lorsqu'il est préfixé. L'expression précédente est équivalente à:

```
value b is behaviour { via a send "Hello" }
```

L'opérateur de condition : **if…then…else…**

- **if(** *condition* **) then** $comportement_1$ **else** $comportement_2$ définit un comportement qui s'exécute comme $comportement_1$ si la condition est vérifiée et comme $comportement_2$ sinon.

L'expression suivante est la définition d'un comportement qui définit la réception de deux chaînes de caractères. Si celles-ci sont équivalentes, il renvoie la première, sinon il renvoie un message d'erreur.

```
behaviour {
     value x is connection(String);
     via x receive y : String;
```

```
      via x receive z : String;
      if y == z then via x send y
      else via x send "Error"
}
```

L'opérateur de choix : **choice**

- **choice{***comportement*$_1$ **or** … **or** *comportement*$_n$ **}** définit un comportement qui s'exécute comme l'un des premiers comportements capable de s'exécuter. Un comportement n'est pas capable de s'exécuter (bloqué) si sa prochaine action est une action de communication et qu'il n'a pas de correspondant immédiat.

L'expression suivante est un comportement qui peut se comporter de deux manières différentes : soit recevoir un message via une connexion *x*, soit par une connexion *z*.

```
behaviour {
      value x is free connection(String); value z is free connection(String);
      choose{
            {via x receive y : String; unobservable}
      or    {via z receive u : String; unobservable}
      }
}
```

L'opérateur de composition : **compose**

- **compose{** *comportement*$_1$ **and** … **and** *comportement*$_n$ **}** définit un comportement par composition de plusieurs comportements. Les comportements composés sont concurrents et peuvent communiquer lorsqu'ils partagent des connexions.

L'expression suivante est la composition de deux comportements, *p* et *q*. Ces deux comportements possèdent des connexions libres (free) qui sont unifiées de telle sorte que *p* puisse envoyer un message à *q*.

```
compose {
      p is behaviour {
            value y is free connection(String);
            value message is "Hello";
            via y send message }
      and
      q is behaviour {
            value z is free connection(String);
            via z receive message : String;
            unobservable}
            where {p::y unifies z}
}
```

L'opérateur de réplication : **replicate**

- **replicate{** *comportement* **}** est équivalent à une composition infinie du même comportement *comportement*.

L'expression suivante est un comportement qui peut être déclenché une infinité de fois par la réception d'une chaîne de caractères par la connexion *x*.

```
behaviour {
```

```
        replicate {
            value x is connection(String);
            via x receive y : String;
            unobservable
        }
}
```

Afin de réutiliser des définitions de comportement, ArchWare ADL fournit un mécanisme de réutilisation appelé ***abstraction***. L'abstraction permet d'encapsuler des définitions paramétrables de comportement. On obtient un comportement d'une abstraction par l'*application* de cette dernière, en fournissant éventuellement une liste de paramètres.

En plus de permettre la réutilisation, les abstractions permettent de définir des comportements récursifs, c'est-à-dire des comportements qui retournent à leur état initial après un certain nombre d'étapes.

L'exemple suivant définit l'abstraction d'une architecture Client-Serveur appelée *oneClientOneServer*. Cette architecture est composée d'un client et d'un serveur qui communiquent. Notez que leurs connexions sont unifiées afin que ces éléments puissent communiquer. Le client effectue un appel au serveur, puis il attend une réponse. Après avoir reçu une réponse du serveur, il effectue un traitement inobservable à la suite duquel il repart depuis son état initial. Pour rendre compte du processus répétitif, le client et le serveur sont définis par des abstractions récursives. Nous pouvons noter que chacune des abstractions contient, comme dernière action, une application d'elle-même, ce qui génère un comportement récursif.

```
recursive value client is abstraction ();{
     via call send; via wait receive; unobservable; client()};
recursive value server is abstraction ();{
     via request receive; unobservable; via reply send; server()};
value oneClientOneServer is abstraction ();{
     value call, wait, request, reply is connection();
     compose { c is client where {reply unifies wait}
     and       s is server where {call unifies request}
     }
}
```

Nous avons vu que les comportements communiquent par des envois de données, nous étudions les aspects concernant ces données dans la section suivante.

Les valeurs et les types

ArchWare ADL est un langage fortement typé : tout est valeur et tout est typé. Il fournit un système de types de données complet et des opérateurs pour manipuler les données. Nous présentons ici plus ou moins en détail ces types et ces opérateurs.

ArchWare ADL fournit des types qui sont classés en types de base, en types d'architectures, en types construits et en types collections.

Les *types de base* sont **Natural**, **Integer**, **Real**, **Boolean**, et **String**.

Les *types d'architectures* sont les suivants : **connection[***Type de donnée***]**, **Behaviour**, **abstraction[***Types des paramètres***]**. Ils représentent les valeurs architecturales présentées dans les sections précédentes.

Les *types construits* représentent des valeurs composites. Ces types sont les suivants : **tuple[***Liste de types***]**, **view[***Liste de types étiquetés***]**, **union[***Liste de types***]**, **variant[***Liste de types étiquetés***]**,

quote[*identifiant***]**, **Any**, et **location[***Type***]**. Parmi ces types, nous présentons le type **view** ci - dessous.

Les vues (views) sont des tuples dont les composantes sont étiquetées. Dans l'exemple suivant, une valeur *v* de type view est définie. Elle est composée d'une valeur ***true*** étiquetée *b,* et d'une valeur "ArchWare" étiquetée *s*. Le type de la valeur *v, MaVue*, est défini à la suite.

```
value v is view(b is true, s is "ArchWare")
```

```
type MaVue is view[b : Boolean, s : String]
```

Des mécanismes de projection permettent d'accéder aux composantes d'une valeur composée. La ligne de code suivante exprime que les composantes de *v* sont associées aux identifiants *b* et *s* qui sont utilisés par la suite (si *b* vaut ***true*** alors on envoie *s* par la connexion *c*).

```
project v as b, s; if b do via c send s
```

Le code suivant est une expression équivalente utilisant un raccourci d'écriture.

```
if v::b do via c send v::s
```

Les *types collections* représentent des valeurs composées d'un ensemble indéfini d'autres valeurs de même type. Ces types sont **bag[***Type***]**, **set[***Type***]** et **sequence[***Type***]**. Toutes les valeurs d'un ensemble sont du même type.

Un mécanisme d'itération permet d'effectuer un traitement pour chacun des éléments d'un ensemble. Dans l'exemple suivant, chaque valeur contenue dans *ensemble* est ajoutée à la valeur *somme* (initialisée à zéro). Ceci est réalisé par un mécanisme d'itération : pour tout élément *i* d'*ensemble*, la valeur *somme* (initialisée à 0) est incrémentée de la valeur de l'élément *i* ; le résultat final est stocké dans la valeur *resultat*.

```
value ensemble is set(1,2,3);
iterate ensemble by i : Integer;
from value somme is location(0)
accumulate {somme:='somme + i}
as resultat;
```

Des mécanismes de projection permettent d'accéder aux éléments des séquences. Dans l'exemple suivant, *resultat1* identifie le deuxième élément de la séquence *seq*, c'est-à-dire *4*. *resultat2* identifie quant à lui le premier élément de *seq*.

```
value seq is sequence(2,4,1);
project seq at 2 as resultat1;
value resultat2 is ensemble::1
```

D'autres mécanismes permettent d'ajouter ou de retirer des éléments d'un ensemble. Dans l'exemple suivant, la valeur 2 est ajoutée à ensemble. Ensuite une valeur 3 est retirée.

```
value ensemble is set(1,2,3);
ensemble includes 2;
ensemble excludes 3;
```

Architectures dynamiques

ArchWare ADL fournit des mécanismes qui promeuvent la définition d'architectures dynamiques. D'une part, l'abstraction est associée à un mécanisme permettant la création dynamique. En effet une abstraction est appliquée à la volée, ainsi plusieurs éléments peuvent être créés dynamiquement à partir d'une même définition. D'autre part, ArchWare ADL hérite des propriétés du

π-calcul d'ordre supérieur et promeut ainsi la dynamique par la mobilité des éléments. C'est à dire que des connexions ainsi que des comportements peuvent transiter par d'autres connexions.

La section suivante concerne ArchWare AAL, le langage de description de propriétés associées à ArchWare ADL.

1.2.2 Description du langage AAL

ArchWare AAL est un langage formel pour exprimer les propriétés des architectures logicielles dynamiques modélisées en ArchWare ADL. AAL est défini comme une extension du μ-calcul pour exprimer à la fois des propriétés structurelles et comportementales. Comme le μ-calcul est limité à l'expression de propriétés comportementales, AAL recouvre aussi la logique des prédicats pour l'expression de propriétés structurelles. Ainsi, AAL fournit un support à la vérification de propriétés architecturelles. Dans cette section, nous donnons un bref aperçu de l'AAL[13].

En ArchWare AAL les propriétés sont définies comme des formules prédicatives. Pour construire ces formules le langage fournit des prédicats prédéfinis et des mécanismes (opérateurs et quantificateurs) permettant de construire de nouveaux prédicats. Nous différencions les prédicats concernant les données et les prédicats concernant des propriétés comportementales. En plus des prédicats, AAL fournit un ensemble de fonctions.

Opérateurs et quantificateurs

ArchWare AAL fournit des opérateurs booléens ainsi que des quantificateurs. Ceux-ci permettent de construire de nouveaux prédicats à partir de prédicats définis. Il s'agit des opérateurs et des quantificateurs classiques de la logique des prédicats.

Le vocabulaire utilisé pour les opérateurs booléens est le suivant : ***not***, ***or***, ***and***, ***xor***, ***implies***, ***equivalent***.

Les quantificateurs sont le quantificateur existentiel, ***exists***, et le quantificateur universel ***forall***.

Un prédicat peut être appliqué à une collection via l'opérateur ***to*** *…* ***apply****.*

```
to collection apply prédicat
```

Ce dernier opérateur est généralement utilisé avec les quantificateurs ***exist*** et ***forall*** pour appliquer un prédicat à tous les éléments d'une collection. L'expression suivante est vraie si le prédicat *p* est vrai pour chacun des éléments de la collection.

```
to collection apply forall{element | p(element)}
```

Prédicats et fonctions sur les données

AAL fournit un éventail de prédicats et de fonctions prédéfinis concernant les données en général. Nous présentons ci-dessous un sous-ensemble des prédicats prédéfinis.

isValue(*name****,*** *type****)*** est vrai si la valeur *name* est de type *type.*
isType(*alias****,*** *type****)*** est vrai si l'identifiant *alias* réfère le type *type.*
isAbstraction(*name****)*** est vrai si la valeur *name* est une abstraction.
isInstance(*name****,*** *abstractionName****)*** est vrai si le comportement *name* est une instance de l'abstraction *abstractionName.*
isConnection(*name****,*** *type****)*** est vrai si ***isValue(****name****,*** ***connection[****type****])*** est vrai.

Concernant les fonctions prédéfinies, nous présentons également un sous-ensemble.

[13] Pour plus d'information sur le langage se reporter aux documents de références [All et al. 02].

parameters(*name****)*** – cette fonction s'applique à une abstraction et retourne l'ensemble de ses paramètres.
name(*data****)*** – cette fonction retourne le nom d'une donnée, elle existe pour tout type de donnée ainsi que pour les paramètres.
value(*name****)*** – cette fonction retourne la valeur d'une donnée.

Afin d'illustrer l'utilisation des prédicats sur les données, considérons l'exemple suivant : il s'agit d'une architecture pipe-filter. (cf. Figure 21)

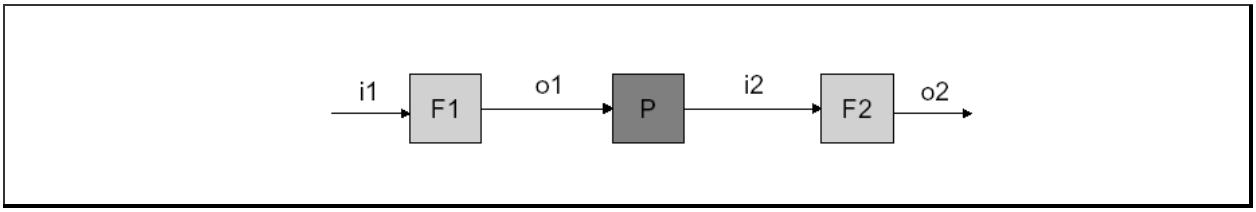


Figure 21 - Architecture Pipe-Filter

Les filtres F1 et F2 reçoivent des données sur leurs connexions d'entrées i1 et i2, les transforment et émettent ensuite le résultat sur leurs canaux de sortie. Le tube (pipe) assure la transmission entre les deux filtres.

La propriété suivante est décrite en utilisant le prédicat *isin*. Ce prédicat est vrai si un élément fait partie d'un ensemble. La fonction *channels* aussi est utilisée, elle retourne l'ensemble des connexions d'un comportement. Ainsi la propriété signifie que tout élément de type Filter est connecté à un élément de type Pipe par une connexion.

```
forall { f:Filter |
     exists { c1, c2:channel |
          exists { p:Pipe |
               (c1 isin channels (f)) and (c2 isin channels (p)) and connect (c1,
c2)
          }
     }
}
```

Prédicats sur les comportements

ArchWare AAL permet de décrire des patrons comportementaux. Ce sont des prédicats qui sont vrais si l'ordonnancement des actions d'un comportement vérifie certaines règles qu'on exprime.

Ces règles expriment l'agencement possible entre des actions de communication. Elles sont décrites à l'aide d'opérateurs spécifiques :

- la concaténation, '**.**',
- le choix, '**|**',
- la fermeture transitive réflexive, '*****',
  - elle correspond à la concaténation de zéro ou plusieurs actions.
- la fermeture transitive, '+'.
  - elle correspond à la concaténation d'une ou plusieurs actions.

L'expression suivante dénote un patron pour lequel une réception via la connexion *c2* ne peut être précédée directement d'une ou plusieurs émissions via la connexion *c1*.

```
(not(via c1 send any))*.(via c2 receive any)
```

De plus, AAL fournit deux autres opérateurs en se basant sur les opérateurs modaux de *nécessité* et de *possibilité* du μ-calcul [All et al. 02]. Le premier permet de spécifier que *toute séquence d'actions* vérifiant un patron comportemental donné doit aboutir à un état donné. Le second permet

de spécifier qu'il *doit exister au moins une séquence d'actions* vérifiant un patron comportemental donné et aboutissant à un état donné. La propriété suivante exprime qu'il ne peut y avoir d'envoi par la connexion *c2* que s'il y a eu réception par la connexion *c1* juste avant. Dit autrement, chaque séquence dans laquelle un envoi par la connexion *c2* n'est pas précédé par une réception par la connexion *c1* conduit vers un état faux.

```
every sequence { not via c1 receive any. via c2 send any} leads to state {false}
```

La propriété suivante exprime qu'il doit exister au moins une séquence dans laquelle une action de réception par la connexion *c1* précède un envoi par la connexion *c2*.

```
some sequence { via c1 receive any } leads to state { via c2 send any }
```

Enfin, AAL fournit deux derniers opérateurs se basant sur les quantificateurs de points fixes maximaux et minimaux du μ-calcul :

- **finite tree** Y (x:T) **given by** { s (Y) } (v)
- **infinite tree** Y (x:T) **given by** { s (Y) } (v)

Le premier opérateur est basé sur les points fixes minimaux. Cet opérateur spécifie qu'en partant d'un état satisfaisant *Y*, on arrive après un nombre fini de pas (un "pas" = un dépliage récursif de la formule s) à un état satisfaisant *s* (false). L'intuition est que les points fixes minimaux se comportent comme des fonctions récursives en langage de programmation, sauf qu'ils sont interprétés sur des systèmes de transitions. En d'autres termes, un patron d'actions peut être répété un nombre fini de fois avant que le système atteigne un état donné.

Par exemple, la formule suivante exprime l'existence d'une succession finie d'actions **via** x **receive any**.

```
finite tree Y given by { some sequence { via x receive any } leads to state { true } or Y }
```

Après un nombre fini de pas (transitions franchies en dépliant le corps "**{ some sequence { via** x **receive any } leads to state { true } or** Y" de la formule), on doit arriver à un état qui satisfait **{ some sequence { via** x **receive any } leads to state { true } or false**, c.a.d. **{ some sequence { via** x **receive any } leads to state { true }**, c.a.d. l'état source d'une action d'envoi par la connexion *x*.

Le second opérateur est basé sur les points fixes maximaux. Cela permet d'itérer indéfiniment un même patron, ce qui les rend utiles pour spécifier des comportements infinis (le système peut ne jamais s'arrêter).
Par exemple, la formule suivante signifie l'existence d'une séquence infinie de réception par la connexion *x*.

```
infinite tree Y given by { some sequence{ via x receive any } leads to { Y }}
```

2. Formalisation d'un style

Dans cette section, nous présentons le langage ASL à travers les mécanismes qu'il fournit pour la formalisation des styles. La description d'un style en ASL s'appuie sur trois concepts : les *constructeurs* qui fournissent un support à la description architecturale, les *contraintes* et les *analyses*. Dans les sections suivantes, nous allons étudier ces trois concepts. Mais, au préalable, nous présentons des généralités concernant la description des styles.

2.1. Généralités

Les styles architecturaux sont intrinsèques aux développements centrés architecture. Ils ont pour vocation de promouvoir la réutilisation des connaissances acquises dans un domaine architectural et de favoriser la compréhension. Ces styles sont souvent utilisés de manière informelle, mais un effort de formalisation a été entrepris dans ce domaine (cf. chapitre 3). Cependant, jusqu'à

aujourd'hui, on ne pouvait pas formaliser des styles ne prenant pas en compte la dynamique des architectures.

Ainsi, nous définissons ASL pour offrir aux architectes des outils formels pour le développement logiciel centré style dans le cadre de la production de systèmes dynamiques.

Les sections suivantes présentent les formalisations des styles et la façon de les décrire.

2.1.1 Définition d'un style

Nous avons vu qu'un style est l'ensemble des propriétés partagées au sein d'une famille d'architectures. Ces propriétés sont appelées contraintes de styles. Nous avons aussi vu qu'un style est associé à un support à la description et à l'analyse architecturale.

Dans le cadre d'ASL, la définition d'un style formalisé est la suivante.

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

Un style formalisé est structuré de trois entités :

- les *constructeurs* (qui fournissent un support à la description architecturale),
- les *contraintes,*
- les *analyses.*

Le langage ASL fournit aussi des mécanismes permettant d'organiser les styles de manière à promouvoir leur réutilisation et la compréhension de leurs relations. Ces mécanismes sont l'*héritage* et l'*agrégation*. L'*héritage* est un mécanisme de spécialisation (cf. chapitre 2). Il permet de décrire un nouveau style tout en conservant les caractéristiques d'un autre style. L'*agrégation* est un mécanisme de composition. Il permet de structurer un style par plusieurs autres styles. Par exemple, un style qui définit les architectures Client-Serveur peut être constitué d'un style Client et d'un style Serveur. Le style client définit des caractéristiques propres à un aspect particulier du Client-Serveur[14].

2.1.2 Description d'un style

Une description en ASL consiste en la définition d'un style. L'encadré ci-dessous montre la structure[15] générale de la description d'un style dont nous allons étudier chaque aspect.

```
Nom_du_style is style extending Style_parent where {
     types { Définitions de types     }
     styles {  Définitions de styles }
     constructors {  Définitions de constructeurs  }
     constraints {  Définitions de contraintes    }
     analyses { Définitions d'analyses      }
}
```

Nom du style

Un style est nommé. Son nom est l'unique référence vers ce style. Il est nécessaire de faire référence à un style à plusieurs occasions : lors de son instanciation, lors de la vérification de la satisfaction d'une architecture par rapport à ce style, lors de l'héritage par un autre style.

Par convention le nom, d'un style commence par une lettre majuscule.

[14] Des définitions plus détailllées de ces termes seront fournies par la suite.

[15] Il ne s'agit pas de la BNF du langage.

Style parent

ASL fournit un mécanisme de spécialisation : l'héritage.

> L'***héritage*** donne la possibilité de construite un nouveau style *S* sur la définition d'un autre style *S'*. Le style *S'* est alors appelé style parent du style *S* et le style *S* un sous-style de *S'*. Les éléments définis au niveau du style *S'* (les constructeurs, les contraintes et les analyses) sont des acquis pour le style *S*.

L'héritage répond à un besoin de :

- réutilisation,
- hiérarchisation.

L'héritage promeut la réutilisation dans le sens où, à partir d'un seul style, on peut en définir plusieurs autres. L'héritage promeut la hiérarchisation afin de définir des styles du plus abstrait au plus concret. Cette relation spécifie le style *S*, définit une famille d'architectures dont toutes les architectures satisfont le style *S'* ; toute architecture qui satisfait *S,* satisfait *S'*.

On spécifie qu'un style en spécialise un autre par le mot clé ***extending***. Dans l'exemple ci-dessous, le style *Style* hérite du style *Style_parent*.

```
Style is style extending Style_parent where { … }
```

Nous étudierons plus en détail l'héritage dans les prochaines sections pour comprendre les mécanismes associés aux niveaux des constructeurs, des contraintes et des analyses.

Types

La définition d'un style peut englober la définition d'un ensemble de types. Bien que les types puissent être définis dans une description en ArchWare ADL, nous avons choisi de permettre la définition des types au niveau styles pour une plus grande réutilisation de ceux-ci.
Les types sont construits par rapport aux types prédéfinis dans ASL. Ils sont utilisables dans tout le reste de la description d'un style.

```
types {
     type nom_type is definition_type,
     …
}
```

Tous les types prédéfinis en ASL sont :

- les types définis en ArchWare ADL,
  - les types de base : **Boolean**, **Natural**, **Integer**, **Real**, **String**,
  - les types de construction : **Tuple**, **…**,
  - les types collections : **Bag**, **Set**, **Sequence**,
  - les types liés aux architectures : **Connection**, **Behaviour**, **Abstraction[**T_1,…,T_n**]**,
- des types introduits en ASL,
  - **Type**, **Expression[** T **]**, **Alias**,
  - ces types sont seulement utilisables pour les paramètres des constructeurs. Nous les étudierons en détail dans la section dédiée aux constructeurs.

Styles

ASL fournit un mécanisme de composition : l'agrégation.

> L'***agrégation*** donne la possibilité de structurer un style en terme d'autres styles dits agrégats. Les styles agrégats encapsulent des constructeurs, de contraintes et des analyses qui alimentent le style contenant.

Les bénéfices de l'agrégation sont :

- une meilleure réutilisation,

- une meilleure lisibilité et une meilleure compréhension du style.

Le mécanisme d'agrégation promeut la réutilisation. En effet, un même style peut servir à la définition de différents autres styles. On a ainsi la possibilité de construire une bibliothèque de styles propres à un domaine particulier.
La structure d'un style en termes d'agrégats peut être construite parallèlement à celle des architectures qu'il représente. Un style qui représente les architectures Client-Serveur peut-être structuré avec un style Client et un style Serveur. Un style agrégat donne un support pour des sous-parties des architectures.
Les styles agrégats qui composent le style sont définis dans la clause ***styles***. Comme le montre l'exemple suivant, ils peuvent être définis par une *description de style* (comme *Client*) ou par *référence* à un style déjà défini en dehors de la description (comme *Serveur*). Ainsi la définition de *Client* est donnée explicitement (`Client is style where{…}`) tandis que celle de Serveur est celle d'un autre style appelé *SERVEUR* (`Serveur is SERVEUR`).

```
ClientServeur is style where {
     styles {
          Client is style where{…};
          Serveur is SERVEUR;
     }
     …
}
```

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

Constructeurs

Les constructeurs fournissent le support à la description architecturale. Ils permettent à la fois de fournir des éléments, et des mécanismes architecturaux comme des patrons et des actions. Ils sont étudiés dans la section 2.2.

Contraintes

Les contraintes de style caractérisent une famille d'architecture. Ces architectures présentent des qualités spécifiques en termes de structure ou de comportement. Par exemple, une structure en étoile promeut la modifiabilité du système[16] : on peut modifier une des branches sans affecter les autres. On s'assure qu'une architecture présente l'ensemble des qualités entraînées par un style lorsqu'elle satisfait ses contraintes. Ainsi, les contraintes guident l'architecte dans la conception architecturale. Elles sont étudiées dans la section 2.3.

Analyses

Les analyses de style sont spécifiques aux architectures suivant le style. Elles sont étudiées dans la section 2.4.

2.2. Constructeur

Un style architectural concentre l'expérience et la connaissance acquises dans un domaine ou un problème particulier. Une partie de cette expérience et de cette connaissance se caractérise par un support à la 'construction' architecturale. Ainsi un style définit des modèles de construction à travers des patrons architecturaux (par exemple, une architecture en étoile), et des briques de construction à travers des éléments architecturaux réutilisables (par exemple, connecteur gérant le broadcasting).

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

[16] Le style DataIndirection est basé sur une structure étoilée. Celui-ci est etudié dans le chapitre 5.

- la génération d'architectures à partir d'un style.
- la réutilisation de code. Un constructeur permet de définir :
  - o des valeurs architecturales,
  - o des mécanismes architecturaux.
- la définition d'une syntaxe spécifique. Un constructeur permet de définir une syntaxe plus proche des concepts d'un domaine particulier. Cela favorise la lecture et la compréhension d'une définition architecturale.

2.2.1 Définition d'un constructeur

Les constructeurs offrent un support à la description d'architectures en regard d'un style particulier. Leur fonction est de construire des parties de descriptions d'architectures.

> Un *constructeur* est la définition d'une valeur[17] dépendant ou non d'un contexte donné et d'un ensemble de paramètres.

La valeur définie par un constructeur peut représenter aussi bien une donnée, qu'une connexion, qu'une architecture, qu'un élément architectural ou qu'un comportement.

> Nous appelons *espace de construction* d'un constructeur, l'ensemble des valeurs qu'on peut générer à partir du constructeur.

A titre d'exemple, le constructeur *complexe* défini ci-dessous permet de générer des tuples en fonction de deux valeurs *a* et *b*. Le tuple représente un nombre complexe.

```
complexe is constructor ( a : Real, b : Real);
{ tuple(a, b)}
```

L'espace de construction de *complexe* est l'ensemble des valeurs de la forme **tuple(**a,b**)** et de type **tuple[Real**, **Real]**.

Un constructeur est un mécanisme de réutilisation. L'ensemble des constructeurs d'un style constitue une bibliothèque de définitions réutilisables. Des mécanismes permettent de les utiliser afin de générer d'autres constructeurs plus spécifiques ou des valeurs. Ces mécanismes sont appelés applications. Nous les étudierons par la suite.

Parmi les constructeurs, nous différencions deux types : les *constructeurs de support* et les *constructeurs d'architectures*.

> Un ***constructeur d'architecture*** encapsule la définition d'une valeur destinée à satisfaire les contraintes de style. Ainsi, un constructeur d'architectures permet d'instancier le style dans lequel il est défini.

> Un ***constructeur de support*** permet de générer des valeurs utilisables pour la définition d'architectures. Ces valeurs n'ont pas de contraintes spécifiques à satisfaire. Le but de ces constructeurs est d'encapsuler des descriptions réutilisables. Par exemple, le constructeur *complexe* vu précédemment est un constructeur de support.

Nous rediscuterons des différences entre ces types de constructeurs à travers les prochaines sections.

2.2.2 Description d'un constructeur

Dans cette section nous montrons comment décrire un constructeur. Nous proposons d'étudier ce concept à travers la structure syntaxique de sa définition.

```
nom_du_constructeur is constructor (p1:Type_p1, …, pn:Type_pn);
```

[17] Une valeur est une description "architecturale" au sens d'ArchWare ADL.

```
context(c1:Type c1, ..., cn:Type cn);
{ corps_du_constructeur}
as { notation_mixfix }
```

Un constructeur est défini avec un nom (*nom_du_constructeur*), un ensemble de paramètres typés (p_1, p_2, …, p_n), un contexte (c_1, c_2, …, c_m), un corps (*corps_du_constructeur*) et une notation mixfix (*notation_mixfix*).

Nous illustrons notre discours sur le constructeur avec l'exemple suivant. Il s'agit d'un constructeur de valeurs de type ***abstraction***. Ces abstractions représentent des éléments clients pour une architecture client-serveur. Le constructeur sera expliqué au fil des paragraphes suivants.

```
Client is constructor( nom:String, requete:String);
context(   entreprise:String    );{
     value envoi is tuple(nom, entreprise, requete);
     abstraction ();{
          via call send envoi;
          via wait receive reponse:String;
          unobservable
     }
} as {
     client named $nom requiring $requete
}
```

Nom du constructeur

Le nom du constructeur est une référence vers celui-ci. On peut appliquer un constructeur en utilisant cette référence.
Plusieurs constructeurs peuvent porter le même nom. Dans ce cas, ils doivent avoir un ensemble de paramètres différents (cf. paragraphe suivant).
Concernant les conventions de nommage, elles sont différentes suivant la nature du constructeur. Dans le cas d'un constructeur de support, la première lettre est en minuscule. Dans le cas d'un constructeur d'architectures, le nom est celui du style qui le contient ; la première lettre est en majuscule.

Dans notre exemple, le constructeur est nommé *Client*. Il s'agit d'un constructeur d'architectures permettant de créer rapidement des éléments clients en leur donnant un nom et une requête spécifique.

```
Client is constructor(…);{…}…
```

Paramètres

Un constructeur est défini avec de paramètres. Les résultats obtenus lors de l'utilisation d'un constructeur dépendent des valeurs attribuées à ses paramètres.

Dans notre exemple, le constructeur est défini avec deux paramètres, *nom* et *requete*, qui représentent respectivement le nom du client construit et la requête qu'il envoie au serveur.

```
Client is constructor( nom:String, requete:String);
```

Les types des paramètres peuvent être de n'importe quel type. ASL définit les types supplémentaires suivant pour les paramètres des constructeurs :

- **Type**,
  - Le type ***Type*** est un type dont les valeurs associées sont des types,
- **Expression[** T **]**,

- Le type ***Expression[T]*** permet de capturer des définitions de valeurs de type T. Par exemple, un paramètre de type ***Expression[Behaviour]*** est la définition d'un comportement et non pas une instance de comportement qui s'exécute,
- Le type ***Expression[T]*** est associé à un opérateur qui permet de créer une instance à partir d'une définition. Cet opérateur est ***eval***. Son argument est obligatoirement l'identifiant d'un paramètre d'un constructeur. Par exemple on peut définir un constructeur de clients dont le paramètre sera le comportement du client construit,

```
Client is constructor(comportement:Expression[Behaviour]);{
     eval comportement
}
```

- **Alias**,
  - Le type ***Alias*** est défini pour pouvoir passer un alias en paramètre (et non l'instance qu'il référence),
  - Un alias passé en paramètre d'un constructeur peut être transformé en chaîne de caractères au sein du corps du constructeur. L'opérateur qui permet cela est ***toString***. Cet opérateur permet de faciliter les descriptions en permettant de passer un identifiant en paramètre d'un constructeur en lieu et place d'une chaîne de caractères. Il s'agit de sucre syntaxique. Par exemple, on peut définir un paramètre *nom* de type ***Alias*** pour un constructeur de clients.

```
Client is constructor(nom:Alias);{
     via call send toString nom; via wait receive; unobservable
}
```

Contexte

Le contexte, dans lequel est appliqué le constructeur, est représenté par l'ensemble des identifiants dont la portée englobe l'application d'un constructeur. Il est possible d'exprimer les identifiants qui doivent faire partie du contexte de l'application.

> Le ***contexte*** d'un constructeur définit un ensemble de valeurs qui doivent être définies dans le contexte d'application d'un constructeur. Il s'agit de sucre syntaxique permettant de passer certains paramètres de manière implicite.

Nous avons défini ce mécanisme afin de simplifier l'utilisation des constructeurs et de rendre transparents à l'architecte les mécanismes sous-jacents à un style.

Par exemple, supposons qu'un élément architectural est composé d'un ensemble de ports et d'un comportement. Supposons qu'il existe un constructeur *include_port* utilisable pour la description du comportement. L'appel de ce constructeur permet d'inclure un nouveau port à l'élément architectural. Logiquement, pour appliquer ce constructeur, il faut donner une valeur pour le port et une valeur désignant l'ensemble dans lequel nous voulons l'inclure. Ainsi, on écrirait au sein du comportement :

```
…
include_port (ensemble_de_port,nouveau_port)
…
```

Mais, il est évident que le comportement d'un élément architectural est toujours lié au même sous-ensemble de port. La déclaration de *ensemble_de_port* dans le contexte permet d'écrire simplement :

```
…
include_port (nouveau_port)
…
```

La description du corps d'un constructeur peut prendre en compte la valeur des paramètres et des éléments du contexte.

Définir un identifiant dans la clause contexte d'un constructeur présente quelques conséquences :

- L'expression du contexte contraint la possibilité d'appliquer un constructeur. Si un identifiant de la clause contexte ne fait pas partie du contexte, on ne peut pas appliquer.
- Si un identifiant du contexte correspond à un identifiant des paramètres, c'est la valeur du paramètre qui sera retenue dans la définition de la valeur. Néanmoins, le contexte contraindra toujours l'application par cet identifiant.

Dans l'exemple, *entreprise* qui est un identifiant pour une valeur de type String et le sous-ensemble minimal d'identifiants devant faire partie du contexte lors d'une application du constructeur. On suppose ainsi qu'un client est créé dans un contexte ou le nom de l'entreprise est défini. Ainsi, plusieurs clients peuvent être créés dans le même contexte sans avoir à rappeler le nom de l'entreprise à chaque fois.

```
Client is constructor( nom:String, requete:String);
context(   entreprise:String    );…
```

Corps

> Le ***corps*** d'un constructeur *contient la description d'une valeur*. Plus précisément, dans le corps d'un constructeur, on décrit comment une valeur est construite relativement à des paramètres et à un contexte.

On utilise une version d'ArchWare ADL étendue avec de nouveaux types et de nouveaux opérateurs pour la définition du corps d'un constructeur. Nous rappelons qu'une description en ArchWare ADL est une suite de clauses séparées par '**;**'.

Chaque clause peut être soit :

- une déclaration de valeur ou de type,
  - **value** *identifiant* **is** *valeur*
  - **type** *identifiant* **is** *type*
- une action de communication ou une action inobservable,
- une valeur d'un type quelconque parmi ceux d'ArchWare ADL (***Integer***, ***sequence[…]***, ***connection[…]***, ***Behaviour***, ***abstraction[…]***, …).

La dernière clause du corps exprime la valeur construite par le constructeur lors d'une application. Les clauses précédentes sont des clauses pouvant préparer la définition de cette valeur.

Dans notre exemple, la dernière clause est une abstraction. Cette abstraction représente le comportement d'un client : l'envoi d'une requête et des informations concernant le client ; la réception de la réponse du serveur ; puis une action de traitement inobservable.

```
{    value envoi is tuple(nom, entreprise, requete);
     abstraction ();{
          via call send envoi;
          via wait receive reponse:String;
          unobservable
     }
}
```

Il est possible que la dernière clause d'une description ne soit pas une valeur. Dans ce cas, la valeur construite est un comportement (de type ***Behaviour***) défini par l'ensemble du corps du constructeur.

Les types supplémentaires, sont ceux que nous avons étudiés dans le paragraphe sur les paramètres : ***Type***, ***Expression [T]***, ***Alias***. Les opérateurs sont ceux associés à ces types : ***eval*** et ***toString***. Il y a un certain nombre de contraintes associées à ces types :

- On ne peut pas déclarer de valeur de type ***Type***.
  - Ces valeurs sont prédéfinies, ce sont les types définis dans le langage.
- On ne peut pas déclarer de valeur de type ***Expression [T]***.
  - Le type ***Expression [T]*** est seulement un type pour un paramètre, il permet au constructeur de savoir si une description passée en paramètre doit être considérée comme une valeur ou comme la définition d'une valeur.
- Les alias sont déclarés à la déclaration d'une valeur.

Mixfix

Un constructeur peut être associé à une notation spécifique. Ainsi, l'architecte appuyant sa conception architecturale sur un style peut avoir à sa disposition une syntaxe adéquate aux concepts définis par le style. Il peut ainsi s'abstraire des concepts basiques du langage ASL. Par exemple, on facilite la compréhension d'un système client-serveur en parlant de client et de serveur plutôt que de composants.

Le mécanisme de notation introduit est appelé notation mixfix.

> La notation mixfix est un mécanisme de notation associée à l'utilisation d'un constructeur.

Ce mécanisme permet de fournir un langage spécifique à un domaine dans le cadre de la description d'architectures. La sémantique associée à cette syntaxe est spécifiée par le constructeur.

A l'application d'un constructeur celui-ci peut être référencé :

- soit par son nom,
- soit par la notation mixfix.

Lorsqu'on définit cette notation, on exprime l'organisation d'un ensemble de mot-clés autour des paramètres du constructeur. Cette notation est dite mixfix car elle permet d'écrire les paramètres avant, après ou entre les mots clés identifiant le constructeur.

L'exemple montre une notation mixfix pour le constructeur *Client*. On remarquera que les identifiants des paramètres sont précédés d'un '$'. C'est de cette manière qu'on différencie les noms d'identifiants des mots-clés utilisés pour la notation.

```
client named $nom requiring $requete
```

En utilisant cette notation mixfix, on peut appliquer le constructeur pour définir un nouvel élément client. Ce client est appelé *horloge*, il demande l'heure au serveur.

```
client named "horloge" requiring "heure"
```

Nous verrons plus tard plusieurs manières d'appliquer un constructeur. En fonction de la nature de l'application, il n'est pas toujours nécessaire de donner une valeur pour chacun des paramètres. Ainsi, on peut adapter la notation mixfix pour qu'elle convienne à différents types d'applications. Pour cela, on utilise un opérateur propre à la description d'une BNF, l'opérateur d'optionalité noté […].

```
client named $nom [requiring $requete]
```

Une autre particularité en regard de l'application d'un constructeur via sa notation mixfix est liée aux paramètres de type collection (***Bag***, ***Set***, ***Sequence***). Nous introduisons un raccourci d'écriture permettant de se passer des mots-clés ***sequence***, ***set*** ou ***bag***. On facilite ainsi l'écriture et la

lecture. Une collection peut être définie de la manière suivante[18] : **{**element$_1$**,** element$_2$**, …,** element$_n$**}**.

2.2.3 Applications des constructeurs

Un constructeur est utilisé afin de générer soit une valeur soit un autre constructeur plus spécifique. Dans les deux cas, il s'agit d'applications. Dans le premier cas, il s'agit d'application totale et dans le second, d'application partielle.

Application totale

> L'***application totale*** d'un constructeur est la génération d'une valeur en fonction d'une liste de paramètres et d'un contexte spécifique. Des valeurs sont données pour l'interprétation des paramètres.

L'application totale est un mécanisme introduit pour générer des valeurs architecturales. L'application est dite 'totale' car il est nécessaire de donner une valeur pour la totalité des paramètres définis au niveau du constructeur.

Une application totale s'effectue dans une description en ArchWare ADL [Cim et al. 02] dans le corps d'un constructeur. La syntaxe classique utilisée suit la structure suivante.

```
nom_du_constructeur (v1, …, vn)
```

Une application totale est exprimée en faisant référence à un constructeur et en donnant une liste de valeurs. Chacune de ces valeurs fixe un paramètre correspondant au même emplacement dans la liste.
Nous verrons dans un exemple que nous pouvons aussi utiliser la notation mixfix pour appliquer un constructeur.

En regard de la BNF d'ArchWare ADL l'application totale d'un constructeur est une expression.

```
expression ::=   …
          |      constructor_application
```

En d'autres termes, il est possible de spécifier une application totale en lieu et place de l'expression d'une valeur. Cependant, il faut respecter certaines règles. D'une part, le contexte minimal requis par le constructeur doit être un sous-ensemble du contexte dans lequel l'application a lieu. D'autre part, il faut respecter les règles de typage. Le type d'une application totale correspond au type de la valeur définie par le constructeur.

Une règle de typage se vérifie sur une expression. Par exemple l'expression peut être une somme :

```
1 + 2
```

Le mécanisme de typage permet de déterminer le type d'une expression et de retourner une erreur si les règles de typage ne sont pas respectées.

Une règle de typage est construite de la manière suivante. Elle est séparée en une partie *prémisse* et une partie *conclusion*. Lorsque la partie prémisse est vérifiée la partie conclusion est vérifiée. Ci dessous se trouve, la règle de typage pour la somme.

T' ∈ { **Natural**, **Integer**, **Real** }

$$\frac{\tau,\delta \mapsto e_1 : T' \quad \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 + e_2 : T'}$$

[18] Cette valeur est évaluée suivant le type attendu en paramètre du constructeur : **Set**, **Bag** ou **Sequence.**

La règle de typage[19] concernant la définition d'un constructeur est la suivante :

$$\frac{\tau^i, \delta_1^i :: < x_i, \mathrm{T_1} > :: \ldots :: < x_n, \mathrm{T_n} > :: < c_i, \mathrm{Tc_1} > :: \ldots :: < c_n, \mathrm{Tc_m} > :: \delta_2^i \mapsto \mathrm{e} : \mathrm{S}}{\tau^i, \delta^i \mapsto \mathrm{constructor}(\mathrm{x_1} : \mathrm{T_1}, \ldots, \mathrm{x_n} : \mathrm{T_n}); (\mathrm{c_1} : \mathrm{Tc_1}, \ldots, \mathrm{c_m} : \mathrm{Tc_m}); \{e\} : \mathrm{constructor}[\mathrm{T_1}, \ldots, \mathrm{T_n}][\mathrm{Tc_1}, \ldots, \mathrm{Tc_m}] \rightarrow \mathrm{S}}$$

La règle de typage concernant l'application d'un constructeur est la suivante :

$$\frac{\tau^i, \delta^i \mapsto \mathrm{e} : \mathrm{constructor}[\mathrm{T_1}, \ldots, \mathrm{T_n}][\mathrm{Tc_1}, \ldots, \mathrm{Tc_m}] \rightarrow \mathrm{S} \quad \tau^i, \delta^i \mapsto \mathrm{e_1} : \mathrm{T_1} \ldots \tau^i, \delta^i \mapsto \mathrm{e_n} : \mathrm{T_n}}{\tau^i, \delta^i \mapsto \mathrm{e}(\mathrm{e_1}, \ldots, \mathrm{e_n}) : \mathrm{S}}$$

L'exemple suivant est une application totale du constructeur *Client*. Notez que nous prenons soin de déclarer l'identifiant *entreprise* pour spécifier le contexte minimal requis pour l'application.

```
value entreprise is "Thésame";
value horloge is client("horloge","heure");
…
```

Cette description est équivalente à la description suivante.

```
value entreprise is "Thésame";
value horloge is {
     value nom is "horloge";
     value requete is "heure";
     value envoi is tuple(nom, entreprise, requete);
     abstraction ();{
          via call send envoi;
          via wait receive reponse:String;
          unobservable
     }
};
…
```

Donc, la valeur évaluée pour *horloge* (dans son contexte) lors de l'application totale est la suivante.

```
abstraction ();{
     via call send tuple("horloge", "Thésame", "heure");
     via wait receive reponse:String;
     unobservable
}
```

Comme nous l'avons mentionné, nous pouvons utiliser la notation mixfix pour appliquer un constructeur. On aurait pu écrire :

```
value entreprise is "Thésame";
value horloge is client named "horloge" requiring "heure";
…
```

Concernant le typage, nous pouvons noter que le type du constructeur *Client* est :

Constructor[String,String][String]→ Abstraction[]

Le type de l'application de *Client* est donc : **Abstraction[]**

Nous venons de voir que nous pouvons générer des valeurs par application totale. Cette valeur peut être de n'importe quel type et peut aussi bien représenter une donnée qu'une architecture ou un élément architectural. Pour simplifier la compréhension, nous nous sommes basés sur un

[19] Nous nous appuyons sur les notations utilisées pour définir les règles de typage d'ArchWare ADL [Cim et al. 02].

constructeur de données. Nous verrons par la suite des exemples concernant des constructeurs pour des éléments architecturaux (par exemple, dans la section 2.2.4).

Une deuxième manière d'appliquer un constructeur est l'application partielle.

Application partielle

> L'***application partielle*** est la génération d'un constructeur en fonction d'un constructeur plus générique et un ensemble de paramètres. Elle consiste à fournir des valeurs pour seulement une partie des paramètres.

Nous avons introduit ce concept par souci de réutilisation. Par application partielle, on peut obtenir plusieurs constructeurs à partir de la définition d'un seul.

Cette application est dite partielle car un sous-ensemble des paramètres est fixé lors de l'application. L'application partielle est un mécanisme de spécialisation. A partir d'un constructeur *C*, il permet d'obtenir un autre constructeur *C'* plus spécifique. Cela signifie que l'espace de construction de *C'* est un sous-espace de l'espace de construction de *C*.

Une application partielle est exprimée en faisant référence à un constructeur et en donnant une liste de paramètres. Syntaxiquement, à la différence d'une application totale, il est nécessaire d'associer l'identifiant du paramètre à la valeur donnée. En effet, comme seul un sous-ensemble des paramètres est fixé, il n'est pas possible de différencier ces paramètres par leurs positions.

```
nom_du_constructeur (v1 as p1, …, vn as pn)
```

Une application partielle s'effectue lors de la définition d'un constructeur[20].

```
constructors{
…
constructeur_specifique is constructeur_generique(v1 as p1, …, vn as pn),
…
}
```

Le constructeur résultant d'une application partielle est défini avec un ensemble de paramètres différents de celui du constructeur appliqué. La règle est la suivante : les paramètres du constructeur résultant sont les paramètres du constructeur appliqué moins les paramètres fixés à l'application si ces derniers ne sont pas des collections (set, bag, sequence).
Les paramètres de type collection sont considérés à part. En effet, nous suggérons de pouvoir donner seulement une partie des éléments d'une collection et de pouvoir compléter cette collection à une prochaine application. Ainsi, un paramètre de type collection reste un paramètre après une application partielle.
Concernant les éléments du contexte, ceux-ci sont des paramètres qui ne peuvent pas être fixés lors d'une application partielle. En effet, une application partielle s'effectue en dehors du corps d'un constructeur. De ce fait, le contexte est entièrement transmis.
Concernant la notation mixfix, celle-ci n'est pas transmise lors de l'application partielle. Comme plusieurs applications partielles peuvent être effectuées sur un même constructeur, on ne peut pas leur donner la même notation. Ainsi, par défaut, elle n'est pas transmise.

Supposons la structure suivante pour un constructeur appelé *constructeur_generique*. Ce constructeur est défini avec trois paramètres de types différents (*p1:**Integer**, p2:**Real**, p3:**set[Integer]***) et un constructeur *c1*.

```
constructeur_generique is constructor (p1:Integer, p2:Real, p3:set[Integer]);
     ( c1:Type_c1 );
     { corps_du_constructeur}
```

[20] On ne peut définir un constructeur à l'intérieur d'un autre constructeur. Ainsi, une application partielle n'a jamais lieu au sein d'un constructeur.

```
     as { notation_mixfix }
```

Supposons l'application partielle suivante : on ne donne des valeurs que pour p_1 et p_3.

```
constructeur_generique (1 as p1, set(1,2,3) as p3);
```

Le constructeur équivalent à cette application est le suivant : le paramètre p_1 n'est pas paramètre du constructeur généré, sa valeur est directement fixée à 1 ; le paramètre p_3 est paramètre du constructeur généré mais, une partie des éléments de l'ensemble sont aussi fixés directement.

```
constructor (p2:Real, p3:set[Integer]);
     ( c1:Type_c1 );
     {    p1 is 1;
          p3 includes 1;
          p3 includes 2;
          p3 includes 3;
          corps_du_constructeur
     }
```

La notation mixfix n'est pas conservée après application partielle cependant, il est possible d'en définir une nouvelle.

```
nom_du_constructeur (v1 as p1, …, vn as pn) as {nouvelle_notation mixfix}
```

L'exemple suivant est la définition du constructeur *Horloge* en appliquant partiellement le constructeur *Client*. Le constructeur *Horloge* permet de créer des clients qui envoient une requête pour demander l'heure.

```
Horloge_Client is Client("heure" as requete) as {
     horloge named $nom
}
```

Horloge_Client est alors équivalent au constructeur suivant :

```
constructor(nom:String)
context(   entreprise:String    );
{    value requete is "heure";
     value envoi is tuple(nom, entreprise, requete);
     abstraction ();{
          via call send envoi;
          via wait receive reponse:String;
          unobservable
     }
}
```

Nous verrons dans ce qui suit que l'application partielle est sous-jacente aux mécanismes d'héritage de style.

2.2.4 Constructeurs et héritage

Un sous-style hérite des constructeurs[21] de son parent. Il peut aussi définir ses propres constructeurs. Ces derniers peuvent surcharger les constructeurs du style parent. Ainsi, lorsqu'il y a ambiguïté, afin de pouvoir différencier un constructeur défini dans un style parent d'un constructeur défini dans le sous-style, un constructeur d'un parent peut être appelé par : nom_parent@nom_constructeur.

[21] Nous rappelons qu'un constructeur est juste un support à la définition et ne garantit pas la satisfaction d'une architecture à un style.

Le mécanisme d'application partielle des constructeurs est utile lors de l'héritage. Il permet de spécialiser des constructeurs du style parent pour qu'ils cadrent mieux avec le sous-style. Par exemple, supposons un style *ClientServeur*, proposant un constructeur pour générer une architecture à partir des définitions d'un client et d'un serveur, ainsi que des quantités de chaque élément.

```
ClientServeur is style where {
    constructors{
        ClientServeur is constructor(client:Behaviour, serveur:Behaviour,
                                     nb_clients:Integer,    nb_serveurs:Integer,);
        {…}
    }
    …
}
```

Un sous-style, *DeuxClients_UnServeur*, propose un constructeur pour générer une architecture à partir des définitions d'un client et d'un serveur. Ce constructeur est l'application partielle de celui du *ClientServeur*.

```
DeuxClients_UnServeur is style where {
    constructors{
        DeuxClients_UnServeur is ClientServeur (2 as nb_clients, 1 as
nb_serveurs)
    }
    …
}
```

2.3. Contraintes

Un style architectural définit une famille d'architectures à travers un ensemble de propriétés. Ces propriétés sont l'essence d'un style : on décide de suivre un style pour les garantir. On les appelle contraintes de style car elles définissent les limites d'un espace de définition d'architecture.

Formaliser un style, c'est avant tout formaliser des contraintes quelle que soit leur nature. Cette section montre comment décrire des contraintes.

2.3.1 Définition d'une contrainte

> Une ***contrainte*** est une assertion définissant des caractéristiques architecturales pouvant être évaluées sur une architecture.

Les contraintes sont un support à la conception architecturale ; elles guident l'architecte afin qu'il puisse construire une architecture suivant le style. Une architecture satisfaisant les contraintes d'un style bénéficie des qualités architecturelles induites par ce style.

Comme nous l'avons vu dans le chapitre 2, on différencie plusieurs types de contraintes :

- les contraintes structurelles,
  - par exemple : une architecture ne contient pas de cycles,
- les contraintes comportementales,
  - par exemple : il n'y a pas d'interblocage,
- les contraintes d'attributs,
  - par exemple : la durée de traitement d'un élément critique ne doit pas excéder 0,5 secondes.

La formalisation des contraintes est nécessaire pour vérifier automatiquement qu'une architecture satisfait un style[22].
Ceci peut être fait à différents moments :

- on a une architecture et on veut savoir si elle satisfait un style, et sinon pourquoi ?
- on instancie un style et on vérifie que l'instance satisfait bien le style.

2.3.2 Description d'une contrainte

Dans cette section, nous proposons d'étudier les concepts de définition de contraintes à travers leur description. La structure syntaxique d'une description de contrainte est la suivante. Une contrainte est définie avec un nom et un corps.

```
nom_de_la_contrainte is constraint {
     corps_de_la_contrainte
},
```

Une contrainte peut aussi être définie par référence à une contrainte d'un autre style.

```
constraints {
     nom_de_la_contrainte is référence_contrainte
}
```

Par exemple, on peut écrire l'expression qui suit. La contrainte *contrainte1* du style *autre_style* est référencée par *autre_style@contrainte1* pour définir la contrainte *contrainte*.

```
constraints {
     contrainte is autre_style@contrainte1
}
```

Nous illustrons notre discours sur les contraintes avec l'exemple suivant. La contrainte *deadLockFreedom* spécifie qu'à n'importe quel moment le système peut exécuter une action. C'est une contrainte portant sur le comportement du système. De manière littérale, elle signifie que pour toute séquence d'actions on peut trouver une action prochaine telle que celle-ci débouche elle-même sur une action. Ainsi, cette propriété signifie qu'une action peut toujours être effectuée, donc que le système ne peut pas bloquer.

```
deadLockFreedom is constraint {
     to styleInstance.behaviour apply{
          every sequence {true*}leads to state
          {some sequence { true } leads to state {true}}
     }
}
```

Nom de la contrainte

Le nom d'une contrainte permet en premier lieu de la référencer et si le nom est bien choisi, de la décrire brièvement. Ainsi, lorsqu'un architecte vérifie la satisfaction d'une architecture à un style, les contraintes non satisfaites lui sont indiquées sans ambiguïté par leur nom. Ainsi plusieurs contraintes ne devraient pas porter le même nom pour pouvoir être référencées correctement.

Dans notre exemple, le nom de la contrainte est *deadLockFreedom*.

Corps

Le corps d'une contrainte est une propriété. Cette propriété est décrite en ArchWare AAL car c'est le langage de description de propriétés associé à ArchWare ADL.

[22] Une contrainte décrite en ASL est vérifiée pour une architecture décrite en ArchWare ADL.

La contrainte concerne l'instance d'un style. Généralement, il s'agit d'une architecture représentée sous forme d'abstraction, mais il peut s'agir de n'importe quel autre type de donnée. Au sein de la description cette instance est représentée par le mot-clé *styleInstance*.
Ainsi, dans l'exemple défini plus haut, la propriété de deadlock s'applique au comportement défini par une instance.

Afin de faciliter la description des propriétés, un certain nombre de mécanismes ont été définis :

- un prédicat ***in style*** a été défini. Ce prédicat est vrai si la valeur à laquelle il est appliqué satisfait toutes les contraintes d'un style. La contrainte définie dans l'exemple suivant oblige la valeur l'*architecture* instanciée à suivre le style *Client*.

```
isClient is constraint {
     styleInstance in style(Client)
}
```

- il est permis d'utiliser les constructeurs au sein des contraintes pour définir des valeurs. Toutefois, seuls les constructeurs sans contexte défini sont utilisables dans la description d'une contrainte.

- des fonctions peuvent être définies par l'utilisateur. Ces fonctions sont décrites au sein des analyses et nous les étudierons en détail dans la section 2.4. Nous verrons qu'elles peuvent être décrites dans n'importe quel langage dont ARCHWARE ADL et ARCHWARE AAL.

- comme nous le verrons dans la section 3.2.2 nous offrons aussi des facilités d'écriture afin de définir des contraintes portant sur des actions, autres que les actions de communication, comme les attachements ou la création dynamique.

Dans la section suivante, nous discutons sur les différents types de contraintes.

2.3.3 Description des différents types de contraintes

Dans le chapitre 2, nous avons relevé différentes sortes de contraintes : les contraintes ***topologiques***, les contraintes ***comportementales*** et les contraintes ***informationnelles*** (d'attribut ou non-fonctionnelles). Dans cette section, nous étudions comment décrire ces contraintes en ASL.

Contraintes topologiques

> Les ***contraintes topologiques*** concernent la structure de l'architecture ou des éléments architecturaux.

Les contraintes sur la topologie d'une architecture ne fixent pas un cadre seulement à l'initialisation de l'architecture, mais tout au long de son évolution. En d'autres termes, une contrainte topologique doit être vérifiée à n'importe quel état de l'architecture. Ainsi, les contraintes topologiques sont aussi des contraintes sur la dynamique d'une architecture.

Parmi ces contraintes, on peut identifier différentes sortes. Elles peuvent contraindre :

- l'interface, en terme de type et de cardinalité,
- la nature des constituants,
- la quantité des constituants,
- les connexions entre les constituants,
- les connexions entre les constituants et l'interface de l'architecture.

Considérons *elements* comme l'ensemble des éléments constituants une architecture. Supposons qu'on veuille réduire les styles suivis par ces constituants à deux "espèces". L'expression suivante signifie littéralement que pour tout élément *x* de *elements*, *x* satisfait soit le style *Client* soit le style *Server*.

```
to styleInstance.elements apply {
```

```
        forall (x | x in style Client xor x in style Server)
}
```

Ceci est une contrainte forte dans le sens où elle laisse peu de flexibilité. Une contrainte plus flexible pourrait seulement obliger l'existence de constituants suivant un certain style sans empêcher l'existence d'autres styles de constituants. L'expression suivante signifie littéralement qu'il existe au moins un élément satisfaisant le style *Client*.

```
to styleInstance.elements apply {
     exists (x | x in style Client)
}
```

De plus, on peut imposer des contraintes sur le nombre de constituants satisfaisant un style. L'expression suivante signifie qu'il existe un élément du style *Client* tel qu'il soit compris dans un ensemble d'éléments vérifiant cette même propriété et tel que cet ensemble comprenne entre 1 et 5 éléments. En bref, cette propriété signifie que le système contient entre 1 et 5 éléments satisfaisant le style *Client*.

```
to styleInstance.elements apply {
     exists (x |     x in style Client and
                apply on x_set union set(x) and
                x_set.size>=1 and x_set.size<=5 )
}
```

ASL fournit un raccourci d'écriture pour spécifier les cardinalités. Ainsi, l'expression suivante est équivalente à la précédente.

```
to styleInstance.elements apply {
     exists ([1..5] x | x in style Client)
}
```

Concernant les liens de communication entre les éléments, AAL fournit un prédicat, *rename*, permettant de vérifier si deux connexions sont unifiées. ASL fournit un prédicat, *attached to*, basé sur le précédent afin de vérifier si deux comportements partagent une connexion. L'expression suivante signifie littéralement que pour tout couple d'élément (*x*, *y*) du système si les deux éléments suivent le style *Client* ceux-ci ne sont pas attachés. En bref, deux éléments satisfaisant le même style *Client* ne peuvent être connectés.

```
to styleInstance.elements apply {
     forall(x, y | x in style Client and y in style Client
                implies not(x attached to y))
}
```

Contraintes comportementales

> Les ***contraintes comportementales*** définissent des patrons sur l'ordonnancement des actions dans un comportement.

Les descriptions de contraintes topologiques sont basées sur la description de prédicats comportementaux tels qu'on les a étudiés dans la section consacrée à l'AAL (cf. 1.2.2).

Par des contraintes comportementales, on peut imposer la satisfaction de propriétés telles que la vivacité ou la sûreté.

> Un système est ***vivace*** s'il a toujours la possibilité d'effectuer une action.

> Un système est ***sûr*** lorsqu'il ne s'exécutera jamais d'une manière explicitement interdite.

Considérons un élément architectural. La contrainte suivante signifie littéralement qu'il existe deux connexions nommées *request* et *answer*, et qu'à chaque séquence d'actions terminée par une réception via la connexion *request* suit une séquence aboutissant à un envoi via la connexion *answer*. En bref, une réception par la connexion *request* entraîne un envoi par la connexion *answer***.** Cette contrainte est une contrainte de vivacité.

```
to styleInstance.connections apply{
     exists(r,a | r.name=request and a.name=answer and
          every sequence{true*.via r receive any}
          leads to state{true*.via a send any}
}
```

Contraintes d'attribut

Les contraintes d'attribut sont des contraintes sur les données contenues par un élément architectural. Ces données peuvent, par exemple, rendre compte de la capacité de stockage, de la vitesse de traitement d'un élément.

On peut définir des contraintes sur :

- les attributs que doit définir un élément,
- sur les valeurs permises pour un attribut,
- les relations entre les attributs.

Tout comme les contraintes topologiques, ces contraintes sont décrites par des prédicats sur les données (cf. 1.2.2). Elles doivent être vérifiées pendant toute évolution de l'architecture.
La propriété suivante signifie littéralement qu'il doit exister une valeur dont le nom est *latency*. En bref, une instance du style doit définir l'attribut *latency*.

```
to styleInstance.values apply{
     exists(x | x.name=latency )
}
```

On peut contraindre la valeur de cet attribut. Par exemple, l'expression suivante signifie littéralement qu'il existe une valeur nommée *latency* et que sa valeur est strictement supérieure à 0 et strictement inférieure à 100. En bref, la valeur de l'attribut *latency* doit être comprise strictement entre 0 et 100.

```
to styleInstance.values apply{
     exists(x | x.name=latency and x.value>0 and x.value<100)
}
```

De plus, il est possible de définir des relations de contraintes entre deux ou plusieurs attributs. Par exemple, l'expression suivante signifie littéralement qu'il existe un couple de valeurs (*x*, *y*) tel que x est nommé *height*, y nommé *width* et *x* soit inférieur strictement à *y*. On suppose que ces attributs servent à définir la taille des représentations graphiques des éléments architecturaux. En bref, il est spécifié qu'il doit exister les attributs *height* et *width*, et que la valeur de *height* doit être plus petite que celle de *width*.

```
to styleInstance.values apply {
     exists (x,y|    x.name=height and
                     y.name=width and
                     x.value<y.value)
}
```

2.3.4 Contraintes et héritage

Lors de l'héritage d'un style, toutes les contraintes du style parent sont héritées. Toutes les contraintes d'un style sont des contraintes de tous ses sous-styles. Un sous-style rajoute des

contraintes sans modifier les contraintes héritées. Ainsi, il décrit un espace de définition strictement inclus dans celui de son parent.

L'héritage permet à l'architecte de style de bénéficier des caractéristiques d'un style pour les appliquer à celui qu'il construit. Cependant, il est probable que l'architecte soit intéressé par plus d'un style, et qu'il désire bénéficier des caractéristiques de plusieurs. Pour imager cela, supposons qu'il dispose d'un style château et d'un style gothique. L'architecte voudrait composer ces deux styles en un style château gothique. Ce dernier style satisfait à plusieurs styles parents ; il s'agit d'*héritage multiple*.

L'expression des contraintes permet de définir l'héritage multiple. On peut définir un style par la composition de plusieurs styles en composant leurs contraintes. L'héritage multiple est obtenu par une composition avec l'opérateur booléen ***and*** comme le montre le code ci-dessous.

```
Style_Composite is style where{
     constraints {
          composition_style_1_2 is constraint {
                    styleInstance in style Style_1
               and
                    styleInstance in style Style_2
          }
     }
}
```

La Figure 22 montre les espaces de définition de deux styles et d'un style héritant de ces deux styles. L'espace de définition du sous-style correspond à l'intersection des espaces de définition de ses styles parents. En d'autres termes, une architecture satisfaisant le sous-style, satisfait tous ses styles parents.

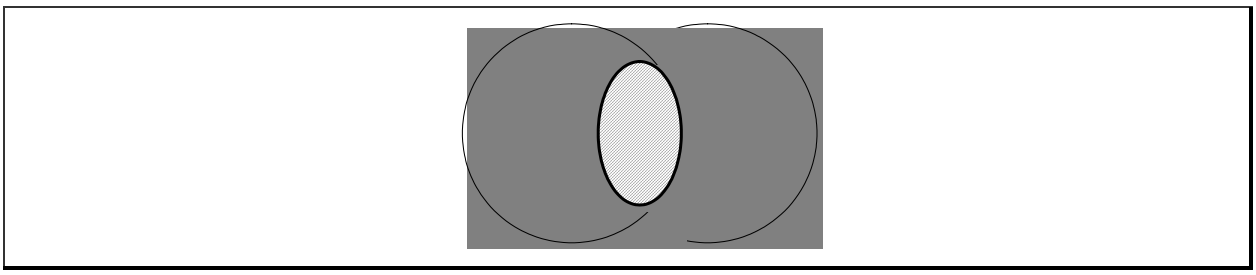


Figure 22 - Espace de définition d'un style héritant de deux autres styles

2.4. Analyses

Un style définit des concepts propres à un domaine particulier. Il peut définir de nouvelles sortes d'éléments architecturaux et de nouvelles sortes de relations entre les éléments architecturaux. Ce sont des concepts que l'architecte doit prendre en compte dans l'étude des caractéristiques de son architecture. Afin d'automatiser l'étude d'une architecture suivant un style particulier, il est nécessaire de définir des outils d'analyse spécifiques à ce style.

Dans cette section, nous abordons la description des analyses propres à un style.

2.4.1 Définition d'une analyse

Les analyses sont un support à la conception architecturale. Dans le contexte de notre étude, elles permettent d'analyser les architectures sur des aspects relatifs à un style.

> Une ***analyse architecturale*** est une étude ou un traitement automatisé d'une architecture pour en déterminer ou quantifier une ou plusieurs caractéristiques.

Les caractéristiques identifiées peuvent être de différentes sortes. Il peut s'agir d'une vérification ou d'une mesure. Une analyse peut vérifier la satisfaction d'une propriété par l'architecture ; par exemple, une analyse permet de vérifier que la structure de l'architecture est cyclique. Une analyse peut retourner une mesure ; par exemple, l'analyse retourne le nombre d'éléments d'une architecture.

Les analyses sont associées aux styles car leur capacité à évaluer telle ou telle caractéristique d'une architecture dépend fortement de la façon dont cette dernière est construite. En d'autres termes, elles sont dépendantes d'un style. Il y a différents moyens de quantifier une caractéristique en fonction des propriétés topologiques, comportementales ou d'attribut d'une architecture. Ainsi, la performance d'un système s'évalue différemment selon la nature de son architecture. S'il s'agit d'une architecture client-serveur, on évaluera la capacité de connexion du serveur. S'il s'agit d'une architecture pipe-filter, l'évaluation se basera sur le temps de traitement des données par les filtres. Un autre exemple figurant dans [Kle&Kaz 99] propose deux façons de quantifier la modifiabilité d'une architecture suivant qu'elle suit un style en couche ou un style tableau noir.

Dans ASL, les analyses peuvent être utilisées de deux manières différentes : elles sont soit appliquées à une architecture par l'architecte, soit utilisées comme support à la description des contraintes.
Dans le premier cas, l'architecte a, à sa disposition, un éventail d'analyses proposées par support à la critique de l'architecture. Ces analyses peuvent être appliquées sur l'ensemble de l'architecture ou sur des sous-parties de celle-ci.
Dans le deuxième cas, une analyse est considérée comme une opération retournant une valeur. D'une part, cela permet de structurer l'expression des contraintes et d'avoir une meilleure lisibilité. D'autre part, cela permet de définir des contraintes plus élaborées, basées sur d'autres mécanismes[23] que ceux de ARCHWARE AAL. Nous donnons plus de détails dans les prochaines sections.

2.4.2 Description d'une analyse

Dans cette section, nous montrons comment décrire une analyse. Nous proposons d'étudier ce concept à travers la structure syntaxique de sa définition.

```
nom_de_l_analyse is analysis {
     kind type_d_analyse
     input { parametres_d_entrée     }
     output {  type_de la valeur retournée  }
     body{
          corps de l'analyse
     }
}
```

Une analyse est définie avec un nom, un type d'analyse, un ensemble de paramètres d'entrée, un type de donnée pour le résultat retourné et un corps contenant la description du traitement.

Nous illustrons notre discours sur la description des analyses avec l'exemple suivant[24]. Il s'agit d'analyses pour un style Pipe-Filter (nous nous basons sur ce style plutôt que le Client-Server du fait des exemples d'analyses simples et démonstratifs que nous avons pu trouver pour ce style). Le style Pipe-Filter a été présenté dans le chapitre 2. Une première analyse, *reachable*, permet de savoir s'il existe un chemin de communication entre deux filtres (*f1* et *f2*). Nous rappelons que deux filtres sont liés directement par un pipe. Nous supposons l'existence des analyses *source of* et *sink of* qui permettent de vérifier respectivement si un filtre est en amont ou en aval d'un pipe. L'analyse retourne la valeur ***true*** s'il existe un pipe connectant *f1* à *f2*, ou s'il existe un autre filtre *f* tel qu'il

[23] Nous allons voir que nous pouvons intégrer des analyses de natures différentes.

[24] Cet exemple a été adapté à partir d'une documentation concernant le langage Armani [Mon 01].

existe un chemin de communication entre *f1* et *f* et tel qu'il existe un pipe reliant *f* à *f2*. On remarquera que cette analyse fait appel à elle-même.

```
reachable is analysis {
      kind AAL
      input {f1:Abstraction[],f2:Abstraction[]}
      output { Boolean }
      body {
            to styleInstance.elements apply{
            exists{p|f1 source of p and f2 sink of p}
            or
            exists{f| reachable(f1,f) and exists{p|f source of p and f2 sink of p)}
            }
      }
}
```

Une seconde analyse, *hasCycle*, permet de vérifier s'il y a des cycles dans l'architecture ; en d'autres termes, s'il existe un élément pour lequel il existe un chemin de communication vers lui-même. Ainsi, cette analyse fait appel à l'analyse *reachable* définie ci-dessus.

```
hasCycle is analysis{
      kind AAL
      input {}
      output { Boolean }
      body{
            to styleInstance.elements apply{
                  exists{f | reachable(f,f)}
            }
      }
}
```

Nom de l'analyse

Le nom d'une analyse permet en premier lieu de la référencer et si le nom est bien choisi, de la décrire brièvement. Il permet à l'architecte d'identifier l'analyse qu'il veut appliquer à une architecture. Il est aussi utilisé comme référence lorsque l'analyse est appliquée au sein d'une contrainte.
Plusieurs analyses peuvent porter le même nom. Dans ce cas, elles doivent définir des ensembles de paramètres d'entrée différents.

Paramètres d'entrée

Les paramètres d'entrée d'une analyse forment un ensemble d'identifiants typés. Ces paramètres sont utilisés dans la description du corps de l'analyse comme s'il s'agissait de valeurs définies. Les valeurs qui sont passées en paramètre représentent à la fois des éléments sur lesquels l'analyse porte et des éléments pour configurer (régler) l'analyse.

Dans notre exemple, l'analyse *reachable* a deux paramètres d'entrée : f1 et f2.

Type de la valeur retournée

Dans le cas où une analyse retourne un résultat sous forme de donnée, le type de cette donnée doit être spécifié. Ceci est nécessaire pour pouvoir appliquer une analyse depuis une contrainte ou depuis une autre analyse.

Dans notre exemple, l'analyse *reachable* retourne un booléen.

Corps

Le corps contient la description de l'analyse dans un langage associé au type de l'analyse.

Type de l'analyse

Les descriptions des analyses précédentes sont basées sur ARCHWARE AAL. Cependant, ASL est ouvert à d'autres types d'analyses. Ceci est nécessaire afin de donner la possibilité de faire évoluer le support analytique. Des traitements spécifiques à un domaine peuvent ainsi être introduits avec des langages de spécification adaptés.

La clause ***kind*** permet de spécifier le type d'analyse. Le nom de ce dernier référence[25] l'outil d'analyse capable de traiter la description contenue au niveau du corps.

L'exemple suivant est l'expression de l'analyse *hasCycle* en utilisant le langage ARMANI. Le corps est ainsi décrit dans le langage des prédicats d'ARMANI pour l'expression des analyses. Littéralement, cette analyse retourne la valeur s'il existe un composant *c1* dans l'architecture analysée tel que l'analyse *reachable(c1,c1)* retourne ***true*** .

```
hasCycle is analysis{
     kind ARMANI
     input {}
     output { Boolean }
     body{
          exists c1 : Component in sys.Components | reachable(c1,c1);
     }
}
```

2.4.3 Analyses et héritage

A l'héritage d'un style, le style hérite des analyses de son parent. Un sous-style peut définir ses propres analyses. Une analyse est conventionnellement définie pour pouvoir être applicable sur toutes les architectures satisfaisant le style associé. La pérennité d'une analyse est garantie par relation d'héritage ; un sous-style ne doit pas redéfinir les contraintes dont il a hérité. En supposant que les analyses d'un style sont applicables à chaque architecture suivant le style, elles sont applicables à chaque architecture de chaque sous-style.

Nous avons montré comment la description d'un style est structurée et comment on définit les constructeurs, les contraintes et les analyses. La prochaine section traite de l'utilisation du langage et des styles formalisés pour la conception d'architectures évolutives.

3. Usage du langage et des styles pour la description d'architectures dynamiques

Nous venons d'étudier les mécanismes et les concepts du langage que nous proposons pour la description des styles architecturaux. Nous avons remarqué qu'il fournit des concepts génériques quant à la description architectural compte tenu de ce que nous avons relevé dans l'étude du domaine (chapitre 2) et dans l'état de l'art (chapitre 3). De plus, nous n'avons pas montré explicitement comment ce langage permet de formaliser des styles pour les systèmes dynamiques. Les systèmes dynamiques sont généralement des systèmes à longue durée d'exécution ou des systèmes critiques. Leur évolution s'effectue en cours d'exécution. Ainsi, leur architecture change de manière dynamique.

[25] Cette référence est abstraite, le lien vers l'outil se fait ultérieurement, afin de donner plus de flexibilité dans l'évolution de l'environnement.

Dans cette section, nous montrons comment ASL répond aux exigences d'un langage de description d'architectures logicielles et de styles architecturaux pour les systèmes dynamiques.

Dans cette section, nous présentons d'abord une vue d'ensemble discutant de l'utilisation des styles dans un processus de développement centré architecture et pour la modélisation des systèmes dynamiques. Ensuite, nous donnons des méthodologies pour la définition d'un support pour la conception architecturale. Enfin, nous discutons de l'utilisation des styles pour la formalisation des familles et des lignes de produits.

3.1. Vue d'ensemble

Dans cette section, nous allons étudier l'utilisation des styles sur deux niveaux. Nous allons montrer qu'ASL fournit les outils pour utiliser les styles comme cadre et comme support à la conception architecturale. Puis, nous allons rappeler la nécessité de pouvoir définir des styles pour les systèmes dynamiques.

3.1.1 Un style = cadre et support à la conception architecturale

Les styles sont présents tout au long du cycle de développement centré architecture, depuis la spécification des besoins jusqu'à la maintenance d'une architecture (cf. Figure 23). Les styles offrent un *cadre* et un *support* à la conception architecturale, mais un style en particulier peut promouvoir un aspect plutôt que l'autre suivant sa définition.
Le schéma ci-dessous montre que les styles peuvent être utilisés pour formaliser les spécifications d'une famille (ou d'une ligne) de produits à partir d'un cahier des charges, pour apporter un support à la description avec des notations spécifiques, pour apporter un support analytique ou pour apporter un support à la conception avec des styles répondant à des problèmes spécifiques.

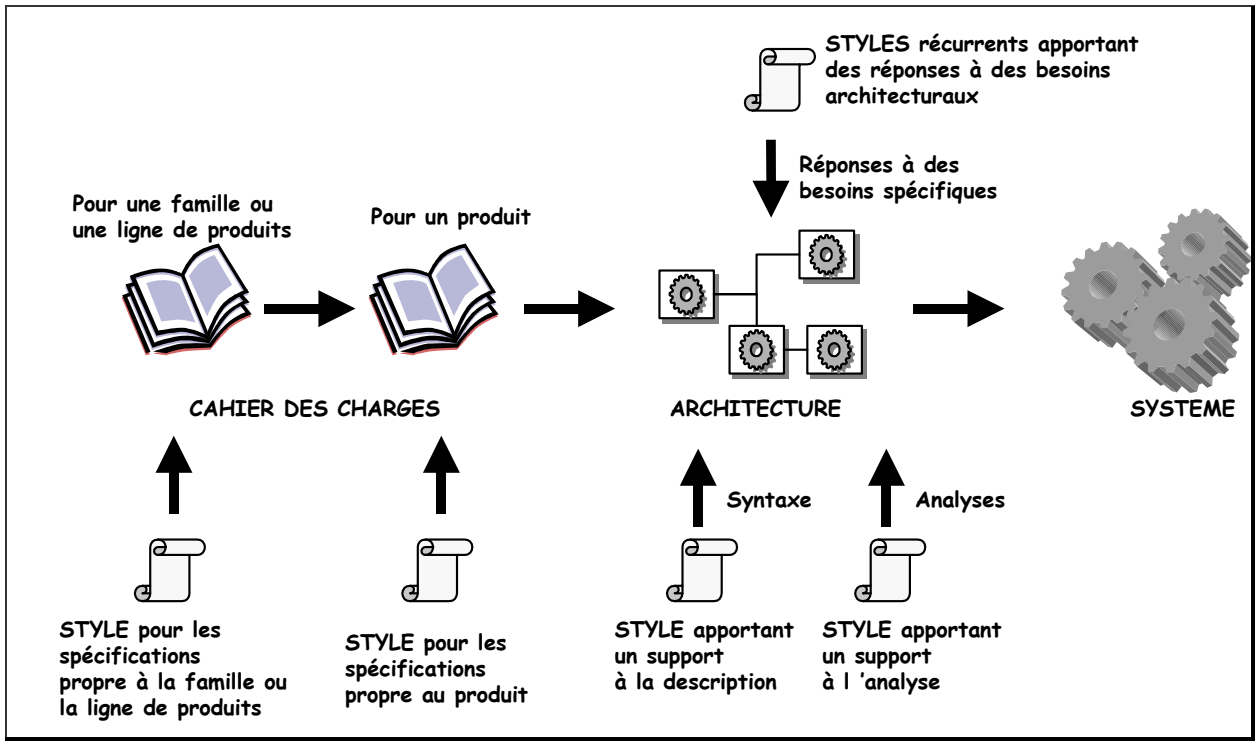


Figure 23 - Les styles dans un processus de développement centré architecture

Nous différencions trois sortes de styles.

> Les ***styles de domaines*** offrent un support à la conception d'architectures pour un domaine particulier.

Le support qu'ils fournissent est :

- un ensemble de concepts et de mécanismes,
- des éléments de construction spécifiques,
- un vocabulaire et une syntaxe adéquate à la description des architectures de ce domaine,
- des analyses.

> Les ***styles patrons*** offrent des solutions architecturales à des problèmes récurrents.

Ces styles promeuvent certaines propriétés architecturales appelées attributs qualité. Ainsi, le style Client-Serveur, Pipe-Filter, ou le style en couche font partie de ces styles qui ne sont pas propres à un domaine en soi.

> Les ***styles de spécification*** offrent un cadre à la conception d'un produit ou d'une famille de produits.

Ces styles représentent le cahier des charges propres à une famille (ou une ligne) de produits ou un produit. Un cahier des charges donne un ensemble de contraintes dont une partie est applicable au niveau architectural. Afin de garantir le respect de ces contraintes, nous pouvons les formaliser au sein d'un style. Nous étudierons comment formaliser ces styles dans une prochaine section consacrée à la représentation des familles et des lignes de produits.

3.1.2 Description de styles pour les systèmes dynamiques

Les styles peuvent cibler aussi bien des systèmes statiques que des systèmes dynamiques. La motivation de formaliser des styles pour des systèmes dynamiques réside dans la raison d'être des styles : capturer l'expérience et les connaissances en conception architecturale. Ces expériences et ces connaissances peuvent concerner l'aspect dynamique des architectures. Ne pas pouvoir prendre en compte les aspects dynamiques au niveau des styles signifie qu'on prive une classe de systèmes de la possibilité de tirer avantages d'une approche centrée style.

Les architectures dynamiques sont celles qui peuvent être modifiées après la construction du système. En définissant de telles architectures, l'architecte est capable d'exprimer des propriétés dynamiques telles que l'extensibilité, la customisabilité, et l'évolutivibilité des gros systèmes logiciels à un haut niveau d'abstraction [Med 96]. Les manières communes selon lesquelles une architecture peut être modifiée après avoir été construite sont les suivantes : [Med 96]

- l'addition ou la suppression de composants,
- la mise à jour d'un composant (par exemple, le réglage de la performance),
- la reconfiguration de la topologie (par exemple, la reconnexion des composants et des connecteurs).

Des solutions architecturales basées sur la dynamique ont été développées. Ces solutions entraînent des propriétés que les systèmes peuvent requérir telles que la performance, la modifiabilité ou la fiabilité. Considérons un système client-serveur. Moins il y a de clients connectés au serveur, meilleure est la performance du système. En adoptant une architecture statique, tous les clients sont connectés au serveur en dépit de la fréquence de leur besoin en services. Ceci peut induire une décroissance des performances du système. Afin de garder un nombre raisonnable de connexions, une solution est de connecter et déconnecter les clients dynamiquement en fonction de leurs besoins.

Les avantages principaux d'une approche centrée style en regard des architectures dynamiques sont les suivants.
D'abord, un style peut offrir une bibliothèque d'éléments incluant des mécanismes propres à la gestion de l'évolution dynamique. Par exemple, un mécanisme pourrait gérer la création dynamique des éléments architecturaux.
Puis, un style peut fournir des solutions architecturales reposant sur des propriétés liées à la dynamique. Une approche formelle permet d'exprimer des contraintes imposant une architecture à satisfaire les propriétés dynamiques insufflées par une solution architecturale.
Enfin, un style peut contraindre l'évolution d'une architecture. On sait qu'un style délimite, à travers les contraintes de style, un espace de définition d'architecture. Si une architecture suit un style

architectural, nous pourrions vouloir que tout en évoluant, elle demeure dans l'espace de définition qu'il définit. Par exemple, on peut vouloir qu'une architecture client-serveur évolue tout en satisfaisant le style Client-Serveur.

3.2. Définir un support à la conception architecturale

Dans cette section nous allons étudier comment utiliser ASL pour la description d'architectures, comment définir de nouvelles actions et comment définir des patrons architecturaux.

3.2.1 Définir une architecture avec ASL

ASL est un langage centré sur la description des styles. Néanmoins, il permet de décrire des architectures. Nous proposons de nous servir des styles comme langages de description d'architecture.

ASL permet de décrire une architecture au sein d'un style. Celle-ci est alors décrite dans un constructeur. En effet, nous avons vu qu'un constructeur peut encapsuler n'importe quelle valeur. Il peut ainsi encapsuler la définition d'une abstraction représentant une architecture. Dans ce cas le constructeur est un constructeur sans paramètre, comme nous pouvons le voir dans la description d'une architecture Client-Serveur ci-dessous (le style *ArchitectureCS* ne contient que la description d'une architecture).

```
ArchitectureCS is style where {
     constructors{
          ArchitectureCS is constructor();{
               abstraction() ;{
                    compose{
                         client is {…}
               and
                         server is {…}
                    }
               }
          }
     }
}
```

Jusque là il n'y a pas plus d'avantage à utiliser ASL qu'à utiliser ArchWare ADL pour la description d'une architecture. La description équivalente en ADL est la suivante.

```
abstraction() ;{
     compose{
          client is {…}
     and
          server is {…}
     }
}
```

L'avantage d'ASL est de pouvoir utiliser la syntaxe fournie par les constructeurs d'un style. L'exemple ci-dessous montre l'utilisation d'une syntaxe définie juste après dans un style *ClientServeur*.

```
ArchitectureCS is style extending ClientServeur where {
     constructors{
          ArchitectureCS is constructor();{
               clientServeur with{
                    c is client,
                    s is server
               }
          }
```

```
        }
}
```

L'expression suivante montre comment est implémentée la syntaxe.

```
ClientServeur is style where {
    styles{
        Client is style where{
            constructors {  Client is constructor();{…} as { client } }
        }
        Server is style where{
            constructors {  Server is constructor();{…} as { server } }
        }
    }
    constructors{
        ClientServeur is constructor(clientAlias:behaviour, client:Alias,
                                     serverAlias:behaviour, server:Alias);{
            compose{
                clientAlias is client
            and
                serverAlias is server
            }
        } as {
            clientServeur with '{
                $clientAlias is $client,
                $serverAlias is $server
            '}
        }
    }
}
```

3.2.2 Définir de nouvelles actions

Le langage définit des actions de base : la réception, l'émission et l'action inobservable. Il fournit aussi les mécanismes pour contraindre l'ordonnancement de ces actions dans le comportement d'une architecture. Cependant, ces actions sont élémentaires et ne permettent pas de refléter la richesse des concepts sous-jacents à un style. Par exemple, si on considère le style BlackBoard (tableau noir), un des mécanismes spécifiques aux architectures suivant ce style est le broadcasting ; c'est-à-dire l'envoi d'un message vers plusieurs destinataires. Dans le cas du tableau noir, un événement est envoyé à toutes les entités réceptrices lorsqu'un message est déposé dans le tableau.

Dans cette section, nous allons exposer une méthode pour définir de nouvelles actions. Notre objectif est de faciliter la description d'une architecture en utilisant des actions de haut niveau. De plus, nous voulons pouvoir décrire simplement des contraintes sur ces nouvelles actions. En effet, l'expression des contraintes étant relativement compliquée, la définition de nouvelles actions ne doit pas rajouter de la complexité.

Nous définissons une action comme l'appel à un service qui effectue un traitement correspondant à l'action. Dans ce contexte, un élément architectural est défini comme la composition d'un comportement et des différents services liés aux actions. Ces actions sont déclenchées depuis le comportement appelé *computation* (cf. Figure 24).

```
compose {
    computation
    and  service1
    …
    and  servicen
```

```
}
```

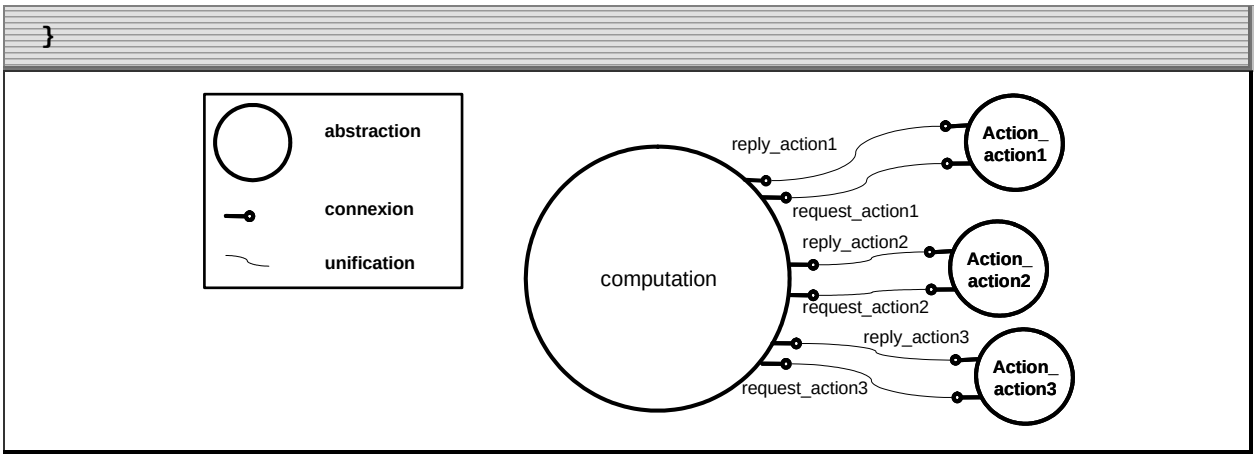


Figure 24 - Les actions comme services

Un service est défini avec une connexion d'entrée et une connexion de sortie. Il effectue un traitement en fonction de données reçues en entrée et peut retourner un résultat en sortie.

```
Action_nom_action is abstraction(){
     value request_nom_action is free connection(…);
     value reply_nom_action is free connection(…);
     via request_nom_action receive …
          traitement
     via reply_nom_action send …
}
```

Une action est déclenchée depuis le comportement *computation* par l'envoi d'un message via la connexion *request_nom_action*. La réception via la connexion *reply_nom_action* permet la synchronisation en attendant la fin de l'action.
La description suivante est un exemple concernant l'action de broadcasting. Cette action est déclenchée par la réception d'une liste de connexion. L'action consiste à effectuer des envois sur chacune des connexions de la liste.

```
Action_broadcasting is abstraction(){
     value request_broadcasting is free connection(sequence[connection[]]);
     value reply_broadcasting is free connection();
     via request_broadcasting receive connections :sequence[connection[]]
     iterate connections by c: connection[]
     do via c send;
     via reply_broadcasting send
}
```

Le déclenchement de cette action s'effectue de la manière suivante[26].

```
…
via request_broadcasting send sequence(c1, c2, c3);
via reply_broadcasting receive;
…
```

Finalement, du point de vue du comportement *computation*, le déclenchement d'une action est équivalent à celui d'une action de base. Ainsi, on a la possibilité de décrire simplement des contraintes comportementales adressant d'autres actions que les actions de base.

Par exemple, la propriété suivante exprime la possibilité d'un déclenchement de l'action $action_1$. Elle signifie que chaque séquence ne contenant pas un envoi via la connexion *request_action1* conduit à un non-respect de cette propriété.

[26] c1, c2, et c3 sont supposés être des connexions de type **connection[].**

```
every sequence {true*. not (via request_action1 send any)} leads {false}
```

Pour faciliter la description du comportement et pour rendre transparents les mécanismes d'appel de l'action, nous définissons un constructeur. Une notation mixfix propre à l'action est associée à ce constructeur.

```
broadcasting is constructor(connections:sequence[connection[]]);
{    via request_broadcasting send connections ;
     via reply_broadcasting receive
} as { broadcast $connections }
```

Le déclenchement de l'action peut alors s'effectuer de la manière suivante.

```
…
broadcast {c1, c2, c3};
…
```

De même pour simplifier la description des contraintes comportementales, nous avons étendu ARCHWARE AAL avec un raccourci d'écriture. La sémantique de l'expression suivante est celle qui la suit.

```
nom_action(paramètres)
```

Elle signifie l'envoi de paramètres via la connexion *request_nom_action* (le déclenchement de l'action), une suite d'actions quelconques, puis la réception via la connexion *reply_nom_action* (la terminaison de l'action).

```
via request_nom_action send parametres. true* .via reply_nom_action receive
```

Ainsi, on peut décrire une contrainte imposant la possibilité d'un déclenchement de l'action *broadcast* et de la réception de son résultat de la manière suivante.

```
every sequence {true*. not broadcast(any)}leads {false}
```

Nous venons de voir comment on peut définir une action avec ASL. La section suivante montre comment décrire des patrons structurels.

3.2.3 Comment définir des patrons ?

Un style définit généralement un ou plusieurs patrons.

> Un ***patron*** définit des règles sur l'organisation des éléments architecturaux.

Nous différencions un style d'un patron dans la mesure où un patron doit définir clairement les règles topologiques, c'est-à-dire comment les éléments architecturaux sont disposés les uns par rapport aux autres. Par exemple, le style Client-Serveur définit un patron imposant une topologie en étoile : plusieurs clients sont connectés autour d'un serveur. L'utilité de ces patrons et de promouvoir telle ou telle propriété architecturale. Ainsi, suivant les propriétés que l'architecte désire mettre en avant dans son architecture il choisit les patrons adéquats.

Description d'un patron

Un patron peut être défini par les contraintes du style. Cependant, il n'est pas aisé pour l'architecte de décrypter une liste de contraintes afin de suivre un patron ; il est plus simple de suivre un patron à partir d'un schéma.

Ainsi, pour décrire un patron, nous préférons utiliser un mécanisme de construction : le constructeur. Dans cette section, nous montrons comment décrire des patrons avec des constructeurs. Les constructeurs offrent un mécanisme de réutilisation adéquat pour des outils aussi récurrents que les patrons. Nous illustrons nos propos avec un patron de type étoile (cf. Figure 25).

Dans ce patron, tous les éléments sont liés à un élément central. Il fournit une structure de base aux styles Client-Serveur et BlackBoard (tableau noir).

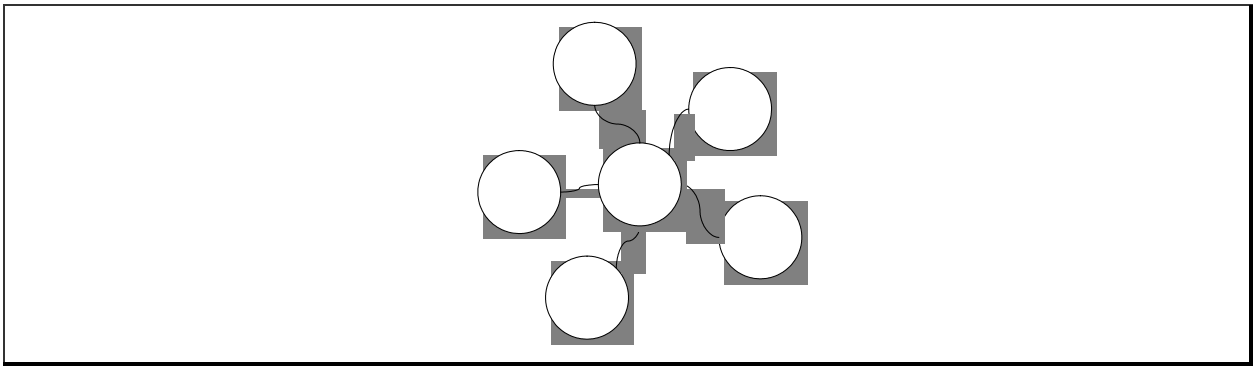


Figure 25 - Diagramme informel du patron étoile

Un constructeur pour un patron définit un traitement pour agencer un certain nombre d'éléments. Ainsi, ce type de constructeur ne génère pas une entité architecturale. Dans l'exemple suivant, le constructeur *etoile* à deux paramètres, un ensemble d'éléments périphériques et un élément central. A l'application de ce constructeur, chacun des éléments périphériques est composé avec l'élément central.

```
etoile is constructor(centre:Behaviour, elements:sequence[Behaviour]);
{    iterate elements by e: Behaviour
     do compose{
          centre
     and
          e
     }
} as { disposer $elements autour de $centre }
```

L'exemple suivant montre l'utilisation du patron étoile dans la définition d'une architecture.

```
abstraction();
{    value c is behaviour{…};
     value p is abstraction();{…};
     value p1 is p();     value p2 is p();  value p3 is p();
     disposer {p1, p2, p3} autour de c
}
```

L'avantage d'une telle description est de pouvoir composer des patrons et de pouvoir ainsi créer de nouveaux patrons à partir de patrons existant.

Composition de patrons

Comme le montre l'exemple d'utilisation des styles proposé par la documentation sur les ABAS [Kle&Kaz 99], the Sea Buoy System[27], l'architecture d'un système est généralement construite sur plusieurs patrons. Chaque patron apporte ponctuellement les propriétés requises. Chaque zone du système ne nécessite pas les mêmes propriétés, certaines zones critiques nécessitent d'être fiables, d'autres nécessitent d'être modifiables.

Ainsi, il faut pouvoir composer les patrons au sein d'une même architecture. Cette composition peut mener à la création de patrons plus complexes comme le montre l'exemple ci-dessous. *NouvelleEtoile* est un patron défini à partir des patrons *Etoile* et *Ligne* (cf. Figure 26).

[27] Ce système relève des données météorologiques et des données de navigation transmises par des bouées pour la gestion des trafics maritimes et aériens. L'architecture de ce système se base sur plusieurs patrons selon les attributs qualités que le système doit mettre en avant.

```
NouvelleEtoile is constructor(
      elements : sequence[sequence[Behaviour]],
      centre:Behaviour);{
      iterate elements by s: sequence[Behaviour]
      do { Etoile(centre, s);
      Ligne(s)
      }
}
```

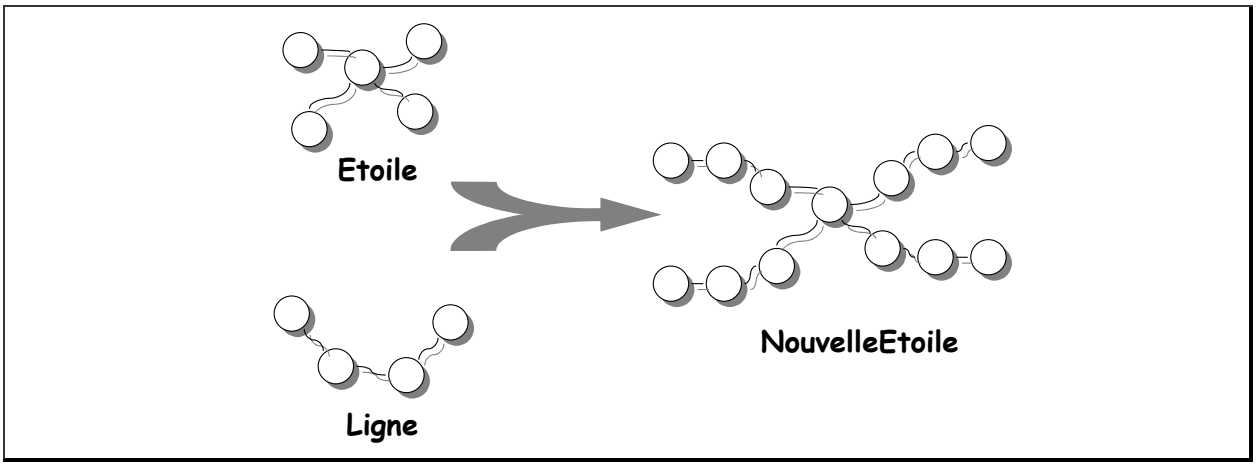


Figure 26 - Composition de patrons

3.3. Formalisation des familles et des lignes de produits

Pour des raisons de réutilisation et d'uniformité, les développements s'orientent vers le développement de *lignes de produits* ou de *familles de produits*.

> Une ***ligne de produits*** est un ensemble de produits différents de par leur utilisation mais qui partagent des composantes en communs [DiN&Fug 96].

Par exemple, la suite office de Microsoft représente une ligne de produit : Word, Excel, et PowerPoint sont des logiciels à vocation différente mais partageant un ensemble de composants en commun (comme la correction orthographique).

> Une ***famille de produits*** est un ensemble de produits qui ont la même utilité mais qui présentent des variations [DiN&Fug 96].

Par exemple, WindowsXP représente une famille de produits avec la version Pro et la version Familiale.

Les familles et les lignes de produits sont généralement [Har 01] modélisées sur deux axes : les aspects communs et les aspects variants. Nous rajoutons un troisième axe, les aspects optionnels. Ainsi une famille (ou une ligne) de produits est représentée par ses *invariants*, ses *options* et ses *variants*.

> Les ***invariants*** sont les spécifications communes à tous les produits.

> Les ***options*** sont les spécifications qui peuvent être présentes ou non dans un produit. Il peut s'agir par exemple d'un ensemble de composants qui peuvent être ou non utilisés dans le produit.

> Les ***variants*** représentent les parties où les produits doivent différer.

En transposant au niveau architectural, si un produit est représenté par une architecture, une famille ou une ligne de produits peut être représentée par un style architectural. Cette transposition a déjà été proposée dans des études faites par [Lal 99].

Dans cette section, nous présentons comment formaliser une famille ou une ligne de produits en ASL à travers la formalisation des styles.

Supposons d'abord que nous avons un constructeur de base permettant de générer une architecture à partir d'une liste d'éléments et une liste de liens reliant ces éléments. Examinons comment nous pouvons définir les invariants, les options et les variants à partir d'un tel constructeur.

Invariants

En termes d'architecture, les invariants correspondent :

- à une architecture minimale (commune à tous les produits de la famille) que l'architecte complète pour obtenir l'architecture d'un produit spécifique,
- à des contraintes de construction commune.

Dans le cadre que nous nous sommes donné, l'application partielle peut-être utilisée afin de compléter une description architecturale. Nous avons vu qu'un constructeur permet de définir une architecture. Supposons un constructeur, appelé constructeur de base, représentant une architecture 'vide'. Il est possible de définir un constructeur représentant la partie invariante (une architecture minimale) en fournissant les éléments architecturaux et les liens qui en font partie. Il est possible d'obtenir l'architecture d'un produit spécifique par une nouvelle application partielle en donnant en paramètres les éléments spécifiques à ce produit.

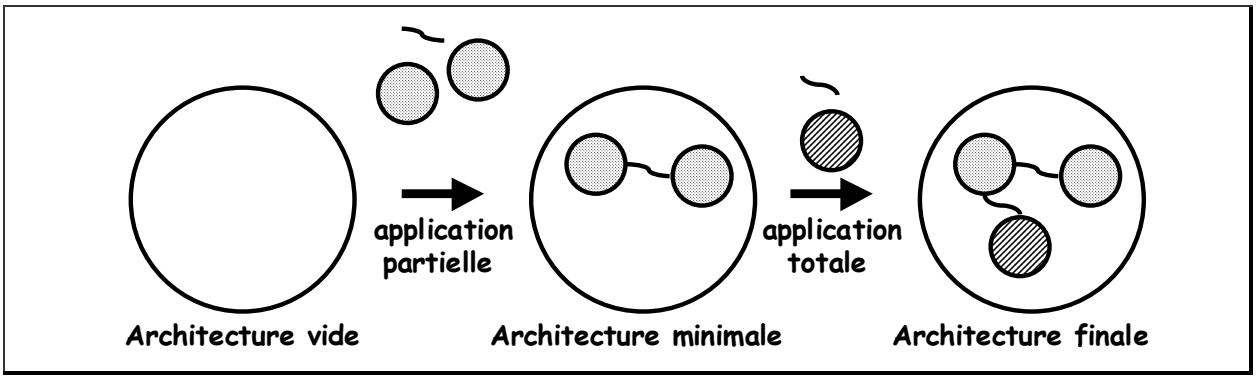


Figure 27 – De l'architecture vide à l'architecture finale en passant par l'architecture minimale

Ainsi, à travers un constructeur, un style peut fournir un noyau architectural que l'architecte peut réutiliser et compléter.

Les contraintes de style offrent un support supplémentaire à la description des invariants. Elles permettent notamment de donner un cadre plus générique à la conception du produit. Par exemple, une contrainte peut seulement stipuler l'absence d'interblocage.

L'avantage des constructeurs et d'offrir une base concrète au développement de produits spécifiques, tandis que les contraintes ont meilleure utilité à exprimer des propriétés génériques.

Options

La partie optionnelle peut être représentée par un constructeur englobant celui des invariants. Chaque paramètre de ce premier constructeur représente une option. A partir de chacun de ces paramètres, des éléments architecturaux et des liens sont créés et donnés en paramètres d'application au constructeur des invariants.

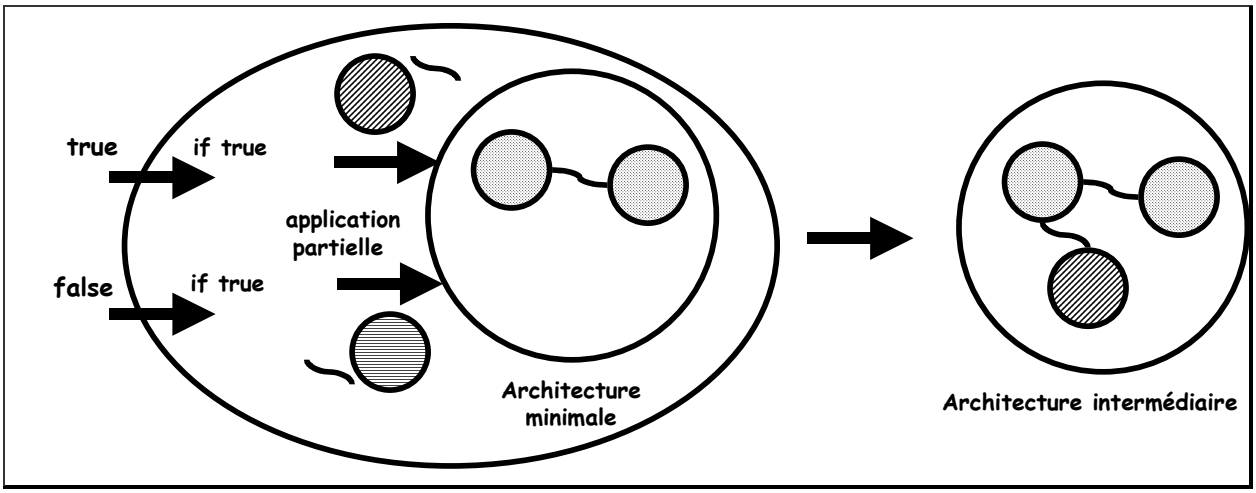


Figure 28 – Les options dans les styles pour les familles et lignes de produits

Variants

Exprimer les variants signifie de donner des informations concernant les localisations où des adaptations sont nécessaires, et si possible, de fournir des conseils pour effectuer cette adaptation. Une manière de spécifier les variants est de définir des éléments architecturaux vides de toute spécification [Lal 99].

Les variants peuvent être exprimés par des contraintes. Celles-ci permettent d'exprimer ce que nous venons d'évoquer : la présence d'un élément, sa localisation, et des conseils d'implémentation. Par exemple, l'expression suivante est une contrainte :

- signifiant l'existence d'un élément tel que sa localisation est représentée par le fait d'être connecté à l'élément *core*,
- guidant l'architecte en signifiant que l'élément ne doit pas pouvoir bloquer.

```
variant is constraint(
     to styleInstance.constituents apply
          exists{element |
               --propriétés de localisation
               element connected to styleInstance.constituent("core") and
               --propriétés guidant l'architecte
               no-deadlock(element)
          }
}
```

4. Conclusion

Dans ce chapitre, nous avons présenté ASL, un langage pour la description des styles architecturaux. C'est un langage qui à une place de front-end dans un environnement de conception orienté architecture. En effet, les styles architecturaux concentrent l'expérience et les connaissances de conception architecturale en général ou pour des domaines spécifiques. Ainsi, les styles décrits en ASL offre un cadre à la conception architecturale en permettant notamment la formalisation de famille ou de ligne de produit. De plus, ASL permet la définition d'un support à la conception architecturale à travers les styles. Nous avons montré comment ce langage permet de répondre aux attentes d'utilisation d'un style tout au long du processus de conception.

Ce langage présente de nombreux mécanismes :

- pour l'organisation des styles à travers l'héritage et l'agrégation,
- pour l'expression d'un support à la description architecturale à travers les constructeurs et la notation mixfix,

- pour l’expression de contraintes à la fois structurelles et comportementales,
- pour l’expression d’analyses à la fois dans les langages ArchWare mais aussi dans des langages externes.

ASL se distingue des autres langages par les aspects suivants :

- les aspects comportementaux,
  - ASL supporte aussi bien les aspects structurels et non-fonctionnels que les aspects comportementaux. Il permet la description de comportement en s'appuyant sur ARCHWARE ADL. Il permet la définition de contraintes et d'analyses sur les comportements en s'appuyant sur ARCHWARE AAL.
- les aspects dynamiques,
  - ASL supporte non seulement le comportement en terme d'ensemble d'interactions au sein d'un élément architectural mais aussi en terme de changement topologique à l'exécution. Cet aspect comportemental est appelé la dynamique et ASL permet d'y associer quelques aspects propres à la mobilité.
- la puissance d'expression,
  - ASL propose un large ensemble de concepts architecturaux permettant ainsi de s'adresser à un très large ensemble de systèmes. De plus, ces concepts sont peu contraignants et permettent à l'architecte d'adapter ses propres règles de construction.
- la flexibilité.
  - ASL permet, à travers les mécanismes de notation qu'il fournit, de définir de nouveaux formalismes adaptés aux besoins de l'architecte et du domaine des systèmes que ce dernier modélise.

Nous avons évoqué à plusieurs reprises les capacités du langage concernant la description de styles pour les systèmes dynamiques. Maintenant, l’objectif est de définir des styles avec ASL. Ainsi, dans les prochains chapitres, nous allons développer :

- un style basé sur le modèle composant-connecteur et offrant des mécanismes pour gérer la dynamique,
- un style courants tels que Client-Serveur et Pipe-Filter,
- des styles répondant à des cas industriels.

Chapitre 5

LE STYLE COMPOSANT-CONNECTEUR

Chapitre 5 - Le style Composant-Connecteur

1. Introduction

Dans le chapitre 4, nous avons proposé un langage pour la description des styles architecturaux. Les styles définis avec ce langage offrent un cadre et un support à la conception architecturale. En effet, ils délimitent un espace de définition par un ensemble de contraintes et fournissent un support à la description et à l'analyse des architectures appartenant à cet espace de définition. La particularité de ce langage est de supporter la description des architectures et des styles pour les systèmes dynamiques.

Le langage que nous proposons est basé sur des concepts génériques de bas niveau (abstractions, comportements et connexions) pour la description des architectures. Ces concepts paraissent éloignés des concepts architecturaux traditionnels. En effet, il y a un consensus au sein de la communauté (cf. chapitre 2) pour identifier les concepts des architectures Composant-Connecteur comme la base dans la conception des architectures logicielles. Nous allons voir comment le langage ASL supporte la description des architectures Composant-Connecteur. Ainsi, en définissant un style Composant-Connecteur en ASL, nous obtenons un langage de description architecturale aux concepts plus 'classiques'.

Nous proposons une démarche de conception des logiciels qui s'appuie sur des bases formelles et qui permet à l'utilisateur ses propres concepts de bases dans la conception architecturale. Dans le processus de conception, le styliste définit des styles (cf. Figure 29). Il a la possibilité de définir un style proposant un formalisme définit en termes d'ASL et reflétant les concepts architecturaux sur lesquels il s'appuie ; c'est le cas du style Composant-Connecteur que nous définissons. En d'autres termes, le styliste à la possibilité de définir son propre ADL basé sur ASL. Ensuite, il peut définir des styles plus spécifiques propres au produit ou à la famille de produits à développer. L'architecte se base sur ces derniers styles pour définir les architectures.

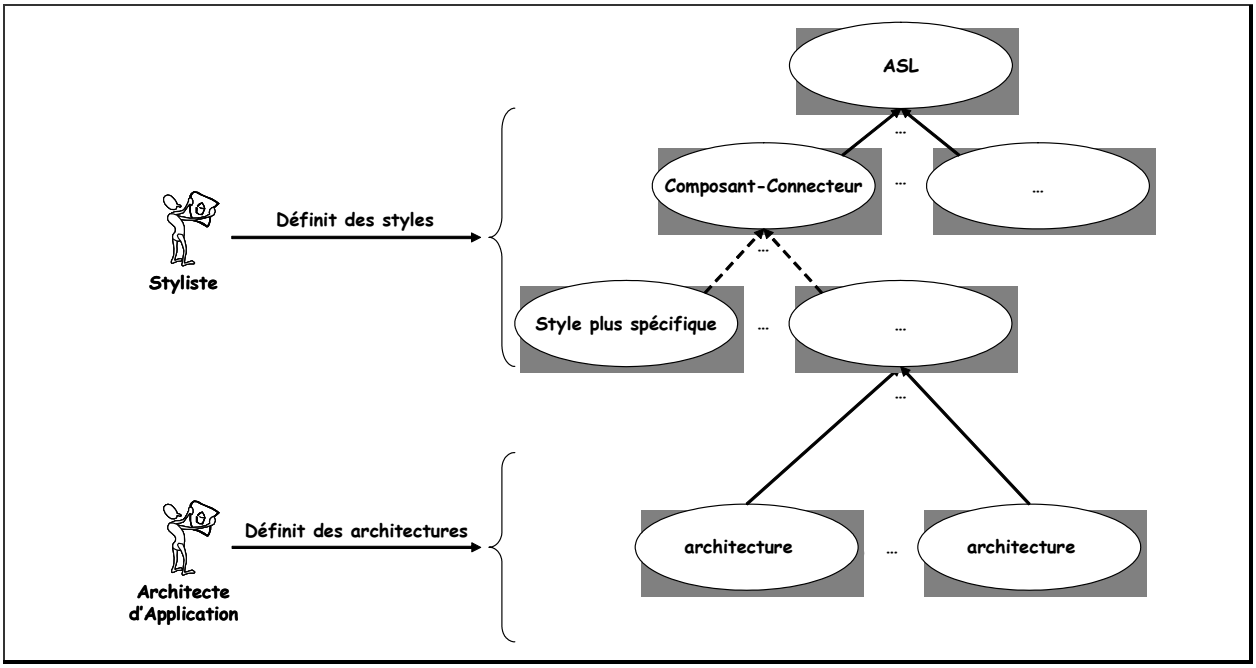


Figure 29 - Rôles du styliste et de l'architecte

Ce chapitre est dédié à la validation du langage ASL à travers la définition de styles récurrents et en particulier le style Composant-Connecteur. Ce dernier est considéré comme le style de base. Il est défini pour fournir des mécanismes de haut niveau (comparé à ASL) pour la conception des styles et des architectures composant-connecteur. Le style Composant-Connecteur supporte les concepts de dynamique et fournit un outil flexible à l'utilisateur pour définir de nouveaux concepts. Nous

montrons ainsi, que le langage permet de supporter et de fournir les concepts sous-jacents à ces architectures. Parmi ces concepts, il y a ceux associés à la dynamique des architectures.

Nous structurons ce chapitre de la manière suivante. Dans un premier temps nous présentons une vue d'ensemble des concepts sous-jacents au style Composant-Connecteur. Dans un deuxième temps, nous donnons plus de détails sur les mécanismes structuraux et comportementaux en apportant des explications sur l'implémentation. Dans un troisième temps, nous montrons l'utilisation du style Composant-Connecteur pour la description d'une architecture et pour la description de styles récurrents que nous appelons styles de fondations.

2. Vue d'ensemble

Nous l'avons dit, les concepts de composants et de connecteurs sont considérés comme la base de la conception architecturale. Il est généralement admis que les architectures sont faites de *composants* (des éléments pour le calcul) et de *connecteurs* (des éléments pour transiter l'information entre des composants).

Les concepts de composants et de connecteurs sont très génériques et on peut trouver plusieurs définitions comme nous avons pu le noter dans l'étude du domaine (chapitre 2) et dans l'état de l'art (chapitre 3). Dans ce chapitre, nous proposons les concepts relatifs aux architectures composant-connecteur et notamment en ce qui concerne la dynamique. Cette vue d'ensemble présente brièvement les concepts architecturaux principaux définis par le style Composant-Connecteur.

2.1. Composants et connecteurs

Une architecture est un ensemble d'éléments architecturaux connectés (cf. Figure 30). Ces éléments sont des *boîtes noires* ; ils sont définis indépendamment de leur l'environnement et leur comportement interne n'est pas visible depuis l'extérieur. Pour communiquer entre eux, ils fournissent des services accessibles par une interface. Les services qu'ils proposent définissent leur rôle dans l'architecture. Il y a un consensus dans la communauté pour différencier deux types d'éléments suivant leur rôle : les composants et les connecteurs. Les ***composants*** sont des unités de traitement ou de stockage de données ; ils représentent les fonctionnalités du système. Un exemple simple de composant est un composant qui additionne deux entiers.
Les ***connecteurs*** sont des éléments communication d'une architecture ; ils transitent des informations entre les composants. Des exemples de connecteurs courants sont l'appel procédural, les requêtes de données ou l'envoi d'évènement. Nous considérons qu'un connecteur peut effectuer un traitement sur les données qui transitent tant que celui-ci ne modifie pas la sémantique de l'information. Ainsi, un connecteur peut changer le format d'une information afin qu'elle puisse être traitée par un composant. Dans une architecture, les composants ne communiquent pas directement, l'information doit passer par un ou plusieurs connecteurs (cf. Figure 30 – les connecteurs y sont plus petits que les composants).

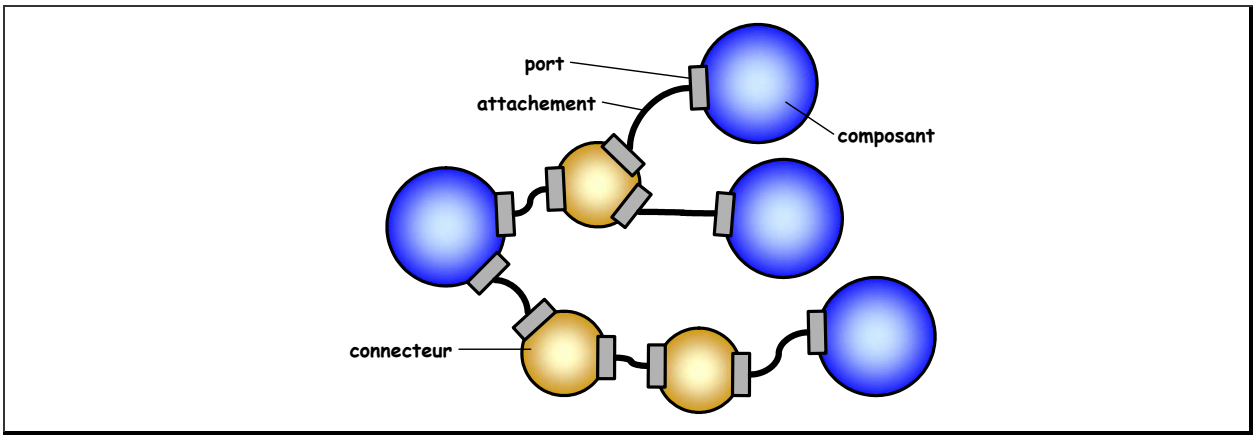


Figure 30 - Diagramme d'une architecture Composant-Connecteur

Les concepts de composants et de connecteurs sont supportés par la plupart des ADLs étudiés dans l'état de l'art. Cependant, certains se passent de la notion de connecteur tels que DARWIN [Mag et al. 95] et RAPIDE [RDT 97]. Nous pouvons aussi noter que le langage AML [wil 99] ne supporte pas ces concepts mais qu'il permet de les formaliser tout comme ASL.

Ces éléments architecturaux sont structurés (cf. Figure 31) par un ensemble de ***ports***, un ***noyau*** et un ensemble d'***attributs***. Les ports constituent l'interface d'un élément ; ils sont constitués de ***connexions*** qui sont des points d'interaction permettant la communication entre les éléments. Le noyau représente le mécanisme interne de l'élément. Les attributs sont des informations pouvant refléter les propriétés du système.

La décomposition de l'interface des éléments architecturaux (les ports dans notre cas) ne sont subdivisés en entités plus petites (les connexions) que dans très peu de langage dont π-SPACE [Cha 02], WRIGHT [All 97] et DARWIN [Mag et al. 95]. Cette subdivision permet de ne pas lier les ports complètement. Concernant les attributs, un mécanisme similaire est adopté par ACME [GarMon&Wil 00] et UNICON [Sha et al. 95] sous le nom de *propriétés*.

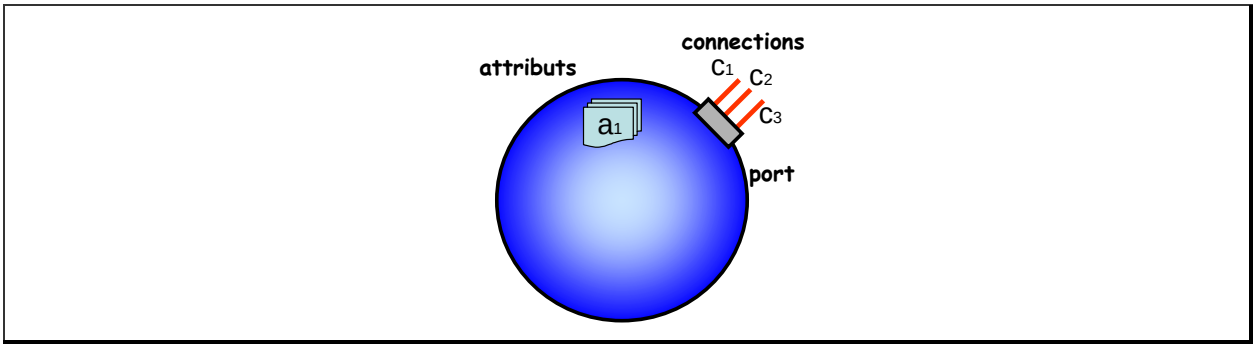


Figure 31 - Structure d'un élément architectural

2.2. Eléments Atomiques et Composites

Le noyau d'un élément architectural peut être défini de deux manières (cf. Figure 32), soit par un comportement, soit par la composition[28] d'autres éléments architecturaux. Dans le premier cas, on dit que l'élément est ***atomique***, dans le second cas, on dit qu'il est ***composite***. Ainsi, une architecture est un élément composite qui ne possède pas d'interface.

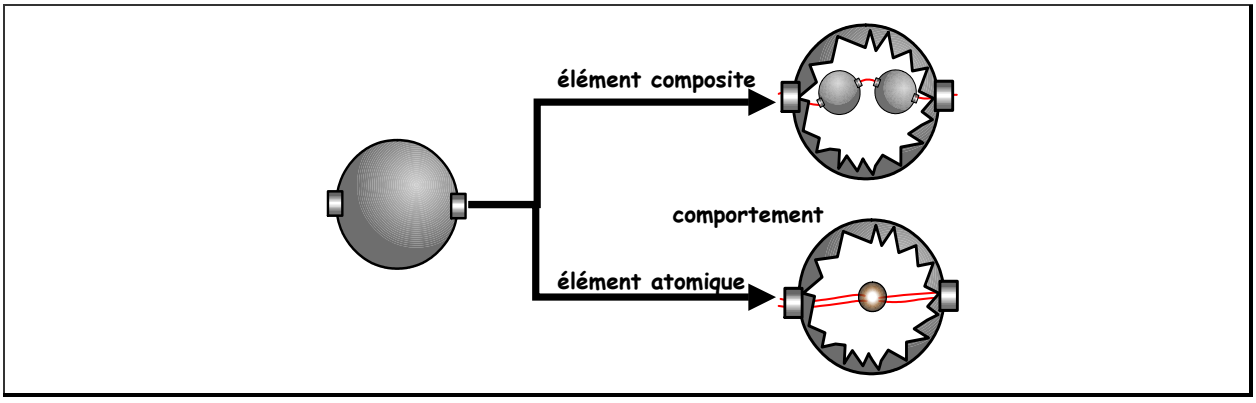


Figure 32 - Elément atomique et élément composite

Dans le contexte d'un *élément atomique*, un comportement a la même sémantique qu'un comportement en ASL : un comportement est une séquence d'actions effectuées au sein de

[28] Nous préférons utiliser le terme 'composition' qui fait partie du vocabulaire liée à ArchWare ADL, mais le terme 'configuration' est souvent utilisé dans le domaine des architectures logicielles.

l'élément. Nous verrons que le style Composant-Connecteur fournit des actions autres que les actions de base d'ASL, notamment des actions relatives à la dynamique.

Dans le cas d'un *élément composite*, ses constituants sont liés par leurs interfaces à la fois entre eux et avec l'interface du composite. Comme dans plusieurs ADLs (ACME [GarMon&Wil 00], AESOP [GarAll&Ock 94],…) nous différencions deux sortes de liens. Les premiers sont les **attachements** ; ils permettent la communication entre deux modules. Les seconds sont des **passerelles** (nous utilisons ***bindings*** en anglais) ; ils permettent de faire correspondre l'interface d'un constituant à l'interface du composite.

La composition est un concept adopté par la plupart des langages de description d'architecture. Le terme *composite* utilisé aussi par π-SPACE [Cha 02] et UNICON [DeL 96] n'est pas toujours le terme adopté. Par exemple, on parle de *représentation* avec AESOP [GarAll&Ock 94] et ACME [GarMon&Wil 96] ou encore de *macro* avec Meta-H [Bin et al. 96].

2.3. Dynamique

Le style Composant-Connecteur supporte la conception des architectures évolutives. Ainsi, il définit des concepts associés à la gestion de la dynamique.

> La **dynamique** d'une architecture consiste en sa capacité à pouvoir modifier sa topologie à l'exécution. Il peut s'agir de la création et de la suppression dynamique d'éléments ou de leur reconfiguration dynamique [Med 96].

Parmi ces concepts, il y a les actions dynamiques, le chorégraphe, les méta-éléments et la mobilité.

La dynamique est un aspect pris en compte par quelques uns des ADLs étudiés : ACME [GarMon&Wil 00], WRIGHT [All 97], π-SPACE [Cha 02], DARWIN [Mag et al. 95] et RAPIDE [RDT 97]. Parmi ceux-ci, π-SPACE est le plus abouti concernant cet aspect : il permet de programmer la dynamique à la description de l'architecture et pose peu de contraintes à cette dynamique. Nous nous sommes basés sur des concepts de π-SPACE pour définir nos mécanismes en améliorant les lacunes que nous avons pu relever dans π-SPACE.

2.3.1 Les actions dynamiques

Le style Composant-Connecteur fournit un certain nombre d'actions pour la dynamique. Ces actions concernent la création dynamique (des connexions, des ports et des éléments architecturaux) et la reconfiguration dynamique (attachement et détachement dynamique des liens, intégration et exclusion dynamique des éléments architecturaux).

Au sein d'un élément atomique, la dynamique concerne les ports et les connexions. Des actions spécifiques peuvent être effectuées au sein de l'élément. Nous étudierons ces actions dans la section 4 consacrée à la description des comportements.

2.3.2 Le chorégraphe

Nous avons mentionné que la dynamique est gérée par des actions spécifiques déclenchées au sein de comportements. Cependant, les éléments composites ne sont pas définis par un comportement mais par un ensemble de constituants. Les composants et les connecteurs sont des boîtes noires. Ainsi, nous introduisons la notion de ***chorégraphe*** pour gérer la dynamique au sein des composites.

> Le **chorégraphe** est un élément qui gère la dynamique au sein d'un composite. Il permet la création et la suppression dynamique, et les reconfigurations topologiques.

Nous étudierons en détail le chorégraphe dans la section 3.11.

2.3.3 Les méta-entités

Les architectures décrites à partir du style C&C sont constituées d'entités toutes potentiellement dynamiques (connexions, ports, composants et connecteurs). Cela signifie que n'importe quelle entité peut être dupliquée plusieurs fois. En d'autres termes, la définition d'une entité peut servir à la création dynamique de plusieurs occurrences.
Pour gérer cela, nous introduisons le concept de ***méta-entité***[29].

> Une ***méta-entité*** est une matrice contenant la définition d'une entité. Elle contient les informations permettant la création et la suppression (dynamique ou non) de plusieurs occurrences, et la gestion de ces occurrences.

Nous verrons plus tard que certaines actions permettent de gérer ensemble les occurrences issues d'une même méta-entité.

La structure de l'implémentation d'une méta-entité est présentée à travers la définition de type suivante.

```
type MetaEntité is view   [
          name:String,
          producer:abstraction[],
          entity_list:sequence[Element],
          last_created:Integer
          ]
```

Une méta-entité est définie par :

- un nom qui permet de la référencer,
- une unité productrice permettant de créer de nouvelles occurrences,
- une liste d'entités contenant les occurrences créées à partir de cette méta-entité,
- la référence, en terme de place dans la liste, de la dernière occurrence créée.

Concernant la définition d'une méta-entité, la syntaxe a la structure qui suit.

```
Nom_meta_entite is entite_definition
```

La syntaxe utilisée pour cette description dépend de la nature de l'entité. Nous étudierons plus en détail l'aspect descriptif d'une méta-entité pour les différents types d'entités, plus loin dans le chapitre.

2.3.4 La mobilité

> La **mobilité** consiste en la capacité d'envoi et de réception d'éléments architecturaux, de ports, ou de connexions.

Nous considérons la mobilité comme un aspect dynamique[30] car lorsqu'un élément transite d'un endroit à un autre, la topologie de l'architecture du système évolue. La mobilité est intrinsèque au langage ASL : une connexion peut faire transiter n'importe quel type de valeur dont les valeurs représentant des entités architecturales.

Dans les sections suivantes, nous étudions en détails les concepts du style Composant-Connecteur en donnant la syntaxe propre au style et les parties clefs de l'implémentation. Nous étudions d'abord les concepts structuraux, puis les concepts comportementaux.

---

[29] Afin d'être plus précis dans notre discours, nous parlerons de méta-connexions, de méta-ports et de méta-éléments.

[30] Ainsi, nous n'étudions pas la mobilité en profondeur, mais nous donnons quelques mécanismes pour la mobilité.

3. Concepts structuraux

La section précédente présente globalement les concepts sous-jacents au style Composant-Connecteur. Cette section, ainsi que la section qui suivra, donne plus de détail quant à la syntaxe et l'implémentation de ces concepts.

Dans ce qui suit, nous discutons des concepts structuraux, c'est à dire des différentes entités définissables et de leurs relations dans une architecture Composant-Connecteur. Nous structurons leur présentation par leur définition, la syntaxe qui leur est associée et la manière dont ils sont décrits en ASL. Mais, avant tout, nous proposons d'étudier notre approche pour la description d'une entité structurelle.

3.1. Approche pour la description des entités structurelles

La vue d'ensemble a montré que la structure compositionnelle des entités est prédominante dans les architectures Composant-Connecteur : les éléments composites sont composés d'autres éléments architecturaux, les éléments architecturaux sont composés de ports, et les ports sont composés de connexions.
Chacune des entités est potentiellement dynamique. Ainsi, lorsqu'on donne la définition d'une entité, on précise la nature des entités qu'elle peut contenir en lui donnant une liste de méta-entités. Afin de définir la topologie d'une architecture à l'état initial, les occurrences existantes et leurs relations sont définies dans une clause spécifique appelée ***configuration***.

La syntaxe suivante illustre le schéma que nous venons de décrire.

```
entity with {
    compound_entities {  liste de méta-entités    }
    configuration {      description de la configuration }
    autres spécifications
}
```

Nous retrouverons ce schéma adapté aux différents types d'entités dans les sections suivantes.

De plus, dorénavant lorsque nous illustrerons nos propos par des diagrammes d'architectures nous utiliserons une notation graphique [All&Oqu 03] basée sur UML. La figure suivante en donne la légende.

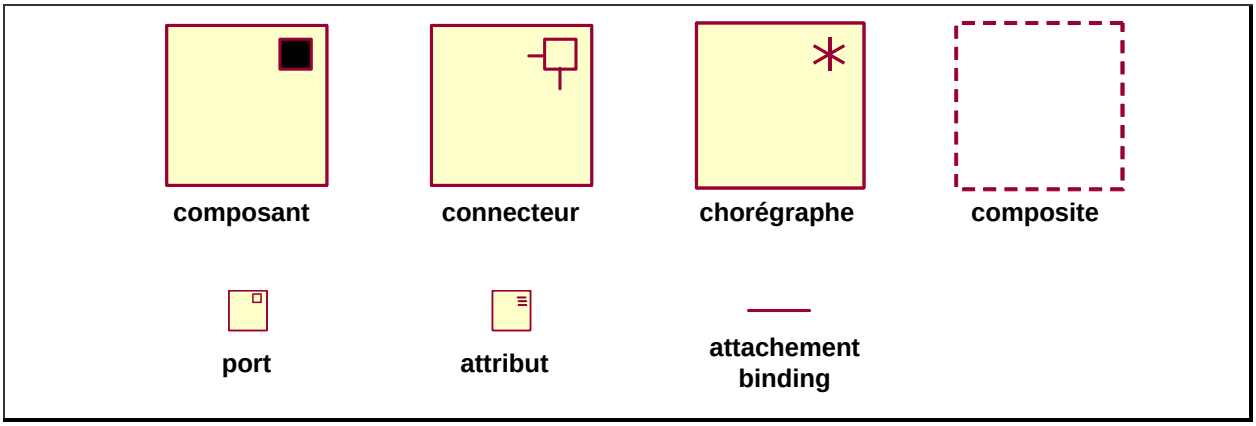


Figure 33 - Légende pour les diagrammes d'architectures

3.2. Connexion

3.2.1 Définition

> Une ***connexion*** est un point d'interaction atomique. C'est par les connexions que transitent les messages et que sont liés les éléments architecturaux.

3.2.2 Syntaxe

Une connexion est définie de la même manière qu'en ASL. On y précise le type de l'information qui y transite.

```
connection(Type)
```

3.2.3 Description

Un type est défini pour désigner une connexion et son nom. Le nom est de type ***String*** tandis que la connexion est de type ***Any***. Le type ***Any***[31] est choisi car, comme nous l'avons vu dans le chapitre précédent, une connexion est typée par rapport au type de message qu'elle transite.

```
type Connection is view{
     name : String,
     instance : Any
}
```

3.3. Méta-Connexion

3.3.1 Définition

> Une ***méta-connexion*** contient la définition d'une connexion et les informations permettant la création dynamique de plusieurs occurrences et leur gestion.

3.3.2 Syntaxe

Une méta-connexion est définie avec un *nom*, qui permet de la référencer et de référencer les occurrences.
La structure de la syntaxe pour déclarer une méta-connexion est la suivante (*connection_name* doit être un identifiant et *connection_definition* doit être l'expression d'une connexion).

```
connection_name is connection_definition
```

3.3.3 Description

Ceci est implémenté avec un constructeur permettant de générer des méta-connexions à partir d'un nom et de la définition d'une connexion. Une méta-connexion est implémentée comme une vue constituée du nom de la méta-connexion, de la définition d'une connexion et d'une liste de connexions (cette liste est vide à la création de la méta-connexion).

```
meta_connection is constructor(
     _name : Alias,
     _connectionExpression : Expression[connection[AnyType]]
);{
view {
     name is toString(_name),
     connection_definition is _connectionExpression,
     instances is nilsequence(Connection)
}
}as{
     $_name is $_connectionExpression
}
```

Le type correspondant à l'implémentation d'une méta-connexion est le suivant.

```
type Méta-Connection is view{
```

[31] Le type Any représente des valeurs encapsulant des valeurs de n'importe quel autre type [Cim et al. 02].

```
      name : String,
      connection_definition : Expression[connection[AnyType]],
      instances is sequence(Connection)
}
```

3.4. Ports

3.4.1 Définition

> Un ***port*** est un point d'interaction pour un élément architectural. Il structure l'interface de l'élément en groupant des connexions. De plus, il spécifie comment l'élément interagit avec l'extérieur via un protocole d'interaction.

Il donne accès à un service en groupant les connexions permettant de transiter l'information depuis et vers ce service. Les ports permettent ainsi d'avoir une interface structurée et donc plus lisible.

Un port définit aussi un protocole de manière analogue au domaine matériel[32]. Ce dernier définit l'ordonnancement et le sens des messages qui passent par le port.

3.4.2 Syntaxe

La syntaxe utilisée pour la définition d'un port est structurée de la manière suivante.

```
port with {
      connections {   liste de connexions     }
      configuration {      description de la configuration }
      protocol {      définition du protocol  }
}
```

Nous illustrons notre discours avec l'exemple de définition de port suivant. Il s'agit de la description d'un port pour un client dans une architecture client-serveur ; le port permet la communication avec un serveur. Il est expliqué dans les prochains paragraphes.

```
port with {
      connections {
            call is connection(Any),
            wait is connection(Any)  }
      configuration {
            new call;  new wait   }
      protocol  {
            replicate
                  via call send    ;
                  via wait receive }
}
```

Connexions

Dans la partie *connexions*, on déclare les méta-connexions du port. Autrement dit, on spécifie les sortes de connexions que le port peut contenir.

Configuration

Dans la description d'un port, la partie configuration décrit la création des connexions à l'initialisation du port. Dans l'exemple, on instancie une occurrence de *call* (`new call`) et une occurrence de *wait* (`new wait`). Des détails sur l'action ***new*** seront présentés plus tard (cf. 4.3).

[32] Par exemple, un port USB est associé à un protocol USB [USB 00]

```
configuration {      new call;  new wait      }
```

Protocole

> Le ***protocole*** décrit le schéma des interactions possibles via le port en définissant l'ordonnancement des actions de communication.

Le protocole est une projection[33] du comportement (comportement équivalent dans le cas d'un élément composite) sur un ensemble de connexions défini. Ainsi, il permet de vérifier que la communication entre deux composants est possible en vérifiant la compatibilité de leurs ports sensés être attachés. Le protocole permet ainsi d'éviter l'interblocage en ne permettant pas de lier des ports dont les schémas d'interactions sont incompatibles.

Ainsi dans l'exemple, on spécifie qu'après une réception via une connexion call suit un envoi via une connexion wait.

```
protocol  {
     replicate
          via call send    ;
          via wait receive }
```

3.4.3 Description

Concernant l'implémentation, un type *Port* est défini. Ainsi, un port est défini comme une vue constituée d'une liste de méta-connexions et d'un protocole.

```
type Port is view{
     name:String,
     meta_connexions:sequence[MetaConnection],
     protocol:Expression[Behaviour]
}
```

Via un constructeur, nous implémentons une abstraction dédiée à générer des ports à partir d'une même définition. Cette abstraction définit un traitement consistant à appliquer les configurations (ce qui met à jour les listes de connexions des méta-connexions) et à générer un port à partir d'un protocole et des méta-connexions.

```
port_producer is constructor(
                meta_connexions:sequence[MetaConnection],
                protocol: Expression[Behaviour],
                configurations:sequence[BehaviourExpression]);{
-- application des configurations (ce qui modifie les méta-connexions)
-- génération d'un port en fonction du protocole et des méta-connexions mise à jour

}as{
     port with {
          connexions $meta_connexions
          configuration $configurations
          protocol  { $protocol }
     }
}
```

[33] A la différence d'un comportement, le protocole ne s'exécute pas. Il s'agit d'un ensemble de propriétés sur les interactions.

3.5. Méta-port

3.5.1 Définition

> Un ***méta-port*** contient la définition d'un port et les informations permettant la création dynamique de plusieurs occurrences et leur gestion.

Un méta-port est défini avec un nom, qui permet de le référencer et de référencer les occurrences qui sont issues du méta-port.

3.5.2 Syntaxe

La structure de la syntaxe pour déclarer un méta-port est la suivante*. meta_port_name* doit être un identifiant et *port_definition* doit être l'expression d'un port.

```
meta_port_name is port_definition
```

3.5.3 Description

Ceci est implémenté avec un constructeur permettant de générer des méta-ports à partir d'un nom et de la définition d'un port.

```
meta_port is constructor(
     _name : Alias,
     _portDefinition : PortDefinition
);{
     --meta-port definition
}as{ $_name is $_portDefinition }
```

Un méta-port est implémenté comme une vue constituée du nom du méta-port, d'une unité productrice de ports (l'unité productrice correspond au constructeur de port) et d'une liste de ports (cette liste est vide à la création du méta-port). Le type *Meta_Port* défini pour désigner un méta-port est le suivant.

```
type Meta_Port is view{
     name : String,
     port_producer : abstraction[],
     instances is sequence(Port)
}
```

3.6. Attributs

3.6.1 Définition

> Un ***attribut*** est une entité qui représente une propriété au sein d'un élément architectural.

Sa raison d'être est de représenter les propriétés[34] d'un élément qui ne sont ni structurelles, ni comportementales. Ces attributs permettent à l'architecte d'apporter des données techniques et de les prendre en compte pour la conception de l'architecture. Par exemple, cela peut-être le *temps de traitement* d'un composant de calcul ou la *capacité* d'un composant de stockage. Les attributs contiennent ainsi des informations essentielles aussi bien lors de la conception que de l'analyse d'une architecture. Ces informations aident l'architecte à faire des choix entre les différents éléments qu'il peut utiliser au sein d'une architecture.

De plus, les attributs peuvent être utilisés en dehors de la conception, c'est-à-dire à l'exécution. En effet, la valeur d'un attribut peut être accédée à l'exécution en lecture et en écriture depuis l'élément

[34] Dans d'autre ADLs, on parle de propriété (***property***) comme dans ACME [GarMon&Wil 96], ARMANI [Mon 01] ou UNICON [DeL 96].

qui la contient ou par d'autres éléments. Un attribut n'est pas fixe, sa valeur peut changer au cours du temps. Cette particularité permet au système de prendre lui-même des décisions lors de ses changements topologiques. Nous verrons plus loin que cela joue un rôle particulier à l'instanciation d'un élément architectural : les éléments issus d'un même méta-élément peuvent être différenciés à l'instanciation par les valeurs de leurs attributs.

3.6.2 Syntaxe

La structure syntaxique pour la déclaration d'un attribut est la suivante. Un attribut est défini par un *nom* qui permet de le référencer, le *type* des données qu'il contient et une *valeur par défaut*.

```
nom_attribut : type default value is value
```

L'exemple ci-dessous est celui d'un attribut représentant un temps de latence exprimé en réel. Cette latence peut être l'attribut d'un serveur donnant une évaluation de son temps de réponse.

```
latency : real default value is 10.0
```

Nous verrons comment on accède aux attributs dans la section sur les aspects comportementaux (cf. 4).

3.6.3 Description

Un attribut est représenté par une vue contenant le nom de l'attribut et sa valeur (celle ci est encapsulée dans une valeur de type ***Any***). Nous avons défini le type *Attribute* pour représenter les attributs.

```
type Attribute is view{
     name:String,
     val:Any
}
```

3.7. Eléments architecturaux

3.7.1 Définition

> Les ***éléments architecturaux*** sont les constituants d'une architecture. Ils évoluent à l'exécution et peuvent communiquer entre eux. Ces éléments sont structurés par un ensemble de ***ports***, un ***noyau*** (contient les mécanismes internes de l'élément) et un ensemble d'***attributs***.

Les éléments architecturaux sont classés à la fois selon leur rôle et leur structure. Au sein d'une architecture, les éléments architecturaux ont plusieurs rôles[35] : ils peuvent être composants, connecteurs ou chorégraphes. Concernant leur structure, ils peuvent être soit atomiques, soit composites. Nous étudierons les éléments en fonction de leurs rôles dans les prochaines sections, ici nous nous intéressons plus particulièrement à l'aspect structurel.

Nous allons étudier la description d'un élément qu'il soit atomique ou composite dans ce qui suit.

3.7.2 Syntaxe

Les points communs entre la description d'un élément atomique et celle d'un élément composite sont les suivantes :

- les ports,
- les attributs,
- la configuration.

[35] Le concept d'élément n'est pas utilisé directement par l'utilisateur. Il sert de support à la définition des concepts de composant et de connecteur.

Les différences entre la description d'un élément atomique et celle d'un élément composite concernent le noyau de l'élément. Pour l'élément atomique le noyau est un *comportement* tandis qu'il est un ensemble de *constituants* pour un élément composite.

La structure syntaxique[36] suivante est celle d'un élément atomique.

```
element with{
     ports {    liste de méta-ports    }
     attributes { liste d'attributs }
     traitement { comportement }
     configuration { configuration    }
}
```

La structure syntaxique suivante est celle d'un élément composite.

```
element with{
     ports {    liste de méta-ports    }
     attributes { liste d'attributs }
     constituents {  liste de méta-éléments  }
     choreographer { chorégraphe      }
     configuration { configuration    }
}
```

Les paragraphes suivants donnent des détails sur les différentes parties de la description.

Ports

Dans cette partie, on définit les méta-ports de l'élément (cf. section 3.5).

Attributs

Dans cette partie, on définit les attributs de l'élément (cf. section 3.6).

Traitement

Le traitement effectué au sein d'un élément atomique est décrit comme un comportement. Il est appelé *computation* dans le cas d'un composant et *routing* dans le cas d'un connecteur. Il effectue une succession d'actions qui fait évoluer l'état de l'élément.
Nous étudierons comment décrire un comportement dans la section 4.

Constituants

Dans la partie *constituents* de la description d'un élément composite, on définit les méta-éléments utilisables. Les méta-éléments sont décrits avec un nom qui permet de le référencer et de référencer les occurrences qui en sont issues.

Chorégraphe

Dans cette partie, on définit le comportement du chorégraphe de l'élément composite. Le chorégraphe est étudié plus en détail dans une prochaine section 3.11.

Configuration

La partie configuration décrit l'initialisation de l'élément architectural. La description est semblable à celle d'un comportement[37]. On y décrit :

- l'instanciation des ports,
- l'instanciation des éléments constituants (dans le contexte d'un composite),

[36] Les mots-clés en italique dépendent de la nature de l'élément : composant ou connecteur.
[37] Nous étudierons les détails concernant la description des comportements dans la section 4.

- les liens entre les différents éléments.

Considérons l'architecture client-server suivante (cf. Figure 34). Elle est composée de deux clients et d'un serveur. Les clients communiquent avec le serveur par un connecteur pc supportant la connexion de deux clients. Cette architecture est définie comme un composite proposant un port d'accès liés directement au serveur.

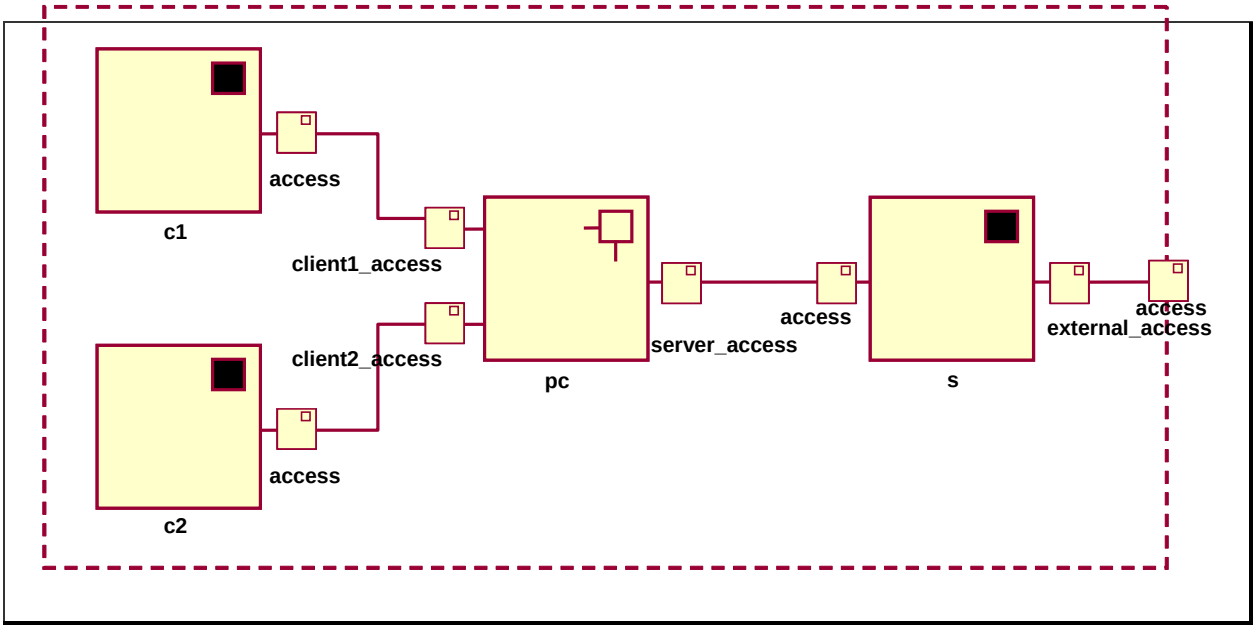


Figure 34 - Architecture Client-Server

L'expression de la configuration du composite exprime la création de plusieurs éléments et de leurs liens avec eux-mêmes et leur environnement.

```
configuration {
      new c1; new c2; new pc; new s; new access;
      bind access to s~external_access;
      attach c1~access to pc~client1_access; attach c2~access to pc~client2_access;
      attach s~access to pc~server_access
}
```

Il y a cinq créations[38]: *c1* et *c2* (qui correspondent à des clients), *s* (qui correspondent à un serveur), *pc* (qui est un connecteur pour appel de procédure), *access* (qui correspondent au port du composite et donne un accès externe au serveur). Ensuite, il y a plusieurs liens de créés :

- le port *access* du composite est lié au port *external_access* de *s*,
- le port *access* du client *c1* est lié au port *client1_access* de *pc*,
- le port *access* du client *c2* est lié au port *client2_access* de *pc*,
- le port *access* de *s* est lié au port *server_access* de *pc*.

3.7.3 Description

Concernant l'implémentation, le style *Architectural_Element* définit tout ce qui attrait au concept d'élément. Ce style définit notamment le constructeur permettant la génération d'unité productrice d'éléments architecturaux (nous allons voir cela en détail dans cette section).

```
Architectural_Element is style with {
      contructors{
            Architectural_Element is constructor(…);…
      }
}
```

[38] Ce à quoi correspondent ces créations dépendent du contexte dans lequel est définie la configuration.

Un élément architectural est défini par une abstraction. L'expression de cette abstraction est décrite ci-dessous.

```
recursive value element is abstraction(
     meta_ports:sequence[MetaPort],
     meta_constituents:sequence[MetaConstituent],
     attributes:sequence[Attribute],
     attachments:sequence[Attachment],
     bindings:sequence[Binding]
);{
…
}
```

L'abstraction est définie *récursive* pour pouvoir prendre en compte des comportements récursifs (nous verrons dans la section 4.4.2 le mécanisme permettant de définir de tels comportements). Nous ne donnons pas l'ensemble de la description qui est longue (cf. Annexe 5). Nous choisissons d'expliquer ce qui s'y trouve. Cette abstraction définit :

- un ensemble de connexions propres aux déclenchements des actions,
  - par exemple, la connexion suivante est déclarée pour l'action d'instanciation, *new* (cf. 4.3),

```
value request_new is connection(String);
```

- un ensemble de connexions propres aux connexions des ports,
  - par exemple, cette connexion est déclarée pour la connexion *wait* du port *access*. Cette connexion est libre (***free***) car accessible depuis l'extérieur,

```
value access_wait is free connection();
```

- un ensemble de connexions propres aux connexions des ports des constituants,
  - par exemple, cette connexion est déclarée pour la connexion *wait* du port *access* du constituant *client*,

```
value client_access_wait is connection();
```

- un ensemble de connexions propres aux attributs,
  - par exemple, cette connexion est déclarée pour l'attribut *latency*. Cette connexion est libre (***free***) car accessible depuis l'extérieur,

```
value attribute_latency is free connection();
```

- un ensemble de comportements correspondant aux constituants sont déclarés pour créer les bindings. Ces déclarations dépendant évidemment des informations du paramètre *bindings*,
  - par exemple, ce behaviour est déclaré pour le constituant *f* dont la connexion *externalaccess_in* est liée à la connexion *access_in* de son contenant. Nous supposons que la valeur `meta_constituents::1::instances::1` représente le constituant *server* dans le paramètre *meta_constituents*. L'unification associée à ce behaviour réalise le lien,

```
value server is meta_constituents::1::instances::1 where {
     externalaccess_in unify access_in
};
```

- le comportement de l'élément.

L'élément est exécuté par application de l'abstraction avec des paramètres représentant l'état de l'élément à son initialisation. Ces paramètres sont : un ensemble de méta-constituants, un ensemble de méta-ports, un ensemble d'attributs, un ensemble d'attachements et un ensemble de bindings.

La description d'un élément atomique et d'un composite est la même. On obtient un élément atomique lorsque les paramètres *meta_constituents*, *attachments* et *bindings* sont des listes vides et lorsque le comportement défini dans l'abstraction est le comportement de l'élément. On obtient un élément composite lorsque le paramètre *meta_constituents* est une liste non vide et lorsque le comportement défini dans l'abstraction est le comportement du chorégraphe (dans le cas où le composite n'a pas de chorégraphe le comportement est ***done***).

Un élément est défini avec les informations qui sont visibles depuis son environnement : son nom, ses ports et ses attributs. Un type *Element* est défini pour représenter cela. C'est une vue constituée d'un *nom*, d'une liste de *méta-ports*, d'une liste *d'attributs* et de l'élément lui-même (*body*).

```
type Element is view{
     name:String,
     meta_ports:sequence[MetaPort],
     attributes:sequence[Attribute],
     body:Behaviour
}
```

Afin de produire des éléments depuis un méta-élément ce dernier est constitué d'un producteur d'éléments. C'est une abstraction qui, à l'exécution, envoie un élément (de type *Element*) initialisé. Ce producteur est généré par le constructeur *Architectural_Element*.

```
Architectural_Element is constructor(
     meta_ports:sequence[MetaPort],
     meta_constituents:sequence[MetaConstituent],
     attributes:sequence[Attribute],
     treatment: Expression[behaviour],
     configuration:sequence[Expression[behaviour]]
);{
     abstraction();{
          -- traitement des configurations pour initialiser l'élément
          -- création de l'élément par application
     }
}
```

Ce constructeur permet la création d'unités productrices d'éléments. Il nécessite les informations concernant les éléments à créer dont leur configuration à l'initialisation et leur comportement. Le traitement des configurations permet de définir l'état de l'élément à l'initialisation en définissant les entités et les liens présents à cet état. L'élément est ensuite créé par application de l'abstraction *element* (déjà définie plus haut) à partir de ses informations.

3.8. Méta-élément

3.8.1 Définition

> Un ***méta-élément*** contient la définition d'un élément et les informations permettant la création dynamique de plusieurs occurrences et leur gestion.

Comme pour le concept d'élément, le concept de méta-élément n'est pas utilisé directement par l'utilisateur. Il sert de support à la définition des concepts de méta-composant et de méta-connecteur.

3.8.2 Syntaxe

Un méta-élément est défini avec un nom, qui permet de le référencer et de référencer les occurrences qui en sont issues.
La structure de la syntaxe pour déclarer un méta-élément est définie ci-dessous (*meta_element_name* doit être un identifiant et *element_definition* doit être l'expression d'un élément).

```
meta_element_name is element_definition
```

3.8.3 Description

Ceci est décrit avec un constructeur permettant de générer des méta-éléments à partir d'un nom et de la définition d'un élément.

```
meta_element is constructor(
      _name : Alias,
      _element : Element
);{
      -- définition du meta-element en fonction des paramètres d'entrée
      -- le méta-élément est une vue
      …
      view ( … )
}as{ $_name is $_element }
```

Ce constructeur génère un méta-élement comme une vue constituée de son nom, d'une unité productrice d'éléments et d'une liste d'éléments (cette liste est vide à la création du méta-élément). Le type *Meta_Element* est défini pour désigner les méta_éléments.

```
type Meta_Element is view{
      name : String,
      element_producer : abstraction[],
      instances is sequence(Element)
}
```

Nous avons étudié en détail le concept d'élément architectural et de méta-élement. Dans les sections suivantes, nous spécialisons ce concept pour définir les concepts de composant et de connecteur.

3.9. Composant

3.9.1 Définition

> Les ***composants*** sont des unités de traitement ou de stockage de données ; ils représentent les fonctionnalités du système.

3.9.2 Syntaxe

La structure syntaxique suivante est celle d'un composant atomique.

```
component with{
      ports {    liste de méta-ports    }
      attributes { liste d'attributs }
      computation { comportement}
      configuration { configuration    }
}
```

La structure syntaxique suivante est celle d'un composant composite.

```
component with{
      ports {    liste de méta-ports    }
      attributes { liste d'attributs }
      constituents {  liste de méta-éléments  }
```

```
     configuration { configuration    }
}
```

La description suivante est celle d'un serveur semblable à ceux que nous avons pu décrire jusqu'à présent.

```
server is component with{
     ports {
          access is port with {
               connections {
                    request is connection(Any),
                    reply is connection(Any)  }
               configuration {
                    new request;     new reply    }
               protocol  {
                    replicate
                         via request receive    ;
                         via reply send
                    }
          }
     }
     attributes {
          latency : real default value is 10.0
     }
     computation {
          via access_request receive requete:Any;
          unobservable;
          via access_reply send any();
          recurse
     }
     configuration {
          new access
     }
}
```

3.9.3 Description

Un composant est décrit comme un style au même titre qu'on a décrit un style *Architectural_Element*. Le style *Component* étend le style *Architectural_Element* et définit un constructeur et des contraintes.

```
Component is style extending Architectural_Element with {
     contructors{
          Component is constructor(…);…
     }
     constraints{
          …
     }
}
```

Le style *Component* propose plusieurs constructeurs, un pour la définition de composants atomiques, l'autre pour la définition de composants composites. Ces deux constructeurs sont des applications partielles du constructeur *Architectural_Element*. Il s'agit d'applications partielles sans passage de paramètres. Nous utilisons l'application partielle pour fournir des notations spécifiques pour l'application. La première notation est pour la définition de composants composites tandis que la deuxième est pour la définition de composants atomiques. On notera que cette deuxième notation ne prend pas en compte de paramètre *metaconstituents*. Celui-ci sera donc fixé à une liste vide. On

notera aussi, comme nous l'avons expliqué dans la section 3.7.3, que dans le premier cas le paramètre *treatment* représente le comportement du chorégraphe et que dans le deuxième cas il représente le traitement interne du composant (*computation*).

```
constructors{
     Component is Architectural_Element() as {
          component with {
               ports      $metaports
               attributes  $attributes
               constituents    $metaconstituents
               choreographer { $treatment }
               configuration  $configurations
          }
     }
     Component is Architectural_Element() as {
          component with{
               ports      $metaports
               attributes  $attributes
               computation { $treatment     }
               configuration  $configurations
          }
     }
}
```

Le style composant est aussi défini par des contraintes. La contrainte pour un composant est d'avoir au moins un port.

```
constraints{
     onePortAtLeast constraint {
          ports.size>0
     }
}
```

3.10. Connecteur

3.10.1 Définition

> Les ***connecteurs*** sont les éléments de communication au sein d'une architecture ; ils transitent les informations entre les composants.

3.10.2 Syntaxe

La structure syntaxique suivante est celle d'un connecteur atomique.

```
connector with{
     ports {    liste de méta-ports    }
     attributes { liste d'attributs }
     routing { comportement     }
     configuration { configuration    }
}
```

La structure syntaxique suivante est celle d'un connecteur composite.

```
connector with{
     ports {    liste de méta-ports    }
     attributes { liste d'attributs }
     constituents {  liste de méta-éléments  }
     choreographer { chorégraphe      }
```

```
     configuration { configuration    }
}
```

La description suivante est celle du connecteur *pc* représenté plus haut sur la Figure 34. Le comportement décrit pour cet élément consiste en recevoir une requête de l'un des clients, de faire suivre cette requête au serveur, de recevoir la réponse du serveur et de la retourner à ce client. L'autre client ne peut pas envoyer sa requête tant que le premier n'a pas reçu de réponse. Nous verrons l'emploi de l'opérateur ***choose*** plus tard.

```
pc is connector with{
     ports {
          clientAccess is port with {
               connections {
                    wait is connection(Any),
                    call is connection(Any)  }
               configuration {
                    new wait;  new call     }
               protocol  {
                    replicate
                         via call receive;
                         via wait send
               },
          serverAccess is port with {
               connections {
                    request is connection(Any),
                    reply is connection(Any)  }
               configuration {
                    new request;     new reply    }
               protocol  {
                    replicate
                         via request send;
                         via reply receive
               }
          }
     }
     routing {
          choose {
               {    via clientAccess_client1Access_call receive requete:Any;
                    via serverAccess_request send requete;
                    via serverAccess_reply receive reponse:Any;
                    via clientAccess_client1Access_call send reponse  }
          or
          {    via clientAccess_client2Access_call receive requete:Any;
                    via serverAccess_request send requete;
                    via serverAccess_reply receive reponse:Any;
                    via clientAccess_client2Access_call send reponse  }
          } then recurse
     }
     configuration {
          new clientAccess named client1Access;
          new clientAccess named client2Access;
          new serverAccess
     }
}
```

3.10.3 Description

Un connecteur est décrit comme un style au même titre qu'on a décrit le style *Architectural_Element* et le style *Component*. Le style *Connector* étend le style *Architectural_Element* et définit un constructeur et des contraintes.

```
Connector is style extending Architectural_Element with {
     contructors{
          Component is constructor(…);…
     }
     constraints{
          …
     }
}
```

Le style *Connector* propose plusieurs constructeurs, un pour la définition de connecteurs atomiques, l'autre pour la définition de connecteurs composites. Ces deux constructeurs sont des applications partielles du constructeur *Architectural_Element*. Il s'agit d'applications partielles sans passage de paramètres. Nous utilisons l'application partielle pour fournir des notations spécifiques pour l'application. La première notation est pour la définition de connecteurs composites tandis que la deuxième est pour la définition de connecteurs atomiques. On notera que cette deuxième notation ne prend pas en compte de paramètre *metaconstituents*. Celui-ci sera donc fixé à une liste vide.

```
constructors{
     Connector is Architectural_Element() as {
          connector with{
               ports      $metaports
               attributes  $attributes
               constituents    $metaconstituents
               choreographer { $treatment   }
               configuration  $configurations
          }
     }
     Connector is Architectural_Element() as {
          connector with{
               ports      $metaports
               attributes  $attributes
               routing { $treatment  }
               configuration  $configurations
          }
     }
}
```

Le style *Connector* est aussi défini par des contraintes. La contrainte pour un connecteur est d'avoir au moins deux ports.

```
constraints{
     twoPortsAtLeast constraint {
          ports.size>1
     }
}
```

3.11. Chorégraphe

3.11.1 Définition

> Un ***chorégraphe*** gère dynamiquement la structure d'un composite. En d'autres termes il gère la création des différentes entités (les ports, les connexions, les composants et les

connecteurs) ainsi que la reconfiguration (attachements et détachement) au sein d'un composite.

Un chorégraphe peut communiquer avec tout autre constituant du composite qui le contient. La différence par rapport aux autres constituants est qu'il a connaissance de son environnement et qu'il n'a pas besoin d'être relié explicitement aux autres éléments. À l'intérieur du composite, il a une visibilité sur tous les constituants et leurs interfaces (ports) et peut leur faire référence. De plus, il est aussi lié implicitement à l'interface du composite qui le contient et ses attributs sont les attributs du composite.

Par exemple, un chorégraphe peut attacher un composant à un connecteur. L'architecture présentée ci-dessous (Figure 35) est composée d'un composant *A*, d'un connecteur *B* et d'un chorégraphe. Le composant exhibe un attribut *isolated*, de nature booléenne. Lorsque cet attribut présente une valeur vraie et que le chorégraphe lit cette valeur, il attache le composant A au connecteur *B*. Nous verrons comment décrire le chorégraphe juste après.

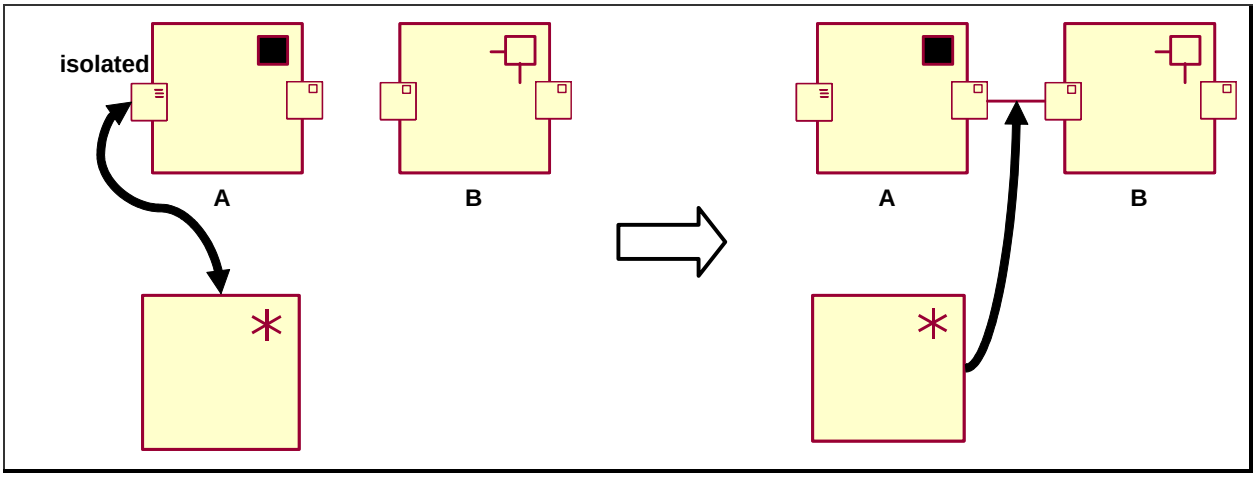


Figure 35 - Reconfiguration dynamique

3.11.2 Syntaxe

La description d'un chorégraphe ne s'appuie pas sur la même structure syntaxique que les autres éléments architecturaux. Comme nous l'avons déjà vu, la description d'un chorégraphe est liée à celle du composite qui le contient. Ses ports et ses attributs sont définis de manière implicite, donc de manière transparente pour l'utilisateur. Ainsi, seul le comportement du chorégraphe est spécifié dans sa description.

L'exemple de description ci-dessous est celle du chorégraphe de l'exemple précédent. Après avoir reçu l'ordre d'attacher *A* et *b*, le chorégraphe exécute cet ordre.

```
choreographer {
     if (A~attribute_isolate=true)
          attach A~p to B~p;
     recurse;
}
```

3.11.3 Description

Nous l'avons évoqué dans la section 3.7.3, la description d'un chorégraphe est incluse dans celle de l'élément composite qui le contient. Le chorégraphe est décrit comme un comportement.

A travers cette section sur les concepts structuraux, nous avons vu que la description des comportements est souvent nécessaire. De plus, nous avons souligné la possibilité d'effectuer des actions propres au style Composant-Connecteur. Nous allons étudier cela dans la section suivante.

4. Concepts Comportementaux

> Un ***comportement*** est une suite d'actions qui s'effectuent les unes après les autres.

Dans le style Composant-Connecteur, les descriptions de comportements sont utilisées pour :

- décrire les clauses configurations, c'est à dire pour l'initialisation des différentes entités (ports et éléments architecturaux),
- décrire le traitement interne des composants et des connecteurs,
- décrire les chorégraphes.

Les actions de base du langage ASL sont des actions de communication. Le style Composant-Connecteur définit de nouvelles actions ainsi que de nouveaux opérateurs.

Nous avons classé les actions introduites en plusieurs groupes :

- les actions de communication,
- les actions topologiques,
- les actions d'instanciation.

4.1. Actions de communication

> Les ***actions de communication*** sont des réceptions et des envois de messages via des connexions.

Parmi ces actions, il y a les actions basiques qui permettent d'envoyer un message par une connexion.

```
via connection_reference receive identifier : type
via connection_reference send identifier
```

Dans cette section, nous montrons comment référencer des connexions, comment faire des envois groupés, comment la mobilité est gérée et comment on accède aux attributs par ce type d'action.

4.1.1 Référencer des connexions

Une connexion est référencée[39] par rapport à un port, qui à son tour peut être référencé par rapport à un constituant. De plus, dans ce cas, le nom d'une connexion est préfixé par le nom de sa méta-connexion. Par exemple, pour faire un envoi vers la connexion *wait_wait1* du port *access_access1*, on écrira.

```
via access_access1_wait_wait1 send
```

Dans le cas où on fait un envoi vers la connexion *wait_wait1* du port *access_access1* qui interface le constituant c*lient_c1*, on écrira.

```
via client_c1_access_access1_wait_wait1 send
```

Cependant, une entité ne porte pas toujours un nom. Dans ce cas, on peut obtenir une référence vers celle-ci avec les actions **get entity**. Dans le cas d'une connexion, on utilise l'action **get connection** permettant d'obtenir la référence d'une connexion (ou d'un ensemble de connexions).

```
get connection client#comp~access#port~wait#conn as reference
```

[39] On peut noter qu'on peut utiliser le nom d'un méta_élément pour référencer sa première instance.

Dans l'expression précédente, `client#comp~access#port~wait#conn` , permet de cibler la connexion. Le symbole `~` sépare les entités et le symbole `#` sépare la référence vers une méta-entité de la référence vers l'une de ses occurrences. Cette dernière est représentée soit par *comp*, *port* ou *conn* qui sont des valeurs des types suivants :

- **Alias** : dans ce cas l'identifiant représente le nom de l'entité.
- **Natural** : dans ce cas la valeur représente le numéro de l'occurrence de la méta-entité.
    - Notons qu'il est possible de ne pas spécifier la référence d'une entité. Dans ce cas, la référence retenue est la première entité de la méta-entité.
    - L'expression suivante permet de récupérer la connexion *wait* (équivalent à wait##1) du port *access#access1* du deuxième filtre.

```
get connection client##2~access#access1~wait as reference
```

- **quote**[all]: dans ce cas la valeur **all** représente l'ensemble des occurrences d'une méta-entité.
    - Le style Composant-Connecteur définit une valeur prédéfinie, **all**, de type **quote**[all]. Le type quote est utilisé car il ne représente qu'une valeur : **quote**(all). Ainsi, **all** vaut **quote**(all).
    - L'expression suivante permet de récupérer une séquence incluant la connexion *wait* du port *access#access1* de chaque filtre.

```
get connection client#all~access#access1~wait as reference
```

- **quote**[last]: dans ce cas la valeur **last** représente la dernière occurrence d'une méta-entité.
    - Le style Composant-Connecteur définit une valeur prédéfinie, **last**, de type **quote**[last].
    - L'expression suivante permet de récupérer la connexion *wait* du port *access#access1* du dernier filtre créé.

```
get connection client#last~access#access1~wait as reference
```

4.1.2 Communication via des ensembles de connexions

Comme nous l'avons laissé supposer précédemment, le style Composant-Connecteur fournit des actions évoluées permettant l'envoi et la réception de données vers un ensemble de connexions[40]. Les références vers les listes de connexions peuvent être utilisées.

Lorsqu'on utilise des listes de connexions, on a accès a deux types de communications : la communication sur *tous* ou la communication sur *un parmi tous*. On peut envoyer un message vers un ensemble de connexions ou recevoir une liste de messages depuis ces connexions. Les expressions ci-dessous montrent ces deux actions.

```
via sequence(liste de connexions) send valeur
via sequence(liste de connexions) receive identifier: sequence[Type]
```

Il faut noter qu'en réception on reçoit une séquence contenant les données reçues par chaque connexion. Ces données sont ordonnées par correspondance à leur connexion associée. Il faut aussi noter que les si une connexion n'est pas disponible pour effectuer la transmission, la liste est bloquée (l'élément indisponible bloque les éléments suivant dans la liste).

On peut envoyer un message vers une connexion choisie aléatoirement parmi les connexions prêtes à interagir, ou de recevoir un message d'une occurrence choisie aléatoirement parmi celles qui sont prêtes pour envoyer. Cela consiste à utiliser le mot-clé ***any*** lors de l'action, comme montré ci-dessous.

[40] Il faut noter que dans ce cas, les communications ne sont pas polyadiques : en d'autres termes, on ne peut pas envoyer un tuple de valeurs. Cependant, on peut envoyer une donnée de type tuple, ce qui permet de contourner le problème.

```
via any sequence(liste de connexions) send valeur
via any sequence(liste de connexions) receive identifier:Type
```

4.1.3 Mobilité

La mobilité est un mécanisme intrinsèque aux actions de communication. En effet, tout type de données définissable en ASL peut transiter par les connexions. Parmi ces données, il y a les connexions, les ports et les éléments architecturaux. Ainsi, toutes les entités architecturales sont mobiles ; elles peuvent transiter d'un élément à un autre.

Cependant, avant de pouvoir transiter à travers les connexions, ces différentes entités doivent être extraites[41] de l'architecture qui les contient. Elles deviennent alors des entités mobiles. Une fois arrivée à destination une entité mobile peut être intégrée[42] dans son nouvel environnement. Une entité transite avec une méta-entité correspondante, ainsi l'entité peut être dupliquée dans son nouvel environnement.

Le style Composant-Connecteur définit des types spécifiques propres aux entités mobiles : *Mobile_Connection*, *Mobile_Port* et *Mobile_Element*. Les valeurs associées à ces types contiennent à la fois l'entité qui transite mais aussi la méta-entité correspondante. L'expression ci-dessous montre la définition d'un type pour une entité mobile.

```
type Mobile_Entity is view[
     meta_entity: Meta_Entity,
     entity: Entity
]
```

4.1.4 Accès aux attributs

L'accès aux attributs d'un élément passe par des actions de communication. Les attributs sont accédés de la même manière qu'on communique par un port. C'est comme si un port *attribute* était défini et que chacune de ses connexions représente un attribut (cf. Figure 36). Nous rappelons que les attributs sont décrits comme des vues.

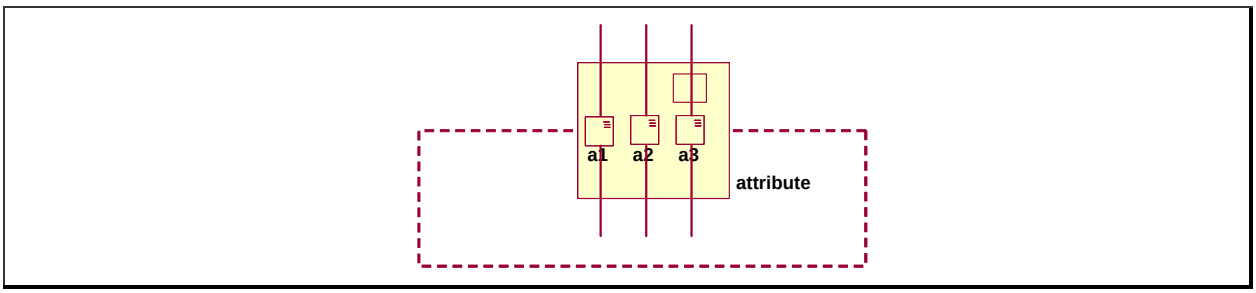


Figure 36 - Des attributs similaires à un port

Ainsi, depuis le comportement d'un élément, on accède à un attribut, *nomAttribut*, respectivement en écriture et en lecture, de la manière suivante :

```
via attribute_nomAttribut send donnee
```

```
via attribute_nomAttribut receive donnee:TypeDonnee
```

Depuis l'environnement d'un élément *element*, on accède à un attribut, *nomAttribut*, respectivement en écriture et en lecture, de la manière suivante :

[41] Nous verrons comment dans la section consacrée aux actions topologiques.
[42] Nous verrons comment dans la section consacrée aux actions topologiques.

```
via element_attribute_nomAttribut send donnee
```

```
via element_attribute_nomAttribut receive donnee:TypeDonnee
```

Ainsi, l'attribut *latency* du serveur *s* est lu de la manière suivante :

```
via server_s_attribute_latency receive latency:Real
```

4.2. Actions topologiques

> Les ***actions topologiques*** sont des actions pour la configuration structurelle d'un élément. Ces actions permettent d'extraire des entités, de les intégrer, de les lier ou de les délier.

4.2.1 Extraction et Insertion

Le style Composant-Connecteur fournit des mécanismes pour extraire des entités architecturales de leur environnement et pour les inclure dans un nouvel environnement. Ces mécanismes sont valables pour les connexions, les ports et les éléments architecturaux.

Extraction

> Le mécanisme d'***extraction*** consiste à isoler une entité de son contexte.

Lors d'une extraction, tous les liens que l'entité concernée partage avec d'autres entités sont détruits. L'entité devient alors une entité dite "mobile". Elle n'est plus référencée que par une variable dans un comportement et peut transiter par des connexions.

Les structures syntaxiques de l'extraction sont présentées ci-dessous. Elles diffèrent en fonction du type d'entité concerné. On y précise la référence de l'entité à extraire et un identifiant permettant de la référencer.

```
extrudes element reference_element as identifiant
```

```
extrudes port reference_port as identifiant
```

```
extrudes connection reference_port~reference_connection as identifiant
```

Insertion

> Le mécanisme d'***insertion*** consiste à intégrer une entité mobile dans un nouvel environnement, c'est-à-dire un élément architectural.

A l'intégration, l'entité est associée à une méta-entité dont la nature correspond à celle de l'entité.

Les structures syntaxiques des inclusions sont présentées ci-dessous. Elles diffèrent en fonction du type d'entité concerné. On y précise l'identifiant de l'entité à intégrer et son nom. On peut aussi indiquer le nom de la méta-entité qui sera associée.

```
inserts element identifier [in meta] as name
```

```
inserts port identifier [in meta] as name
```

```
inserts connection identifier [in meta] as name
```

On peut différencier trois cas différents lors d'une insertion :

- on donne le nom d'une méta-entité existante dans le nouvel environnement,
    - si l'entité partage une définition commune avec cette méta-entité alors elle est considérée comme une nouvelle instance de celle-ci,
- on donne le nom d'une méta-entité inexistante dans le nouvel environnement,
    - la méta-entité transitant avec l'entité est incluse dans l'environnement avec ce nom et sa première occurrence est l'entité,
- on ne donne pas de nom de méta-entité,
    - la méta-entité transitant avec l'entité est incluse dans l'environnement mais n'est pas nommée. Elle ne pourra jamais être réutilisée.

Dans le cas où une nouvelle méta-entité est créée et nommée, elle peut-être utilisée comme n'importe quelle autre pour la création dynamique de nouvelles entités.

4.2.2 Liaisons

Définition

Les éléments sont mis en relation par des liens de communication. Ces liens relient leurs interfaces, c'est à dire les ports, et plus précisément les connexions. On peut relier directement deux ports, ou deux connexions. Relier deux ports consiste à relier toutes leurs connexions de même nom.

Pour que des liens soient valides, il y a plusieurs conditions à respecter :

- les connexions doivent être de même type,
- les protocoles d'interactions des ports doivent être compatibles.

Pour pouvoir faire transiter une information par un lien, il faut un élément émetteur et un élément récepteur. Le premier doit effectuer un envoi et le second doit effectuer une réception, chacun sur leur connexion concernée par le lien.

Nous différencions deux sortes de liens : les attachements et les bindings.

> Un **attachement** est un lien entre deux éléments. Il peut se faire entre un composant et un connecteur, ou entre deux connecteurs.

> Un **binding** est un lien entre l'interface d'un élément composite et l'interface d'un de ces constituants.

Ces liens sont créés au sein d'un élément composite à l'initialisation ou pendant l'exécution. Ils peuvent aussi être défaits à l'exécution. Il n'y a pas de restriction à l'application de ces deux actions : on peut créer un lien déjà existant et détruire un lien inexistant, cela aura simplement aucun effet.

Syntaxe

Les structures syntaxiques présentées ci-dessous correspondent à l'attachement de deux connexions et à l'attachement de deux ports. La syntaxe est similaire en ce qui concerne les bindings, il suffit de remplacer 'attach' par 'bind'.

Les références des différentes entités sont données suivant le schéma suivant :

```
Meta_element#element~meta_port#port~meta_connection#connection
```

- Attachement de deux connexions.

```
attach connection_ref to connection_ref
```

- Attachement de deux ports. Cela consiste à attacher toutes leur connexions qui portent le même nom.

```
attach port_ref to port_ref
```

En ce qui concerne la destruction d'un attachement nous différencions plusieurs cas. Les actions similaires existent en ce qui concerne les bindings, il suffit de remplacer detach par unbind.

- Détacher deux connexions.

```
detach connection_ref from connection_ref
```

- Détacher une connexion de toutes les connexions qui y sont attachées.

```
detach connection_ref
```

- Détacher deux ports.

```
detach port_ref from port_ref
```

- Détacher un port de tous les ports qui y sont attachés.

```
detach port_ref
```

- Détacher un élément de tous les éléments qui y sont attachés.

```
detach element_ref
```

Description

Concernant la description de ces actions, les liaisons sont réalisées en s'appuyant sur le mécanisme de liaison d'ArchWare ADL : l'*unification*. Pour faire un attachement, on crée une *composition* des deux éléments concernés par l'attachement en unifiant les connexions à attacher. L'expression[43] suivante montre le mécanisme utilisé. Deux éléments, *element1* et *element2* sont reliés par des connexions *in* et *out* appartenant à des ports tous deux appelés *inout*.

```
attachment_element1_inout_out__ element2_inout_in is
compose{
     e1 is element1 where {inout_out unifies element2::inout_in}
and
     e2 is element2
}
```

Le détachement est réalisé de la manière suivante.

```
decompose attachment_element1_inout_out__ element2_inout_in
```

Pour faire un binding, on utilise à nouveau l'unification. Supposons qu'un élément *element* est lié à la connexion *error_out* de son contenant par sa connexion *inout_out*. Il est déclaré de la manière suivante au sein de son composite.

```
value e is element where {inout_out unifies error_out}
```

4.3. Instanciation

> L'***instanciation*** est l'action consistant à créer une entité que ce soit à l'initialisation de l'architecture ou à l'exécution.

[43] C'est une expression volontairement simplifiée pour améliorer la compréhension.

Les entités pouvant être instanciées sont les connexions, les ports, les composants et les connecteurs. Elles sont instanciées à l'initialisation de leur contenant/environnement ou en cours d'exécution, c'est à dire dynamiquement.

Le mot-clé pour l'instanciation est ***new***. La structure syntaxique[44] utilisée est la suivante.

```
new nom_meta-entité [named nom_entite] [initialised with {parameters}]
```

Afin d'instancier une méta-entité, on donne son nom et optionnellement le nom de l'entité et une liste de paramètres pour configurer cette entité. Nous montrons l'utilisation de l'instanciation sur quelques exemples. Considérons le méta-composant nommé *Server* qui définit un attribut *latency*.

```
Server is component with {
     attributes{
          latency: real default value is 10
     }
     …
}
```

L'exemple suivant montre comment *Server* est instancié. Le nouveau composant créé est nommé *Agilium.*

```
new Server named "Agilium"
```

Il n'est pas nécessaire de nommer une entité lorsque c'est la première occurrence.

```
new Server
```

Dans le cas de l'instanciation des composants et des connecteurs, il est possible d'initialiser les valeurs des attributs. Cela permet d'obtenir des variations parmi les entités issues d'une même méta-entité. Dans l'exemple suivant, on montre la création de deux serveurs possédant des temps de latence différents.

```
new Server named "Agilium" initialised with {5 as latency};
new Server named "ArchWare" initialised with {200 as latency }
```

L'initialisation des attributs apporte beaucoup de flexibilité lorsqu'elle est conjuguée avec une configuration adaptée. Elle permet de rendre l'initialisation totale d'un élément entièrement configurable en s'appuyant sur la valeur des attributs. Par exemple, les attributs peuvent évoquer la présence ou non de constituants au sein d'un élément composite.

Une utilisation simple des attributs dans une configuration est de garder certaines actions avec des conditions sur la valeur des attributs. Par exemple, considérons un serveur qui peut disposer ou non d'un port *externalAccess* qui permet entre autre de communiquer les erreurs rencontrées au niveau du serveur. Un attribut appelé *redirection_erreur* permet d'indiquer le serveur, communique les erreurs qu'il rencontre et permet de déclencher la création d'un port *externalAccess* lorsque sa valeur est *true*.

```
Server is component with {
     ports{
          access is port with{…},
          externalAccess is port with{
               connections {
                    error is connection(Any),
                    …  }
               …
          }
     }
```

[44] Les parties entre crochets sont optionnelles.

```
        attributes{
            redirection_erreur: real default value is false
        }
        computation { … }
        configuration{
            new access;
            if(redirection_erreur)then{
                new externalAccess
            }
        }
}
```

4.4. Opérateurs

Le style Composant-Connecteur ne bouleverse pas la manière d'agencer les actions dans la description d'un comportement. Cependant, il propose quelques extensions concernant le choix et la récursivité, et suggère quelques restrictions.

4.4.1 L'opérateur de choix

L'opérateur de choix d'ASL a été étendu pour permettre la succession après un choix comme le montre l'expression suivante.

```
choose {comportement_1 or … or comportement_n } then comportement
```

Cela signifie que le comportement *comportement* succède au comportement qui a été choisi lorsque ce dernier prend fin. Ainsi cette extension permet d'alléger la description.
Le ***choose…then*** est implémenté comme un ***choose*** normal ou chaque choix est la concaténation d'un choix ($comportement_n$) et du comportement *comportement*.

```
constructor( behaviours:sequence[Expression[behaviour]],
       next_behaviour:Expression[behaviour]);{
choose{
     eval(behaviours::1++next_behaviour)
or
     …
or
     eval(behaviours::n++next_behaviour)
}
} as {
choose $behaviours then $next_behaviour
}
```

4.4.2 Récursivité

L'opérateur ***recurse*** est introduit. Il permet de revenir à l'état initial du comportement d'un élément.

```
recurse
```

Par exemple, prenons le cas d'un serveur qui reçoit une requête, qui la traite, qui envoie une réponse et qui recommence.

```
server is component with{
     ports {
          access is port with {…}
     }
     computation {
          via access_request receive requete:Any;
          unobservable;
```

```
            via access request receive requete:Any;
            recurse
      }
      configuration {
            new access
      }
}
```

Cet opérateur est décrit de la manière suivante.

```
recurse is constructor();context(     element:abstraction[
                        meta_ports:sequence[MetaPort],
                        meta_constituents:sequence[MetaConstituent],
                        attributes:sequence[Attribute],
                        attachments:sequence[Attachment],
                        bindings:sequence[Binding]],
                        meta_ports, meta_constituents,  attributes,
                        attachments, bindings);{
element(meta_ports, meta_constituents,      attributes, attachments, bindings)}
as {
      recurse
}
```

Nous rappelons qu'un élément est décrit comme une abstraction récursive (cf. 3.7.3). L'opérateur ***recurse*** permet le rappel de l'abstraction avec les paramètres courants. L'abstraction *element* ainsi que ces paramètres doivent faire partie du contexte de ***recurse***. Lors d'une utilisation correcte, c'est-à-dire au sein d'un comportement d'un élément architectural c'est toujours le cas. En effet, le comportement est décrit au sein de l'abstraction *element*.

5. Styles de fondation – Le Client Serveur

Les styles de fondation sont des styles récurrents dans le domaine des architectures logicielles. Nous les appelons styles de fondation car ils sont la base pour la définition de styles plus spécifiques. Ces styles de fondations sont définis à partir du style Composant-Connecteur (cf. Figure 37). Dans le cadre de nos travaux, nous avons implémenté le style Client-Server, le style Pipe-Filter, le style BlackBoard (ou DataIndirection) et le style en couche (Layered style).

La figure suivante montre que plusieurs styles de fondation peuvent être utilisés ensemble pour la définition d'une architecture ou d'un style plus spécifique. Chacun des styles de fondation apporte une solution à un besoin ponctuel.

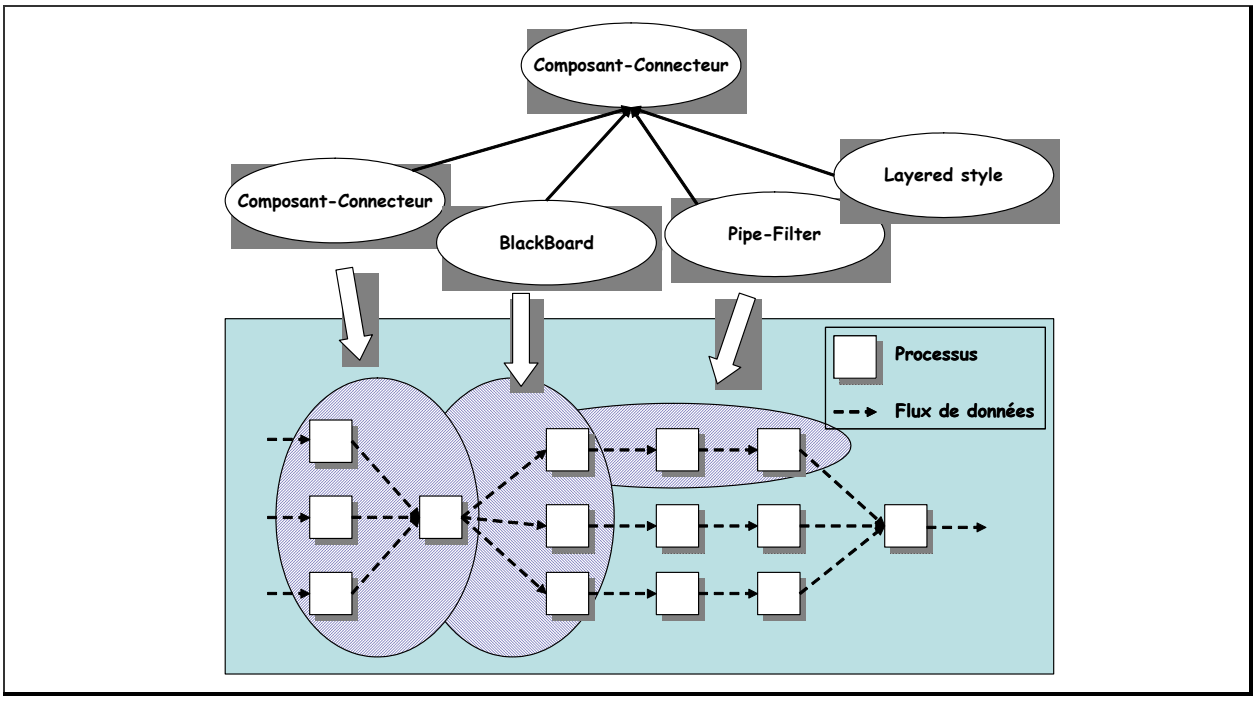


Figure 37 - Styles de fondation

Pour la description de ces styles, nous nous sommes inspirés de plusieurs travaux et notamment des travaux sur ARMANI [Mon 01] et sur les ABAS [Kle&Kaz 99].

Dans cette section, nous étudions le style Client-Server, les autres sont donnés en annexe. Nous structurons sa présentation de la manière suivante. Nous le présentons d'abord de manière informelle. Ensuite, nous donnons sa description. Puis, nous définissons un sous-style du Client-Serveur, le style *Dynamic_CS* qui impose à l'architecture d'être dynamique. Enfin, nous donnons un exemple d'architecture suivant le style.

5.1. Concepts

Le style Client-Serveur est un style générique populaire utilisé fréquemment pour construire des systèmes d'information distribués.
Les composants, dans une architecture client-serveur, sont soit des *clients*, soit des *serveurs*. D'un point de vue topologique, les clients encerclent un serveur. Les clients envoient des requêtes pour des données ou des traitements à des serveurs. Ces derniers effectuent la recherche de données ou le traitement requis.
Les connecteurs sont des appels de procédure distants ou locaux. Ces appels peuvent être soit bloquants soit non bloquants.

Le diagramme informel suivant illustre le style Client-Serveur par un exemple d'architecture.

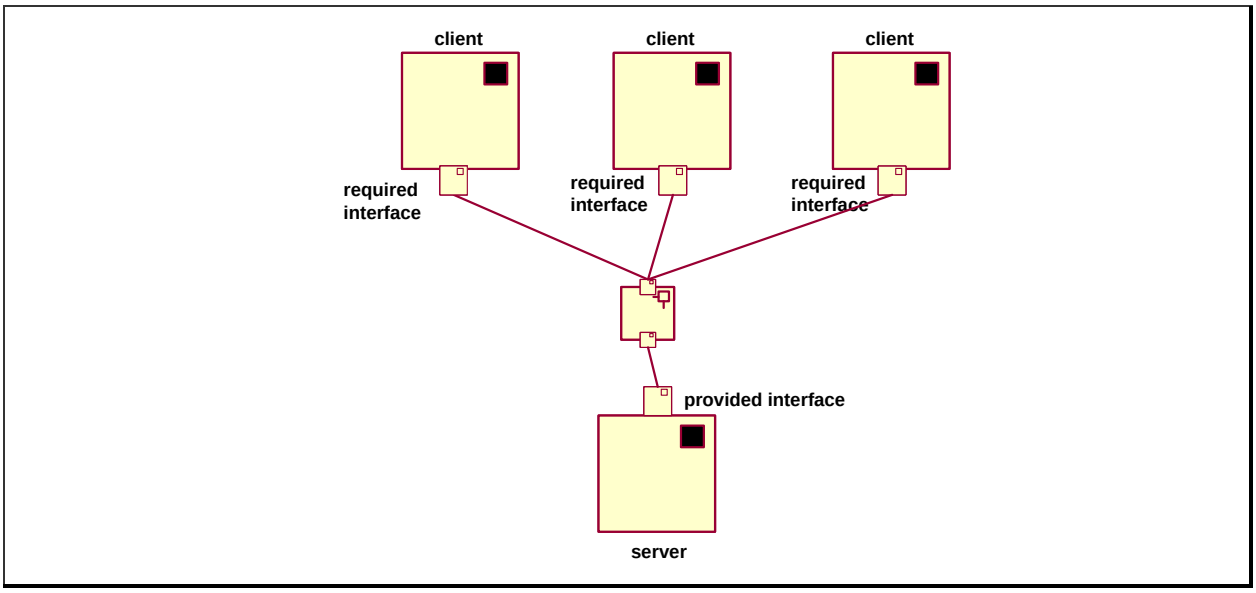


Figure 38 - Architecture Client-Serveur

5.2. Implémentation

La formalisation de style présentée dans cette section est inspirée de la description du style Client-Serveur basée sur Armani.

Style Client-Serveur

La description que nous présentons ici spécifie le haut niveau d'abstraction du style *Client_Serveur*. Il montre ce que le style fournit comme support structurel (la définition des éléments est remplacée par des pointillés pour le moment ; nous étudierons leurs définitions plus tard) :

- les composants sont des *Clients* et des *Servers*,
- les connecteurs sont des appels de procédure (*PC*) qui peuvent être distants (*RPC*) ou non.

```
Client_Serveur is style extending component where {
styles{
     Client is style extending Component …
     Server is style extending Component …
     PC is style extending Connector …
     RPC is style extending PC …
}
```

Le style *Client_Serv*eur définit une syntaxe adaptée par la définition du constructeur suivant.

```
constructors {
     Client_Serveur is constructor(
          meta_ports:sequence[MetaPort],
          meta_clients:sequence[MetaConstituent],
          meta_servers:sequence[MetaConstituent],
          attributes:sequence[Attribute],
          treatment: Expression[behaviour],
          configurations:sequence[Expression[behaviour]]
     );{  Component(      meta_ports,
                          meta_clients inserts meta_servers,
                          attributes,
                          treatment, configurations      )
     } as {
          client_server with {
               [ ports    $meta_ports  ]
```

```
                [ attributes  $attributes ]
                clients    $meta_clients
                servers    $meta_servers
                choreographer { $treatment   }
                [ configuration  $configurations ]
            }
        }
}
```

Les contraintes du style *Client_Serveur* sont les suivantes :

- les composants sont soit des Clients, soit des Serveurs et rien d'autre,
- un Client ne peut-être connecté qu'à un Serveur,
- un Serveur ne peut-être connecté qu'à un Client.

```
constraints {
     only_PC_connectors is constraint {
          to styleInstance.connectors apply
               forall{c |      c in style PC}
     },
     only_Client_Server_components is constraint {
          to styleInstance.components apply
               forall{c |      c in style Client
                         or    c in style Server}
     },
     only_Client_Server_connexions is constraint {
          to styleInstance.components apply
               forall{c1, c2 | c1 connected to c2 implies
                         (c1 in style Client and
                         c2 in style Server) or
                         (c1 in style Server and
                         c2 in style Client)
               }
     }
}
```

Le style *Client_Server* fournit une analyse permettant de vérifier si deux composants sont connectés.

```
analysis  {
     connected is analysis {
          kind AAL
          input { c1:Component, c2:Component      }
          output {  Boolean   }
          body {
               to styleInstance.connectors apply
                    exists{conn |    c1 attached to conn and
                                          c2 attached to conn )
          }
}
```

Eléments

Le style Client est défini par héritage du style Component. Il définit le style Required_Port représentant le port de communication d'un client. Il définit un constructeur par application partielle du constructeur Component avec un port *Required_Port* en paramètre. Les contraintes définies par le style sont "contenir au moins un port *Required_Port*" et "avoir un comportement vérifiant bien l'alternance entre les envois et les réceptions".

```
Client is style extending Component where {
     styles {
          Required_Port is style extending Port {… }
     }
     constructors {
          Client is component with {
               ports {
                    p is Required_Port()
               }
          } as {
               basicclient { &treatment }
          }
     }
     constraints {
          -- have a RequiredPort
          have_Required_Port is constraint {
               to styleInstance.ports apply {
                    exists(p | p in style Required_Port )
               }
          },
          -- safety alternation between request sending and answer sending
          safety_alternation is constraint {
                    to styleInstance.connexions apply{
                    exists(c1,c2|
                    every sequence { (not via c1 receive any)*.
                         via c2 send any}
                    leads to state { false } and
                    every sequence { true*. via c1 receive any.
                         (not via c2 send any)*. via c1 receive any}
                    leads to state { false } and
                    every sequence { true*. via c2 send any.
                         (not via c1 receive any)*. via c2 send any}
                    leads to state { false }
               }
          }
     }
}
```

Le style Server est défini comme sous-style du style Component. Il définit le style Provided_Port représentant le port de communication d'un client. Il définit un constructeur par application partielle du constructeur Component avec un port Provided_Port en paramètre. Les contraintes définies par le style sont "contenir au moins un port Provided_Port" et "avoir un comportement vérifiant bien l'alternance entre les envois et les réceptions".

```
Server is style extending Component where {
     styles {
          Provided_Port is style extending Port {… }
     }
     constructors {
          Server is component with {
               ports {
                    p is Provided_Port()
               }
          } as {
               basicserver { &treatment }
          }
     }
```

```
        constraints {
            -- have a Provided_Port
            have_Provided_Port is constraint {
                to styleInstance.ports apply {
                    exists(p | p in style Provided_Port)
                }
            },
            -- safety alternation between answer sending and request sending
            safety_alternation is constraint {
                to styleInstance.connexions apply{
                    to connexions apply{
                        exists(c1,c2|
                        every sequence { (not via c1 send any)*.
                            via c2 receive any}
                        leads to state { false } and
                        every sequence { true*. via c1 send any.
                            (not via c2 receive any)*. via c1 send any}
                        leads to state { false } and
                        every sequence { true*. via c2 receive any.
                            (not via c1 send any)*. via c2 receive any}
                        leads to state { false }
                    }
                }
            }
        }
    }
```

Le style *PC* (Procedure Call) est défini comme sous-style du style *Connector*. Il définit un constructeur par application partielle du constructeur *Connector* avec un attribut blocking permettant de stipuler si les appels sont bloquants ou non.

```
    PC is style extending Connector where{
        constructors{
            PC is connector with{
                attributes {
                    blocking: Boolean default value is true
                }
            }
        }
    }
```

Le style *RPC* (Remote Procedure Call) est défini comme sous-style du style *PC*. Il définit un constructeur par application partielle du constructeur *PC* avec des attributs représentant les adresses de l'appelant et de l'appelé.

```
    RPC is style extending PC where{
        constructors{
            RPC is PC with{
                attributes {
                    callerAddress: String,
                    calleeAddress: String
                }
            }
        }
    }
```

5.3. Héritage

En se basant sur le style Client-Serveur, nous définissons, par héritage, un style imposant la dynamique. Ce style, *Dynamic_CS*, définit une contrainte pour la création dynamique des éléments.

```
Dynamic_CS is style extending Client_Serveur where {
constructors{
     Dynamic_CS is …
}
constraints{
     dynamic_creation is constraint {
          to styleInstance.behaviour apply {
               everysequence { true*. not new(any) } leads {false}
          }
     }
}
}
```

Il définit aussi un constructeur, ce dernier représentant une instance de ce style que nous étudions ci-après.

5.4. Instanciation

Une instance est définie au sein du style *Dynamic_CS*. Cette architecture est décrite en utilisant la syntaxe fournie par le style Client-Serveur. Elle donne la définition des clients et des serveurs. Ceux-ci respectent les contraintes définies dans les styles *Client* et *Server*. A l'initialisation, seul un serveur est créé. Les clients sont créés et liés au serveur dynamiquement. Ainsi, cette architecture satisfait aux contraintes du style *Dynamic_CS*.

```
     Dynamic_CS is client_server with {
          clients{
               Client is basicclient {
                    via p_out send;
                    via p_in receive;
                    inobservable
               }
          }
          servers{
               Server is basicserver {
                    via p_in receive;
                    inobservable;
                    via p_out send;
                    recurse
               }
          }
          choreographer {
               new Client;
               attach Client#last~p to Server~p;
          }
          configuration {
               new Server
          }
     }
}
```

6. Positionnement

Le style Composant-Connecteur dont nous venons d'étudier les différents concepts fournit un langage (que nous appelons C&C) pour la description des architectures Composant-Connecteur. Cependant, il existe déjà de nombreux ADLs pour la description de ce type d'architecture. Dans cette section, nous mettons en valeur les points forts de ce langage comparé aux autres ADLs existants [Med 97][LeyCim&Oqu 02].

Les points forts que nous avons mis en avant par C&C sont les suivants :

- la flexibilité,
- l'extensibilité du langage,
- la description d'architectures dynamiques,
    - o les créations dynamiques,
    - o les reconfigurations dynamiques,
    - o la mobilité.

Ces points sont détaillés dans les paragraphes suivants.

6.1. Flexibilité du langage

Le style Composant-Connecteur permet de montrer la flexibilité du langage ASL dans le sens où il est possible de décrire en ASL une couche adaptée au besoin de l'utilisateur. Nous appelons C&C la couche définie par le style Composant-Connecteur. Tout comme une majorité d'ADLs (ACME [GarMon&Wil 96], UNICON [DeL 96], π-SPACE [Cha 02]), C&C propose l'utilisation des composants et des connecteurs. Chaque ADL aborde ces concepts d'une manière plus ou moins différente, et impose des règles de construction spécifiques. Par exemple, ACME contraint un port d'être lié à un seul autre port. π-SPACE ne permet pas à deux connecteurs d'être reliés entre eux. Des règles de construction trop contraignantes peuvent être conflictuelles en regard de certains systèmes [Sha et al. 95]. Ainsi, nous avons construit C&C afin qu'il soit flexible dans le sens ou il contraigne le moins possible l'utilisateur lors de la conception ; les concepts C&C sont relativement génériques ; les propriétés propres aux architectures composant-connecteurs sont limitées à leur strict minimum.

Dans cette optique de flexibilité, nous avons décidé de relier explicitement l'interface d'un composite à l'interface de ses constituants. C'est le cas dans la plupart des ADLs. Mais il existe d'autres concepts, comme dans π-SPACE où les ports 'libres' (non reliés) des constituants sont considérés comme les ports du composite. Notre choix repose sur deux points. Premièrement, pour une bonne réutilisation, il est important qu'un élément puisse être utilisé dans différents environnements. Or, tous les services d'un composant ne sont pas forcément utilisés au sein du composite qui le contient. De plus, ces services 'libres' n'ont peut être pas de signification en tant que services du composite. Pour cette raison, ils ne sont pas visibles depuis l'extérieur du composite. Deuxièmement, nous modélisons des architectures dynamiques. Ainsi, des ports peuvent être attachés et détachés dynamiquement.

6.2. Aspects comportementaux et dynamiques

Comme c'est le cas pour d'autres ADL (WRIGHT, RAPIDE et π-SPACE), C&C permet la description de comportements. Ces langages sont généralement basés sur des algèbres de processus. En se basant sur le π-calcul, C&C peut décrire des comportements supportant la dynamique dans les architectures.
Peu d'ADLs gèrent la *dynamique*, parmi ceux-ci, π-SPACE semble le plus abouti dans ce domaine. En connaissance des concepts de π-SPACE, nous avons amélioré la gestion de la dynamique sur quelques points. Premièrement, nous avons simplifié la définition des entités. Nous ne différencions pas les entités dynamiques des autres : toute entité est potentiellement dynamique. Deuxièmement, la gestion de la dynamique est différente. Dans π-SPACE une évolution dynamique, (un attachement par exemple), est initiée par un composant atomique. Dans notre cas, afin de garantir une bonne réutilisation des composants et des connecteurs, nous les considérons comme des boîtes noires ; ils sont définis indépendamment de leur environnement et n'ont aucune

connaissance sur leurs attachements avec d'autres éléments. Ainsi, nous proposons d'introduire un nouveau type d'élément, le *chorégraphe*, pour gérer l'évolution au sein du composite qui le contient. Cet élément est très peu réutilisable tant il est étroitement lié au composite qui le contient. Troisièmement, π-SPACE gère la décomposition et la recomposition des composites. C&C ne propose pas de mécanismes explicites pour ce genre de dynamique. Mais il les supporte à travers les actions d'extraction et d'insertion qui permettent des reconfigurations dynamiques.
Nous avons vu aussi que nous traitons partiellement le problème de la *mobilité*. Cependant, notre formalisme est le seul parmi les ADLs étudiés à apporter des mécanismes pour cette approche. Ainsi, il permet à des entités architecturales de transiter d'un environnement à un autre via des connexions.

6.3. Extensibilité

ASL est un langage extensible dans le sens où il permet de décrire des formalismes qui sont de nouvelles couches sur lui-même : C&C est défini comme une extension d'ASL. Cette extension passe par l'utilisation du mécanisme de notation mixfix qui permet d'associer une syntaxe à de nouveaux concepts et à de nouveaux mécanismes. Ces concepts et ces mécanismes sont définis au niveau des constructeurs d'un style comme nous avons pu le voir dans le chapitre précédent.

La plupart des ADLs permettent seulement de redéfinir un vocabulaire pour les entités architecturales. Parmi les langages étudiés dans le chapitre 3, seul AML propose des mécanismes d'extension aussi poussés. Toutefois, il reste difficile avec AML de pouvoir fournir une syntaxe permettant de s'abstraire de la syntaxe et des mécanismes de base.

C&C hérite des propriétés extensibilité d'ASL. Ainsi il peut être directement utilisé pour définir des formalismes pour des domaines spécifiques ; ainsi les utilisateurs peuvent avoir des langages de description adaptés à leur domaine précis. Pour cela, le style Composant-Connecteur doit servir à la définition de sous-styles tels que ceux que nous avons présentés dans la section 5.

6.4. Récapitulatif

Le tableau suivant donne une vue d'ensemble sur les critères du formalisme C&C associé au style Composant-Connecteur par rapport à ceux que nous avons définis dans l'état de l'art concernant la formalisation des architectures.

	C&C
Architecture	*component* composite
Elément de base (englobant la fonctionnalité)	*component*
- interface	*port*
Elément de connexion	*connector*
- interface	*port*
Attachement	*attachement*
Comportement	*behaviour* basé sur le π- calcul
Attributs	*attributes*
Implémentation	*wrappers*
Composition	*component* et *connector* composite
Dynamicité	Elément de gestion : *choreographer* Les attachements peuvent être crées dynamiquement. Les éléments architecturaux peuvent être instanciés à l'exécution.
Mobilité	Les components, connectors, et ports peuvent transiter via les attachements.

Figure 39 - Caractéristiques du C&C concernant la définition d'architectures

7. Conclusion

Dans ce chapitre, nous avons proposé un style Composant-Connecteur définissant les concepts et mécanismes pour la description d'architectures composant-connecteur dynamiques. De plus, ce style est associé à une syntaxe spécifique plus abordable par les utilisateurs.

Le formalisme issu de ce style se démarque des ADLs étudié sur les aspects comportementaux. Non seulement il permet de décrire le comportement des éléments architecturaux, mais il permet de décrire la dynamique d'une architecture, c'est à dire les changements topologiques au cours de l'exécution. Au travers de ce formalisme, nous avons étudié la dynamique en profondeur et sommes allés plus loin que les travaux existants concernant les langages étudiés. Dans cette optique, nous avons introduit un nouveau type d'éléments : le chorégraphe. Cet élément gère la dynamique au sein d'une architecture.

De plus, c'est un formalisme flexible dans les concepts qu'il propose et extensible. Ainsi, il permet aux utilisateurs de l'adapter à leur domaine spécifique.
Nous avons aussi défini des styles récurrents dans la conception architecturale : le style Client-Serveur et le style Pipe-Filter. Ces styles sont définis comme des spécialisations du style Composant-Connecteur.

Enfin, ses "liens de parenté" avec ArchWare ADL lui permet de bénéficier de ses fondements formels ainsi que de tous les outils associés à ce dernier.

Dans le chapitre suivant, nous allons présenter l'implémentation d'un outil pour l'utilisation des styles dans un processus de développement centré architecture.

Chapitre 6

IMPLEMENTATION

Chapitre 6 - Implémentation

ASL est un langage pour la description des architectures et des styles architecturaux. Ce langage permet notamment de définir des architectures et des styles pour les systèmes dynamiques. Nous avons défini le style Composant-Connecteur pour supporter la conception des architectures dynamiques en utilisant les concepts de composant et de connecteur qui sont traditionnellement considérés comme la base de la conception architecturale.

Cependant, il est ardu d'exploiter un langage s'il n'est pas associé à des outils logiciels spécifiques.

Dans ce chapitre, nous proposons d'automatiser l'utilisation d'ASL et des styles avec un outil logiciel. Le but est de pouvoir ***compiler*** une description, ***instancier*** un style, ***vérifier la satisfaction*** d'une architecture à un style, et effectuer des ***analyses*** définies par un style.

Cette section est structurée de la manière suivante :

- une vue d'ensemble sur la solution logicielle apportée par ArchWare à l'approche d'un processus de développement centré architecture et style,
- l'implémentation d'un outil pour la gestion des styles,
- l'interface utilisateur permettant l'accès à cet outil,
- l'intégration de cet outil dans l'environnement de développement d'ArchWare,
- des expérimentations avec l'outil.

1. Vue d'ensemble

Les styles architecturaux et leurs formalisations apportent de nouveaux concepts pour le développement des logiciels. Cela suscite de nouvelles méthodes et techniques de développement. L'environnement ArchWare propose un ensemble d'outils pour le développement centré architecture et style architectural. Au sein de cet environnement, un outil est spécifique à la gestion des styles architecturaux, l'ASL ToolKit.

Dans cette section nous présentons l'environnement ArchWare, puis l'ASL ToolKit, l'outil associé au langage ASL et à l'utilisation des styles.

1.1. L'environnement ArchWare

Le projet ArchWare [Arc 02] fournit un environnement pour supporter le développement centré architecture dans le cadre des systèmes évolutifs. Les systèmes logiciels évolutifs sont ceux qui sont capables de changements à travers un cycle de vie prolongé avec un impact réduit pour les coûts et la planification, et un impact contrôlé sur la qualité. Le souci principal est de garantir le développement (développement initial et évolution) de logiciels évolutifs, en tenant compte des styles architecturaux spécifiques à un domaine, de la réutilisation de composants existants et de l'évolution de l'exécution du système.

Ce projet a pour objectif de concevoir, développer, et disséminer :

- des langages innovateurs centrés architecture (langage de description d'architecture et langage d'analyse),
- des styles architecturaux,
- des modèles de raffinement,
- des environnements logiciels personnalisables,
- des outils dédiés à l'ingénierie de systèmes logiciels évolutifs.

La nouveauté principale de l'approche du projet ArchWare est sa vue holistique des systèmes logiciels évolutifs. Cela commence par une description à un niveau élevé du système logiciel exprimé en une description formelle de l'architecture. Cette description à un haut niveau peut alors être incrémentalement raffinée dans des descriptions de plus bas niveau jusqu'à atteindre un niveau

concret qui peut alors être employé pour la génération d'une application. Des raffinements sont ainsi appliqués sur les descriptions d'architecture. La description d'une architecture est basée sur des styles architecturaux. Les styles architecturaux sont très puissants du fait qu'ils fournissent une possibilité de réutilisation avec tout le "savoir" des concepteurs ayant été confrontés à des problèmes de conception semblables. Employer des styles architecturaux permet ainsi à un architecte de réutiliser le savoir de la communauté de conception d'architecture [Kle&Kaz 99].

L'environnement est séparé en deux parties (Figure 40) : les *outils architecturaux* et une *plate-forme* ("framework").
Les outils architecturaux fournissent un support pour :

- la description visuelle des architectures par l'application de styles définis. L'outil s'appelle le *Visual Modeller*. La modélisation visuelle des descriptions d'architectures est supportée par des profils UML. Ces profils sont générés selon des styles architecturaux,
- la validation par animation des descriptions architecturales en fonction des cahiers des charges des systèmes. L'outil s'appelle l'*Animator,*
- l'analyse des descriptions d'architectures selon des attributs qualités. Afin de supporter un raisonnement sur les attributs qualités, ArchWare fournit trois outils : Le *Theorem-Prover*, le *Model-Checker*, et le *Model-Specific Evaluator*. Le *Theorem-Prover* permet de vérifier des propriétés sur la structure et sur les données, le *Model-Checker* permet de vérifier des propriétés sur les comportements, et le *Model-Specific Evaluator* permet d'intégrer des analyses propres à des domaines spécifiques,
- le raffinement des descriptions d'architectures depuis un niveau abstrait jusqu'à un niveau concret. L'outil s'appelle le *Refiner*,
- la synthèse de systèmes dans différents langages de programmation pour les utilisateurs finaux. L'outil s'appelle le *Synthetiser*. Cette synthèse est effectuée par génération de code (en utilisant explicitement des règles de synthèse).

La *plate-forme* inclue le moteur pour déclencher des processus de développement évolutionnaires, incluant le raffinement des descriptions d'architectures, et les mécanismes pour supporter l'interopérabilité parmi les outils implémentant la conception et l'analyse. Le coeur de cette plate-forme est la Machine Virtuelle Archware qui représente le système d'exécution pour l'ArchWare ADL (et par conséquent la représentation cible pour les outils). L'interface de la plate-forme est appelée *Tower*. Cette interface permet l'intégration des outils dans l'environnement et offre une interface homme-machine commune.

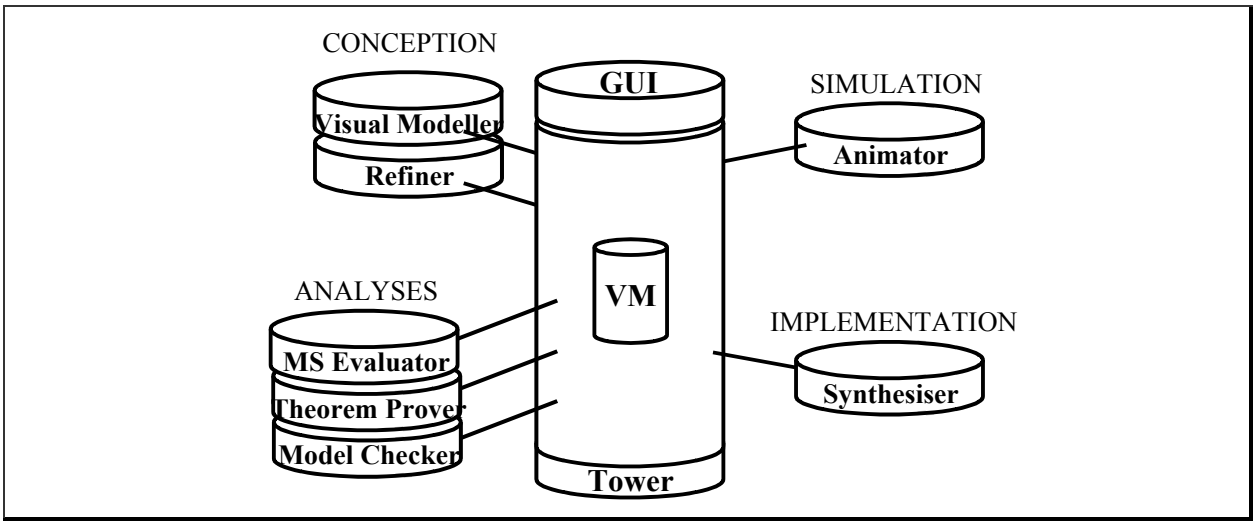


Figure 40 - Environnement ArchWare

Dans la section suivante nous présentons un nouvel outil à intégrer à cet environnement : un outil pour la conception et l'utilisation des styles architecturaux.

1.2. L'ASL ToolKit

Dans un développement de logiciel centré style, les styles sont utilisés à plusieurs reprises (cf. Chapitre 2), ils permettent :

- de définir des architectures d'un domaine spécifique en bénéficiant d'un support de conception adéquat,
- de s'assurer qu'une architecture vérifie des propriétés adéquates,
- d'appliquer des analyses spécifiques sur une architecture.

L'ASL ToolKit répond à cette approche.

L'ASL ToolKit a été implémenté pour donner un support logiciel aux nouvelles techniques de conception liées à l'utilisation du langage ASL. Ainsi, il permet de gérer et d'utiliser les descriptions de styles architecturaux. Il permet :

- la **compilation** des descriptions de style. C'est une étape précédant toute utilisation des styles. Il faut notamment compiler un style avant de pouvoir y faire référence dans d'autres descriptions,
- l'**instanciation** des styles. Ce service permet de générer une architecture définie en ArchWare ADL à partir d'un style,
- la **vérification de satisfaction** d'une architecture à un style. Ce service permet de vérifier en quelle mesure une architecture suit un style,
- l'**analyse**. Ce service permet d'appliquer des analyses proposées dans un style sur des architectures.

Ces services sont placés ci-dessous dans la Figure 41 par rapport à leur utilisation dans le processus de développement.

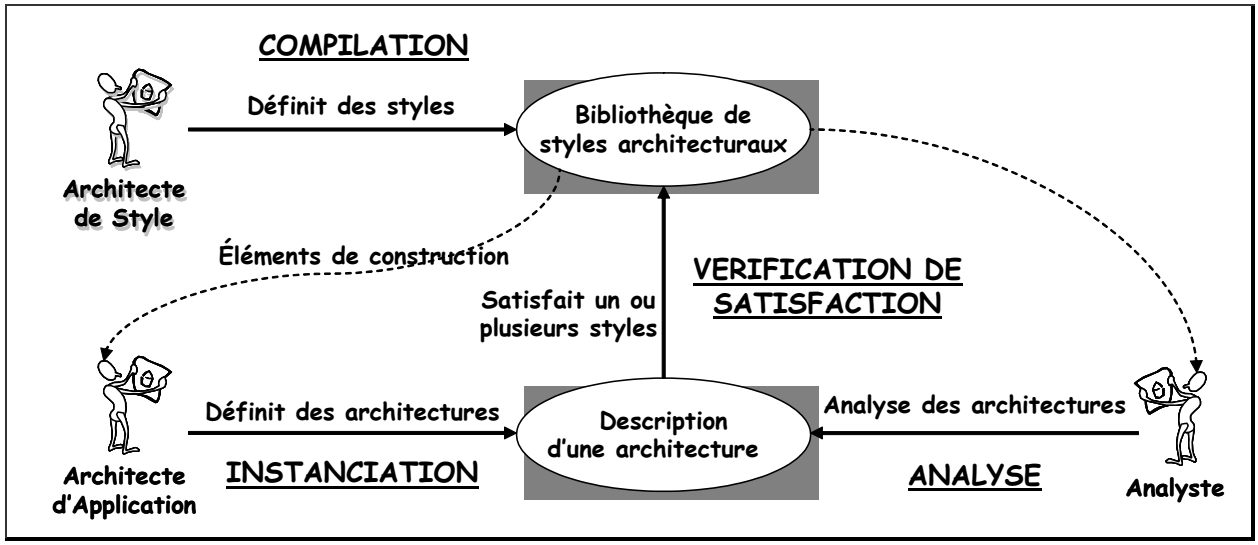


Figure 41 – Utilisation des styles

Dans l'environnement ArchWare, l'ASL ToolKit est un "front-end" pour l'utilisateur. D'une part, il est utilisé à la conception des styles architecturaux, c'est à dire dès les premières étapes d'un développement centré styles architecturaux. D'autre part, il offre à l'architecte une interface pour travailler avec des concepts de haut niveau, propres à un domaine particulier, tout en bénéficiant des autres outils adaptés à des concepts de bas niveau.

La prochaine section donne des informations concernant l'implémentation de l'ASL ToolKit.

2. Implémentation

Lors du développement associé à l'ASL ToolKit, plusieurs technologies et plusieurs langages ont été utilisés. Dans cette section, nous étudions l'implémentation du *noyau de l'outil*, puis nous présentons sa façade (wrapper) qui permet notamment son intégration dans l'environnement ArchWare.

2.1. Noyau

Le noyau de l'outil a été implémenté en utilisant le langage Prolog. Dans une première partie, nous présentons des modules que nous avons définis et qui servent de composants de base à la définition des services. Puis, nous présentons l'implémentation des services.

2.1.1 Modules

Afin d'implémenter les services de l'ASL ToolKit, nous avons d'abord défini plusieurs modules. Nous avons fait cela, d'une part pour structurer l'implémentation, d'autre part parce que de nombreux modules sont communs à plusieurs services.
Nous avons défini les modules principaux suivants[45]: *scanner*, *parser*, *typechecker*, *architecturegenerator*.

Scanner

Le module *scanner* fournit les mécanismes pour transformer une description sous forme de chaîne de caractères en informations pertinentes pour la compilation. Ces mécanismes sont les suivants :

- l'élimination des commentaires d'une description,
- la transformation d'une description en liste de mots appelés *tokens*,
- l'ajout d'informations concernant la place d'un mot (token) dans la description en termes de ligne et de position sur la ligne.

Parsers

Nous définissons plusieurs modules liés à l'analyse syntaxique. Le premier module, appelé *parser*, fournit le moteur de l'analyse syntaxique. Les autres modules (*parserdcg_asl*, *parserdcg_adl*, *parserdcg_aal*) fournissent les descriptions en DCG du langage. L'analyse syntaxique d'une liste de tokens retourne cette liste sous forme d'un *arbre syntaxique* dans le cas d'un succès.

L'arbre syntaxique généré par l'analyse syntaxique est une représentation en Prolog de la description interne. Dans cette représentation chaque expression est représentée par un mot-clé et par des arguments. Par exemple*, add(Arg1, Arg2)* représente une somme. Les arguments peuvent être eux-mêmes des expressions. Par exemple, *Arg1* peut être la valeur 1 et *Arg2* l'expression *2*3*. Ainsi l'expression *1+2*3* est représentée par *add(1,mul(2,3))*. La représentation graphique de son arbre syntaxique est montrée ci-dessous.

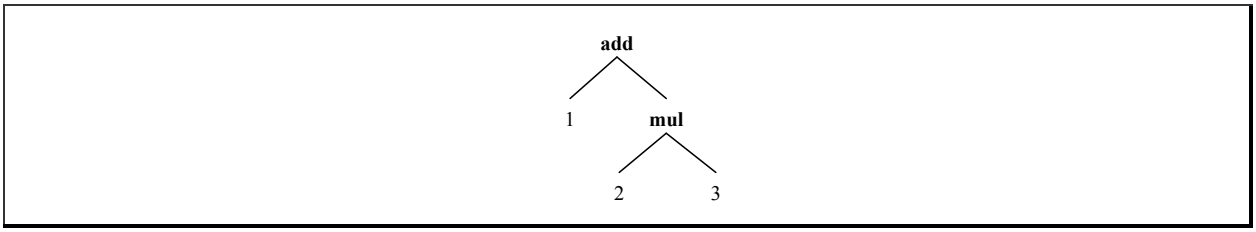


Figure 42 - Arbre syntaxique correspondant à l'expression 1+2*3

Au niveau du module *parser*, nous avons défini des mécanismes d'extension à la DCG. Nous avons optimisé le traitement d'une description par DCG pour qu'il consomme moins de ressources. En effet, l'analyse syntaxique par une DCG classique consomme beaucoup de mémoire, car à chaque prédicat, la description est presque entièrement dupliquée en mémoire. De notre côté, nous avons géré cela par des passages de références. Nous avons aussi défini des mécanismes pour la gestion des erreurs. Lorsqu'une erreur est trouvée, l'analyseur retourne la ligne de l'erreur, une position relative de l'erreur dans la ligne, et propose une correction probable.

[45] En prolog, le nom des modules est donné en minuscule.

Les modules *parserdcg_asl*, *parserdcg_adl*, *parserdcg_aal* définissent les règles de syntaxe pour l'ASL. Chacun de ces modules est dédié à une partie précise ; un pour la structure propre au style (*parserdcg_asl*), un pour les descriptions basées sur ArchWare ADL (*parserdcg_adl*), un pour les descriptions basées sur ArchWare AAL (*parserdcg_aal*).

TypeChecker

Le module *typechecker* est un module lié à l'analyse sémantique d'une description. Il donne les mécanismes permettant de vérifier que les règles de typage du langage (cf. annexe 4) sont vérifiées.
De la même manière qu'on utilise les DCGs pour l'analyse syntaxique, nous adoptons une manière spécifique de noter les règles de typage en Prolog.

Une règle de typage se vérifie sur une expression. Par exemple, l'expression peut être une somme :

```
1 + 2
```

Le mécanisme de typage permet de déterminer le type d'une expression et de retourner une erreur si les règles de typage ne sont pas respectées.

Une règle de typage est construite de la manière suivante. Elle est séparée en une partie *prémisse* et une partie *conclusion*. Lorsque la partie prémisse est vérifiée, la partie conclusion est vérifiée. Ci-dessous se trouve, la règle de typage pour la somme.

T' ∈ { **Natural**, **Integer**, **Real** }

$$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 + e_2 : T'}$$

Pour représenter une règle de typage en Prolog, on procède comme suit. On identifie d'abord la nature de l'expression (depuis l'arbre syntaxique) à partir du mot-clé. Ensuite, on décrit les conditions sur les arguments. Il s'agit généralement de condition sur leurs types. Il peut s'agir aussi, d'autres conditions comme le fait qu'un identifiant soit déjà déclaré. Si les conditions sur les arguments ne sont pas respectées alors, on peut envoyer un message d'erreur. Sinon, on renvoie le type de l'expression tel qu'il est spécifié dans les règles de typage.

Pour illustrer ceci, définissons la règle de typage pour la somme en Prolog. La somme est référencée par le mot-clé *add*. Les conditions sont les suivantes :

- le type de e1 doit être aussi celui de e2,
- le type de e1 et de e2 doit être soit ***Natural***, soit ***Integer***, soit ***Real***.

Si ces conditions sont réunies alors, le type de l'expression est le type de e_1 et de e_2.

L'expression suivante décrit le mécanisme de vérification associé à cette règle de typage pour la somme.

```
  typecheck(EXPRESSION,TYPE):-
        EXPRESSION = add(ARG1,ARG2),
        typecheck (ARG1,TYPE1), typecheck (ARG2,TYPE2),
        (type_nir(TYPE1); writeline(['Wrong type :',TYPE1]) ,fail),!,
        (type_nir(TYPE2); writeline(['Wrong type :',TYPE2]) ,fail),!,
        (TYPE1=TYPE2;  writeline(['Type  mismatch  in  +  :',TYPE1,'  different  from
',TYPE2]),fail),!,
        TYPE=TYPE1
```

ArchitectureGenerator

Le module *architecturegenerator* permet de générer du code ArchWare ADL à partir d'une expression en ASL. Il contient notamment le mécanisme d'instanciation, permettant de générer du code ArchWare ADL à partir de l'application d'un constructeur.

Le mécanisme de génération d'architecture est un mécanisme implémenté similairement au mécanisme de vérification des règles de typage concernant le parcours de l'arbre syntaxique. Pour chaque expression en ASL, on définit la correspondance en ArchWare ADL. L'application d'un constructeur est le mécanisme qui demande un traitement un peu spécifique car il faut retrouver la définition du constructeur et générer le résultat de son application en fonction des paramètres d'entrée.

2.1.2 Services

L'ASL ToolKit offre plusieurs services pour : la compilation, la vérification de satisfaction, l'instanciation et les analyses. Des modules en Prolog sont définis pour chacun des services. Ces modules sont basés sur ceux vus précédemment.

Compilation

Le déroulement de la compilation est le suivant (cf. Figure 43). La compilation commence par la lecture d'une description depuis un fichier. Ensuite cette description est traitée pour la transformer en une liste de tokens. Puis, cette liste est traitée par analyse syntaxique, ce qui retourne un arbre syntaxique dans le cas d'un succès ou une liste d'erreurs sinon. L'arbre syntaxique est alors traité pour vérifier la conformité aux règles de typage et pour vérifier la cohérence. Enfin, si la description est jugée correcte après ces traitements, le style est enregistré sous sa forme d'arbre syntaxique.

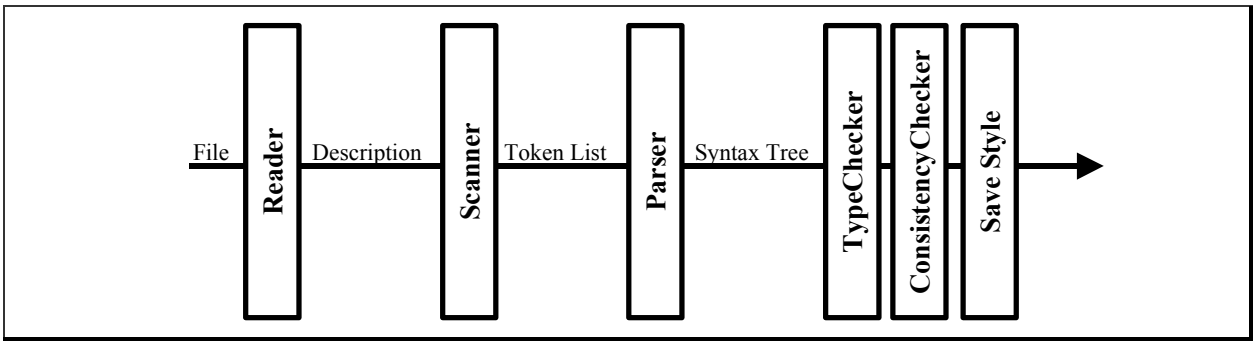


Figure 43 – Processus de la compilation

En Prolog ceci est défini de la manière suivante.

```
compiler(FILE) :-
     %getting an ASCII code list from a text file
     reader(FILE,Code),
     %getting an enhanced token list from an ASCII code list
     scanner(Code,Tokens,LineTokens),
          writeln('The code has been scanned'),!,
     %parsing the code
     parser(LineTokens,Tree),
          writeln('The code has been parsed :'),!,
     %checking typing rules
     type_checker(Tree),
          writeln('Typing rules has been checked'),!,

     %checking typing rules
     consistency_checker(Tree),
          writeln('Consistency has been checked'),!,
     save_style.
```

En réalité, la définition de la compilation est plus compliquée que cela. Le compilateur compile les descriptions des constructeurs, des contraintes et des analyses séparément, mais le principe reste le même.

Pour la *représentation d'un style*, nous utilisons le système de fichiers et Prolog. Chaque style est représenté par un dossier (répertoire) au sein duquel se trouvent les dossiers représentant les styles agrégats et un fichier, nommé *style*, qui contient les informations relatives au style. Nous verrons un exemple concret d'un fichier *style* dans la section 5.2.3. Si on prend l'exemple du style *ClientServer* (cf. Chapitre 5), celui ci est représenté par un dossier du même nom et qui contient un dossier *Client*, un dossier *Server*, un dossier *PC*, et un dossier *RPC*. Ainsi, les représentations des styles sont hiérarchisées.

Vérification de satisfaction

L'ASL ToolKit permet de vérifier qu'une architecture satisfait un style. Nous rappelons qu'une architecture satisfait un style lorsque l'architecture vérifie chacune des contraintes du style.

Pour vérifier la satisfaction d'une architecture, les contraintes sont traitées comme des analyses. L'ASL ToolKit transforme donc chacune des descriptions de contraintes en descriptions d'analyses. Il gère ensuite l'ensemble des résultats fournis pour chacune des contraintes, et génère un bilan à l'utilisateur. Le bilan retourné est la liste de contraintes dans laquelle les contraintes satisfaites sont différenciées des contraintes non satisfaites. Ainsi, l'architecte peut savoir avec précision pourquoi son architecture ne suit pas le style, et il peut décider ou non de garder son architecture telle quelle ou de la modifier pour satisfaire d'autres contraintes.

Instanciation

L'ASL ToolKit permet l'instanciation des styles architecturaux ; en d'autres termes, il permet la génération de descriptions d'architectures satisfaisant un style à partir de la description de ce style.

L'instanciation consiste en l'application totale d'un constructeur d'architectures (cf. Chapitre 4). A l'instanciation, on spécifie le style à instancier ainsi qu'une liste de paramètres. Le constructeur d'architectures à appliquer est sélectionné en fonction du *nombre* et de la *nature des paramètres*.

Lors de l'instanciation :

- les paramètres sont compilés,
  - Ceux-ci sont décrits en ASL. Leurs descriptions peuvent être basées sur la syntaxe proposée par le style qu'on instancie. Par exemple, en supposant qu'on instancie un style Client-Serveur, les paramètres pourraient être des clients et des serveurs décrits suivant la syntaxe vue dans le Chapitre 5.
- du code ArchWare ADL est généré,
  - Chaque application de constructeur (qu'elle soit au niveau des paramètres, ou au niveau du constructeur d'architecture qu'on instancie) conduit à la génération de code en ArchWare ADL. L'architecture est générée en ArchWare ADL afin de pouvoir bénéficier des outils associés à ce langage. C'est aussi sous cette forme qu'une architecture peut être utilisée pour vérifier la satisfaction à un style ou pour appliquer des analyses.
- la satisfaction de l'architecture générée au style est vérifiée.
  - Cette vérification conduit à un fichier log permettant à l'architecte de vérifier que l'architecture instanciée suit bien le style.

Le processus de l'instanciateur est schématisé dans la Figure 44.

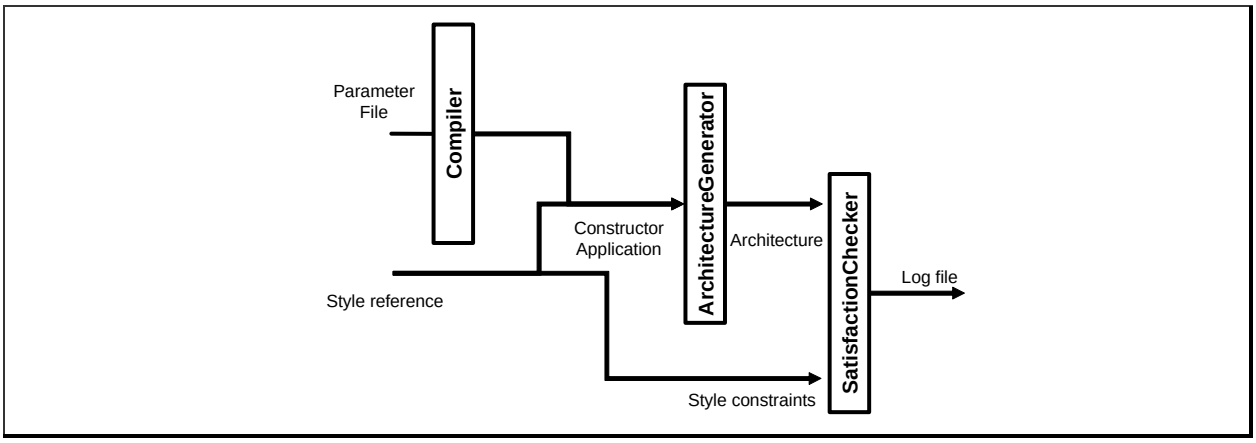


Figure 44 - Processus de l'instanciation

Dans la section suivante, nous présentons en détail le traitement des analyses.

Analyse

L'ASL ToolKit permet d'appliquer à une architecture les analyses fournies par un style.

Pour effectuer une analyse, l'utilisateur fournit une architecture définie en ArchWare ADL, choisit un style, précise l'analyse du style à appliquer, puis fournit des paramètres pour configurer l'analyse si nécessaire.

Pour traiter une analyse sur une architecture, l'ASL ToolKit fait appel aux outils d'analyse. L'outil *Evaluator* est le front-end de tous les outils d'analyses. Notre outil fait appel à cet outil en envoyant l'analyse à traiter, l'architecture et des paramètres. Ensuite, il récupère le résultat des analyses. Le seul traitement effectué lors de l'analyse est de transformer les descriptions d'analyses en des informations traitables par l'Evaluator.

2.2. Wrapper (façade)

Nous avons présenté l'implémentation du noyau de l'ASL ToolKit. Afin de pouvoir facilement intégrer ce noyau dans un environnement et l'interfacer par un module graphique, le noyau est encapsulé dans un wrapper défini en Java.

Le Wrapper est une classe Java proposant quatre méthodes associées à chacun des services :

`public void compile(String style_file),`
: méthode d'accès à la compilation.

`public void instantiate(String parameters_file, String style_reference, String architecture_name),`
: méthode d'accès à l'instanciation.

`public void check_satisfaction(String architecture_file, String style_reference),`
: méthode d'accès à la vérification de satisfaction.

`public void analyse(String architecture_file, String style_reference, String parameters_file).`
: méthode d'accès à l'analyse.

3. Utilisation du système

L'ASL ToolKit est développé pour être accessible. Premièrement, il propose plusieurs types d'accès soit par ligne de commande, soit par interface graphique. De plus, il peut être utilisé soit localement, soit à distance en fonctionnant comme un service Web. Enfin, il fonctionne à la fois sous Windows et sous Linux.

3.1. Le système de répertoire

Une fois l'ASL ToolKit installé, le répertoire principal contient plusieurs sous-répertoires dont nous présentons ici leurs raisons d'être :

- ***bin*** contient les scripts pour appeler les services de l'ASL ToolKit et pour lancer les applications graphiques,
- ***lib*** contient les exécutables en Prolog,
- ***adl_files*** est le répertoire par défaut contenant les architectures générées,
- ***source_files*** est le répertoire par défaut contenant les descriptions de styles,
- ***style_representations*** contient les représentations des styles après compilation.

3.2. Interface utilisateur

L'ASL ToolKit offre deux types d'interface pour les utilisateurs : la console (shell) et une interface graphique.

3.2.1 Console

L'ASL ToolKit fournit des commandes pour la console *Dos* et pour la console *Bash* (système Unix). Nous avons défini une commande par service.

Compilateur

La commande pour la compilation est *customiser_compilation*.

```
customiser_compilation [options] source_file [representation_path]
```

Le paramètre ***source_file*** est l'adresse du fichier contenant la description qu'on veut compiler. Le paramètre ***representation_path*** indique le chemin désignant le répertoire contenant la base de styles (par défaut c'est le répertoire ./style_representation).
Les ***options*** sont les suivantes :

- `-logpath path` permet de spécifier le dossier où le fichier log peut être retrouvé,
- `-nowarn` permet de cacher les messages d'alertes,
- `-verbose` permet d'obtenir des informations sur le fichier source,
- `-version` permet d'obtenir des informations sur l'outil.

La figure suivante montre la capture d'écran de l'exécution d'une compilation sur une console DOS. Le fichier compilé est le fichier *ClientServer.adl* qui contient la description du style Client-Serveur. A l'exécution, la compilation est effectuée par l'appel au prédicat compile (cf. `compile('..\source_files\ClientServer.Adl',"..\\style_representations",'.\',yes,non,;no)`). Sur la fin de la capture, on peut voir des informations concernant le traitement d'une description brute et l'analyse syntaxique.

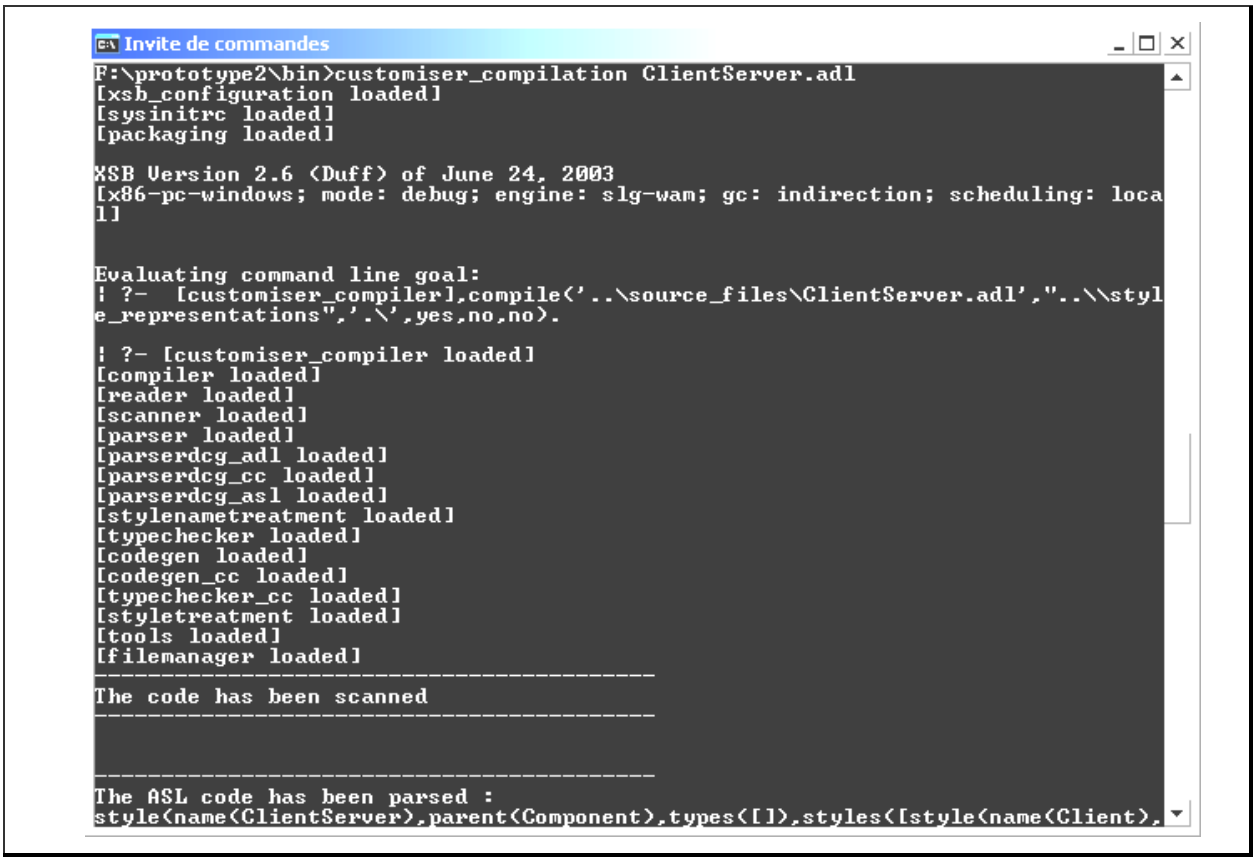


Figure 45 - Capture d'écran de l'exécution de la compilation sur une console DOS

Instantiateur

La commande pour l'instanciation est *customiser_instantiation*.

```
customiser_instantiation [options] style_reference ADL_file [parameter_file]
```

Le paramètre ***style_reference*** est la référence du style (le nom du style, par exemple : ClientServer). Le paramètre ***ADL_file*** est le nom du fichier dans lequel sera générée la description d'architecture. Le paramètre ***parameter_file*** est le nom du fichier dans lequel les paramètres sont définis.

Vérification de satisfaction

La commande pour la vérification de satisfaction est *customiser_satisfaction_checking*.

```
customiser_satisfaction_checking [options] style_reference ADL_file
```

Le paramètre ***style_reference*** est la référence du style (le nom du style). Le paramètre ***ADL_file*** est le nom du fichier contenant la description de l'architecture à examiner.

Analyseur

La commande pour la vérification de satisfaction est *customiser_analyse*.

```
customiser_analyse [options] style_reference ADL_file analyse_reference
[parameter_file]
```

Le paramètre ***style_reference*** est la référence du style (le nom du style). Le paramètre ***ADL_file*** est le nom du fichier contenant la description de l'architecture à analyser. Le paramètre ***analyse_reference*** est le nom de l'analyse qu'on veut appliquer. Le paramètre ***parameter_file*** est le nom du fichier dans lequel les paramètres sont définis.

3.2.2 Interface graphique

L'ASL ToolKit est défini avec une interface graphique offrant des facilités dans l'utilisation de l'outil. Cette interface est en fait composée de plusieurs interfaces graphiques, une pour chaque service.

Ces interfaces graphiques (implémentées en Java) offrent des commodités pour donner les paramètres. Par exemple, elles ouvrent un explorateur pour choisir les fichiers et les répertoires qu'on veut passer en paramètre. Elles affichent aussi, une fenêtre rendant compte du déroulement du traitement et des résultats obtenus.

Nous montrons ces interfaces et leur utilisation dans la partie dédiée à l'utilisation de l'outil.

4. Intégration

L'ASL ToolKit est un outil parmi d'autres dans un processus de développement centré architecture. L'intégration des outils dans un même et unique environnement offre un équipement complet pour le développement, une interface commune, et la possibilité de formaliser et d'automatiser le processus de développement.

De plus, l'intégration dans l'environnement ArchWare de l'ASL ToolKit est nécessaire pour qu'il fonctionne pleinement. En effet, certains services de l'ASL ToolKit reposent sur l'utilisation d'autres outils de l'environnement. C'est le cas pour la vérification de la satisfaction et l'analyse qui sont des services reposant sur l'utilisation du *Theorem Prover*, du *Model Checker* et du *Model Specific Evaluator*.

Comme nous l'avons vu, le cœur de l'environnement est appelé *Tower*. C'est un gestionnaire de processus qui interface la machine virtuelle permettant leur exécution. Il est destiné à gérer un ensemble d'outil constituant l'environnement de développement et à exécuter des modèles de processus représentant des scénarios qui mettent en œuvre les outils d'ArchWare pour le développement d'applications. C'est aussi un outil de navigation permettant de définir des structures de données. Ces données peuvent être des applications ou des descriptions. Les structures de données sont un ensemble de nœuds reliés par des relations diverses. Chaque nœud représente un lieu de stockage pour une donnée.

Les outils sont tous connectés autour de la *Tower* de la même manière, par le biais d'un connecteur propre à chacun appelé *TC* - Transformer Connector. Ce connecteur transite les informations entre la Tower et un outil en transformant le format de cette information. En effet, la Tower ne traite que des informations décrites en ArchWare ADL tandis que les outils peuvent traiter d'autres formats d'information.

Concernant l'ASL ToolKit, il est intégré plus spécifiquement comme le présente l'architecture suivante.

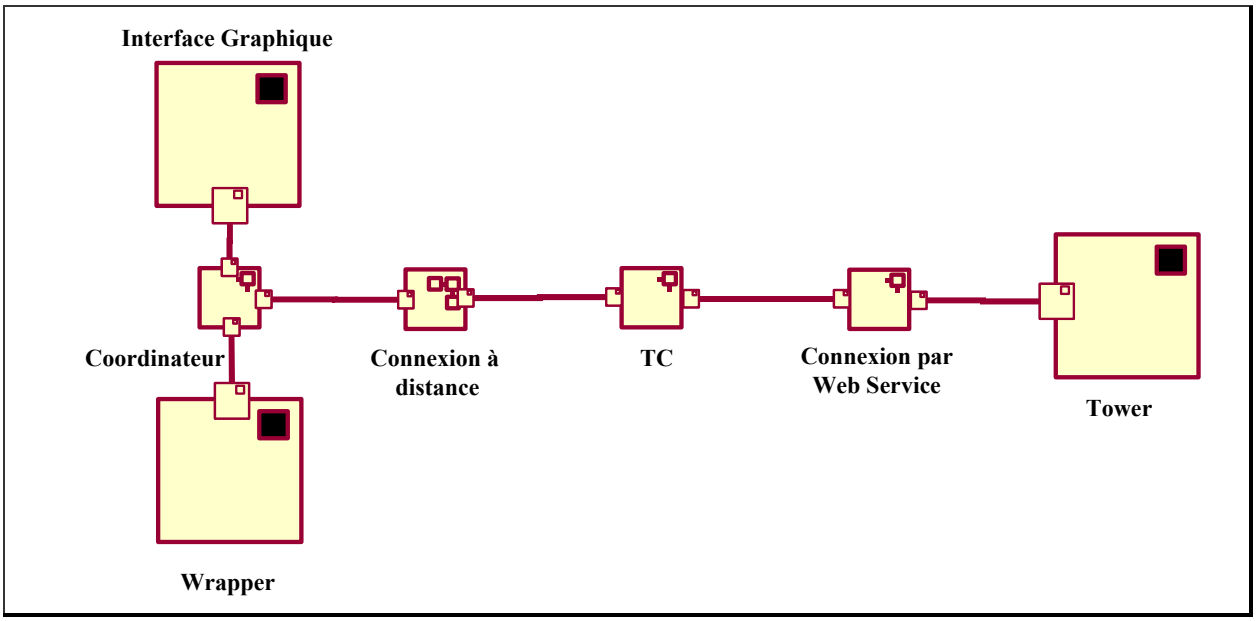


Figure 46 - Architecture de l'intégration de l'ASL ToolKit dans l'environnement ArchWare

L'ASL ToolKit présente ses services à la Tower par un connecteur, *Coordinateur*, qui aiguille les messages entre la Tower, *l'Interface Graphique* et le *Wrapper*. Lorsque la Tower déclenche l'ASL ToolKit, l'interface graphique de celui-ci s'ouvre et déclenche un service en fonction des manipulations de l'utilisateur.

Le *Coordinateur* peut être relié directement au TC ou bien être connecté par un connecteur gérant la connection à distance (cf. Figure 47). La présence de ce connecteur permet d'utiliser l'ASL ToolKit sur une autre machine que sur celle ou la Tower s'exécute. Ce connecteur est défini, par deux composants, un *Client* et un *Serveur*, qui communique ensemble via des *sockets*.

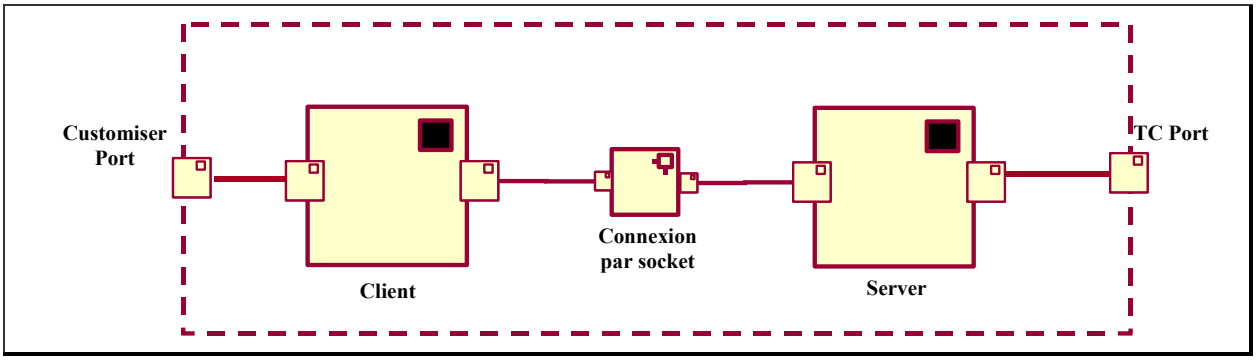


Figure 47 - Détail du connecteur pour la connexion à distance

5. Expérimentation

Pour présenter l'utilisation de l'ASL ToolKit, nous proposons un exemple de scénario concernant le développement d'une application Client-Serveur.

5.1. Description du scénario

Le scénario concerne le développement d'une application Client-Serveur dans laquelle on veut s'assurer que les Clients sont créés dynamiquement sans que leur quantité n'excède 1000 unités.

Ainsi, nous choisissons de définir un sous-style du style Client-Server pour les architectures où la création des Clients est dynamique. Nous appelons ce style *Dynamic_CS*.

Afin de valider la description du style et afin de l'enregistrer dans la base des styles, celui-ci est compilé.

Ensuite, nous instancions une architecture, *Dynamic_CS_Architecture*, définie à partir du style Client-Server. Cette architecture est dynamique : les Clients sont créés dynamiquement. Mais le nombre de Clients créés n'est pas limité. Ainsi, l'outil retourne un fichier log stipulant qu'une contrainte n'est pas vérifiée.

L'architecte décide d'utiliser une analyse pour vérifier combien de Clients peuvent être créés au maximum afin de faire son choix quant à garder son architecture telle quelle ou la modifier pour qu'elle satisfasse la contrainte non satisfaite.

5.2. Détail

Dans cette section, nous donnons en détail le déroulement du scénario.

5.2.1 Description du style

La description suivante est celle du style *Dynamic_CS*. Ce style hérite du style *ClientServer*. Il définit deux contraintes :

- *dynamic_creation* est une contrainte imposant la création dynamique au sein du système,
- *max_clients_1000* est une contrainte imposant le nombre de clients à être inférieur à 1000.

Il définit une analyse spécifique au style et implémentée dans un langage autre (dans le fichier *client_amount_quantification.rc*) : *client_amount_quantification*. Cette analyse évalue le nombre maximum de clients qui peuvent être créés au cours du temps.

```
Dynamic_CS is style extending Client_Server where {
constraint {
     dynamic_creation is constraint {
          to styleInstance.behaviour apply {
               everysequence { true*. not new(any) } leads {false}
          }
     }
     max_clients_1000 is constraint {
          to styleInstance.elements apply {
               forall(c| c in style Client and
                    apply on client_set union c.occurences and
                    client_set.size<=1000)
          }
     }
}
analyses{
     client_amount_quantification is analysis {
          kind Dynamic_CS
          input {}
          output { Integer }
          body {
               "client_amount_quantification.rc"
          }
     }
}
}
```

5.2.2 Définition d'une architecture

Un style peut représenter une architecture. La description suivante est la description d'une architecture. Nous la définissons comme un constructeur d'architectures au sein d'un style *Dynamic_CS_Architecture* pour bénéficier des facilités syntaxiques apportées par le style *Client_Server*. En instanciant le style, on créé une occurrence de l'architecture.
Dans cette architecture, un chorégraphe est défini. Il génère dynamiquement et en continu des clients qu'il attache à un serveur crée à l'initialisation.

```
Dynamic_CS_Architecture is style extending Dynamic_CS where {
constructors {
     Dynamic_CS_Architecture is client_server with {
          -- définition des clients. On définit un client basique.
          clients{
               Client is basicclient {
                    via p_out send;
                    via p_in receive;
                    unobservable
               }
          }
          -- définition des serveurs. On définit un serveur basique.
          servers{
               Server is basicserver {
                    via p_in receive;
                    unobservable;
                    via p_out send;
                    recurse
               }
          }
          -- définition du chorégraphe. Celui-ci définit la création dynamique des clients.
          choreographer {
               new Client;
               attach Client#last~p to Server~p;
               recurse
          }
          -- définition de la configuration initiale. Il n'y a qu'un serveur.
          configuration {
               new Server
          }
     }
}
}
```

5.2.3 Compilation

Afin de valider les descriptions précédentes et afin de les enregistrer dans la base des styles de l'ASL ToolKit, celles-ci sont compilées.

La commande suivante permet de compiler le style *Dynamic_CS* défini dans le fichier *Dynamic_CS.stl*.

```
customiser_compilation Dynamic_CS.stl
```

Après la compilation, la représentation du style est stockée. La description qui suit est celle contenue dans le fichier *style* correspondant au style *Dynamic_CS*. Cette description est *un fait* défini en Prolog. Elle est stockée dans le dossier *style_representation\Dynamic_CS*.

```
style(
     name(['Dynamic_CS']),     %style name
```

```
parent('ClientServer'),    %style parent
types([]),          %types defined in the style : none here
styles([]), %aggregate styles
constructors([])                              %constructors
constraints([
     constraint(name(''),kind(),…
          body('')]
),              %style constraints
analyses([])).                  %style analyses
```

5.2.4 Instantiation

L'instanciation permet de générer la description de l'architecture en ArchWare ADL. En ouvrant l'interface graphique associée à l'instanciation, on a à disposition une fenêtre permettant de choisir le style à instancier.

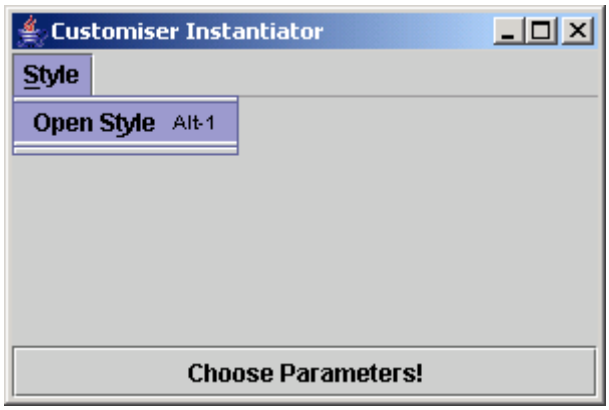


Figure 48 - Interface graphique - Choisir un style

Une fois le style choisi, la liste des constructeurs d'architectures disponibles pour l'instanciation est affichée. Dans notre cas, il n'y a qu'un constructeur sans paramètre. La figure suivante montre un cas où il y a plusieurs constructeurs avec paramètres.

Figure 49 - Interface graphique - Choisir un constructeur

Une fois le constructeur choisi, il faut définir les paramètres. Une fenêtre s'ouvre et propose, pour chacun des paramètres :

- soit de définir une valeur,
- soit de choisir un fichier contenant la description d'une valeur.

Figure 50 - Interface graphique - Définir la valeur des paramètres

Enfin, on peut lancer l'instanciation. Une dernière fenêtre propose de donner le nom du fichier qui contiendra la description d'architecture générée.

Figure 51 - Interface graphique - Choix du nom de l'architecture

Une fois le bouton Ok pressé, un fichier unique pour les paramètres est créé et passé en paramètre de commande de l'instanciation avec le reste des informations. La description suivante est générée et stockée dans le fichier *client-server.adl*. Nous en donnons une partie en mettant en relief des informations qui rappelle la description initiale.

```
…
abstraction ( ) ;
{
value ports is sequence (
       view (
              name is "p" ,
              producer is function(); {
                     value port is view (
                            connections is sequence (
                                   view (
                                          name is "in" ,
                                          producer is function(); {
                                                 any ( connection ( String )  )
                                          },
                                          instances is nilsequence(Any)
                                   ),
                                   view (
                                          name is "out" ,
                                          producer is function(); {
                                                 any ( connection ( String )  )
                                          },
                                          instances is nilsequence(Any)
                                   )
                            )
                     );
                     iterate ports by meta_port:MetaPort
        do if (meta_port::name== "in" ) do meta_port::instances includes
              any(meta_port::producer( "null" )) ;
                     iterate ports by meta_port:MetaPort
        do if (meta_port::name== "out" ) do meta_port::instances includes
              any(meta_port::producer( "null" )) ;
                     port
              }
       )
);
value attributes is nilsequence(Attribute) ;
value attachments is nilsequence(Attachment) ;
iterate ports by meta_port:MetaPort
        do if (meta_port::name== "p" ) do meta_port::instances includes
              any(meta_port::producer( "null" )) ;
```

```
value component is abstraction (_ports:MetaPorts , _attributes:Attributes, _attachments:Attachments) {
            via p_out send;
      unobservable
      done ;

};
component(ports,attributes,attachments)
}
…
```

De plus, l'instanciation génère un fichier log à partir duquel l'utilisateur peut se rendre compte que la contrainte *max_clients_1000* n'est pas vérifiée. Les lignes suivantes sont issues du fichier log généré. Les contraintes satisfaites y sont présentées séparément des contraintes insatisfaites. De plus, le style définissant ces contraintes est spécifié (c'est le style *Dynamic_CS* pour les deux contraintes).

```
INSTANTIATION
      style : Dynamic_CS_Architecture
      parameters : none
      architecture : client-server
…
CONSTRAINT_SATISFACTION
      SATISFIED CONSTRAINTS
            Dynamic_CS : dynamic_creation
      UNSATISFIED CONSTRAINTS
            Dynamic_CS : max_clients_1000
…
```

Devant ce fait, l'architecte peut, soit revoir son architecture, soit faire des analyses pour quantifier le nombre de clients qui peuvent être créés à l'exécution afin de pouvoir prendre une décision quand à garder son architecture telle quelle.

5.2.5 Analyses

En utilisant l'analyse *client_amount_quantification*, l'utilisateur peut se rendre compte qu'une infinité de Clients peuvent être créés. Les lignes suivantes montre le fichier log généré après l'analyse. L'analyse renvoi la valeur -1 pour spécifier que le nombre de clients pouvant être créés est infini.

```
ANALYSE
      style : Dynamic_CS
      architecture : client-server.adl
      analyse : client_amount_quantification
      input : none
      output : -1
```

Devant ce fait, l'architecte aura certainement choisi de revoir son architecture pour que celle-ci limite le nombre de clients créés à l'exécution.

6. Conclusion

L'ASL ToolKit est un outil intégré dans l'environnement ArchWare pour supporter les développements centrés architecture. C'est l'outil associé au langage ASL et à l'utilisation des styles. Il fournit quatre services : la compilation, l'instanciation, la vérification de satisfaction, l'analyse.

6.1. Les technologies

L'implémentation de l'ASL ToolKit a demandé l'emploi de plusieurs technologies : les langages Prolog et Java, des APIs spécifiques à l'interaction entre une application Java et une application Prolog, les Services Web via Axis et le serveur web Tomcat.

6.2. Les chiffres

En terme de chiffres le travail sur l'implémentation de l'ASL ToolKit représente :

- 6 classes Java et 10 modules Prolog,
    - soit environ 7000 lignes de code,
- une documentation d'une soixantaine de pages.

6.3. Perspectives

Concernant, l'implémentation de l'outil, des améliorations et des extensions pourraient être apportées. Parmi les améliorations, on peut noter qu'à la compilation les descriptions d'analyses spécifiques (dont la syntaxe n'est pas connue au sein d'ArchWare) ne sont pas examinées. L'outil pourrait disposer d'une base dans laquelle on implanterait les BNFs et règles de typage propres à des formalismes nouveaux. Ces informations serviraient de support à l'outil pour la compilation des analyses.
D'autre part, nous avons remarqué que les descriptions d'architectures sont encapsulées dans des descriptions de styles. On pourrait donc parfaire l'outil et de même le langage ASL en permettant de distinguer les descriptions d'architectures des descriptions de styles.
Un autre point d'amélioration serait de rendre l'ASL ToolKit complètement indépendant de l'environnement ArchWare. Cela consisterait à laisser la possibilité d'y connecter des outils d'analyse directement. Pour réaliser cela, il faudrait intégrer les fonctionnalités de l'outil Evaluator au sein de l'ASL ToolKit.
Enfin, pour un outil plus abouti, on pourrait l'agrémenter de fonctionnalités offrant un support documentaire à l'architecte. On pourrait, par exemple, fournir un document après compilation, qui fournirait des informations sur l'utilisation du style et une définition de la notation qu'il fournit.

Dans le chapitre suivant, on montre l'utilisation d'ASL et de l'ASL ToolKit sur des scénarios concrets. Ce chapitre décrit l'utilisation du langage ASL, du style Composant-Connecteur et de l'ASL ToolKit sur trois cas industriels. Ces trois cas présentent des scénarios d'utilisation différents.

Chapitre 7

VALIDATION

Chapitre 7 - Validation

Le langage ASL (chapitre 4), le style Composant-Connecteur (chapitre 5) et l'outil ASL ToolKit (chapitre 6) ont été conçus pour supporter le développement de logiciels en s'appuyant sur les concepts d'architecture et de style architectural. La particularité de nos travaux est de pouvoir supporter les systèmes dynamiques.

A ce stade, nous n'avons pas montré que nos travaux étaient utilisables et utilisés dans des cas d'études autres que les exemples que nous avons nous même fournis.

Dans ce chapitre, nous présentons l'utilisation des résultats de ces travaux dans plusieurs cas industriels[46]. Chacun de ces cas présente le développement de systèmes différents et des scénarios différents. Ces cas ont été développés dans des contextes particuliers que nous expliciterons dans le détail. Nous évoquons aussi quelles ont été les interactions avec les acteurs sur ces travaux. D'une manière générale, ceux-ci ont été menés en parallèle aux nôtres, et beaucoup de nos résultats intermédiaires ont servi de supports. Ainsi, il y a eu beaucoup d'interaction et de nombreux retours qui ont influé l'évolution de nos travaux.

Le premier cas concerne la conception d'un système de gestion des connaissances. Ce cas montre l'utilisation des styles architecturaux en vue de définir l'environnement de conception et de maintenance d'un système à base de connaissances. Les interactions se sont déroulées la plupart du temps à distance.

Le second cas concerne la conception d'un système pour la génération d'environnements pour le développement de systèmes d'Intégration d'Applications Industrielles. Ce cas montre l'utilisation des styles architecturaux en vue de définir une famille d'environnements propre au domaine des EAIs (Enterprise Application Integration). Dans ce cas, nous avons été directement impliqués et les interactions avec les autres acteurs ont été très directes.

Le dernier cas concerne la conception d'un système pour le système de surveillance d'un accélérateur de particules. Ce cas montre l'utilisation des styles architecturaux pour contraindre la conception d'interfaces graphiques. Les interactions avec les acteurs travaillant sur ce cas ont été moins nombreuses que dans les cas précédents et les travaux sortent un peu de l'esprit dans lequel nous avons effectué nos travaux. Ce cas présente donc aussi des perspectives intéressantes.

Notre chapitre est structuré en trois parties, chacune consacrée à l'un des cas[47]. Chacune de ces sections est subdivisée en une présentation de la problématique, une présentation de la solution et une présentation de l'implémentation.

1. Développement d'un système de gestion des connaissances

Les résultats des travaux que nous présentons dans cette section sont accessibles dans les livrables [Occ&Fab 03a] [Occ&Fab 03b] [Occ&Fab 04] du projet ArchWare rendus à la commission européenne.

1.1. Contexte

Les travaux présentés dans cette section ont été essentiellement fournis par Engineering Ingegneria Informatica S.p.A., entreprise basée à Rome en Italie. C'est l'entreprise leader d'un groupe constitué par neuf sociétés spécialisées dans différents services de technologies de l'information. Engineering

[46] Les travaux ont été réalisés dans des cadres différents par des tiers. Nous avons participé à ceux-ci par des actions de conseil essentiellement.

[47] Les deux premiers cas industriels sont intégrés dans le projet ArchWare.

est partenaire dans le projet ArchWare. Elle s'appuie sur les résultats du projet ArchWare pour développer un système de gestion des connaissances facilement évoluable.

Le partenaire Engineering est basé à Rome. Les échanges ont commencé début 2003 et vont continuer au moins jusqu'en décembre 2004. Jusqu'à présent ces échanges ont essentiellement consisté à du conseil concernant le langage ArchWare ADL, puis au fil du temps, des concepts sur les styles, du langage ASL, du style Composant-Connecteur et de l'outil ASL ToolKit. Les interactions ont surtout été effectuées par messagerie électronique puis par des rencontres lors de réunions entre partenaires européens. De plus, Engineering a aussi eu des interactions avec d'autres acteurs du projet (Thésame) travaillant sur un autre cas industriel. Les échanges que les deux équipes ont eu ont également permis de faire évoluer leurs travaux respectifs.

1.2. Problématique

Engineering travaille sur une stratégie innovante autour de l'idée du ***partage de connaissances*** afin de permettre aux petites et moyennes entreprises de maintenir leur indépendance, accroître leur marché national et d'être présentes dans le marché global, à travers la création d'un réseau de coopérations et d'alliances stratégiques. Engineering a déjà développé son propre système de gestion des connaissances.

Elle a décidé de l'améliorer pour supporter la collaboration avec d'autres systèmes équivalents. Ainsi, elle veut permettre une fédération à travers plusieurs organisations. Elle veut aussi que le système soit facilement évolutif pour satisfaire les besoins des organisations. Elle tient à obtenir une solution compétitive par l'implémentation d'un système customisable (configurable). Cela signifie que le système de gestion des connaissances d'Engineering doit être capable d'assumer les besoins actuels et futurs de n'importe quel client qui décide d'utiliser ses propres connaissances et de les partager au sein d'une fédération.

1.3. Solution

Engineering utilise l'environnement ArchWare pour accroître la productivité et la sûreté dans le développement d'un système qui évolue. L'utilisation de l'environnement d'ArchWare permettra de définir un environnement pour supporter l'implémentation et l'évolution d'un système de gestion de connaissances fédéré (FKMS - Federation Knowledge Managment System). Cet environnement se base sur un style propre au FKMS. Ce style est lui-même basé sur le style Client-Server pour satisfaire les qualités suivantes : vivacité, scalabilité[48] et sûreté.

1.3.1 Processus de développement adopté

Le scénario adopté par Engineering est le suivant (cf. Figure 52) :

- définir un style de manière informelle,
- définir des patrons et des composants génériques pour la description des architectures,
- définir les contraintes de styles :
    - les règles de configuration,
    - les propriétés que le système doit vérifier : vivacité, scalabilité et sûreté,
- définir une syntaxe propre à la conception de systèmes FKMS,
- génération d'un environnement de développement et de maintenance à partir des styles formalisés.

A partir du cahier des charges et des expériences, une architecture de référence est définie de manière informelle. Elle sert de plan, pour la définition des constituants des styles (éléments architecturaux, patrons et contraintes). Les styles formalisés permettent la génération d'un environnement de développement.

[48] Scalabilité est la francisation du mot anglais *scalability* qui signifie une propension à pouvoir croître tout en conservant les propriétés requises du système.

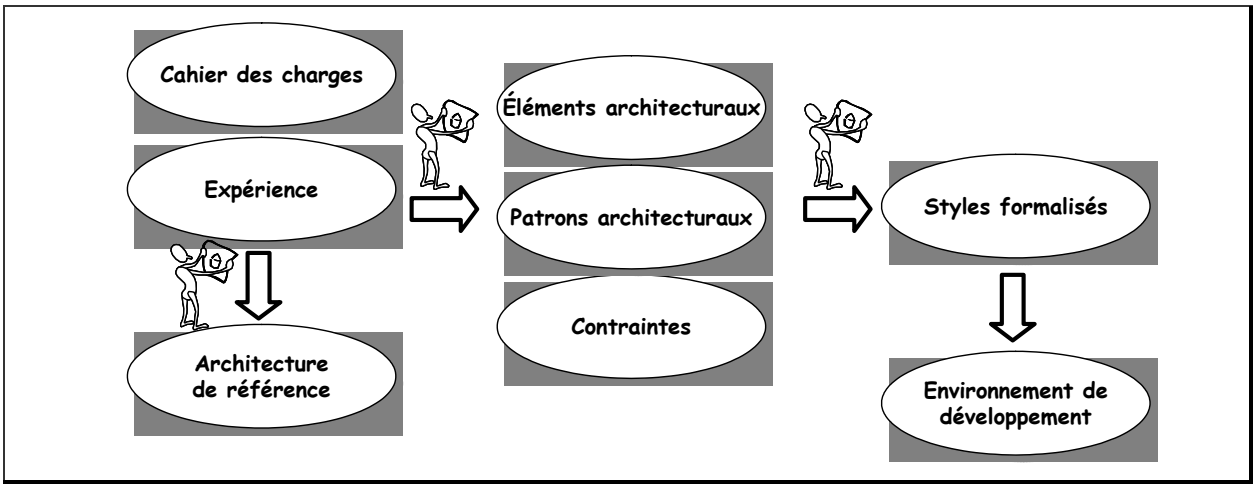


Figure 52 - Utilisation des styles dans le développement du système de gestion des connaissances

1.3.2 Bases architecturales

La conception du style FKMS se base sur une architecture de référence (Figure 53) [Occ&Fab 03a]. Cette architecture de référence a été construite sur l'expérience d'Engineering dans le domaine. Elle présente deux systèmes fédérés (KMS Organisation A et KMS Organisation B). Dans le cas générique (que le style doit prendre en compte) il y a un nombre quelconque de systèmes à fédérer.

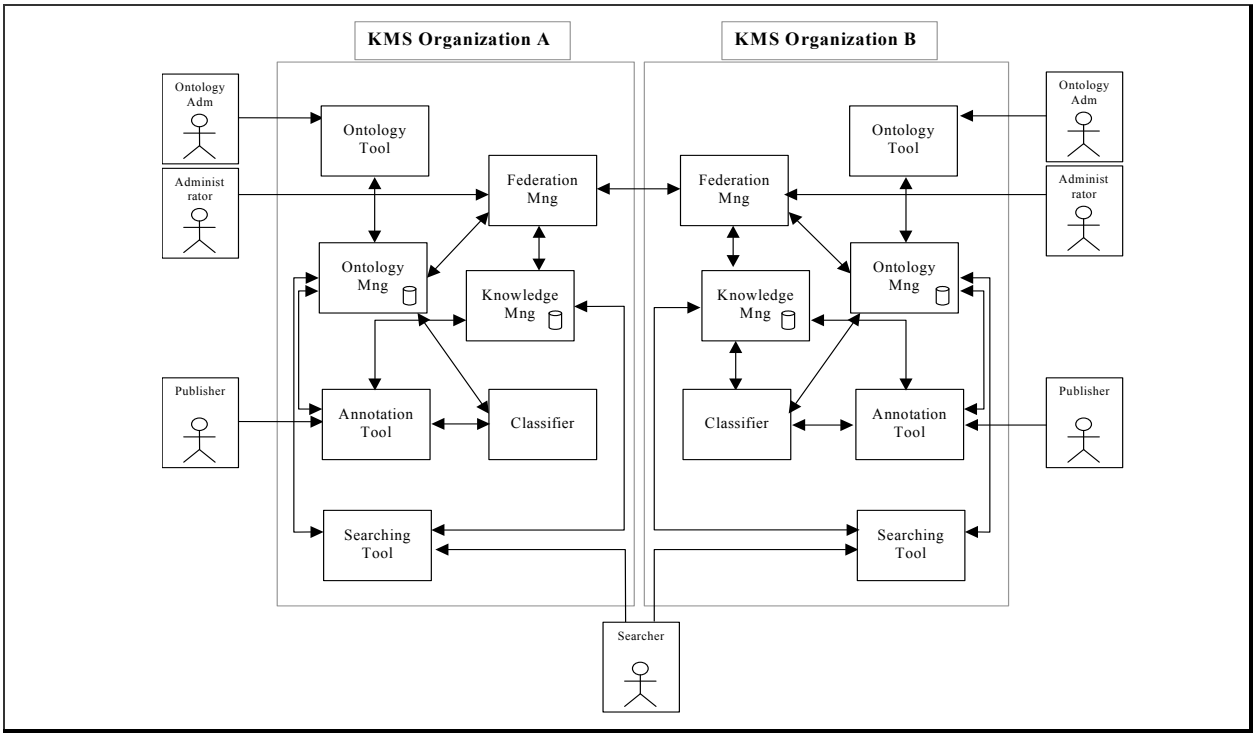


Figure 53 - Architecture de référence

L'architecture de référence définit une fédération de systèmes de gestion de la connaissance de différentes organisations (*KMS Organization A* et *KMS Organization B*). La collaboration entre les organisations est réalisée par l'utilisation de plusieurs ontologies. Une *ontologie* est une manière de décrire la sémantique commune pour un ensemble d'informations. Elle contient des informations sur des objets et leurs relations, de sorte qu'une communication sûre puisse s'établir entre les personnes et les systèmes d'application.
Cela permet ainsi d'utiliser communément des sources d'informations disparates d'une manière contrôlée et croissante. Par conséquent, chaque organisation prenant part à la fédération a sa

propre ontologie et un accès à distance aux ontologies des autres membres à travers un gestionnaire de fédération (*FederationMng*).

1.3.3 Patrons et éléments architecturaux

Dans la définition du style FKMS, Engineering a défini des patrons et des éléments architecturaux leur permettant de répondre aux attentes de leur cahier des charges. Pour cela, ils ont modélisés des architectures [Occ&Fab 03b] suivant un style Composant-Connecteur, proposé par une notation UML spécifique [All&Oqu 03].

Par exemple, un style de "configuration en étoile" est adopté (cf.Figure 54) avec en son centre un système de routage spécifique. En effet, il est attendu que chaque système négocie avec une politique spécifique avec chaque autre système dans la fédération. Il est nécessaire de pouvoir adresser un message à un système spécifique dans la fédération sans annoncer le message à la fédération entière. Un connecteur centralisé (*NegotiationDispatcher*) reçoit les requêtes et les réponses des négociations de chaque système et les expédie au système cible.

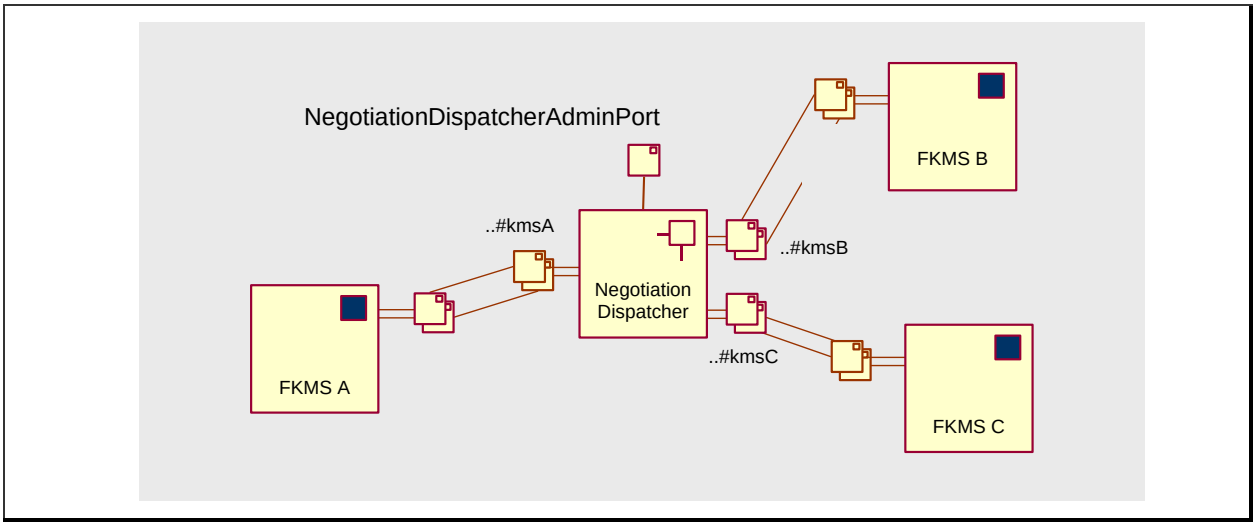


Figure 54 - Configuration étoilée pour le routage des négociations

Un autre style (cf.Figure 55) représente les systèmes liés par un simple connecteur "un-à-un" (*FKMSConnector* - un connecteur ne liant que deux entités). Chaque FKMS expose aussi un port pour chaque système directement relié étant donné que l'accès aux voisins est sujet à une politique d'accords entre les systèmes.

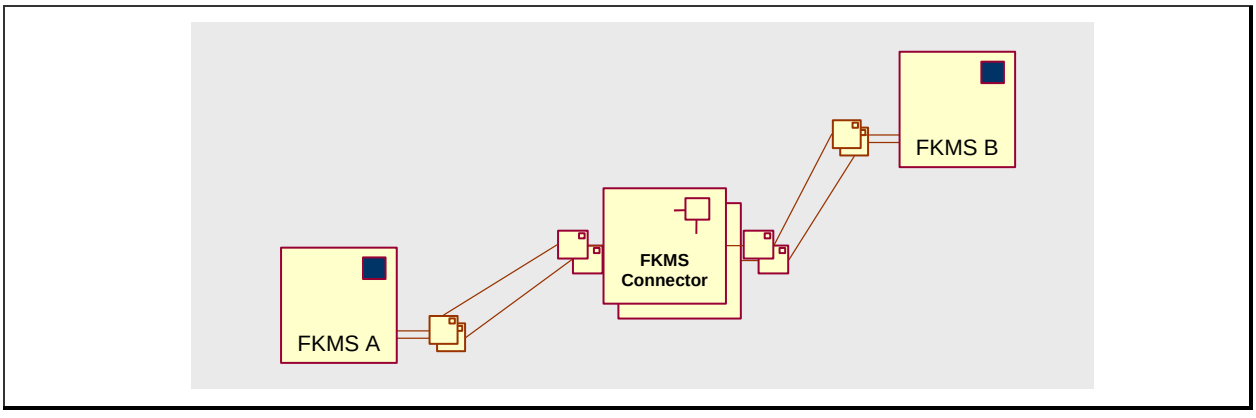


Figure 55 - Liaison un-à-un au niveau 'fédération'

1.3.4 Description des contraintes

Une fois la topologie du système décrite, Engineering a défini des contraintes de styles. Certaines de ces contraintes sont des règles de configuration issues des études sur la structure du système. D'autres contraintes reflètent les propriétés que le système doit vérifier.

Compte tenu des évolutions de nos travaux ces contraintes ont d'abord été exprimées avec le langage ArchWare AAL [All et al. 02]. Engineering a implémenté les propriétés architecturales suivantes : la scalabilité, la sûreté et la vivacité. La ***scalabilité*** d'un logiciel est sa propension à pouvoir croître tout en conservant les propriétés requises du système. La ***sûreté*** d'un logiciel concourant spécifie que "quelque chose de mauvais n'arrive jamais" [Mat 96]. La ***vivacité*** d'un logiciel concourant spécifie que "quelque chose de bon finira par arriver" [Mat 96].

Ces propriétés ont été définies sur plusieurs niveaux. Autrement dit, elles s'appliquent tant au système global qu'à des éléments architecturaux précis. Elles sont de plusieurs natures : structurelles et comportementales.

1.3.5 Formalisation des styles

Engineering a formalisé un grand nombre de styles hiérarchisés en exploitant les différentes facettes offertes par les mécanismes de styles. Ainsi, ils utilisent largement les mécanismes d'agrégation et d'héritage. Par exemple, certains styles sont exclusivement définis pour définir des bibliothèques d'éléments réutilisables.

Les principaux styles ont été définis par rapport aux éléments qu'on peut retrouver sur la Figure 53 :

- ***Federation_Style***: C'est le style FKMS, celui qui englobe les autres styles et qui définit les contraintes globales du système,
- ***GenericKMS_Style***: C'est le style générique pour les éléments Knowledge Management System (KMS),
- ***StandaloneKMS_Style***: C'est le style pour un élément KMS ne faisant pas partie de la fédération,
- ***FederatedKMS_Style***: C'est le style pour un élément KMS ne faisant pas partie de la fédération,
- ***KnowledgeManager_Style***: C'est le style pour un élément Knowledge Manager qui apparaît au sein d'un élément KMS,
- ***FederationManager_Style***: C'est le style pour un élément Federation Manager qui apparaît au sein d'un élément KMS,
- ***FederationManagerServices_Style***: C'est un style spécifique aux services proposés par un élément Federation Manager.

Ces styles ont été d'abord définis directement en ASL. ASL n'étant pas basé sur le style Composant-Connecteur, Engineering a transformé tous les éléments architecturaux en abstractions.

Avec l'évolution de nos travaux, ils ont pu ensuite s'appuyer sur le style Composant-Connecteur et aller plus loin dans la finesse de description de leurs styles.

1.4. Retours d'expérience

Les interactions avec Engineering bien qu'elles furent distantes, la plupart du temps, furent nombreuses. Ces interactions ont commencé dès les premières modélisations jusqu'à leurs formalisations.

Il y eut différentes sortes d'interactions : la dissémination des résultats de nos travaux, les questions sur nos travaux, les conseils et les retours d'erreurs sur l'utilisation de l'outil ASL ToolKit.

Engineering a eu possession des résultats de nos travaux tout au long du projet ArchWare. Le premier support consiste en des livrables qui sont fournis régulièrement et dans lesquels ces résultats figurent. Il y eut aussi de nombreuses réunions lors desquelles nous avons présenté nos

travaux, dont deux vidéoconférences au cours desquelles nous avons présenté un tutorial sur la formalisation des styles et l'utilisation du style Composant-Connecteur.

Nous avons eu un grand nombre de questions concernant les langages ArchWare ADL, ASL et le style Composant-Connecteur. Un grand nombre de ces questions nous ont permis de rendre notre langage plus ergonomique.

Il y eut aussi une demande de conseils pour le passage entre les différents formalismes, et à l'organisation des styles les uns par rapports aux autres.

Lors de ces retours, nous avons pu noter les difficultés que présente l'utilisation des langages de bas-niveau (ArchWare ADL et ASL) pour des systèmes dynamiques. L'utilisation du style Composant-Connecteur a posé beaucoup moins de problèmes. Il est vrai que cette situation peut s'expliquer par le fait qu'au moment d'utiliser le style Composant-Connecteur, les ingénieurs d'Engineering avaient acquis un niveau de compréhension avancé du fait de leurs modélisations précédentes. Nous avons noté aussi le désir d'Engineering de promouvoir la réutilisation à son maximum par l'utilisation des styles et les mécanismes d'héritage et d'agrégation.

2. Développement d'un système pour une gestion agile des processus industriels

Les résultats des travaux que nous présentons dans cette section sont accessibles dans les livrables [Bla et al. 02][Bla et al. 03][Bla et al. 04] du projet ArchWare rendus à la commission européenne.

2.1. Contexte

Les travaux présentés dans cette section ont été fournis par Thésame. Thésame a été créée dans le cadre d'une association semi-public semi-privée comprenant le Conseil Régional de Rhône-Alpes et le Conseil Départemental de Haute-Savoie. C'est une organisation principale dans le secteur de transfert d'innovation et de technologie en France, comprenant un réseau de 3000 conseillers. Thésame est partenaire dans le projet ArchWare. Elle s'appuie sur les résultats du projet ArchWare pour développer ***un système de gestion agile des processus industriels***.

Ces travaux représentent également une partie des travaux de thèse de Lionel Blanc dit Jolicœur [Bla 04] qui s'intitule "*Modélisation des processus métiers mis en œuvre dans une approche EAI en vue de leur pilotage - Le pilotage des applications intégrées*".

Etant sur le même lieu de travail, les échanges ont été réguliers et continus. Nous avons nous même activement participé à ces travaux. Les échanges ont essentiellement consisté à du conseil concernant le langage ArchWare ADL, puis au fil du temps, des concepts sur les styles, du langage ASL, du style Composant-Connecteur et de l'outil ASL ToolKit. De plus, Thésame a également eu des interactions avec Engineering. Les échanges que ces deux équipes ont eus ensemble ont permis de faire évoluer leurs travaux respectifs.

2.2. Problématique

Thésame travaille sur l'intégration d'applications hétérogènes afin de permettre aux entreprises de contrôler leur système de production. Thésame a développé un middleware pour l'intégration d'applications métiers - middleware EAI (Enterprise Application Integration - Intégration d'Applications d'Entreprises). D'une part, celui-ci permet de capturer les descriptions des produits, des processus et de leurs relations ; son formalisme de modélisation permet de décrire différentes applications. D'autre part, il permet de suivre le déroulement de processus et de produits ; il permet de fournir une trace pour chaque exécution d'un processus sur un produit.

Les middlewares EAI sont seulement une partie de la solution. Plusieurs points restent à résoudre dans l'intégration d'applications hétérogènes en ce qui concerne les opérations globales d'une entreprise. Ceci requiert un support à la modélisation et l'automatisation de processus. Ces processus touchent à tous les aspects concernant l'organisation et l'activité intra-entreprise et inter-entreprise.

En général, le middleware EAI est applicable à des systèmes de production à grande échelle se déployant à travers plusieurs organisations. Thésame souhaite développer un environnement générique lui permettant de concevoir des systèmes EAI pour les petites et moyennes entreprises. Ces systèmes sont à chaque fois spécifiques à l'entreprise. En outre, un besoin essentiel est de pouvoir faire évoluer ces systèmes rapidement et sûrement pour conserver leurs intégrités.

2.3. Solution

Thésame s'appuie sur ArchWare pour customiser un environnement de développement pour les systèmes de gestion de production pour les petites et moyennes entreprises. Cet environnement se base sur un style apparenté au style ***abstract data repository middleware*** satisfaisant les qualités suivantes : modifiabilité, performance et sureté.
Dans ce cas industriel, l'utilisation devrait permettre, durant ***l'évolution du système*** et à différents ***niveaux d'abstractions*** de :

- réaliser des ***vérifications formelles*** sur des propriétés telles que la modifiabilité, la performance et la sûreté,
- capturer la connaissance et l'expérience du domaine dans une forme réutilisable. Ceci concerne la formalisation de processus métiers, le développement du système et la réutilisation du code.

2.3.1 Processus de développement adopté

Le processus adopté par Thésame concernant l'utilisation des styles est le suivant (cf. Figure 56) :

- définir un style de manière informelle sur plusieurs niveaux,
- formaliser une architecture typique permettant de valider le style,
- définir les mécanismes propres à la description de processus métiers,
- définir les contraintes de style :
  - les règles de configuration,
  - les propriétés que le système doit vérifier : modifiabilité, sûreté et performance,
- définir une syntaxe propre à la conception de systèmes EAI,
- génération d'un environnement de développement à partir des styles formalisés,
- génération d'environnements de développement et de maintenance pour des PME/PMI spécifiques.

A partir du cahier des charges et des expériences, une architecture de référence est définie de manière informelle. Elle sert de plan, pour la formalisation des architectures spécifiques. Cette architecture est la base de travail pour la définition de plusieurs styles. Ces styles formalisés permettent la génération d'un environnement de développement. Cet environnement est lui même utilisé pour la génération d'environnements plus spécifiques destinés à des PME/PMI particulières.

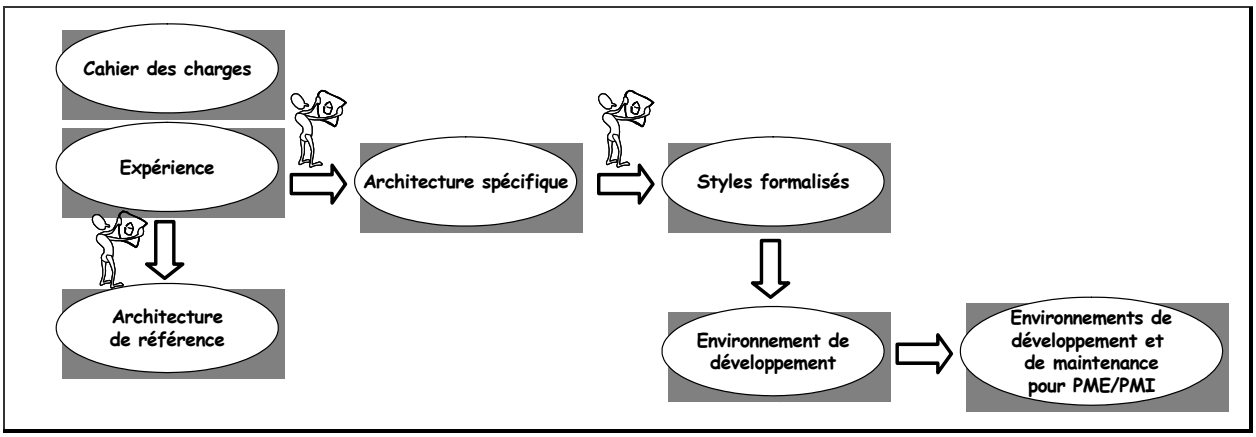


Figure 56 - Utilisation des styles dans le développement de systèmes de gestion de processus industriels

Ce processus est relativement proche de celui adopté par Engineering du fait des interactions entre les deux équipes.

2.3.2 Bases architecturales

Pour développer ses styles, Thésame se base sur l'architecture proposée par son middleware EAI. Elle montre (cf. Figure 57) des systèmes de gestion de processus métiers interconnectables d'une entreprise à l'autre ; le système de gestion d'une entreprise fournisseur communique avec le système de gestion d'une entreprise client. Chaque système est composé d'un gestionnaire de flux (workflow) et de COTS (Component Off The Shelf).

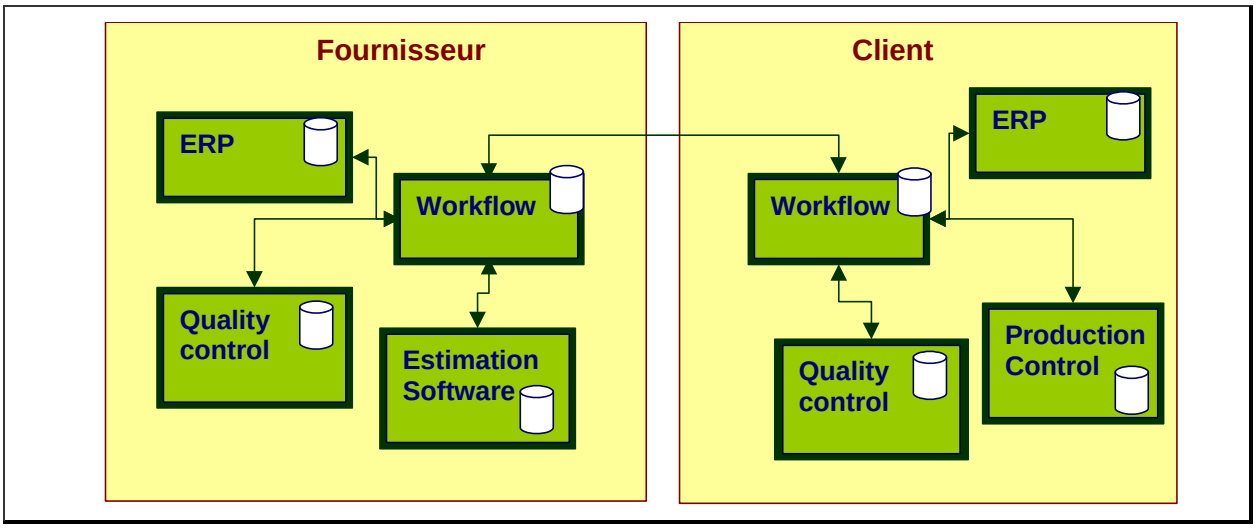


Figure 57 - Connexion inter-entreprises des systèmes de gestion

2.3.3 Architecture Fournisseur-Client

Avant de formaliser un style, Thésame a choisi de décrire d'abord une architecture d'un haut niveau d'abstraction. Cette architecture modélise un système permettant la négociation entre un client et un ou plusieurs fournisseurs.

Cette architecture a été d'abord modélisée en ArchWare ADL, puis, en utilisant le style Composant-Connecteur. Elle a servi de base de travail à Thésame pour mener et valider ses travaux. Cette architecture a servi de base de réflexion pour déterminer les variations possibles et pour mettre en avant un ensemble d'éléments architecturaux, des contraintes, puis finalement des styles.

2.3.4 Description des contraintes

Thésame a défini un ensemble de propriétés en ArchWare AAL par rapport à la description de l'architecture précédemment citée. Thésame a implémenté les propriétés architecturales suivantes : la modifiabilité, la sûreté et la performance. La ***modifiabilité*** d'un système reflète la possibilité de pouvoir le modifier facilement. La ***sûreté*** d'un système reflète le fait que "quelque chose de mauvais n'arrive jamais". La ***performance*** d'un système, dans le cas étudié, se mesure par la rapidité des négociations.

2.3.5 Description des mécanismes propres à la définition de processus

Thésame a besoin de permettre la description de processus métier. Pour permettre cette description des travaux ont été effectués dans l'optique d'utiliser les styles architecturaux pour fournir des mécanismes et une syntaxe spécifiques à la description des processus. Ces mécanismes ont été définis en utilisant le style Composant-Connecteur.

2.3.6 Formalisation des styles

Thésame a formalisé ces styles vis-à-vis de l'architecture de référence comme le montre la Figure 58. Les styles sont développés sur trois niveaux :

- le niveau ***Inter-entreprises***,
- le niveau ***Entreprise***,
- le niveau ***Processus***.

Ces styles représentant les différents niveaux du système ont d'abord été formalisés avec ASL seul, puis en se basant sur le style Composant-Connecteur.

Le niveau Inter-Entreprises

Le point de vue **Inter-entreprises** place le système EAI d'une entreprise au sein de son environnement c'est-à-dire au milieu de la chaîne de production à laquelle elle appartient. Ce point de vue s'intéresse à la synchronisation des différents Systèmes d'Information (SI) des entreprises mises en œuvre au sein de la chaîne de production. Les différentes entreprises en relations communiquent via un médiateur qui transite les informations entre les différents acteurs de la chaîne de production (les clients et les fournisseurs).

Le style *InterEnterprise* traduit ce niveau. Il définit les règles de configuration pour construire la chaîne de production et les types d'éléments peuvent intervenir dans cette chaîne. Il définit aussi des contraintes concernant la gestion de cette chaîne de production, par exemple :

- ajout d'un ou plusieurs fournisseur(s),
- suppression d'un fournisseur,
- gestion des magasins avancés,
- externalisation du transport, etc.

Le niveau Entreprise

Le point de vue ***Entreprise*** montre la structure interne du système de gestion d'une entreprise avec une vision composant. A ce niveau on trouve :

- les ***composants industriels***,
  - un composant industriel est une entité destinée à satisfaire un besoin unique dans l'entreprise. Il est caractérisé par les propriétés suivantes : il est atomique, il exprime un point de vue unique, il est générique, il est autonome,
- le ***système de gestion de la performance***,
  - il identifie les composants industriels capables de satisfaire un besoin et évalue le but atteint par ces composants,
- le ***système de gestion des ressources***,
  - il gère les différentes ressources de l'entreprise.

Le niveau processus

Au niveau processus, on décrit les processus métiers et les COTS (Components Off The Shelf), les éléments représentants les logiciels utilisés au sein de l'entreprise.

Les styles correspondant à ce niveau fournissent :

- un formalisme de description des processus métiers,
- des propriétés concernant les COTSs.

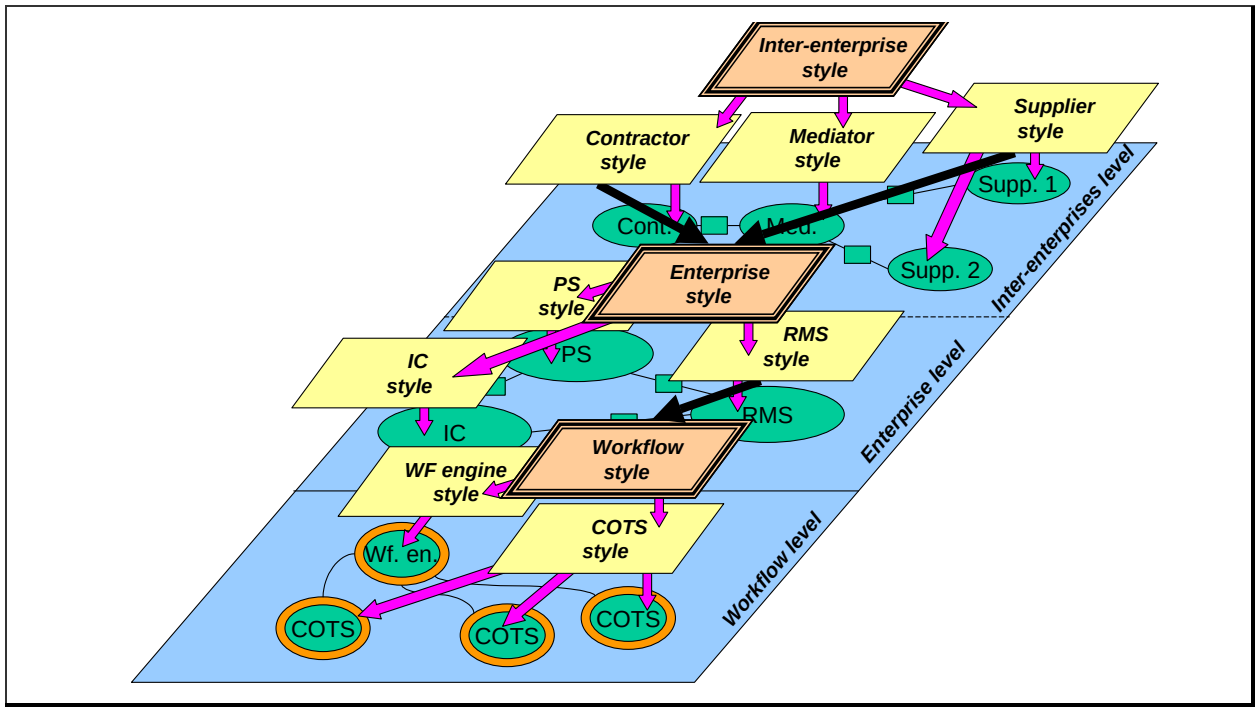


Figure 58 - Des styles adaptés aux trois niveaux de conception de l'architecture de référence

La particularité par rapport à Engineering est que Thésame veut obtenir un environnement permettant de générer d'autres environnements spécifiques à des entreprises. Ces environnements spécifiques seront issus de la spécialisation des styles définis. Afin de pouvoir les générer, l'environnement initial devra être constitué du ASL ToolKit pour permettre la gestion des styles.

2.4. Retours d'expériences

Les interactions avec Thésame furent directes et rapprochées. Nous avons initié les premiers travaux concernant la formalisation des architectures et des styles.

Il y eut différentes sortes d'interactions : les questions sur nos travaux, les conseils et les retours d'erreurs sur l'utilisation de l'outil ASL ToolKit.

Nous avons eu un grand nombre de questions concernant les langages ArchWare ADL, ASL et le style Composant-Connecteur. Un grand nombre de ces questions nous ont permis de relever des erreurs dans nos documents et d'améliorer le langage.

Il y eut aussi une demande de conseils due au fait du passage entre les différents langages, et à l'organisation des styles les uns par rapports aux autres.

Lors de ces retours, nous avons pu noter les difficultés que présente l'utilisation des langages de bas-niveau (ArchWare ADL et ASL) pour des systèmes dynamiques. L'utilisation du style Composant-Connecteur a posé beaucoup moins de problèmes.

Dans la section suivante, nous présentons un cas industriel qui, cette fois-ci, n'est pas directement lié au projet ArchWare.

3. Développement d'un système pour la gestion d'accélérateurs de particules

Les résultats des travaux que nous présentons dans cette section sont accessibles dans les documents [Rat 04][Rat et al. 03][Rat et al. 04a][Rat et al. 04b].

3.1. Contexte

Les travaux ont été réalisés dans le cadre de la thèse d'Olivier Ratcliffe intitulée "*Approche et environnement fondés sur les styles architecturaux pour le développement de logiciels propres à des spécifiques - Application au domaine de la supervision au redémarrage d'accélérateurs de particules*" [Rat 04] et préparée au sein de l'Université de Savoie. Ces travaux se sont effectués au CERN (Centre Européen pour la Recherche sur le Nucléaire) [CERN] et concerne le développement d'applications dédiées à la sécurité des accélérateurs de particules.

Les interactions avec ces travaux ont eu lieu avec Olivier Ratcliffe, par messagerie instantanée et par contact direct.

3.2. Problématique

La Salle de Contrôle Technique du CERN a implanté une méthode utilisée pour définir l'information de monitoring nécessaire pour redémarrer efficacement un accélérateur après une panne majeure. Cette méthode est la base pour le développement d'un ensemble d'Interface Homme Machine (IHM) pour la surveillance des accélérateurs de particules. Dans ce contexte, un logiciel, SEAM (Software for the Engineering of Accelerator Monitoring), est développé pour produire automatiquement des IHMs de surveillance de propriétés et de règles prédéfinies.

Initialement, les programmeurs ont implémenté des prototypes de ces IHMs en prenant un accélérateur de particule spécifique comme exemple. C'est un travail répétitif, ce qui mène à de nombreuses erreurs sur la standardisation graphique et les niveaux de traitement de données. Plusieurs itérations de développement ont été nécessaires pour obtenir des résultats satisfaisants.

Tenant compte de ces considérations, il a semblé nécessaire d'automatiser le processus de développement, pour obtenir des familles complètes d'applications qui seront employées pour surveiller le redémarrage de tous les accélérateurs de particules du CERN.

L'objectif de SEAM est d'utiliser l'expérience première comme directive pour réduire l'effort de développement, et par conséquent, le coût de développement, qui est un point important pour la communauté entière. La nécessité d'avoir un guide formel de développement et un cadre pour réutiliser l'expérience de conception a mené à considérer une solution orientée architecture. En effet, les Langages de Description d'Architectures fournissent des moyens de décrire les propriétés fonctionnelles, structurelles, et comportementales auxquelles l'application doit être conforme.

3.3. Solution

Un processus de développement centré sur la définition et l'exploitation d'un style architectural pour développer une famille d'IHMs de supervision possède plusieurs bénéfices. En effet, l'utilisation des styles architecturaux simplifie la construction des systèmes, réduit le coût d'implémentation en promouvant la réutilisation, et par dessus tout, améliore l'intégrité des systèmes au moyen d'analyses et d'outils spécifiques au style utilisé. Le développement architectural permet de réutiliser des concepts déjà validés et garantit que les futures applications vont respecter des propriétés voulues. De plus, l'utilisation d'un style architectural guide le développement d'une famille de systèmes logiciels (ex : IHMs) en fournissant un vocabulaire architectural commun utilisé pour décrire les structures des systèmes, et en contraignant l'utilisation de ce vocabulaire. L'adoption d'une approche utilisant les styles architecturaux facilite l'implémentation des IHMs en :

- contraignant la construction de ces IHMs (par la spécification et la vérification de propriétés),
- produisant des IHMs standardisées partageant des propriétés graphiques, structurelles et comportementales communes,
- capturant le savoir faire du domaine et en réutilisant l'expérience acquise (réutilisation des concepts utilisés pour les prototypes d'IHMs).

3.3.1 Processus de développement adopté

Le processus adopté pour aboutir à la production d'une famille d'IHMs est le suivant (cf. Figure 56) :

- formaliser directement des architectures à partir du cahier des charges,
- le style est défini par induction à partir de ces architectures,
  - définitions d'éléments architecturaux,
  - définitions des contraintes,
- le style est ensuite intégré dans l'outil SEAM,
- des architectures sont créées à partir de SEAM,
- du code est généré à partir des architectures pour fournir une application exécutable,
- les architectures sont analysées et peuvent conduire à l'évolution du style.

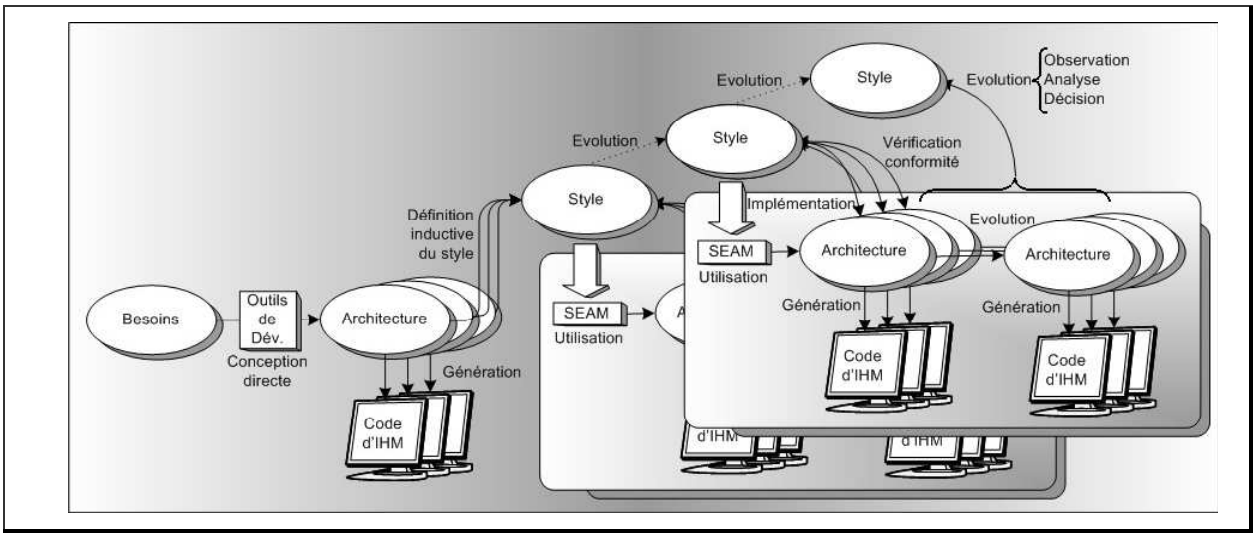


Figure 59 - Processus de développement complet

3.3.2 Bases architecturales

Comme un travail significatif a été réalisé durant la première implémentation (spécialement en collection de données, en standardisation d'affichage et en développement de champs de procédures), il est utile de réutiliser la conception et l'expérience acquise pour le futur développement de la famille complète des applications qui seront employées pour surveiller le redémarrage de tous les accélérateurs de particules du CERN.

Dans le domaine du développement d'IHMs pour la supervision d'accélérateurs de particules, aucun style architectural n'était disponible. Il a donc été nécessaire de définir un style propre à ce domaine. Un processus de développement inductif a été adopté : un style appelé style GTPM (Gestion Technique de Pannes Majeures) a été défini à partir de l'analyse d'architectures existantes pour les IHMs. L'architecture informelle de la Figure 60 montre la composition d'une IHM.

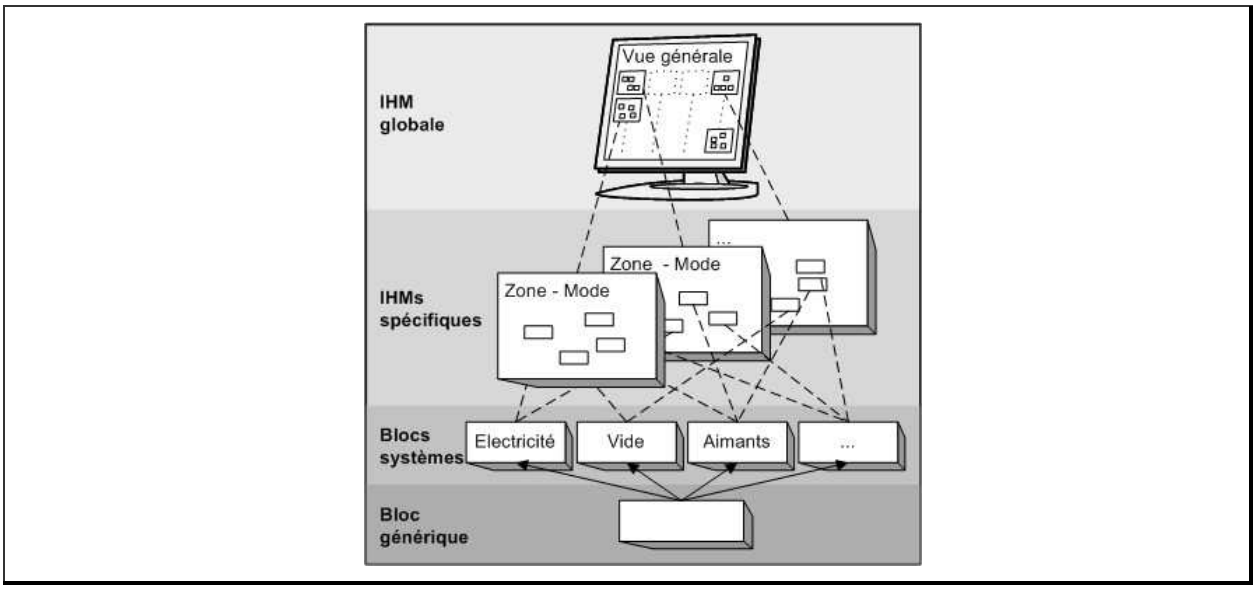


Figure 60 - Structure de l'application de supervision du SPS

La Figure 61 représente, en notation UML, le modèle de construction d'IHMs conçu à partir de l'étude des prototypes.

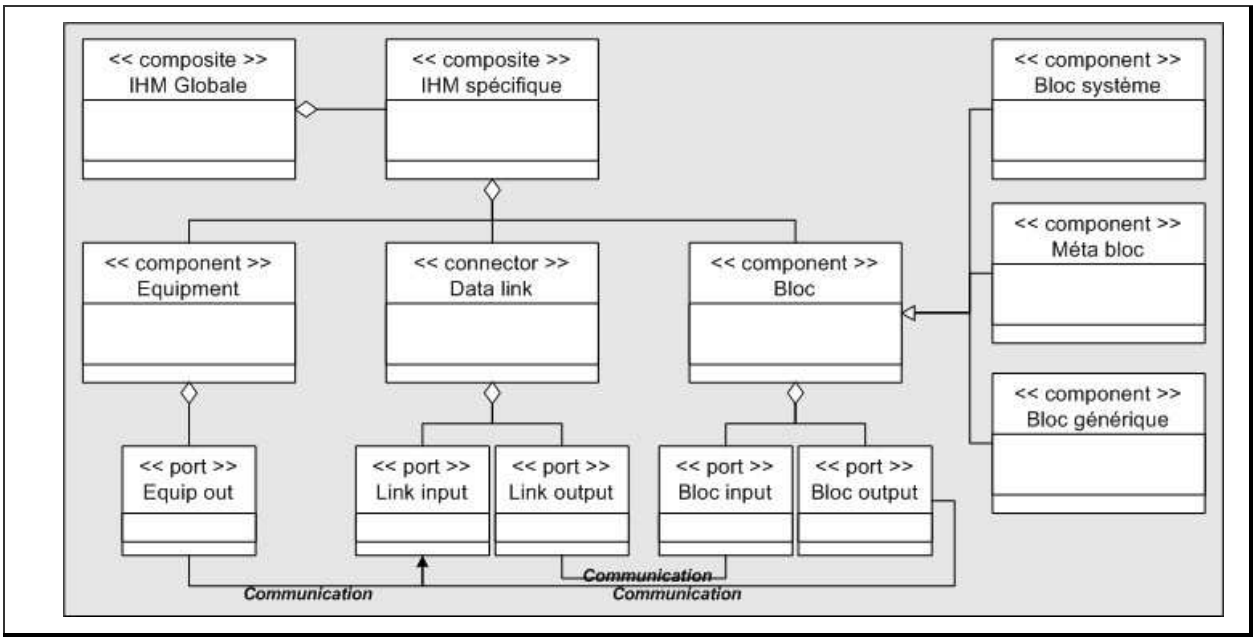


Figure 61 - Structure des IHMs

Ainsi, les IHMs doivent être composées des éléments suivants :

- **IHM globale :** cet élément doit permettre l'affichage de l'état de tous les systèmes nécessaires à l'opération de l'accélérateur. Il s'agit de l'élément du plus haut niveau de l'architecture, il se compose des éléments de niveau inférieur, les IHMs spécifiques,
- **IHM spécifique :** permet l'affichage de l'état des systèmes correspondant à un mode d'accélération ou une zone de l'accélérateur spécifique,
- **Bloc système :** cet élément affiche l'état résultant d'un ou plusieurs équipements d'un système spécifique.

Le style GTPM peut proposer des valeurs par défaut pour les attributs des blocs et des IHMs énumérés ci-dessus, et les utilisateurs doivent toujours avoir la possibilité de paramétrer ceux-ci.

Cette phase a permis l'identification du vocabulaire métier, ainsi que des principales contraintes de style. Ayant adopté une approche inductive, cette phase a pris la forme d'une étude des vingt prototypes d'IHMs préalablement développés. Ainsi, l'expertise accumulée lors des premiers prototypages est capturée dans ce vocabulaire et ces contraintes.

3.3.3 Description des contraintes

Il est capital que les IHMs construites à partir du style GTPM soient standardisées. Par conséquent, il est nécessaire que le style spécifie, en plus des propriétés structurelles citées précédemment, un certain nombre d'autres contraintes s'appliquant notamment aux attributs graphiques. L'analyse des prototypes a permis de répertorier ces contraintes, ainsi que les propriétés et les données utilisées par les IHMs déjà développées. La majorité de celles-ci correspondent aux besoins exprimés par les utilisateurs. Ces propriétés et contraintes se répartissent principalement dans les catégories suivantes :

- contraintes graphiques générales (taille, couleur, position des éléments),
- contraintes de validité des informations,
- contraintes de positionnement des éléments en fonction de leur rôle dans la séquence de redémarrage de l'accélérateur,
- contraintes de cardinalité des éléments,
- contraintes d'interdépendance entre éléments,
- contraintes d'acquisition de données,
- contraintes de traitement de données,
- contraintes concernant l'évolution des éléments architecturaux.

3.3.4 Formalisation du style

Etant donné que ce style architectural a été défini à partir d'architectures prototypes, il était clair que celui-ci ne pourrait être complet dès sa création. En effet, l'objectif était de l'utiliser pour construire de nombreuses architectures, et donc de nouveaux besoins allaient vraisemblablement apparaître. Il a donc été nécessaire d'étendre le processus de développement de façon à permettre une évolution du style de référence en fonction de l'évolution des besoins.

3.3.5 Utilisation du style pour concevoir l'outil SEAM

Le style GTPM a été utilisé pour concevoir l'outil de spécification SEAM. SEAM peut-être utilisé pour construire des architectures d'IHMs.

3.4. Retour d'expériences

Ce retour d'expériences est largement repris au document [Rat 04].

L'étude des concepts architecturaux et la prise en main des techniques associées ont nécessité plusieurs semaines. Il est à noter que la définition du style a été relativement aisée, et ce, principalement grâce à l'utilisation du langage de haut niveau C&C (style Composant-Connecteur). La formalisation du style en elle-même à partir du style Composant-Connecteur n'a, quant à elle, représentée que quelques jours de travail. En effet, un essai de formalisation du style directement à partir de l'ADL ArchWare avait été préalablement réalisé, et il s'agit d'un travail bien plus délicat. Par ailleurs, la formalisation et l'utilisation d'un style supposent que l'architecte maîtrise correctement les différents aspects du développement formel dans le cadre des architectures logicielles.

Concernant la construction d'une bibliothèque d'éléments architecturaux, le langage ASL et le style Composant-Connecteur ont permis de formaliser tous les éléments nécessaires à la modélisation des IHMs GTPM, il s'agit d'une dizaine de composants et connecteurs.

L'approche centrée style a permis la définition d'un vocabulaire spécifique au domaine des accélérateurs de particules. Il a été défini un vocabulaire décrivant des IHMs globales, incluant des IHM spécifiques à des localisations ou modes donnés, affichant des blocs acquérant les données concernant des états d'équipements. Il est en outre possible de matérialiser les dépendances entre

blocs. L'approche centrée style s'est donc avérée efficace pour spécifier le vocabulaire du langage dédié.

Les entretiens avec les utilisateurs du style via l'outil SEAM, et l'analyse des IHMs qu'ils ont produit, ont indiqué que celles-ci respectaient les contraintes du style GPTM : aucune violation n'a été constatée. Lors de la formalisation du style, plusieurs nouveaux besoins ont été exprimés par les membres du projet GTPM. Les contraintes du style déjà formalisées ont du être modifiées en fonction de ces nouveaux besoins. Les modifications des contraintes au niveau du style formel se sont avérées rapides à mettre en œuvre (quelques minutes de travail). Toutefois, la ré-implémentation de ces contraintes modifiées dans les modules logiciels de SEAM est beaucoup plus coûteuse (plusieurs heures de travail).

Quant à garantir l'utilisation du vocabulaire et le respect des contraintes, l'étude de cas a indiqué qu'aucune contrainte du style n'avait été violée. Pour permettre le développement d'IHMs respectant les contraintes du style, il a été nécessaire d'implémenter ces contraintes dans SEAM. Ce travail s'est avéré long et relativement complexe dans le cas étudié ; l'intégration du style dans l'environnement de développement SEAM, c'est à dire l'implémentation de son vocabulaire et de ses contraintes dans les différents modules logiciels a nécessité plusieurs mois de travail. Ceci s'explique principalement par le fait que le contexte industriel du développement de SEAM n'a pas permis d'attendre la fin du développement des outils architecturaux proposés par ArchWare.

En ce qui concerne l'évolution du langage et de l'environnement associé, l'étude de cas a montré que la modification de contraintes formelles dans le style est facile et rapide. Toutefois, la modification du code correspondant dans l'outil SEAM est beaucoup plus délicate. Ce point souligne encore une fois l'intérêt d'utiliser un environnement de développement directement paramétrable par le style. Un tel environnement est automatiquement mis à jour lorsqu'une contrainte formelle du style est modifiée. Dans ce cas, l'approche centrée style faciliterait non seulement l'évolution du langage, mais aussi celle de l'environnement associé.

4. Conclusion

Dans ce chapitre, nous avons présenté trois cas industriels dans lesquels l'objet est le développement d'un système dans une optique centrée architecture. Dans chacun de ces trois cas, les styles jouent un rôle central et les outils formels que nous avons proposés dans les chapitres précédents sont utilisés.

De ces travaux, il en ressort que la formalisation des styles a son importance dans un processus de développement centré architecture. Les styles sont utilisés de plusieurs manières.

Premièrement, ils offrent des outils de conception propres à un domaine avec des notations et des mécanismes spécifiques.

Puis, ils spécifient les propriétés critiques des systèmes, telles que la fiabilité, la modifiabilité, la performance. L'architecte a ainsi à sa disposition un outil pour vérifier si une architecture a les caractéristiques voulues.

Ensuite, ils peuvent définir des règles sur la configuration du système et fournir une bibliothèque de composants prédéfinis.

Enfin, les styles trouvent beaucoup d'intérêts à être intégrés au cœur d'un environnement de développement. Ainsi, on peut obtenir des environnements spécifiques à des domaines ou à des systèmes. De tels environnements peuvent proposer :

- des formalismes spécifiques,
- une vérification des contraintes qui peut avoir lieu
    - lors de la modélisation de l'architecte,
    - à chaque étape de l'évolution du système,
- une bibliothèque d'éléments prédéfinis.

Par rapport à nos travaux, il ressort que le langage ASL répond à l'attente des différents utilisateurs. De plus, ceux-ci ont soulevé plus vivement leur intérêt pour le style Composant-Connecteur qui offre des facilités qui ont été remarquées.

L'utilisation de nos travaux ne s'arrête pas à ces trois cas industriels. Ils sont aussi actuellement en cours d'utilisation dans des travaux de thèses.

Jérôme Revillard prépare une thèse sur les architectures pour les instruments intelligents [Rev et al. 03][Rev 05]. Il développe un style fournissant les concepts sous-jacents à ce type d'architecture.

Selma Azzaiez prépare une thèse sur l'évolution des architectures basée sur les agents [Azz&Oqu.04][Azz 06]. Elle développe un style propre à la conception d'architectures dans lesquelles certains éléments, les agents, peuvent transiter d'un environnement à un autre.

Chapitre 8

CONCLUSION ET PERSPECTIVES

Chapitre 8 - Conclusions et Perspectives

Cette thèse a traité le problème de la formalisation des styles architecturaux pour les systèmes dynamiques et de leurs utilisations dans des processus de développement centré architecture. L'objectif de ces travaux est de proposer à l'utilisateur un langage et des outils associés pour un développement orienté style architectural. A travers ces travaux, il a été possible de répondre à plusieurs problèmes ouverts dans l'approche d'un développement centré architecture.

Nous fournissons au styliste, le concepteur des styles, les outils pour formaliser des styles architecturaux ; c'est-à-dire, son expérience et ses connaissances dans la conception architecturale. Il peut ainsi définir :

- des règles sur la construction des architectures,
- des bibliothèques d'éléments,
- des propriétés architecturales,
- des analyses pour mesurer ou quantifier des caractéristiques architecturales.

En aval, se trouve l'architecte, le concepteur des architectures. Il se base sur les styles architecturaux pour se guider dans la construction d'une architecture. Les styles et les outils formels associés lui permettent :

- de construire des architectures à partir d'éléments de construction,
- de générer des architectures,
- de valider des architectures.

Les systèmes sont de plus en plus dynamiques ; les changements de leurs architectures lors de l'exécution sont programmés lors du développement. En nous ouvrant aux systèmes dynamiques, nous élargissons l'utilisation formelle des styles. Ainsi, nous donnons la possibilité de définir au niveau des styles :

- des éléments et des architectures aux caractéristiques dynamiques,
- des propriétés sur la dynamique,
- des analyses sur la dynamique.

Ensuite, l'architecte peut valider une architecture ainsi que sa manière d'évoluer, vis-à-vis d'un style.

Dans la suite, nous proposons de faire un bilan sur les travaux effectués au cours de la thèse, puis de présenter des perspectives qui s'ouvrent à leur issue.

1. Bilan et Remarques

Nous présentons un bilan pour chacune des parties de notre solution et de sa validation ; elles correspondent aux chapitres 4, 5, 6 et 7.

Le problème sur lequel nous nous sommes penchés est de définir un langage expressif pour la définition des styles pour les systèmes dynamiques. Nous avons implémenté un outil associé à ce langage pour automatiser l'utilisation des styles et il a été un support pour la validation. Les résultats de nos travaux sont utilisés en amont d'un processus de développement centré architecture.

Nous proposons de répondre au problème posé par trois éléments :

- un **formalisme** : ASL,
- des **styles formalisés** : le style Composant-Connecteur et des styles de fondations,
- un **outil logiciel** : l'ASL ToolKit, un outil pour l'utilisation des styles.

1.1. ASL

ASL est un langage pour la description des styles architecturaux. Il permet de définir un support à la conception architecturale à travers les styles et il fournit un langage de haut niveau pour la description architecturale.

1.1.1 Définition du langage

ASL s'appuie sur des fondements formels et bénéficie d'une large bibliothèque d'outils associés aux formalismes sur lesquels il se base. Dans ASL, un style est vu comme un ensemble de contraintes, un ensemble d'analyses et un ensemble de constructeurs.
Les constructeurs sont la partie 'concrète' dans un style. Ils permettent de définir des 'brides' d'architectures. Leurs descriptions s'appuient directement sur ArchWare ADL, un langage de base pour la description d'architectures évolutives, et indirectement sur le π-calcul. Ces formalismes apportent le support pour la dynamique et le système de typage du langage.
Les contraintes permettent de définir des règles de construction et des propriétés architecturales telles que la vivacité, la sûreté ou la complétude. Leurs descriptions s'appuient directement sur ArchWare AAL, et indirectement sur le μ-calcul. Ces bases formelles, permettent la description de contraintes à la fois structurelles et comportementales. Ainsi, ASL permet la spécification de *propriétés critiques* des systèmes telle que la modifiabilité. L'architecte a ainsi à sa disposition un outil formel pour vérifier si une architecture possède les caractéristiques voulues.
Les analyses permettent la mesure ou la quantification des caractéristiques architecturales promues par un style. ASL fournit un mécanisme permettant d'inclure dans la description d'un style des analyses décrites avec n'importe quel langage ou script spécifique.

ASL fournit des mécanismes pour organiser les styles les uns par rapport aux autres. Il s'agit de l'agrégation et de l'héritage de style. L'agrégation permet de structurer les descriptions des styles structurées de manière compositionnelle (un style peut être composé d'autres styles) reflétant la structure compositionnelle des architectures. L'héritage de style est un mécanisme de spécialisation permettant de réutiliser les caractéristiques d'un style pour en définir un nouveau.
Ces mécanismes promeuvent la lisibilité et la compréhension et facilitent le travail des architectes dans leur choix des styles. Cet aspect est crucial car le choix d'un style arrive dans les premières étapes du développement. Ces mécanismes promeuvent aussi la réutilisation. Ils permettent d'adapter des styles pour les utiliser dans un domaine particulier.

1.1.2 Utilisation du langage

Nous avons montré comment ce langage permet de répondre aux attentes d'un style au travers d'un processus de développement de logiciels centrés architecture.

Nous séparons deux facettes dans l'utilisation des styles :

- l'utilisation comme *cadre à la conception*. Dans ce cas, le style est un guide dans le développement pour respecter des propriétés issues du cahier des charges. Le style est défini pour délimiter l'espace de conception pour un système logiciel ou pour une famille de systèmes logiciels,
- l'utilisation comme *support à la conception*. Dans ce cas, le style apporte des solutions architecturales à des problèmes donnés (problèmes de performance, de modifiabilité, de robustesse,…), ou offre un support pour la construction de l'architecture en terme d'éléments architecturaux prédéfinis, en terme d'actions comportementales prédéfinies et en terme de patrons architecturaux.

Ces deux facettes ne permettent pas de définir deux classes distinctes de styles car un style peut servir à la fois de cadre et de support à la conception.

Le diagramme suivant (Figure 62) montre notre vision d'un processus de développement basé sur les styles architecturaux.

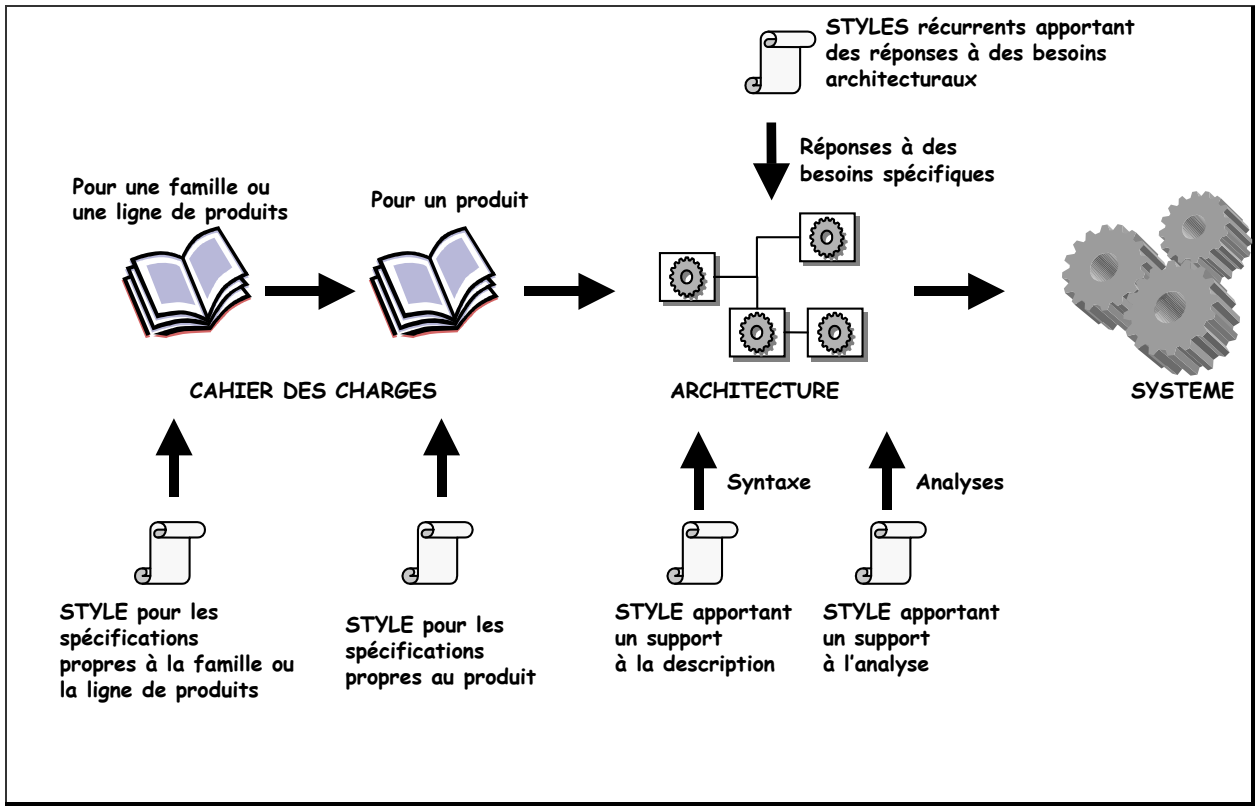


Figure 62 - Utilisation des styles dans le développement

1.2. Style Composant-Connecteur

Fournir des outils de conception pour une approche centrée style, c'est aussi fournir des styles. Il existe plusieurs styles répertoriés dans la littérature (Client-Serveur, Pipe-Filter, BlackBoard,…). Ce sont des styles récurrents qui sont généralement utilisés de manière informelle. Par exemple, le style Pipe-Filter est le style sous-jacent aux architectures Unix. Parmi ces styles, il y a un style reconnu comme la base de la conception architecturale, c'est le style Composant-Connecteur.

Ce style est généralement sous-jacent aux ADLs qui, pour la plupart, sont basés sur ses concepts. Nous avons décidé de fournir une formalisation de ce style. D'une part, parce que le langage ASL est un langage générique et que la formalisation de ce style permet de fournir une couche conceptuelle, plus facile d'utilisation. D'autre part, parce que, d'après nos lectures et nos études, il y a un manque concernant la formalisation des architectures dynamiques. Ainsi, nous avons formalisé un style Composant-Connecteur pour les architectures dynamiques.

Ce style Composant-Connecteur permet la description du comportement des composants et des connecteurs. Ce comportement spécifie notamment comment l'élément interagit avec son environnement. Ces informations sont importantes pour vérifier si une architecture ne présente pas de défaut d'un point de vue comportemental. Par exemple, on peut vérifier si deux éléments interconnectés ne se bloquent pas mutuellement.

Le style Composant-Connecteur permet la description d'architectures dynamiques ; le style offre des mécanismes pour la gestion de la dynamique dans une architecture. Nous différencions plusieurs aspects dans la dynamique : la création de nouveaux éléments à la volée, la reconfiguration dynamique (c.a.d. le changement de la structure) et la mobilité[49] (c.a.d le déplacement d'élément d'un environnement à un autre).

[49] Nous traitons de manière incomplète la mobilité en l'incorporant comme une propriété dynamique.

Une des nouveautés, que nous introduisons dans le style Composant-Connecteur, est la présence d'un élément spécifique à la gestion de la dynamique dans une architecture : le chorégraphe. A la différence des composants et des connecteurs, le chorégraphe n'est pas défini indépendamment de son environnement.

Afin de fournir une base de styles à l'utilisateur, nous avons défini ce que nous appelons les styles de fondation. Ce sont des styles répondant à des problèmes récurrents concernant des propriétés de performance, de modifiabilité,…. Nous avons défini le style Client-Serveur, le style Pipe-Filter, le style "en couche" (ou Layered), le style BlackBoard (ou DataIndirection). Ces styles sont définis comme des spécialisations du style Composant-Connecteur.

1.3. ASL ToolKit

Le ASL ToolKit est un outil intégré dans l'environnement ArchWare pour supporter les développements centrés architecture. C'est l'outil associé au langage ASL et à l'utilisation des styles.

1.3.1 Les services

Il fournit quatre services : la ***compilation***, l'***instanciation***, la ***vérification de satisfaction***, et l'***analyse***. Ce sont des services qui sont utilisés à différents stades d'un processus de développement centré architecture comme le montre la figure ci-dessous.

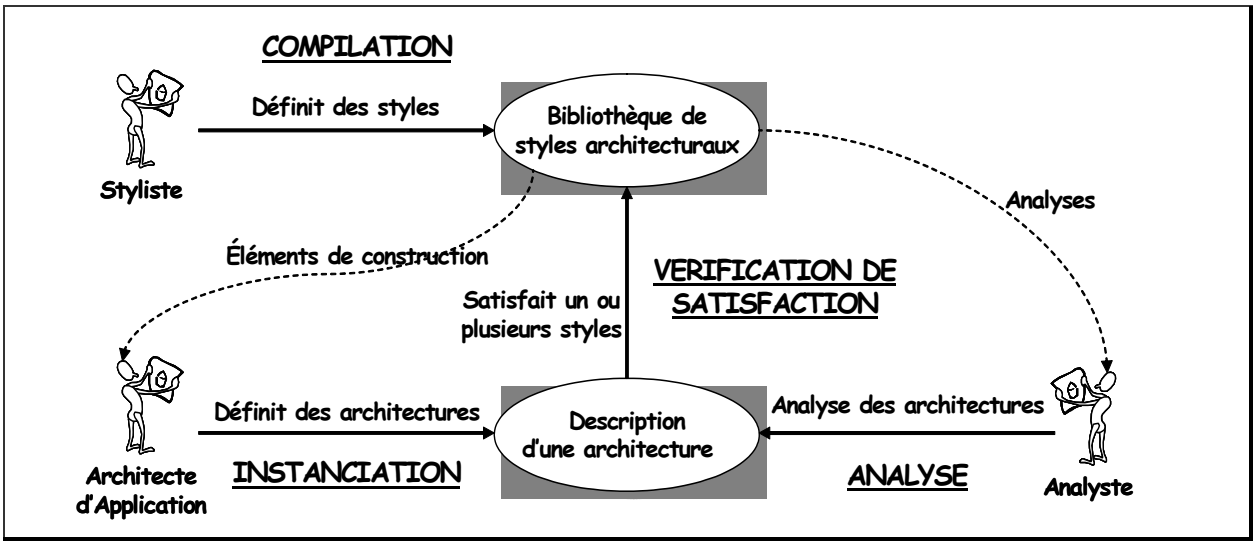


Figure 63 - Les fonctionnalités du ASL ToolKit utilisées dans le développement

1.3.2 Les technologies

L'implémentation du ASL ToolKit a demandé l'emploi de plusieurs technologies :

- les langages Prolog et Java,
- des APIs spécifiques à l'interaction entre une application Java et une application Prolog,
- les Services Web via Axis et le serveur web Tomcat.

1.3.3 Les chiffres

En termes de chiffres, le travail sur l'implémentation de ASL ToolKit représente :

- 6 classes Java et 10 modules Prolog,
- soit environ 7000 lignes de codes,
- une documentation d'une soixantaine de pages.

1.4. La validation

ASL, le style Composant-Connecteur et le ASL ToolKit ont été validés à travers des cas industriels à la fois dans le cadre du projet ArchWare et dans le cadre d'un projet pilote au CERN. Dans chacun des cas, l'objet est le développement d'un système dans une optique centrée architecture avec pour chacun des particularités différentes. Dans chacun de ces trois cas les styles jouent un rôle central.

Le premier cas concerne la conception d'un système de gestion des connaissances. Ce cas montre l'utilisation des styles architecturaux en vue de définir l'environnement de conception et de maintenance d'un système à base de connaissances. Les interactions se sont déroulées la plupart du temps à distance.

Le second cas concerne la conception d'un système pour la génération d'environnements pour le développement de systèmes d'Intégration d'Applications Industrielles. Ce cas montre l'utilisation des styles architecturaux en vue de définir une famille d'environnements propre au domaine des EAIs (Enterprise Application Integration). Dans ce cas, nous avons été directement impliqués et les interactions avec les autres acteurs ont été très directes.

Le dernier cas concerne la conception d'un système pour le système de surveillance du redémarrage d'un accélérateur de particules. Ce cas montre l'utilisation des styles architecturaux pour contraindre la conception d'interfaces graphiques. Les interactions avec les acteurs travaillant sur ce cas ont été moins nombreuses que dans les cas précédents et les travaux sortent un peu de l'esprit dans lequel nous avons effectué nos travaux. Ce cas présente aussi des perspectives intéressantes.

ASL a été validé sur les aspects suivants : la spécification des propriétés critiques des systèmes, la définition d'un support en terme d'éléments architecturaux et de patrons prédéfinis, et l'intégration des styles dans des environnements de développement.

Cette validation nous a conforté sur deux points :

- les utilisateurs ont apprécié fortement de pouvoir utiliser le formalisme associé au style Composant-Connecteur pour formaliser leurs architectures et leurs styles,
- il apparaît que les styles présentent un grand intérêt pour la génération d'environnements de développement spécifiques.

2. Positionnement

2.1. Positionnement d'ASL

Le tableau[50] suivant donne un positionnement d'ensemble du langage ASL par rapport aux critères que nous avons définis dans l'état de l'art concernant la formalisation des styles.

	ASL
Style	*style*
Vocabulaire	styles agrégats
Contraintes	*constraints*
- Topologiques	descriptions basées sur la logique des prédicats
- Comportementales	descriptions basées sur le μ-calcul.
- D'attributs	descriptions basées sur la logique des prédicats
Instanciation	Architecture crée avec l'environnement
Héritage	*extending* Un sous-style hérite des entités (styles, constructeurs, contraintes et analyses) définies dans le style parent. Il peut en redéfinir de nouvelles.
Dynamicité	La dynamicité est gérée comme le comportement.
Mobilité	La mobilité est gérée comme le comportement.

[50] Les mots en italique représente des mot-clés.

Outils d'exploitation	Environnement de développement, analyses, visualisation, …

Figure 64 - Caractéristiques des ASLs concernant la définition de styles

ASL se démarque des autres langages sur les trois points suivants.

2.1.1 ASL est un langage générique

ASL est un langage **générique** qui tient sa généricité d'ArchWare ADL. Les ADLs sont généralement basés sur un style ; généralement le style Composant-Connecteur. En étant un langage plus générique (basé sur les abstractions, les comportements et les connexions), le langage ArchWare ADL n'impose pas de règles propres à un style en particulier. ASL est donc très expressif ; il peut s'adapter à différents concepts architecturaux aussi spécifiques soient-ils.

2.1.2 ASL supporte les aspects dynamiques

ASL est un langage qui permet non seulement la définition de contraintes sur la structure des architectures mais aussi sur leurs **comportements**. Le problème des descriptions de contraintes sur les comportements avait déjà été posé pour WRIGHT [All 97]. En effet, ce langage, basé sur l'algèbre CSP [Hoa 85], propose la description d'architectures et de leurs comportements ainsi que celle des styles architecturaux. Cependant, le langage WRIGHT en est resté à la description de contraintes structurelles.

ASL permet la description de contraintes sur le comportement et par extension sur la **dynamique** d'une architecture. Il supporte les aspects dynamiques aussi bien au niveau de la définition des architectures qu'au niveau de la définition des contraintes. De plus, ces aspects dynamiques incluent quelques mécanismes liés à la mobilité. ASL est ainsi le seul langage à proposer des descriptions de styles pour les systèmes dynamiques. ASL supporte la définition de style autour des trois aspects suivants :

- la reconfiguration dynamique d'une architecture,
- l'instanciation des éléments architecturaux à l'exécution,
- la mobilité des éléments architecturaux.

2.1.3 ASL est flexible

ASL apparaît comme un méta-langage permettant de définir de nouveaux formalismes aux concepts spécifiques. ASL est la base d'un ensemble de formalismes hiérarchisés depuis des niveaux abstraits à des niveaux concrets. Ainsi, nous avons vu qu'ASL est très générique. Dans nos travaux, nous avons défini un formalisme dont les concepts architecturaux sont basés sur le style Composant-Connecteur. Ces concepts sont encore très abstraits est sont généralement considérés comme les concepts de bases pour la conception architecturale. Il s'agit d'une façon de voir les architectures composant-connecteur, l'utilisateur peut définir ces concepts comme il les entend. Enfin, pour des niveaux plus concrets, l'utilisateur peut définir son propre formalisme (à partir du style Composant-Connecteur par exemple) adapté à un domaine particulier. Nous avons vu, dans le chapitre 7, que certains utilisateurs désiraient définir un formalisme propre aux systèmes EAIs.

2.2. Positionnement du style Composant-Connecteur

Le style Composant-Connecteur définit un *formalisme* que nous appelons *C&C* pour la description des architectures Composant-Connecteur. Le tableau suivant donne une vue d'ensemble sur les critères du formalisme C&C associé au style Composant-Connecteur par rapport à ceux que nous avons définis dans l'état de l'art concernant la formalisation des architectures.

	C&C
Architecture	*component* composite
Elément de base (englobant la fonctionnalité)	*component*
- interface	*port*

Elément de connexion	*connector*
- interface	*port*
Attachement	*attachement*
Comportement	*behaviour* basé sur le π- calcul typé (ARCHWARE ADL)
Attributs	*attributes*
Implémentation	*wrappers*
Composition	*component* et *connector* composite
Dynamicité	Elément de gestion : *choreographer* Les attachements peuvent être crées dynamiquement. Les éléments architecturaux peuvent être instanciés à l'exécution.
Mobilité	Les components, connectors, et ports peuvent transiter via les attachements.

Figure 65 - Caractéristiques du C&C concernant la définition d'architectures

Les points forts que nous avons mis en avant avec le langage C&C sont les suivants :

- la généricité,
- la description d'architectures dynamiques,
- un mécanisme de définition d'attributs,
- la flexibilité du langage.

2.2.1 Généricité du langage

Tout comme une majorité d'ADL (ACME [GarMon&Wil 00], UNICON [Sha et al. 95], π-SPACE [Cha 02]), le langage C&C propose l'utilisation des composants et des connecteurs. Chaque ADL aborde ces concepts d'une manière plus ou moins différente, et impose des règles de construction spécifiques. Par exemple, ACME contraint un port d'être lié à un seul autre port ; π-SPACE ne permet pas deux connecteurs d'être reliés entre eux. Des règles de construction trop contraignantes peuvent être conflictuelles en regard de certains systèmes [Sha et al. 95]. Ainsi, bien que C&C soit décrit en ASL, nous l'avons construit le plus générique possible. Ainsi, les propriétés propres aux architectures composant-connecteur sont limitées à leur strict minimum.

2.2.2 Aspects comportementaux et dynamiques

C&C et d'autres ADLs (WRIGHT, RAPIDE et π-SPACE) permettent la description de comportements. Ces langages sont généralement basés sur des algèbres de processus. En se basant sur le π-calcul, C&C permet de décrire des comportements supportant la dynamique dans les architectures.

Peu d'ADLs gèrent la *dynamique*, parmi ceux-ci, π-SPACE nous a servi de base de travail. Nous avons amélioré la gestion de la dynamique sur quelques points. Premièrement, nous avons simplifié la définition des entités architecturales en ne différenciant pas les entités dynamiques des autres : toute entité est potentiellement dynamique. Deuxièmement, la gestion de la dynamique est différente. Dans π-SPACE, la dynamique est initiée par un composant atomique. Dans notre cas, afin de garantir une bonne réutilisation des composants et des connecteurs, nous les considérons comme des boîtes noires (ils sont définis indépendamment de leur environnement et n'ont aucune connaissance sur leurs attachements avec d'autres éléments). Ainsi, nous proposons d'introduire un nouveau type d'élément, le *chorégraphe*, pour gérer l'évolution au sein du composite qui le contient. Troisièmement, π-SPACE gère la décomposition et la recomposition des composites. C&C ne propose pas de mécanismes explicites pour ce genre d'évolution. Mais il le supporte à travers les actions d'extraction et d'inclusion qui permettent des reconfigurations dynamiques à des degrés plus fins car tout ne doit pas être décomposé pour extraire ou introduire un élément.

Concernant la *mobilité*, notre formalisme de C&C est le seul parmi les ADLs étudiés à la permettre, même si le problème de la mobilité n'a pas été approfondi avec C&C. Ainsi, il permet à des entités architecturales de transiter d'un environnement à un autre via des connexions.

2.2.3 Attributs

C&C permet aussi la définition des données qui ne sont ni structurelles ni comportementales à travers les attributs. Pour cela, UNICON et ACME utilisent un mécanisme d'annotation appelé *property*. Ces attributs ont pour but d'apporter des informations additionnelles sur le système. Alors

que les *properties* sont statiques dans UNICON et ACME, en C&C, les attributs sont modifiables à l'exécution et peuvent être accédés en lecture et en écriture par les éléments architecturaux.

2.2.4 Flexibilité

C&C a été écrit en ASL. Il peut lui-même être utilisé pour la définition de sous-styles tels que ceux que nous avons présentés dans le chapitre 5, ainsi que de formalismes adaptés. Parmi les langages étudiés dans le chapitre 3, seul AML propose des mécanismes d'extensions aussi poussés. La plupart des autres langages permettent seulement de redéfinir un vocabulaire pour les entités architecturales.

3. Perspectives

Les perspectives ouvertes par nos travaux peuvent se grouper selon trois axes (cf. Figure 66) : la continuation de nos travaux, les nouvelles applications et de nouveaux thèmes de recherche ouverts à l'issue de nos travaux.

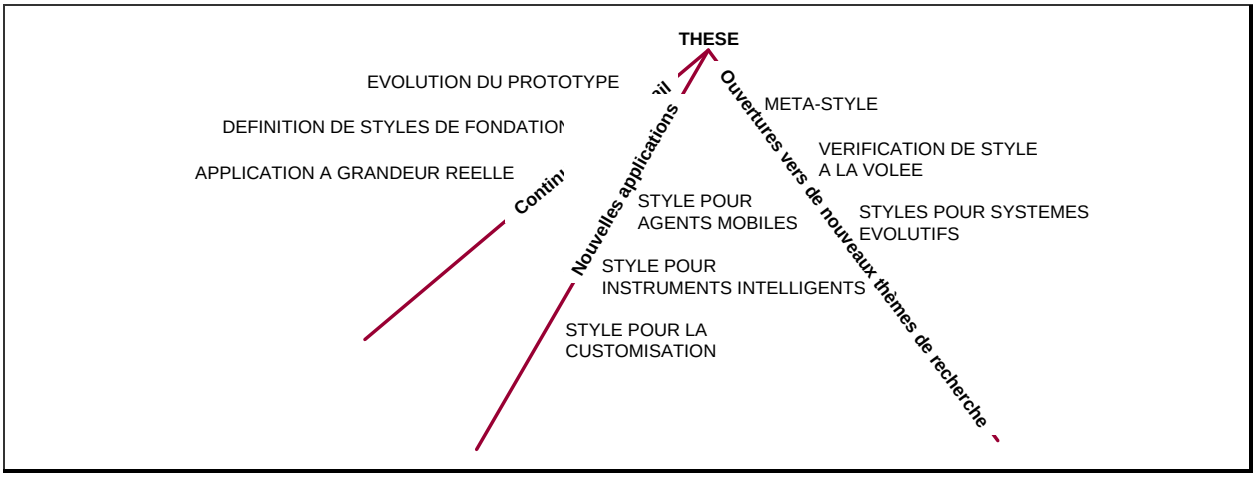


Figure 66 - Perspectives de recherche

Nous allons par la suite développer les perspectives mentionnées selon les trois axes.

3.1. Continuation des travaux entrepris

3.1.1 Style de fondations

Nous avons proposé quelques styles de fondation. Nous n'avons pas épuisé la liste des styles reconnus comme styles récurrents. Ainsi, nous envisageons d'enrichir la base des styles de fondation avec d'autres styles y compris pour les systèmes dynamiques. Nous pouvons notamment envisager de définir différents styles de chorégraphes (ces éléments qui gèrent la dynamique) lorsqu'après en avoir défini plusieurs des experts auront mis au jour différents modèles de ces éléments. Ces styles pourraient apporter des solutions architecturales reposant sur la dynamique en vue de promouvoir des attributs qualités.

3.1.2 Prototype

Concernant l'implémentation de l'outil, des améliorations et des extensions pourraient être apportées. Parmi les améliorations, on peut noter qu'à la compilation, les descriptions d'analyses spécifiques ne sont pas examinées. L'outil pourrait disposer d'une base dans laquelle on implanterait les BNFs et règles de typage propres à des formalismes nouveaux. Ces informations serviraient de support à l'outil pour la compilation des analyses.
Enfin, pour un outil plus abouti, on pourrait l'agrémenter de fonctionnalités offrant un support documentaire à l'architecte. On pourrait par exemple, fournir un document, après compilation, qui fournirait des informations sur l'utilisation du style et une définition de la notation qu'il fournit.

3.1.3 Validation

Les études de cas fournies par les travaux de nos partenaires industriels nous ont permis une première validation de nos travaux. L'application en grandeur réelle sur des projets industriels permettra une deuxième validation pour notre proposition. Dans ce contexte et dans la continuation de nos travaux, nous envisageons la mise en place de protocoles expérimentaux afin de pouvoir quantifier les gains relatifs en temps, en coût et en qualité dus à l'utilisation des styles. Dans cette approche, nous pourrions avoir des résultats à chacune des étapes suivantes : définition d'un style depuis le cahier des charges, définition d'une architecture et validation d'une architecture. De plus, cela permettra de valider l'utilisation de ASL ToolKit pour les analyses.

3.2. Nouvelles applications envisageables

3.2.1 Un style pour les agents mobiles

Le traitement de la mobilité, même de manière incomplète, est un des aspects qui distingue ASL des autres ADLs. Le domaine des agents mobiles est un domaine de recherche pour lequel on trouve des applications industrielles. Par exemple, Google, le célèbre moteur de recherche, procède à sa recherche par l'envoi de robots logiciels qui ont pour mission de récupérer des informations sur la toile. Cependant, pour définir des architectures et des styles pour des systèmes à agents mobiles il est nécessaire de pouvoir formaliser la mobilité et de la contraindre. Ainsi, ASL viendrait en première ligne pour formaliser un style pour les systèmes à agents mobiles. Des études sur la mobilité sont en cours (certains résultats étant déjà publiés [Azz&Oqu.04]) et pourraient permettre d'améliorer ASL dans cette direction.

3.2.2 Style pour les instruments intelligents

Dans la conception des instruments intelligents, les ingénieurs doivent suivre des règles de construction très strictes, afin de respecter des modèles topologiques et comportementaux, afin de garantir la sûreté de ces systèmes. Pour guider ces ingénieurs, ces modèles peuvent être formalisés comme des styles architecturaux. Cela est déjà le thème d'une thèse en cours à l'Université de Savoie (certains résultats étant déjà publiés [Rev et al. 03]).

3.2.3 Utilisation des styles pour la customisation des outils

Les styles peuvent fournir un support à la customisation. Dés lors qu'un style représente un outil, il peut présenter des points de variation pour permettre de spécialiser un outil. Il y a donc un début de piste quant à l'utilisation et l'adaptation des styles pour la customisation d'outils. Ceci peut mener à une étude englobant les lignes de produits, compte tenu qu'on peut considérer la customisation d'un outil comme un produit dans une famille de produits.

3.3. Nouveaux thèmes de recherche

3.3.1 Méta-style

Qu'est-ce qu'un méta-style? C'est un style de style. Un méta-style contraint l'architecture d'un style. Alors, qu'est ce que l'architecture d'un style? L'architecture d'un style est l'ensemble des entités composant un style et leurs relations. Par exemple, un style décrit en ASL présente des constructeurs, des contraintes et des analyses qui partagent les relations suivantes. Les constructeurs et les contraintes sont liés : un constructeur génère une description en ADL qui peut être validée vis à vis de certaines contraintes. Les contraintes et les analyses sont liées : les contraintes peuvent être composées d'analyses.

Mais à quoi peut donc servir un méta-style? Nous avons vu dans notre étude du domaine que les définitions concernant ce qu'est un style divergent. Pour certains, un style est un patron. Pour d'autres, il s'agit d'architectures très abstraites. Pour d'autres encore, ils permettent de formaliser une ligne ou une famille de produits. Les méta-styles permettraient de formaliser ces différentes définitions de styles. La formalisation d'un méta-style permettrait de vérifier la bonne écriture d'un style. Cela permettrait à l'utilisateur de choisir un style en fonction du méta-style qu'il suit. Il pourrait y avoir un méta-style destiné aux styles de support, un méta-style pour les familles de produits, un méta-style pour des styles supportant l'implémentation du code, etc.

C'est évidemment une idée qui mérite plus de réflexion et d'approfondissement. Mais ceci laisse entrevoir que la notion de style peut s'étendre au delà de ce que nous avons déjà pu voir.

3.3.2 Vérification de style à la volée

Les besoins en systèmes dynamiques ne cessent d'augmenter. Il serait intéressant de pouvoir faire évoluer les architectures de ces systèmes par les styles et avec les styles. Ainsi des styles pourraient être encapsulés dans une architecture afin de pouvoir générer des éléments architecturaux et afin de pouvoir vérifier des propriétés à la volée. De plus, les styles pourraient être modifiés au sein de l'architecture pour s'adapter à de nouveaux besoins.

3.3.3 Styles pour systèmes évolutifs

Nous donnons la possibilité de décrire des styles pour les systèmes dynamiques. C'est à dire des systèmes dont l'évolution est totalement programmée à la conception. Mais nous ne couvrons pas la totalité de la famille des systèmes évolutifs, ceux pouvant être modifiés en cours d'exécution, d'une manière prévue à la conception ou non. Quelques ADLs, comme π-SPACE, proposent la formalisation des architectures évolutives. Mais il reste une étude à mener pour savoir comment les styles peuvent offrir un support à la conception de ces systèmes.

REFERENCES BIBLIOGRAPHIQUES

Références Bibliographiques

[ABL] ABLE Group. *Toward an ADL Toolkit*. http://www-2.cs.cmu.edu/~acme/adltk/tools.html.

[Abd 96] A. Abd-Allah. *Composing Heterogeneous Software Architecture*. Thèse. Center for Software Engineering, University of Southern California, 1996.

[AboAll&Gar 93] G. Abowd, R. Allen et D. Garlan. *Using Style to Give Meaning to Software Architecture.* SIGSOFT'93:Foundation Software Eng., ACM, New York, 1993.

[AboAll&Gar 95] G. Abowd, R. Allen et D. Garlan. *Formalizing style to understand descriptions of software architecture.* ACM transaction on Software Engineering and Methodology, vol. 4, pp. 319-364, octobre 1995.

[All 97] R.J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, School of Computer Science, available as TR# CMU-CS-97-144, mai 1997.

[AllDou&Gar 98] R. Allen, R. Douence et D. Garlan. *Specifying and Analazing Dynamic Software Architectures*. Proceedings of 1998 Conference on Fundamental Approches to Software Engineering. Lisbon, Portugal, mars, 1998.

[AllDou&Gar 97] R. Allen, R. Douence, D. Garlan. *Specifying Dynamism in Software Architectures*. Proceedings of the workshop on foundations of component-based systems, pp. 11-22, Zurich, Switzerland, septembre 1997.

[All&Gar 94] R. Allen et D. Garlan. *Formalizing Architectural Connection.* Proceedings of 16th International Conference Software Engineering, IEEE Computer Soc. Press, Los Alamitos, Californie, pp. 71-80, 1994.

[All&Gar 97] R. Allen et D. Garlan. *A Formal Basis for Architectural Connection.* ACM Transactions on Software Engineering and Methodology, 6(3), juillet 97.

[All et al. 02] I. Alloui, H. Garavel, R. Mateescu et F. Oquendo. *The ArchWare Architecture Analysis Language*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D3.1b, décembre 2002.

[All&Oqu 03] I. Alloui et F. Oquendo. *Final UML ArchWare/Style-based ADL*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D1.4b, juin 2003.

[Ame et al. 00] P. America, H. Obbink, R. van Ommering et F. van der Linden. *CoPAM: A Component-Oriented Platform Architecting Method Family for Product Family Engineering*. Software Product Lines: Experience and Research Directions, Kluwer Academin Publishers, 2000.

[Arc 02] ArchWare team. *Architecting Evolvable Software*. ARCHWARE European RTD Project IST-2001-32360. www.arch-ware.org. 2002-2004.

[Azz et al.03] S. Azzaiez, F. Pourraz, H. Verjus et F. Oquendo. *Final ArchWare Architecture Animator - Release 1*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D2.2b, décembre 2003.

[Azz&Oqu.04] S. Azzaiez et F. Oquendo. *GAMA: Towards Architecture-centric Software Engineering of Mobile Agent Systems*. Proceedings of the Third

International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'04), 2004.

[Azz 06] S. Azzaiez. *Proposition d'une approche basée sur les agents pour l'évolution des architectures logicielles*. Thèse en cours. Université de Savoie. Soutenance prévue en 2006.

[Bal et al. 02] D. Balasubramaniam, R. Morrison, G. Kirby, K. Mickan et S. Norcross. *ArchWare Code Synthesiser*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D6.4a, mars 2004.

[Bar et al. 95] M. Barbacci, M.H. Klein, T.A. Longstaff, C.B. Weinstock. *Quality Attributes*. Technical Report CMU/SEI-95-TR-021. ESC-TR-95-021, décembre 95.

[BasCle&Kaz 99] L. Bass, P. Clements et R. Kazman. *Software Architecture in Practice*. Addison Wesley, ISBN 0-201-19930-0, 1999.

[Bat&O'M 92] D. Batory et S. O'Malley. *The Design and Implementation of Hierarchical Software Systems with Reusable Components*. ACM Transactions on Software Engineering and Methodology, 1(4), octobre 1992.

[Bat 98] D. Batory. *Product-Line Architectures*. Proceedings of the International Conference on Software Engineering (ICSE 1998), 1998.

[Bat&Ger 97] D. Batory et B. Geraci. *Composition Validation and Subjectivity in GenVoca Generators*. Proceedings of IEEE Transactions on Software Engineering, février 1997.

[Ber et al. 03] J. Bergey, S. Cohen, M. Fisher, L. Jones, L. Northrop, W. O'Brian. *Fifth DoD Product Line Practice Workshop Report*. Technical Report CMU/SEI-2003-TR-007, ESC-TR-2003-007, juin 2003.

[Ber&Inv 03] M. Bernardo et P. Inverardi (Eds.). *Formal Methods for Software Architectures*. LNCS 2804, septembre 2003.

[Ber et al. 04] D. Bergamini, D. Champelovier, N. Descoubes, H. Garavel, R. Mateescu et W. Serwe. *ArchWare Architecture Analysis Tool by Model-Checking*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D3.6b, juin 2004.

[BerDon&Cia 02] M. Bernardo, L. Donatiello et P. Ciancarini. *Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language*. Performance Evaluation of Complex Systems: Techniques and Tools, LNCS 2459:236-260, 2002.

[Ber&Bou 92] G. Berry et G. Boudol. *The Chimical Abstract Machin*. Theoritical Computer Science, 96:217-248, 1992.

[Bin&Ves 93] P. Binns et S. Vestal. *Formal real-time architecture specification and analysis*. Proceedings of 10th IEEE Workshop on Real-Time Operating Systems and Software, mai 1993.

[Bin et al. 96] P. Binns, M. Engelhart, M. Jackson et S. Vestal. *Domain-Specific Software Architectures for Guidance, Navigation, and Control*. International Journal of Software Engineering and Knowledge Engineering, 1996.

[Bla et al. 02] L. Blanc dit Jolicoeur, C. Braesch, R. Dindeleux, S. Gaspard, D. Le Berre, F. Leymonerie, A. Montaud, C. Chaudet, A. Haurat, et F. Théroude. *Final*

Specification of Business Case 1, Scenario and Initial Requirements. ARCHWARE European RTD Project IST-2001-32360. Deliverable D7.1b, décembre 2002.

[Bla et al. 03] L. Blanc dit Jolicoeur, R. Dindeleux, A. Montaud, F. Leymonerie, S. Gaspard et C. Braesch. *Definition of Architecture Styles and Process Model for Business Case1*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D7.2b, juin 2003.

[Bla et al. 04] L. Blanc dit Jolicoeur, C. Braesch et R. Dindeleux. *Customised ArchWare Engineering Environment for Business Case 1*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D7.3, mars 2004.

[Bla 04] L. Blanc dit Jolicoeur. *Modélisation des processus métiers mis en œuvre dans une approche EAI en vue de leur pilotage - Le pilotage des applications intégrées*. Thèse, Université de Savoie, soutenance est prévue en décembre 2004.

[Boa 95] M. Boasson. *The Artistry of Software Architecture.* Guest editor's introduction, IEEE Software, 1995.

[Boe et al. 94] B. Boehm, P. Bose, E. Horowitz et M. J. Lee. *Software requirements negotiation and renegociation aids: A theory-W based spiral approach*. Proceedings of the 17th International Conference on Software Engineering, 1994.

[Bol 04] T. Bolusset. *β-SPACE : Raffinement de descriptions architecturales en machines abstraites de la méthode formelle B*. Thèse, Université de Savoie, septembre 2004.

[Bos 99] J. Bosch. *Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study*. Software Architecture, Kluwer Academic Publishers, 1999.

[Bos 00] J. Bosch. *Design & Use of Software Architectures*. Addison-Wesley, 2000.

[Bus et al. 96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad et M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[CCC] Site de Life Cycle Management. http://www3.ca.com/Solutions/Product.asp?ID=255.

[CERN] Site web du CERN. http://www.cern.ch.

[Cha 02] C. Chaudet. *π-Space : Langage et outils pour la description d'architectures évolutives à composants dynamiques. Formalisation d'architectures logicielles et industrielles*. Thèse, Université de Savoie, décembre 2002.

[Cha&Oqu 01] C. Chaudet et F. Oquendo. *π-SPACE: Modeling Evolvable Distributed Software Architectures*. Proceedings of International Conference PDPTA'01, Las Vegas, juin 2001.

[Cia&Mas 96] P. Ciancarini et C. Mascolo. Analysing and Refining an Architectural Style. Proceedings of 10th International Conference of Z Users (ZUM'97), 1996.

[Cim et al. 02] S. Cimpan, F. Oquendo, D. Balasubramaniam, G. Kirby et R. Morrison. *The ArchWare ADL: Definition of the Textual Concrete Syntax*. ArchWare European RTD Project IST-2001-32360, Deliverable D1.2b, décembre 2002.

[Cim et al. 03] S. Cimpan, F. Leymonerie, H. Verjus et F. Oquendo. *Formalism for Model Specific Evaluation*. ArchWare European RTD Project IST-2001-32360, Deliverable D3.4, juillet 2003.

[CimLey&Oqu 03a] S. Cimpan, F. Leymonerie et F. Oquendo. *State of the art on architectural styles: Classification and Comparison for Software Architecture Description Languages*. ArchWare European RTD Project IST-2001-32360, ArchWare Document, février 2003.

[CimLey&Oqu 03b] S. Cimpan, F. Leymonerie et F. Oquendo. *ADL Foundation Style Library*. ArchWare European RTD Project IST-2001-32360, ArchWare Document, juin 2003.

[ClC] Page web sur ClearCase. http://www-306.ibm.com/software/awdtools/clearcase.

[Cle et al. 95] P. Clements, L. Bass, R. Kazman et G. Abowd. *Predicting software quality by architecture-level evaluation*. Proceedings of the Fifth International Conference on Software Quality, Austin, Texas, octobre 1995.

[Cle et al. 02] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord et J. Stafford. *Documenting Software Architectures: View and Beyond*. Addison Wesley, 2002.

[Cle 96] P. Clements. *A survey of Architecture Description Languages*. Proceedings of the Eight International workshop on Software Specification and Design, Germany, mars 1996.

[Coh 02] S. Cohen. *Product Line State of the Practice Report*. Technical Note, CMU/SEI-2002-TN-017, septembre 2002.

[Cog&Szy 93] L. Coglianese et R. Szymanski. *DSSA-ADAGE: An Environment for Architecture-based Avionics Development*. Proceedings of AGARD'93, mai 1993.

[DeB&Sch 99] J-M. DeBaud et K, Schmid. *A Systematic Appraoch to Derive Scope of Software Product Lines*. Proceedings of the International Conference on Software Engineering (ICSEE'99), 1999.

[DeL 96] R. DeLine. *Toward User-Defined Element Types and Architectural Styles*. Proceedings of the Second International Software Architecture Workshop (ISAW), San Francisco, 1996.

[DIp 89] R. D'Ippolito. *Using Models in Software Engineering*. Proceedings of Tri-Ada 89 ACM, 1989.

[DiN&Fug 96] E. Di Nitto et A. Fuggetta. *Product lines: what are the issues?*. Proceedings of the 10th International Software Process Workshop (ISPW '96), Dijon, France, juin 1996.

[DiN&Ros 99] E. Di Nitto et D. Rosenblum. *Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures.* Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, mai 1999.

[Doo] Pages Web sur Doors. http://www.telelogic.com/products/doorsers/doors/index.cfm

[Fro 89] B. Fromme. *HP Encapsulator: Bridging the Generation Gap*. Technical Report SESD-89-26, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, novembre 1989.

[Fux 00] A.D. Fuxman. *A survey of Architecture Description Languages*. CSC2108 Automated Verification, Fall '99, février 2000.

[Gam et al. 94] E. Gamma, R. Helm, R. Johnson et J. Vlissides. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[Gam et al. 95] E. Gamma, R. Helm, R. Johnson et J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design.* Addison-Wesley, Reading, mars 1995.

[Gar 95] D. Garlan, What is Style?, Proceedings of Dagshtul Workshop on Software Architecture, 1995.

[Gar 00] D. Garlan. *Software Architecture: a Road Map*. Proc. of the conference on The future of Software engineering, mai 2000.

[Gar 01] D. Garlan. *Software Architecture*. Wiley Encyclopedia of Software Engineering, J. Marciniak (Ed.), John Wiley & Sons, 2001.

[Gar 03] D. Garlan. *Formal Modeling and Analysis of Software Architecture : Components, Connectors and Events*. Présentation à the Third International School on Formal Methods for the Design of Computer, Communication and Software Architectures, SFM 2003 Bertinoro, Italy, septembre 2003. http://www.sti.uniurb.it/events/sfm03sa/slides/garlan-A.ppt

[GarAll&Ock 94] D. Garlan, R. Allen et J. Ockerbloom. *Exploiting Style in Architectural Design Environments.* SIGSOFT'94, ACM Press, New York, décembre 1994.

[GarMon&Wil 96] D. Garlan, R.T. Monroe et D. Wile. *ACME: An Architecture Description and Interchange Language.* Technical report, Carnegie Mellon University, Pittsburgh, 1996.

[GarMon&Wil 97] D. Garlan, R. Monroe et D. Wile. *ACME: an Architectural Description Interchange Language*. Proceedings of CASCON'97,Toronto, novembre 1997.

[GarMon&Wil 00] D. Garlan, R.T. Monroe et D. Wile. *ACME: Architectural Description of Composed-Based Systems.* Gary Leavens and Murali Sitaraman, ed.s Kluwer, 2000.

[Gar&Sha 93] D. Garlan et M. Shaw. *Introduction to Software Architecture.* Advances in Software Engineering and Knowledge Engineering. World Scientific Publishing Company, 1993.

[Gar et al. 92] D. Garlan, M. Shaw, C. Okasaki, C. Scott, and R. Swonger. *Experiences with a Course on Architectures for Software Systems*. Proceedings of the 6th SEI Conference on Software Engineering Education, 1992.

[Gom et al. 94] R.E. Lopez-Herrejon et D. Batory. *A Standard Problem for Evaluating Product-Line Methodologies*. Proceedings of the Third International Conference on

Generative and Component-Based Software Engineering, Springer-Verlag, 2001.

[Har 01] M. Harsu. *A Survey of Product-Line Architectures*. Technical report, Software Systems Labratory, Tampere University of Technology, octobre 2001.

[Hoa 85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hun 81] H. Hunke. *Software Engineering Environments.* North-Holland, 1981.

[Inv&Wol 95] P. Inverardi et A. Wolf. *Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model*. Proceedings of the IEEE transactions on software engineering, avril 1995.

[Kaz et al. 94] R.Kazman, L. Bass, G. Abowd et M. Webb. *SAAM: A Method for Analysing the Properties of Software Architectures*. Proceedings of the 16th International Conference on Software Engineering (ICSE), 1994.

[Kle et al. 99] M.H. Klein, R. Kazman, L.J. Bass , S.J. Carrière, M. Barbacci et H.F. Lipson. *Attribute-Based Architecture Styles*. Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), février, 1999.

[Kle&Kaz 99] M. Klein et R. Kazman. *Attribute-Based Architectural Styles*. Technical Report CMU/SEI-99-TR-022. ESC-TR-99-022, octobre 1999.

[Kog&Cle 95] P. Kogut et P. Clements. *Features of Architecture Description Languages*. Proceedings of the Software Technology Conference, Salt Lake City, avril 1995.

[Koz 83] D. Kozen. *Results on the Propositional Mu-Calculus*. Theoretical Computer Science, 1983.

[Lal 99] Philippe Lalanda. *Style-Specific Techniques to Design Product-Line Architectures*. Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, 1999.

[Lan 02] R. Land. *A Brief Survey of Software Architecture*. Technical report MRTC Report, Malardalen Real-Time Research Centre, Malardalen University, Suède, février 2002.

[LeBAll&Oqu 03] D. Le Berre, I. Alloui et F. Oquendo. *Preliminary ArchWare Architecture Analysis Tool by Theorem Proving*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D3.5a, décembre 2003.

[LeM 96] D. Le Métayer. *Software Architecture Styles as Graph Grammars*. Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering, ACM SIGSOFT, 1996.

[LeyCim&Oqu 01] F. Leymonerie, S. Cîmpan et F. Oquendo. *Extension d'un langage de description architecturale pour la prise en compte des styles architecturaux : Application à J2EE*. Proceedings of the 14th International Conference on Software and Software Engineering and their Applications (ICSSEA), Paris, décembre 2001.

[LeyCim&Oqu 02] F. Leymonerie, S. Cîmpan et F. Oquendo. *Etat de l'art sur les styles architecturaux : Classification et Comparaison des Langages de Description d'Architectures Logicielles*. Revue Génie Logiciel, No. 62, septembre 2002.

[Ley&Cim 04] F.Leymonerie et S. Cîmpan. *Style-Based Customiser*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D2.4, juin 2004.

[Ley et al. 03] F. Leymonerie, S. Cîmpan, H. Verjus et F. Oquendo. *Prelim ArchWare Arch Analysis Tool by Mode--Specific Evaluation*. ARCHWARE European RTD Project IST-2001-32360. Deliverable D3.7a, décembre 2003.

[Ley et al. 04] F. Leymonerie, L. Blanc Dit Jolicoeur, S. Cîmpan, C. Braesch et F. Oquendo. *Towards a business process formalisation based on an architecture centred approach*. Proceedings of the International Conference on Enterprise Information Systems (ICEIS 04), avril 2004.

[Li&Ram 85] C.H. Li et J. Ramanathan. *Beyond Isolated Systems for Programming-in-the-Small and Development-in-the-Large*. Proceedings of the Workshop on Software Engineering Environements for Programming-in-the-Large, Harwichport, juin 1985.

[LiuMcG&Epp 87] L. Liu, A. McGee et W. Epperley. *Recent Developments in Chemical Process and Plant Design*. Wiley, 1987.

[LoH&Bat 01] R.E. Lopez-Herrejon et D. Batory. *A Standard Problem for Evaluating Product-Line Methodologies*. Proceedings of the Third International Conference on Generative and Component-Based Software Engineering, Springer-Verlag, 2001.

[Luc et al. 95] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan et W. Mann. *Specification and analysis of system architecture using Rapide*. Proceedings of the IEEE Transaction on Software Engineering, 1995.

[Luc&Ver 95] D.C. Luckham et J. Vera. *An event-based architecture definition language*. Proceedings of the IEEE Transactions on software engineering, septembre 1995.

[Mag et al. 95] J. Magee, N. Dulay, S. Eisenbach et J. Kramer. *Specifying Distributed Software Architectures*. Proceedings of the Fifth European Engineering Conference (ESEC'95), Barcelona, septembre 1995.

[Mak 92] V.W. Mak. *Connection: An Inter-Component Communication Paradigm for Configurable Distributed Systems*. Proceedings of the International Workshop on Configurable Distributed Systems, Londres, UK, mars 1992.

[Mat 96] R. Mateescu. *Formal Description and Analysis of a Bounded Retransmission Protocol*. INRIA Research Report N°2965, août 1996.

[Med 96] N. Medvidovic. *ADLs and Dynamic Architecture Changes*. Proceedings of the Second International Software Architecture Workshop (ISAW-2). San Francisco, CA, octobre 1996.

[Med&Tay 97] N. Medvidovic. *A Classification and Comparison Framework for Software Architecture Description Languages.* Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, février 1997.

[Med&Tay 00] N. Medvidovic et R. Taylor. *A Framework for Classifying and Comparing Architecture Description Languages.* Proceedings of the IEEE Transactions on Software Enginering, 2000.

[MedRos&Tay 99] N. Medvidovic, D.S Rosemblum et R. Taylor. *A Language and Environment for Architecture-Based Software Development and Evolution.* Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, mai 1999.

[MedTay&Whi 96] N. Medvidovic, R. Taylor et E.J. Whitehead Jr. *Formal Modeling of Software Architectures at Multiple levels of abstraction*. Proceedings of the california software symposium 1996, pp 22-40, Los Angeles, CA, avril 1996.

[Med et al. 96] N. Medvidovic, P. Oreizy, D.S Rosemblum et R. Taylor. *Using Object-Oriented Typing to Support Architectural Design in the C2 style.* Proceedings of ACM SIGSOFT'96: Fourth symposium on the foundations of software engineering (FSE4), pp. 24-32, San Francisco, CA, octobre 1996.

[MedOre&Tay 97] N. Medvidovic, P. Oreizy et R. N. Taylor. *Reuse of offthe-shelf components in c2-style architectures*. Proceedings of the 1997 Symposium on Software Reusability, 1997.

[Meg&Oqu 04] K. Megzari et F. Oquendo. *ArchWare Architecture Refinement Tools*. ArchWare European RTD Project IST-2001-32360, Deliverable D6.3b, avril 2004.

[Met&Gra 92] E. Mettala et M. H. Graham. *The domain-specific software architecture program*. Technical report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, juin 1992.

[Mil 89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil et al. 1992] R. Milner, J. Parraw et D. Walker. *A calculus of mobile processes*. Information and Computation, septembre 1992.

[Mil 99] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge university press, 1999.

[Mon 01] R.T. Monroe. *Capturing Software Architecture Design Expertise With Armani*. Tech. Report CMU-CS-98-163, 2001.

[Mon et al. 97] R.T. Monroe, A. Kompanek, R. Melton et D. Garlan. *Architectural Styles, Design Patterns, and Objects.* Proceedings of the IEEE Transactions on Software Enginering, 1997

[Mon&Gar 96] R. Monroe, D. Garlan. *Style-Based Reuse for Software Architectures*. Proceedings of the 1996 International Conference on Software Reuse, avril 1996.

[Mor et al. 04] R. Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cîmpan, B. Warboys, B. Snowdon, R. M. Greenwood. *Support for Evolving Software Architectures in the ArchWare ADL*. Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04). Oslo, Norvège, juin 2004.

[Occ&Fab 03a] C. Occhipinti et P. Fabriani. *Final Specification of Business Case 2 Scenario and Initial Requirements*. ArchWare European RTD Project IST-2001-32360, Deliverable D8.1b, février 2003.

[Occ&Fab 03b] C. Occhipinti et P. Fabriani. *Definition of Architectural Styles and Process Models for the Business Case 2*. ArchWare European RTD Project IST-2001-32360, Deliverable D8.2, décembre 2003.

[Occ&Fab 04] C. Occhipinti et P. Fabriani. *Customised ArchWare Engineering Environment for Business Case 1*. ArchWare European RTD Project IST-2001-32360, Deliverable D8.3, mars 2004.

[Occ&Zav 03] C. Occhipinti et C. Zavattari. *Final ArchWare Architecture Modeller User Manual*. ArchWare European RTD Project IST-2001-32360, Deliverable D2.1b, décembre 2003.

[Oqu 03] F. Oquendo. *The ArchWare Architecture Description Language: Tutorial*. Report R1.1-1, ArchWare European RTD Project, IST-2001-32360, mars 2003.

[Oqu et al. 02] F. Oquendo, I. Alloui, S. Cimpan et H. Verjus. *The ArchWare ADL: Definition of the Abstract Syntax and Formal Semantics*. ArchWare European RTD Project IST-2001-32360, Deliverable D1.1b, décembre 2002.

[Oqu 03b] F. Oquendo. *Final Definition of the ArchWare. Architecture Refinement Language (ArchWare ARL)*. ArchWare European RTD Project IST-2001-32360, Deliverable D6.1b, décembre 2003.

[Par 01] D.L. Parnas. *On the Design and Development of Software Families*. Software fundamentals: collected papers by David L. Parnas, Addison-Wesley Longman Publishing Co, 2001.

[Per 98] D. Perry. *Generic Architecture Descriptions for Product Lines*. F. van der Linden, editor, Development and Evolution of Software Architectures for Product Lines. Proceedings of the Second International ESPRIT ARES Workshop, LNCS 1429, pages 51–56. Springer, 1998.

[Per&Wol 92] D. Perry et A. Wolf. *Foundations for the Study of Software Architecture.* ACM SIGSOFT, Software Engineering Notes, Vol. 17, no 4, pp. 40-52, octobre 1992.

[Pre 95] W. Pree. Design *Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, Mass. 1995.

[Rap] Page web sur Rapide : http://www-2.cs.cmu.edu/~acme/adltk/adls.html#rapide

[Rat] Page web sur Rational Rose. http://www-306.ibm.com/software/rational/

[Rat et al. 03] O. Ratcliffe, S. Cîmpan, F. Oquendo et L. Scibile. *Formalizing an architectural style specific to process monitoring applications*. Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications (ICSSEA 2003), décembre 2003.

[Rat et al. 04a] O. Ratcliffe, S. Cîmpan, F. Oquendo et L. Scibile. *Formalization of an HCI Style for Accelerator Restart Monitoring*. Proceedings of the First European Workshop on Software Architectures (EWSA 2004), mai 2004.

[Rat et al. 04b] O. Ratcliffe, S. Cîmpan, F. Oquendo et L. Scibile. *Towards Inductive Definition an Evolution of Architectural Styles*. Proceedings of the 17th International Conference on Software & Systems Engineering and their Applications (ICSSEA 2004), décembre 2004.

[Rat 04] O. Ratcliffe. *Approche et environnement fondés sur les styles architecturaux pour le développement de logiciels propres à des spécifiques - Application au domaine de la supervision au redémarrage d'accélérateurs de particules*. Thèse, Université de Savoie, décembre 2004.

[RDT 97] Rapide Design Team. *Rapide 1.0. Pattern Language*. Reference Manual, juillet 1997.

[Req] Page web sur Requisite Pro. http://www-306.ibm.com/software/awdtools/reqpro/

[Rev et al. 03] J. Revillard, E. Benoit, S. Cîmpan et F. Oquendo. Architectural Style for Intelligent Instrument. Proceedings of ICSSEA, Paris, décembre 2003.

[Rev 05] J. Revillard. *Styles architecturaux pour la conception d'instruments intelligents*. Thèse en cours. Université de Savoie. Soutenance prévue en 2005.

[Rha] Page web sur Rhapsody. http://www.ilogix.com/rhapsody/rhapsody.cfm

[Ric&Sei 97] M.D. Rice et S.B. Seidman. *Using Architectural Style to Support Software Understanding and Reuse*. Position Paper, WISR8 - 8th Annual Workshop on Software Reuse, Ohio State University, mars 1997.

[Ric&Sei 99] M.D. Rice et S.B. Seidman. *Describing Software Architectures and Architectural Styles*. First Working IFIP Conference on Software Architecture, (WICSA1), San Antonio, Texas, février 1999.

[Rum et al. 91] J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Tay et al. 96] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J.W. Jr., J.E. Robbins, K.A. Nies, P.Oreizy et D.L. Dubrow. *A Component- and Message-Based Architectural Style for GUI Software.* IEEE Transactions on Software Engineering and Methodology.22(6), juin 1996.

[Sha 90] M. Shaw. *Prospects for an Engineering Discipline of Software*. IEEE Software, novembre 1990.

[Sha 94] M. Shaw. *Patterns for Software Architectures*. J. Coplien and D. Schmidt (Eds.), Pattern Languages of Program Design, Addison-Wesley, 1995 (from the First Annual Conference on Pattern Languages of Programming, août 1994).

[Sha 95] M. Shaw. *Some Patterns for Software Architecture*. J. Vlissides, J. Coplien and N. Kerth (Eds.), Pattern Languages of Program Design, Vol. 2, Addison-Wesley, 1996 (from the Second Annual Conference on Pattern Languages of Programming, septembre 1995).

[Sha&Cle 96a] M. Shaw et P. Clements. *Toward Boxology: Preliminary Classification of Architectural Styles.* Proceedings of the Second International Software Architecture Workshop, 1996.

[Sha&Cle 96b] M. Shaw et P. Clements. *How Should Patterns Influence Architecture Description Languages? A Call for Discussion*. Working paper for DARPA EDCS community, juillet 1996.

[Sha&Cle 97] M. Shaw et P. Clements. *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*. Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC'97), 1997.

[Sha&Gar 96] M. Shaw et D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Englewood Cliffs, N.J., 1996.

[Sha et al. 95] M.Shaw, R.DeLine, D.Klein, T.Ross, D.Young et G.Zelesnik. *Abstractions for Software Architecture and Tools to Support Them*. Proceedings of the IEEE Transactions on Software Engineering: Special Issue on Software Architecture, avril 1995.

[Spi 89] J.M. Spivey. *The Z Notation, A Reference Manual*. Prentice-Hall, Englewood Cliffs, N.J., 1989.

[StaRic&Wol 98] J. A. Stafford, D. J. Richardson, A. L. Wolf. *Aladdin: A Tool for Architecture-Level Dependence Analysis of Software*. University of Colorado at Boulder, Technical Report CU-CS-858-98, avril 1993.

[Sti 01] C. Stirling. *Modal and Temporal Properties of Processes*. Springer Verlag, 2001.

[Tar 55] A. Tarski. *A Lattice-Theoretical Fixpoint Theorem and its Applications*. Pacific Journal of Mathematics 5:285-309, 1955.

[USB 00] D Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips. *Universal Serial Bus Specification*. Revision 2.0, avril 2000.

[Ver&Oqu 03] H. Verjus et F. Oquendo. *Final XML ArchWare style-based ADL (ArchWare AXL)*. ArchWare European RTD Project IST-2001-32360, Deliverable D1.3b, juin 03.

[Ves 92] S. Vestal. *MetaH Reference Manual*. Honeywell Technology Center, Minneapolis MN, 1992.

[Ves 94] S. Vestal. *Mode Changes in Real-Time Architecture Description Language*. Proceedings of the Second International Workshop on Configurable Distributed Systems, mars 1994.

[VdH 02] A. Van der Hoek. *Representing Product-Line Architectures.* Ground System Architectures Worshops (GSAW2002), 2002.

[Wal 98] A. Wall. *Software Architectures - An Overview*. Department of Computer Engineering, MälardalenUniversity, 1998.

[Wei et Lai 99] D.M. Weiss et C.T.R. Lai. *Software Product-Line Engineering*. Addison-Wesley, 1999.

[Wil 99] D. Wile. *AML: An Architecture Meta Language*. Proceedings of the 14th International Conference on Automated Software Engineering. Cocoa Beach, FL, octobre 1999.

[Wil 01] D. Wile. *Using Dynamic ACME*. Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Australie, décembre 2001.

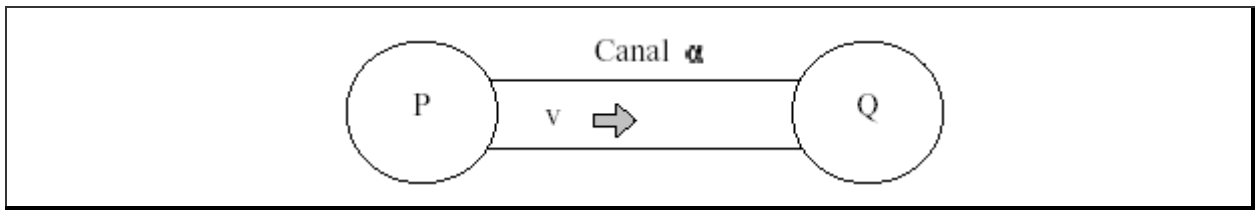
[XSB] Page Web du système XSB Prolog. http://xsb.sourceforge.net.

ANNEXES

Annexes

1. Le π-calcul

Le π-calcul est une algèbre de processus sur laquelle se base de nombreux travaux dans le domaine des architectures. Les algèbres de processus permettent de représenter un ensemble de processus et leurs interactions. Par exemple, la figure ci-dessous illustre un processus *P* qui communique une valeur *v* au processus *Q* à travers un canal de transmission α.



Il existe plusieurs algèbres de processus dont CSP [Hoa 85], CCS [Mil 89] et le π-calcul [Mil 99].

C'est sur le π-calcul qu'est basé ArchWare ADL, un des langages sur lequel nous avons basé nos travaux. Le π-calcul est le support formel pour la description des aspects dynamiques et des aspects comportementaux. Le π-calcul a été choisi car cette algèbre introduit le concept de mobilité, c'est-à-dire la possibilité de faire communiquer les liens de communication entre les processus. Cette propriété permet de définir la dynamique.

La grammaire pour définir un processus P en π-calcul est la suivante :
P ::= **0** | x(y)•P | x y•P | τ•P | $P_1|P_2$ | (ν x)P | P_1+P_2 | [x = y]P | A (x_1, …, x_n)

Dans les lignes suivantes, nous étudions les différents concepts supportés par le π-calcul.

Le processus inactif

0 représente le processus inactif. C'est le constituant de base de tout processus, car un processus finira par ne plus rien faire (sauf dans le cas de la récursivité).
La structure du π-calcul peut être définie de manière inductive, c'est-à-dire qu'elle va être construite autour du processus ***0***. Les autres processus sont construits à partir de ***0*** et d'actions de communications organisées via les opérateurs décrits dans les prochains paragraphes.

La préfixation

• est l'opérateur de préfixation. L'expression $a \bullet P$ spécifie que l'action *a* s'effectue avant le processus P. Il existe trois types d'actions possibles : l'émission, la réception et l'action inobservable τ.
L'expression $P \equiv \overline{\alpha} v \bullet \mathbf{0}$ décrit un processus *P* qui émet la valeur *v* par le canal α puis s'arrête ;
L'expression $Q \equiv (x) \bullet R$ décrit un processus *Q* un processus qui reçoit la variable *x* par le canal α, puis continue en se comportant comme un processus *R* (pouvant contenir des occurrences de *x*).

Le parallélisme

Deux (ou plusieurs) processus peuvent évoluer en parallèle, ceci est noté : *P* | *Q*. Dans ce cas et si ces processus possèdent des canaux de mêmes noms, ils peuvent communiquer.
Dans l'exemple suivant, où *P* vaut $\overline{\alpha} v \bullet \mathbf{0}$ et *Q* vaut $\alpha (x) \bullet R \bullet \mathbf{0}$, *v* va être transmis par le canal α à un moment donné.

$$\overline{\alpha} v \bullet \mathbf{0} \mid \alpha (x) \bullet R \bullet \mathbf{0} \xrightarrow{\tau} \mathbf{0} \mid R\{v/x\} \bullet \mathbf{0}$$

Après une action inobservable τ, le système a évolué en transmettant *v* du processus *P* au processus *Q*. En effet, les *x* libres qui se trouvent dans le processus *R* seront substitués par des *v*. (Les notions de "variables libres" et de "variables liées" sont expliquées dans le paragraphe sur la restriction).

Nous avons mentionné que le π-calcul permet la mobilité. En effet, un canal peut être transmis d'un processus à un autre. C'est-à-dire qu'un canal peut transiter via un autre canal. Cette particularité apporte des priorités dynamiques : un processus peut acquérir dynamiquement un nouveau canal. L'exemple suivant, montre cette capacité de communiquer un canal. Soit le système *P* | *Q* | *R* avec

$$\overline{\alpha}\,v \bullet \mathbf{0} \mid \alpha\,(x) \bullet \overline{\alpha}\,(y) \bullet \mathbf{0} \mid \alpha\,(x) \bullet S \xrightarrow{\tau} \mathbf{0} \mid R\{v/x\} \bullet \mathbf{0}$$

La restriction

L'opérateur de restriction ν permet de spécifier la portée d'une variable (d'un nom). On distingue ainsi les noms libres, ceux qui ne sont restreints à aucune portée, des noms liés.

Considérons le système suivant : P | Q | R avec :

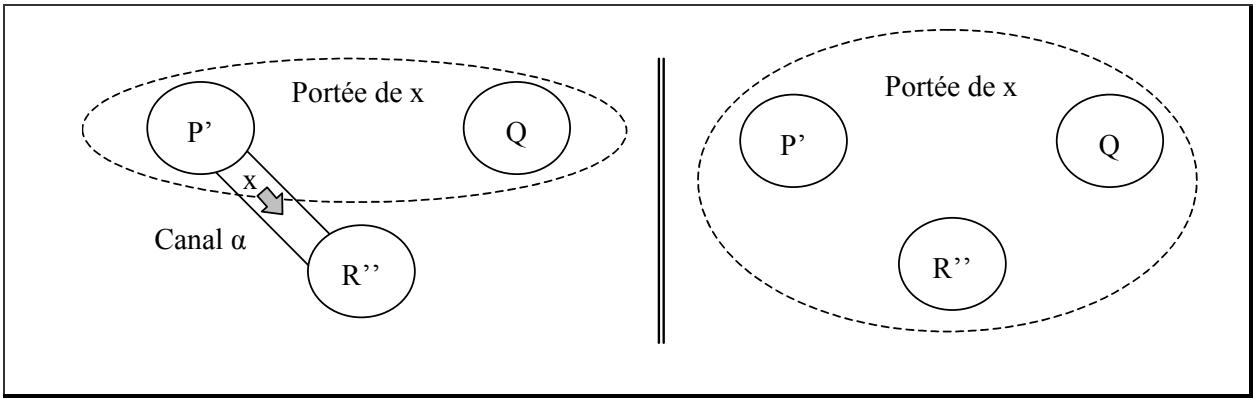
$P \equiv \overline{\alpha}\,v \bullet \mathbf{0}$ et $Q = R \equiv \alpha\,(x) \bullet \mathbf{0}$

Alors *P* peut transmettre la valeur v aussi bien à *Q* qu'à *R*.

Pour restreindre la communication seulement entre *P* et *Q* il faudra faire la restriction *v* sur le canal *α*, de la manière suivante : *(v α) (P | Q) | R*.
La communication sur le canal *α* ne pourra se faire qu'entre *P* et *Q* ; c'est-à-dire que *R* n'a pas accès au même canal *α* que celui utilisé par *P* et *Q*. Dans cet exemple les occurrences du canal *α* dans les processus *P* et *Q* sont dites "variables liées", et dans *R α* est une "variable libre".

Une extension de portée « Scope extrusion » est possible. Par exemple :
Si *P* possède un canal privé *x* avec *Q* : *(vx) (P | Q)*,
et souhaite le transmettre à *R* : *(vx) (P | Q | R),* alors il y a une extension de portée.

Ex : $(vx)\,(\overline{\alpha}\,x \bullet P' \mid Q) \mid \alpha\,(y) \bullet R' \xrightarrow{\tau} (vx)\,(P' \mid Q \mid R'\{x/y\})$



Toutefois, la substitution de *y* par *x* : *R'{x/y}* n'est possible que si *R* ne possède pas déjà un canal *x*, sinon il faut renommer le canal privé *x* pour préserver la différence avec le canal public. On étend alors la portée de *x* aux trois processus *P'*, *Q* et *R''* (avec R'' = R'{x/y} : *R''* est le processus résultant de la substitution de *y* par *x* dans *R'*).

Le choix indéterministe

Il est possible de spécifier plusieurs évolutions possibles d'un processus. L'opérateur + permet de distinguer les différents choix possibles. Le choix de l'évolution est un choix indéterministe.

Un autre opérateur que nous devons introduire est l'opérateur de choix : +

$$(\alpha\,(x) \bullet R + \beta\,(y) \bullet S) \mid \overline{\alpha}\,v \bullet \mathbf{0} \mid \overline{\beta}\,w \bullet \mathbf{0}$$

Ce système peut évoluer :

- soit vers : $R\{v/x\} \mid \mathbf{0} \mid \bar{\beta}\, w \bullet \mathbf{0}$
- soit vers : $S\{w/y\} \mid \bar{\alpha}\, v \bullet \mathbf{0} \mid \mathbf{0}$

L'appariement de forme

L'appariement de forme permet de spécifier une condition qui doit être vraie pour pouvoir exécuter un processus.

Pour spécifier le choix, nous pouvons utiliser l'appariement de forme "matching" :

$$[x = a]P + [x = b]Q$$

Le processus se comportera comme *P* si *x* et *a* sont identiques, comme *Q* si *x* et *b* sont identiques et dans les autres cas comme ***0***.

Il existe aussi le non appariement de forme "mismatching" :

$$[x \neq a]P$$

Le processus se comportera comme *P* si *x* et *a* sont différents.

Ainsi, on peut facilement traduire un "si, alors, sinon" par :

$$[x = a]P + [x \neq a]\, R \qquad (\text{si } x = a \text{ alors } P \text{ sinon } R)$$

Récursivité

Il est possible de définir un processus en fonction de lui-même. Il s'agit alors d'un processus récursif.

Un processus peut être exprimé de la manière suivante :

$$A(x_1, \ldots, x_n) \overset{\text{def}}{=} P$$

C'est-à-dire qu'un processus peut être remplacé par un nom avec ses canaux libres en paramètres.

L'avantage principal de cette notation est la possibilité de décrire des processus récursifs. Par exemple, un processus qui ne ferait qu'émettre indéfiniment la valeur *a* sur le canal *x* serait :

$$\text{Emission}(x) \overset{\text{def}}{=} \bar{x}\, a \bullet \text{Emission}(x)$$

2. Le μ-calcul

Le μ-calcul [Koz 83] est un formalisme basé sur le concept de point-fixe pour la spécification de propriétés temporelles sur les systèmes concourants. Il regroupe virtuellement toutes les logiques temporelles définies dans la littérature et peut être vu comme un langage d'assemblage pour les propriétés temporelles. Une documentation détaillée du μ-calculus modal peut être trouvée dans [Sti 01]. Nous présentons ici une brève vue d'ensemble du μ-calculus en considérant une version étendant légèrement la version utilisée dans [Sti 01].

Nous considérons les Systèmes à Transition Etiquetée (Labelled Transition Systems – LTSs) qui conviennent pour les langages de description basés sur les actions telles que les algèbres de processus. Un modèle de LTS $M = (S, A, T, s_0)$ consiste en un ensemble d'états S, un ensemble d'actions A, une relation de transition $T \subseteq S \times A \times S$ et un état initial $s_0 \in S$. Une transition $(s_1, a, s_2) \in T$ signifie qu'un système peut évoluer de l'état s_1 à l'état s_2 en effectuant l'action *a*. Nous supposons que tout état *S* est atteignable depuis l'état s_0 après une séquence de transitions.

Nous considérons ici une variante du μ-calcul reposant sur des formules d'actions, notées α, et des formules d'états (notées φ) définies selon la grammaire suivante.

$$\alpha ::= a \mid F \mid T \mid \neg\alpha_1 \mid \alpha_1 \vee \alpha_2 \mid \alpha_1 \wedge \alpha_2$$

$\varphi ::= F \mid T \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid < \alpha > \varphi_1 \mid [\ \alpha\ ]\ \varphi_1 \mid X \mid \mu X.\varphi_1 \mid \nu X.\varphi_1$

Les formules d'actions α expriment des prédicats sur l'ensemble des actions A d'un LTS. Elles sont définies à partir d'actions $a \in A$ et de connectives booléennes. D'autres opérateurs booléens peuvent être dérivés, par exemple: $a_1 \Rightarrow a_2 = \neg a_1 \vee a_2$, $a_1 \Leftrightarrow a_2 = (a_1 \Rightarrow a_2) \wedge (a_2 \Rightarrow a_1)$, etc. Les formules d'état expriment des prédicats sur l'ensemble des états S d'un LTS. Elles sont définies par des connectives booléennes, des opérateurs modaux de possibilité ($< \alpha > \varphi$) et de nécessité ($[\ \alpha\ ]\ \varphi$), et des opérateurs[51] de point fixe maximal et minimal $\mu X.\varphi$ et $\nu X.\varphi$, où X appartient à un ensemble de variables propositionnelles $\boldsymbol{X}$.

Les opérateurs de points fixes ont le rôle de lieurs pour les variables propositionnelles de la même manière que les quantificateurs dans la logique du premier ordre. Les ensembles de variables libres et liées dans une formule d'état φ sont notés respectivement $fv(\varphi)$ et $bv(\varphi)$. Une formule d'état est dite *fermée* si $fv(\varphi) = \emptyset$; elle ne contient aucune variable libre. Une formule d'état est dite dans sa *forme normal* si $fv(\varphi) \cap bv(\varphi) = \emptyset$; elle ne contient aucune variable à la fois libre est liée. Afin d'assurer des sémantiques bien définies, les formules d'états doivent être *syntaxiquement monotonique* [Koz 83].

Les sémantiques dénotationnelles des formules d'actions et d'états vis-à-vis d'un modèle LTS $M = (S, A, T, s_0)$ sont définies dans le tableau ci-dessous. L'interprétation $[[\ \alpha\ ]]$ d'une formule d'actions α dénote l'ensemble des actions la satisfaisant. L'interprétation $[[\ \varphi\ ]]\ \rho$ d'une formule d'états, où $\rho : \boldsymbol{X} \rightarrow 2^S$ est un contexte propositionnel associant des ensembles d'états à des variables propositionnelles, dénote l'ensemble des états satisfaisant φ dans le contexte ρ. La notation $\rho[U / X]$ représente l'*extension* d'un contexte propositionnel ρ ; un contexte identique à ρ sauf pour la variable X assignée à l'ensemble d'états U. Pour les formules fermées φ, nous noterons simplement $[[\ \varphi\ ]]$, puisque les sémantiques de ces formules ne dépendent d'aucun contexte propositionnel.

$[[\ a\ ]] = \{\ a\ \}$ $[[\ F\ ]] = \emptyset$ $[[\ T\ ]] = A$ $[[\ \neg\alpha_1\ ]] = A \setminus [[\ \alpha_1\ ]]$ $[[\ \alpha_1 \vee \alpha_2\ ]] = [[\ \alpha_1\ ]] \cup [[\ \alpha_2\ ]]$ $[[\ \alpha_1 \wedge \alpha_2\ ]] = [[\ \alpha_1\ ]] \cap [[\ \alpha_2\ ]]$
$[[\ F\ ]]\ \rho = \emptyset$ $[[\ T\ ]]\ \rho = S$ $[[\ \neg\varphi_1\ ]]\ \rho = S \setminus [[\ \varphi_1\ ]]\ \rho$ $[[\ \varphi_1 \vee \varphi_2\ ]]\ \rho = [[\ \varphi_1\ ]]\ \rho \cup [[\ \varphi_2\ ]]\ \rho$ $[[\ \varphi_1 \wedge \varphi_2\ ]]\ \rho = [[\ \varphi_1\ ]]\ \rho \cap [[\ \varphi_2\ ]]\ \rho$ $[[\ < \alpha > \varphi_1\ ]]\ \rho = \{\ s \in S \mid \exists (s, a, s') \in T.\ a \in [[\ \alpha\ ]] \wedge s' \in [[\ \varphi_1]]\ \rho\ \}$ $[[\ [\ \alpha\ ]\ \varphi_1\ ]]\ \rho = \{\ s \in S \mid \forall (s, a, s') \in T.\ a \in [[\ \alpha\ ]] \Rightarrow s' \in [[\ \varphi_1]]\ \rho\ \}$ $[[\ X\ ]]\ \rho = \rho\ (X)$ $[[\ \mu X.\varphi_1\ ]]\ \rho = \cap \{\ U \subseteq S \mid [[\ \varphi_1\ ]]\ \rho[U / X] \subseteq U\ \}$ $[[\ \nu X.\varphi_1\ ]]\ \rho = \cup \{\ U \subseteq S \mid U \subseteq [[\ \varphi_1\ ]]\ \rho[U / X]\ \}$

L'opérateur modal de possibilité $< \alpha > \varphi$ caractérise les états à partir desquels il y a une transition étiquetée par une action satisfaisant α et menant à un état satisfaisant φ. L'opérateur modal de nécessité $[\ \alpha\ ]\ \varphi$ caractérise les états à partir desquels toutes les transitions étiquetées par une action satisfaisant α mènent à un état satisfaisant φ. Intuitivement, les opérateurs modaux permettent de décrire des arbres d'exécution finis au niveau du LTS. L'opérateur de point fixe minimal $\mu X.\varphi$ et l'opérateur de point fixe maximal $\nu X.\varphi$ caractérisent la plus petite et la plus grande solution sur 2^S de l'équation $X = \varphi$. Intuitivement les opérateurs de points fixes minimaux et

[51] Dans la suite, nous utilisons le symbole σ pour dénoter μ ou ν.

maximaux permettent de décrire de manière récursive des patrons "arborés" respectivement fini et infini dans le LTS.

L'opérateur modal de nécessité est le dual de l'opérateur modal de possibilité : $[\ \alpha\ ]\ \varphi = \neg < \alpha > \neg\varphi$. L'opérateur de point fixe maximal est le dual de l'opérateur de point fixe minimal : $\nu X.\varphi = \neg\mu X.\neg\varphi\ [\neg X\ /\ X]$, ou la notation $\varphi\ [\neg X\ /\ X]$ représente la substitution syntaxique de X par $\neg X$ dans φ. En utilisant ces dualités et les lois de De Morgan, toute formule d'états syntaxiquement monotonique peut être convertie dans une *forme normale positive* (FNP) en "abaissant" ses négations jusqu'à atteindre des sous-formules atomiques. Dans une formule de point fixe $\sigma X.\varphi$ en FNP toutes les occurrences libres de X dans φ sont positives (non précédée par une négation), ce qui garantit la monotonicité de la formule vis-à-vis de X ; le fait que $U \subseteq V \Rightarrow [[\ \varphi\ ]]\ \rho[U\ /\ X] \subseteq [[\ \varphi\ ]]\ \rho[V\ /\ X]$ pour tout ensemble d'état U, $V \subseteq S$ et tout contexte propositionnel ρ. Par le théorème de Tarski , il suit que l'interprétation des formules de points fixes données dans le tableau ci-dessus dénote les solutions de point fixes de l'équation $X = \varphi$. Le tableau ci-dessous indique une liste de tautologies typiques entre les modalités.

$< \alpha > F = < F > \varphi = F$
$[\ \alpha\ ]\ T = [\ F\ ]\ \varphi = T$
$< \alpha_1 \vee \alpha_2 > \varphi = < \alpha_1 > \varphi \vee < \alpha_2 > \varphi$
$[\ \alpha_1 \vee \alpha_2\ ]\ \varphi = [\ \alpha_1\ ]\ \varphi \wedge [\ \alpha_2\ ]\ \varphi$

Un état s du LTS satisfait une formule d'états fermée φ (notation : $s \models \varphi$) si et seulement si $s \in [[\ \varphi\ ]]$. Un LTS satisfait φ si et seulement si son état initial s_0 satisfait φ.

3. Règles syntaxiques du langage ASL

Nous définissons la syntaxe du langage ASL avec une version étendue du méta-langage Backus-Naur Form (EBNF).

Style Declaration

style_declaration ::=
    style_id_declaration
style_id_declaration ::=
    identifier **is** style
style ::=
    style [**extending (** identifier **)**] **where {** style_definition**}**
style_definition ::=
    [type_declarations]
    [style_declarations]
    [constructor_declarations]
    [constraint_declarations]
    [analysis_declarations]
style_declarations ::=
    styles { style_id_declaration_list **}**
style_id_declaration_list ::=
    style_id_declaration **,** style_id_declaration_list

Constructor Declaration

constructor_declarations ::=
    constructors { constructor_declaration_list **}**

constructor_id_declaration_list ::=
constructor_id_declaration, constructor_id_declaration_list
constructor_id_declaration ::=
identifier **is** clause

Constraint Declaration

constraint_declarations ::=
constraints { property_expression **}**

Analysis declaration

analysis_declarations ::=
analyses {analysis_id_declaration_list **}**
analysis_id_declaration_list ::=
analysis_id_declaration [**,** analysis_id_declaration_list]
analysis_id_declaration ::=
identifier **is** analysis_definition
analysis_definition ::=
analysis {
[input_declaration]
output_declaration
analysis_body_definition
[mixfix_definition]
}
input_declaration ::=
input { parameter_id_declaration_list **}**
output_declaration ::=
output { type **}**
analysis_body_definition ::=
body { aal_property_definition **}**
analysis_kind ::=
identifier

Description

description ::=
declaration [**;** description]
| clause [**;** description]
declaration ::=
type_declaration
| value_declaration

Type declaration

types_declaration ::=
types { type_id_declaration_list **}**
type_id_declaration_list ::=
type_id_declaration **,** type_id_declaration_list
type_declaration ::=

type type_id_declaration
| **recursive type** type_id_declaration [**and** type_id_declaration]*
type_id_declaration ::=
identifier **is** type

Value declaration

value_declaration ::=
value value_id_declaration
| **recursive value** literal_id_declaration [**and** literal_id_declaration]*
value_id_declaration ::=
identifier_list **is** clause
literal_id_declaration ::=
identifier **is** literal

Type descriptor

type ::=

Any	-- top type
Natural | Integer | Real | Boolean | String	-- base types
Type | Expression [T **]** | **Alias**	-- constructor specific types
identifier	-- type alias
tuple [type_list **]**	-- tuple type
view [labelled_type_list **]**	-- view type
union [type_list **]**	-- union type
variant [labelled_type_list **]**	-- variant type
quote [identifier **]**	-- quote type
_ identifier	-- shorthand for quote type
set [type **]**	-- set type
bag [type **]**	-- bag type
sequence [type **]**	-- sequence type
location [type **]**	-- location type
connection [[type_list] **]**	-- connection type
Behaviour	-- behaviour type
abstraction [[type_list] **]**	-- abstraction type
function[[type_list] **]** -> type	-- function type

type_list ::=
type [**,** type]*
labelled_type_list ::=
identifier **:** type [**,** identifier **:** type]*

Clause (behaviour clause)

clause ::=

behaviour { [clause] **}**	-- behaviour
action_prefix [**;** clause]	-- action prefix
if clause **do** clause	-- matching prefix
if clause **then** clause **else** clause	-- unmatching prefix

```
|       choose { [choice_list] }                          -- summation
|       iterate clause [by identifier : type]             -- iteration
        from value_declaration accumulate clause
        [as identifier] [; clause ]          -- iteration accumulating value as identifier
|       iterate clause [by identifier : type]
        do clause [; clause ]                -- iteration not accumulating value
|       project clause as identifier_list ; clause       -- projection for tuple and view
|       project clause as identifier ;                   -- projection for variant, union and any
        case { project_list } [; clause ]
|       project clause at clause as identifier ; clause  -- projection for sequence
|       expression [; clause]
action_prefix ::=
        via clause send [clause_list]                    -- output prefix
|       via clause receive [labelled_type_list]          -- input prefix
|       unobservable                                     -- silent prefix
choice_list ::=
        clause [or clause]*
identifier_list ::=
        identifier [, identifier]*
project_list  ::=
        variant_project_list
|       union_project_list
|       any_project_list
variant_project_list ::=
        identifier do clause [or identifier do clause]* or default do clause
union_project_list ::=
        type do clause [or type do clause]* or default do clause
clause_list ::=
        clause [, clause]*
connection_identifier ::=
        clause
|       identifier :: clause
```

Expression

```
expression ::=
        (clause )                            -- parenthesis
|       {description }                       -- brackets
|       literal                              -- literal values
|       not expression                       -- boolean negation
|       expression and expression            -- boolean conjunction
|       expression or expression             -- boolean disjunction
|       expression xor expression            -- boolean exclusive disjunction
|       expression implies expression        -- boolean implication
|       expression relational_operator expression   --comparison expression
|       add_operator expression              -- signed expression
|       expression add_operator expression   -- arithmetic additive expression
```

```
|       expression multiply_operator expression    -- arithmetic multiplicative expression
|       expression ++ expression                   -- string concatenation
|       expression ( clause | clause )             -- substring
|       eval expression
|       toString expression
|       any ([clause] )                            -- injection into any
|       tuple ( clause_list )                      -- tuple value
|       view ( value_id_declaration_list )         -- view value
|       clause :: identifier                       -- shorthand for view projection
|       clause :: clause                           -- shorthand for sequence projection
|       union ( clause ) : type                    -- union value
|       variant ( identifier is clause ) : type    -- variant value
|       quote ( identifier )                       -- quote value
|       _identifier                                -- shorthand for quote value
|       location ( clause )                        -- location value
|       ‘clause                                    -- dereference location
|       identifier := clause                   -- assignment in location returns location
|       set ( collection_clause_list )         -- set value
|       bag ( collection_clause_list )         -- bag value
|       sequence ( collection_clause_list )    -- sequence value
|       sequence using clause values clause    -- sequence value by repetition
|       clause :: clause                       -- shorthand for sequence projection
|       expression includes expression         -- insertion in collection
|       expression excludes expression         -- deletion one from collection
|       expression excludes all expression     -- deletion all from collection
|       expression ( [clause_list] )           --  application
|       mixfix_application
|       component_connector_application
|       identifier
mixfix_application:=
        [identifier*[clause]*]*
value_id_declaration_list ::=
        identifier is clause [, identifier is clause]*
relational_operator ::=
        equality_operator
|       comparison_operator
equality_operator ::=
        == | <>
comparison_operator ::=
        < | <= | > | =>
add_operator ::=
        + | -
multiply_operator ::=
        integer_multiply_operator
|       real_multiply_operator
integer_multiply_operator ::=
        *                               -- integer multiplication
```

```
|               \                         -- integer division
|               %                         -- modulo
real_multiply_operator ::=
                *                         -- real multiplication
|               /                         -- real division
collection_clause_list ::=
                clause_list
|               clause .. clause          -- range
```

Literal

```
literal         ::=
                natural_literal
|               integer_literal
|               real_literal
|               boolean_literal
|               string_literal
|               tuple_literal
|               view_literal                          -- nil constructed literals
|               set_literal
|               bag_literal
|               sequence_literal                      -- empty collection literals
|               connection_literal                    -- connection literal
|               behaviour_literal                     -- inaction literal
|               abstraction_literal                   -- abstraction literal
|               constructor_literal                   -- constructor literal
|               function_literal                      -- function literal
natural_literal ::=
                 # digit [digit]*
integer_literal ::=
                [add_operator] digit [digit]*
boolean_literal ::=
                true | false
string_literal ::=
                "[character]*"
tuple_literal ::=
                tuple () : type
view_literal ::=
                view () : type
set_literal     ::=
                set () : type
bag_literal     ::=
                bag () : type
sequence_literal ::=
                sequence () : type
real_literal                    ::=
                integer_literal.[digit]*[e integer_literal]
```

connection_literal ::=
connection ([type_list] **)**
behaviour_literal ::=
done
abstraction_literal ::=
abstraction ([labelled_type_list] **)**
constructor_literal ::=
constructor ([labelled_type_list] **) ;** clause [**as {** mixfix **}**]
function_literal ::=
function([identifier_type_list] **)** [**->** type] **;** clause
mixfix ::=
[mixfix_term]*
mixfix_term ::=
identifier -- mixfix keyword
| **$**identifier -- parameter identifier
| **[**mixfix**]** -- optionality

Property expression

property_expression ::=
predicate_formula
| action_formula
| regular_formula
| state_formula
value :=
variable
| function(value_list)
value_list :=
value [**,**value]*

Predicate formula

predicate_formula ::=
value -- expression of boolean type
| predicate_indexing
| predicate_application
| predicate_existential_quantification
| predicate_universal_quantification
| predicate_construct_formula
predicate_indexing ::=
collectionValue **.** function (value_list)
predicate_application ::=
to collectionValue **apply** predicate_formula
| **to** collectionValue **apply {** predicate_formula [,predicate_formula]* **}**

predicate_existential_quantification ::=
exists { cardinalised_identifier_list | predicate_formula }
predicate_universal_quantification ::=

forall { identifier_list | predicate_formula }
predicate_construct_formula ::=
project { tupleValue **as** variable1, ..., variablen **|** predicate_formula }
| **project {** viewValue **as** label1, ..., labeln **|** predicate_formula }
| **project {** unionValue **as** variable | **case** valueType1 **do** predicate_formula1 ...
case valueTypen **do** predicate_formulan }
| **project {** variantValue **as** variable **| case** label1 do predicate_formula1...
case labeln **do** predicate_formulan **}**
| **project {** locationValue **as** variable **|** predicate_formula **}**
| **iterate {** collectionValue **by** iterator **from** initialisation
accumulate accumulation **as** result | predicate_formula **}**
cardinalised_identifier_list ::=
cardinalised_identifier [**,** cardinalised_identifier]*
cardinalised_identifier ::=
[literal...literal**]** identifier
| **[**literal**]** identifier

Action formula

action_formula ::=
action_predicate
| action_unobservable
| action_truth_value
| action_negation
| action_disjunction
| action_exclusive_disjunction
| action_conjunction
| action_implication
| action_equivalence
| action_existential_quantification
| action_universal_quantification
action_predicate ::=
action_send
| action_receive
| action_send_wildcard
| action_receive_wildcard
action_send ::=
via value **send** value
action_receive ::=
via value **receive** value
action_send_wildcard ::=
via value **send any**
action_receive_wildcard ::=
via value **receive any**
action_unobservable ::=
unobservable
action_truth_value ::=

false
| **true**
action_negation ::=
not action_formula
action_disjunction ::=
action_formula **or** action_formula
action_exclusive_disjunction ::=
action_formula **xor** action_formula
action_conjunction ::=
action_formula **and** action_formula
action_implication ::=
action_formula **implies** action_formula
action_equivalence ::=
action_formula **equivalent** action_formula
action_existential_quantification ::=
exists (identifier_type_list | action_formula)
action_universal_quantification ::=
forall (identifier_type_list | action_formula)

Regular formula

regular_formula ::=
action_formula
| regular_concatenation
| regular_choice
| regular_star_closure
| regular_plus_closure
| regular_empty_sequence
regular_concatenation ::=
regular_formula **.** regular_formula
regular_choice ::=
regular_formula | regular_formula
regular_star_closure ::=
regular_formula *
regular_plus_closure ::=
regular_formula +
regular_empty_sequence ::=
nil

State formula

state_formula ::=
boolean_formula
| state_negation
| state_disjunction
| state_exclusive_disjunction
| state_conjunction
| state_implication

| state_equivalence
| state_existential_quantification
| state_universal_quantification
| state_modal_ possibility
| state_modal_necessity
| state_propositional_variable
| state_minimal_fixed_point
| state_maximal_fixed_point

state_negation ::=
not state_formula

state_disjunction ::=
state_formula **or** state_formula

state_exclusive_disjunction ::=
state_formula **xor** state_formula

state_conjunction ::=
state_formula **and** state_formula

state_implication ::=
state_formula **implies** state_formula

state_equivalence ::=
state_formula **equivalent** state_formula

state_existential_quantification ::=
exists (identifier_type_list | state_formula)

state_universal_quantification ::=
forall (identifier_type_list | state_formula)

state_modal_ possibility ::=
some sequence { regular_formula }
leads to state { state_formula }

state_modal_necessity ::=
every sequence { regular_formula }
leads to state { state_formula }

state_propositional_variable ::=
prop_variable (value_list)

state_minimal_fixed_point ::=
finite tree prop_variable [(identifier_type_list)]
given by { state_formula } (value_list)

state_maximal_fixed_point ::=
infinite tree prop_variable [(identifier_type_list)]
given by { state_formula } (value_list)

prop_variable ::=
identifier

Identifier

identifier ::= letter [letter | digit | _]*

letter ::= **a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z**

digit ::= **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

character ::= *any ASCII character including:*

'b -- backspace (ASCII code 8)
't -- horizontal tab (ASCII code 9)
'n -- newline (ASCII code 10)
'p -- newpage (ASCII code 12)
'o -- carriage return (ASCII code 13)

4. Règles de typages

Les règles de typage d'ASL sont utilisées en conjonction avec les règles syntaxiques pour déterminer l'ensemble des descriptions bien écrites.
Les lignes qui suivent expliquent la construction de ces règles de typages. D'abord, nous introduisons le concept d'environnement. Deux sortes d'environnements sont utilisées. Ces environnement sont des ensembles de liens : les uns pour les identifiants des valeurs, les autres pour les identifiants de types. δ dénote les environnements où les identifiants de valeurs sont liés à leur types sous la forme <x, T>, où x est une identifiant et T est un type. τ dénote l'environnement dans lequel les identifiants des types sont liés à des expressions de types de la forme <t, T>, où t est un identifiant et T est un type. $A_1::b::A_2$ est utilisé pour représenté une liste A qui contient un lien b. A++B est utilisé pour dénoter la concaténation de deux listes de liens A et B. δ_V, δ_A, τ_V et τ_A sont des environnements globaux et supporte les structures en blocs. (b_1, …, b_n) est une liste contenant les liens de b_1 à b_n. La méta-fonction *type_declaration* prend en paramètre une liste de liens entre les identifiants de types et les expressions de types, et l'ajoute à l'environnement τ.
Similairement, la méta-fonction *id_declaration* prend en paramètre une liste de liens entre les identifiants et les types et l'ajoute dans l'environnement δ.
Les règles de typage utilisent la structure des règles de preuve. Par exemple, si A_i est vrai pour i = 1, . . n alors B est vrai. Chaque A_i et B peuvent être de la forme $X \mapsto Y$ ce qui dénote que Y est déductible d'une collection d'environnement X. Ainsi, la règle de typage

$$\frac{\tau, \delta \mapsto e_1 : \mathbf{Integer} \quad \tau \ \delta \mapsto e_2 : \mathbf{Integer}}{\tau, \delta \mapsto e_1 + e_2 : \mathbf{Integer}}$$

est lu comme "si l'expression e_1 est de type **Integer** dans les environnements τ et δ, et l'expression e_2 est de type **Integer** dans les environnements τ et δ alors le type de l'expression e_1 + e_2 est déduit être **Integer** dans les environnements τ et δ.

Declarations

[declarations]

$$\frac{\tau, \delta \mapsto D_1 : \mathbf{void} \quad \text{type\_declaration}(\Omega_1) \qquad \text{id\_declaration}(\Psi_1) \qquad \tau + +\Omega_1, \delta + +\Psi_1 \mapsto D_2 : T}{\tau, \delta \mapsto D_1 ; D_2 : T}$$

[type declaration]

$$\frac{\tau \mapsto T \in \text{ ValueType}}{\tau, \delta \mapsto \mathbf{let\ type}\ \text{name}\ \mathbf{be}\ T : \mathbf{void} \quad \text{type\_declaration}(<\text{name}, T>)}$$

[recursive type declaration]

$$\frac{\tau \mapsto T_1 \in \text{Type} \quad \tau \mapsto T_2 \in \text{Type}}{\tau' \mapsto \mathbf{recursive\ let\ type}\ t_1\ \mathbf{be}\ T_1\ \&\ t_2\ \mathbf{be}\ T_2 : \mathbf{void} \quad \text{type\_declaration}(<t_1, T_1>, <t_2, T_2>)}$$

where τ stands for $\tau_1::\tau_2::\tau_3$ and τ' stands for $\tau_1::<t_1, T_1>::\tau_2::<t_2, T_2>::\tau_3$

[value declaration]

$$\frac{\tau \mapsto T \in \text{ ValueType} \quad \tau, \delta \mapsto e : T}{\tau, \delta \mapsto \textbf{let } \text{name} = e : \textbf{void} \quad \text{id\_declaration}(< \text{name}, T >)}$$

[recursive value declaration]

$$\frac{\tau, \delta \mapsto e_1 : T_1 \quad \tau, \delta \mapsto e_2 : T_2}{\tau, \delta' \mapsto \textbf{recursive let } i_1 = e_1 \textbf{ \& } i_2 = e_2 : \textbf{void} \quad \text{id\_declaration}(< i_1, T_1 >, < i_2, T_2 >)}$$

where δ stands for $\delta_1 :: \delta_2 :: \delta_3$ and δ' stands for $\delta_1 :: < i_1, T_1 > :: \delta_2 :: < i_2, T_2 > :: \delta_3$

Clauses

[restriction]

$$\frac{\tau \mapsto T_1 \in \text{ConnectionType} \ \ldots \ \tau \mapsto T_n \in \text{ConnectionType} \quad \tau, \delta :: < i_1, T_1 > :: \ldots :: < i_n, T_n > \mapsto B : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{restrict } i_1 : T_1 \textbf{,...,} i_n : T_n \textbf{ in } \{B\} : \textbf{behaviour} \quad \text{id\_declaration}(< i_1, T_1 >, \ldots, < i_n, T_n >)}$$

[composition]

$$\frac{\tau, \delta \mapsto B_1 : \textbf{behaviour} \quad \tau, \delta \mapsto B_2 : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{compose } [B_1 \textbf{ and } B_2] : \textbf{behaviour}}$$

[choice]

$$\frac{\tau, \delta \mapsto B_1 : \textbf{behaviour} \quad \tau, \delta \mapsto B_2 : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{choose } [B_1 \textbf{ or } B_2] : \textbf{behaviour}}$$

[conditioned behaviour]

$$\frac{\tau, \delta \mapsto \text{expr} : \textbf{Boolean} \quad \tau, \delta \mapsto B : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{if } \text{expr } \textbf{do } B : \textbf{behaviour}}$$

[replication]

$$\frac{\tau, \delta \mapsto B : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{replicate } B : \textbf{behaviour}}$$

[unobservable]

$$\frac{}{\tau, \delta \mapsto \textbf{unobservable} : \textbf{void}}$$

[receive 1]

$$\frac{\tau \mapsto T \in \text{ValueType} \ \tau, \delta \mapsto \text{name} : \textbf{connection}\{T\} \quad \tau, \delta \mapsto x : T}{\tau, \delta \mapsto \textbf{via } \text{name } \textbf{receive } x : \textbf{void}}$$

[receive 2]

$$\frac{\tau \mapsto T \in \text{ValueType} \ \tau, \delta \mapsto \text{name} : \textbf{connection}\{T\}}{\tau, \delta \mapsto \textbf{via } \text{name } \textbf{receive } x \textbf{ : } T : \textbf{void} \quad \text{id\_declaration}(< x, T >)}$$

[send]

$$\frac{\tau \mapsto T \in \text{ValueType} \quad \tau, \delta \mapsto \text{name} : \textbf{connection}\{T\} \quad \tau, \delta \mapsto x : T}{\tau, \delta \mapsto \textbf{via } \text{name } \textbf{send } x : \textbf{void}}$$

Expressions

Boolean

[negation]

$$\frac{\tau, \delta \mapsto e : \textbf{Boolean}}{\tau, \delta \mapsto \textbf{not } e : \textbf{Boolean}}$$

[or]

$$\frac{\tau, \delta \mapsto e_1 : \textbf{Boolean} \ \tau, \delta \mapsto e_2 : \textbf{Boolean}}{\tau, \delta \mapsto e_1 \textbf{ or } e_2 : \textbf{Boolean}}$$

[and] $$\frac{\tau,\delta \mapsto e_1 : \textbf{Boolean} \;\; \tau,\delta \mapsto e_2 : \textbf{Boolean}}{\tau,\delta \mapsto e_1 \textbf{ and } e_2 : \textbf{Boolean}}$$

[implies] $$\frac{\tau,\delta \mapsto e_1 : \textbf{Boolean} \;\; \tau,\delta \mapsto e_2 : \textbf{Boolean}}{\tau,\delta \mapsto e_1 \textbf{ implies } e_2 : \textbf{Boolean}}$$

Comparison

[equality] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 = e_2 : \textbf{Boolean}}$$

[non equality] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 \sim= e_2 : \textbf{Boolean}}$$

In the following, T ∈ { **Natural**, **Integer**, **Real**, **String** }

[less] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 < e_2 : \textbf{Boolean}}$$

[less equal] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 <= e_2 : \textbf{Boolean}}$$

[greater] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 > e_2 : \textbf{Boolean}}$$

[greater equal] $$\frac{\tau,\delta \mapsto e_1 : T \;\; \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 >= e_2 : \textbf{Boolean}}$$

Numeric Expression

In the following type rules, T ∈ { **Integer**, **Real** }, T' ∈ { **Natural**, I**nteger**, **Real** } and T" ∈ { **Natural**, **Integer** }.

[plus] $$\frac{\tau,\delta \mapsto e : T}{\tau,\delta \mapsto + e : T}$$

[minus] $$\frac{\tau,\delta \mapsto e : T}{\tau,\delta \mapsto - e : T}$$

[add] $$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 + e_2 : T'}$$

[subtract] $$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 - e_2 : T'}$$

[times] $$\frac{\tau,\delta \mapsto e_1 : T' \;\; \tau,\delta \mapsto e_2 : T'}{\tau,\delta \mapsto e_1 * e_2 : T'}$$

[division] $$\frac{\tau,\delta \mapsto e_1 : T'' \;\; \tau,\delta \mapsto e_2 : T''}{\tau,\delta \mapsto e_1 \textbf{ div } e_2 : T''}$$

[remainder] $$\frac{\tau,\delta \mapsto e_1 : T'' \;\; \tau,\delta \mapsto e_2 : T''}{\tau,\delta \mapsto e_1 \textbf{ rem } e_2 : T''}$$

[real division] $$\frac{\tau,\delta \mapsto e_1 : \textbf{Real} \;\; \tau,\delta \mapsto e_2 : \textbf{Real}}{\tau,\delta \mapsto e_1 / e_2 : \textbf{Real}}$$

String Expression

[concatenation] $$\frac{\tau,\delta \mapsto e_1 : \textbf{String}\ \ \tau,\delta \mapsto e_2 : \textbf{String}}{\tau,\delta \mapsto e_1 ++\ e_2 : \textbf{String}}$$

[substring] $$\frac{\tau,\delta \mapsto e : \textbf{String} \quad \tau,\delta \mapsto e_1 : \textbf{Natural} \quad \tau,\delta \mapsto e_2 : \textbf{Natural}}{\tau,\delta \mapsto e\,(e_1 \,|\, e_2)\ :\ \textbf{String}}$$

Collection Expression

[included equal] $$\frac{\tau,\delta \mapsto e_1 : T\ \ \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 <=\ e_2 : \textbf{Boolean}}$$ where T ∈ { **set**, **bag** }

[included] $$\frac{\tau,\delta \mapsto e_1 : T\ \ \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 <\ e_2 : \textbf{Boolean}}$$ where T ∈ { **set**, **bag** }

[includes equal] $$\frac{\tau,\delta \mapsto e_1 : T\ \ \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 >=\ e_2 : \textbf{Boolean}}$$ where T ∈ { **set**, **bag** }

[includes] $$\frac{\tau,\delta \mapsto e_1 : T\ \ \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 >\ e_2 : \textbf{Boolean}}$$ where T ∈ { **set**, **bag** }

[intersection] $$\frac{\tau,\delta \mapsto e_1 : T\ \ \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto \textbf{intersection}(e_1\,, e_2) : T}$$ where T ∈ { **set**, **bag** }

[reunion] $$\frac{\tau,\delta \mapsto e_1 : T\ \ \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto \textbf{reunion(}e_1\,,\ e_2\textbf{)} : T}$$ where T ∈ { **set**, **bag** }

[concatenation] $$\frac{\tau,\delta \mapsto e_1 : T\ \ \tau,\delta \mapsto e_2 : T}{\tau,\delta \mapsto e_1 ++\ e_2 : T}$$ where T ∈ { **sequence** }

Literals

[natural literal] $$\frac{}{\tau,\delta \mapsto n : \textbf{Natural}}\ n \in \text{Natural}$$

[integer literal] $$\frac{}{\tau,\delta \mapsto i : \textbf{Integer}}\ i \in \text{Integer}$$

[real literal] $$\frac{}{\tau,\delta \mapsto r : \textbf{Real}}\ r \in \text{Real}$$

[boolean literal] $$\frac{}{\tau,\delta \mapsto b : \textbf{Boolean}}\ b \in \text{Boolean}$$

[string literal]

$$\frac{}{\tau,\delta \mapsto s : \textbf{String}}\ s \in \text{String}$$

[connection literal]

$$\frac{\tau \mapsto T \in \text{ValueType}}{\tau,\delta \mapsto \textbf{connection(}T\textbf{)}\ :\textbf{connection[}T\textbf{]}}$$

[behaviour literal]

$$\frac{}{\tau,\delta \mapsto \textbf{done} : \textbf{behaviour}}$$

[quote literal]

$$\frac{i \in \text{Identifier}}{\tau \mapsto i : \textbf{quote}\ \ i}$$

[abstraction literal]

$$\frac{\tau \mapsto VT_1 \in \text{ValueType}, ..., \tau \mapsto VT_n \in \text{ValueType} \quad \tau, \delta ::< x_1, VT_1 >:: ... ::< x_n, VT_n > \mapsto B : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{abstraction}(x_1 : VT_1, ..., x_n : VT_n); B : \ \textbf{behaviour[}VT_1, ..., VT_n\textbf{]}}$$

[function literal]

$$\frac{\tau^i, \delta^i \mapsto e : \textbf{function}[T_1, ..., T_n] \rightarrow S \quad \tau^i, \delta^i \mapsto e_1 : T_1 \ ... \ \tau^i, \delta^i \mapsto e_n : T_n}{\tau^i, \delta^i \mapsto e(e_1, \ ..., \ e_n) : S}$$

$$\frac{\tau^i, \delta^i_1 ::< x_1, T_1 >:: \ ... \ ::< x_n, T_n >:: \delta^i_2 \mapsto e : S}{\tau^i, \delta^i \mapsto \textbf{function}(x_1 \ : \ T_1, \ ..., \ x_n \ : \ T_n) \ ; \ e : \textbf{function}[\ T_1, ..., T_n\ ] \rightarrow S}$$

Block

[parenthesis]

$$\frac{\tau, \delta \mapsto e : T}{\tau, \delta \mapsto (e) : T}$$

[{}]

$$\frac{\tau, \delta \mapsto e : T}{\tau, \delta \mapsto \{e\} : T}$$

Abstraction

[abstraction derivation]

$$\frac{\text{Decl1} \quad \tau, \delta \mapsto A : \textbf{behaviour[}VT_1, ..., VT_n\textbf{]} \ \tau, \delta \mapsto x_{pi1} : VT_{i1}, ..., \tau, \delta \mapsto x_{pik} : VT_{ik}}{\tau, \delta \mapsto A(x_{pi1}\textbf{as}\ x_{i1}, ..., x_{pik}\textbf{as}x_{ik}) \ : \textbf{behaviour[}VT_{j1}, ..., VT_{jl}\textbf{]} \quad \text{Decl2}} \text{ Where}$$

Decl1 stands for $\tau \mapsto VT_1 \in \text{ValueType}, ..., \tau \mapsto VT_n \in \text{ValueType}$

Decl2 stands for

$\text{id\_declaration}(< x_{pi1} : VT_{i1} >, ..., < x_{pik} : VT_{ik} >)$ and $\{VT_{i1}, ..., VT_{ik}, VT_{j1}, ..., VT_{jl}\} = \{VT_1, ..., VT_n\}, k < n$

Behaviour

[abstraction application]

$$\frac{\text{Decl1} \quad \tau, \delta \mapsto A : \textbf{behaviour[}VT_1, ..., VT_n\textbf{]} \ \tau, \delta \mapsto x_{pi1} : VT_{i1}, ..., \tau, \delta \mapsto x_{pin} : VT_{in}}{\tau, \delta \mapsto A(x_{pi1}\textbf{as}\ x_{i1}, ..., x_{pin}\textbf{as}x_{in}) \ : \textbf{behaviour} \quad \text{Decl2}} \text{ where}$$

Decl1 stands for $\tau \mapsto VT_1 \in \text{ValueType}, ..., \tau \mapsto VT_n \in \text{ValueType}$

Decl2 stands for $\text{id\_declaration}(< x_{pi1}, VT_{i1} >, ..., < x_{pik}, VT_i >)$

Identifier

[identifier]

$$\frac{}{\tau, \delta_1 ::< x, T >:: \delta_2 \mapsto x : T}$$

[renaming]

$$\frac{\tau \mapsto T1, ..., Tn \in \text{ConnectionType} \quad \tau, \delta \mapsto e : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{rename}\ e\ \textbf{using [}\ x\ \textbf{as}\ y\ \textbf{]} : \textbf{void}\ \ \text{id\_substitution}(< x, y >)}$$

[naming]

$$\frac{\tau, \delta \mapsto e : \textbf{behaviour}}{\tau, \delta \mapsto \textbf{naming}\,[x_1, ..., x_n] :: e : \textbf{void} \quad \text{id\_declaration}(< x_1, \textbf{behaviour} >, ..., < x_n, \textbf{behaviour} >)}$$

Tuple

[tuple value]

$$\frac{\tau \mapsto T1, ..., Tn \in \text{ValueType} \quad \tau, \delta \mapsto e_1 : T_1 \; ... \; \tau, \delta \mapsto e_n : T_n}{\tau, \delta \mapsto \textbf{tuple}(e_1, ..., e_n) : \textbf{tuple}[T_1, ..., T_n]}$$

[tuple dereference]

$$\frac{\tau \mapsto T1, ..., Tn \in \text{ValueType} \quad \tau, \delta \mapsto e : \textbf{tuple}[T_1, ..., T_n]}{\tau, \delta \mapsto \textbf{use}\; x_1, ..., x_n \; \textbf{from}\; e : \textbf{void} \quad \text{id\_declaration}(< x_1, T_1 >, ..., < x_n, T_n >)}$$

View

[view value]

$$\frac{\tau \mapsto T1, ..., Tn \in \text{ValueType} \quad \tau, \delta \mapsto v_1 : T_1 \; ... \; \tau, \delta \mapsto v_n : T_n}{\tau, \delta \mapsto \; \textbf{view}\,(l_1 = v_1, ..., l_n = v_n) : \textbf{view}[l_1 : T_1, ..., l_n : T_n]}$$

[view dereference]

$$\frac{\tau \mapsto T1, ..., Tn \in \text{ValueType} \quad \tau, \delta \mapsto v : \textbf{view}[l_1 : T_1, ..., l_n : T_n]}{\tau, \delta \mapsto \; v.l_i : T_i}$$

Union

[union-i value]

$$\frac{\tau, \delta \mapsto v : T_i}{\tau_1 ::< T, \textbf{union}[T_1 ... T_n] >:: \tau_2, \delta \mapsto \textbf{union}(v) : T : \textbf{union}[T_1 ... T_n]}$$

[union projection]

$$\frac{\tau, \delta \mapsto e : \textbf{union}[T_1, ..., T_n] \quad \forall i \in \{1, ..., n+1\} (\tau, \delta_1 ::< x, T_i >:: \delta_2 \mapsto e_i : \textbf{T})}{\tau, \delta \mapsto \; \textbf{project}\; e \; \textbf{as}\; x \; \textbf{onto}\; \{T_1 : e_1; ..., Tn : en ; \textbf{default}\; e_{n+1}\} : T \quad \text{id\_declaration}(< x, T_i >)}$$

Infinite Union

[any-injection]

$$\frac{\tau \mapsto T \in \text{ValueType} \quad \tau, \delta \mapsto v : T}{\tau, \delta \mapsto \textbf{any}(v) : \textbf{Any}}$$

[any projection]

$$\frac{\tau \mapsto T1, ..., Tn \in \text{ValueType} \quad \tau, \delta \mapsto v : \textbf{Any} \quad \forall i \in \{1, ..., n+1\} (\tau, \delta_1 ::< x, T_i >:: \delta_2 \mapsto e_i : \textbf{T})}{\tau, \delta \mapsto \; \textbf{project}\; e \; \textbf{as}\; x \; \textbf{onto}\; \{T_1 : e_1; ..., Tn : en ; \textbf{default}\; e_{n+1}\} : T \quad \text{id\_declaration}(< x, T_i >)}$$

Variant

[variant value]

$$\frac{\tau \mapsto T1, ..., Tn \in \text{ValueType} \quad \tau, \delta \mapsto v : T_i}{\tau_1 ::< T, \textbf{variant}[l_1 : T_1, ..., l_i : T_i, ..., l_n : T_n] >:: \tau_2, \delta \mapsto \textbf{variant}(l_i = v) : T : \textbf{variant}[l_1 : T_1, ..., l_i : T_i, ..., l_n : T_n]}$$

[variant projection]

$$\frac{\tau, \delta \mapsto e : \textbf{variant}[l_1 : T_1, ..., l_i : T_i, ..., l_n : T_n] \quad \forall i \in \{1, ..., n+1\} (\tau, \delta_1 ::< x, T_i >:: \delta_2 \mapsto e_i : \textbf{T})}{\tau, \delta \mapsto \; \textbf{project}\; e \; \textbf{as}\; x \; \textbf{onto}\; \{l_1 : e_1; ..., ln : en ; \textbf{default}\; e_{n+1}\} : T \quad \text{id\_declaration}(< x, T_i >)}$$

Location

[location value]

$$\frac{\tau \mapsto T \in \text{ValueType} \quad \tau, \delta \mapsto v : T}{\tau, \delta \mapsto \textbf{location}\,(v) : \textbf{location[}T\textbf{]}}$$

[location dereference]

$$\frac{\tau \mapsto T \in \text{ValueType} \quad \tau, \delta \mapsto l : \textbf{location[}T\textbf{]}}{\tau, \delta \mapsto\ 'l\ \textbf{:}\ T}$$

[assignment]

$$\frac{\tau \mapsto T \in \text{ValueType} \quad \tau, \delta \mapsto v : T \quad \tau, \delta \mapsto n : \textbf{location[}T\textbf{]}}{\tau, \delta \mapsto v\ \textbf{:=}\ n : \textbf{void}}$$

Sequence

[sequence value]

$$\frac{\tau, \delta \mapsto e_1 : T \ldots \tau, \delta \mapsto e_n : T}{\tau, \delta \mapsto \textbf{sequence}(\, e_1, \ldots, e_n) : \textbf{sequence}[\, T\,]}$$

[sequence index]

$$\frac{\tau, \delta \mapsto e : \textbf{sequence}[\, T\,] \quad \tau, \delta \mapsto e_1 : \textbf{Integer}}{\tau, \delta \mapsto e[\, e_1\,] : T}$$

Set

[set value]

$$\frac{\tau, \delta \mapsto e_1 : T \ldots \tau, \delta \mapsto e_n : T}{\tau, \delta \mapsto \textbf{set}(\, e_1, \ldots, e_n) : \textbf{set}[\, T\,]}$$

[set selection - 1]

$$\frac{\tau, \delta \mapsto e : \textbf{set}[\, T\,]}{\tau, \delta \mapsto \textbf{one of}\ e : T}$$

[set selection - 2]

$$\frac{\tau, \delta \mapsto e : \textbf{set}[\, T\,]}{\tau, \delta \mapsto \textbf{rest of}\ e :\ \textbf{set}[\, T\,]}$$

Bag

[bag value]

$$\frac{\tau, \delta \mapsto e_1 : T \ldots \tau, \delta \mapsto e_n : T}{\tau, \delta \mapsto \textbf{bag}(\, e_1, \ldots, e_n) : \textbf{bag}[\, T\,]}$$

[bag selection - 1]

$$\frac{\tau, \delta \mapsto e : \textbf{bag}[\, T\,]}{\tau, \delta \mapsto \textbf{one of}\ e : T}$$

[bag selection - 2]

$$\frac{\tau, \delta \mapsto e : \textbf{bag}[\, T\,]}{\tau, \delta \mapsto \textbf{rest of}\ e :\ \textbf{bag}[\, T\,]}$$

5. Styles de fondations

5.1. Pipe-Filter

5.1.1. Concepts

Le style Pipe_Filter est un support aux systèmes dont les constituants sont organisés comme des traitements asynchrones reliés par des flux de données [GarAll&Ock 94].
Tous les connecteurs dans le modèle de Pipe&Filter sont des pipes[52] qui transportent, de manière asynchrone, des flux de données typées d'une entrée, appelée *source,* vers une sortie, appelée *sink*, maintenant l'ordre des éléments d'informations.
Tous les composants dans le modèle de Pipe&Filter sont des filtres, *Filters*. Ils reçoivent un flux de données en entrée, effectuent un traitement, puis envoient les résultats. Un filtre a deux sortes de ports, les uns pour recevoir des données, les autres pour envoyer des données. Ils sont respectivement appelés *input* et *output.* Un filtre n'a aucune connaissance des filtres en amont et en aval. Le diagramme informel suivant présente une architecture Pipe&Filter.

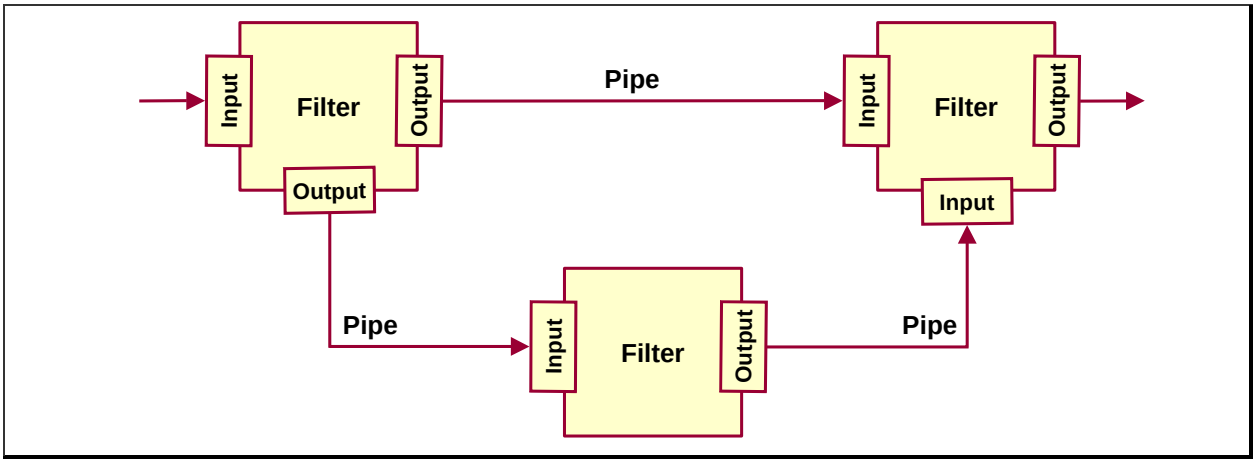


Figure 67 - Architecture Pipe-Filter

5.1.2. Description

Style Pipe-Filter

La description que nous présentons dans ce paragraphe est celle du style *Pipe_Filter*. Le support structurel fournit part ce style est :

- des ports : InputPort (port d'entrée) et OutputPort (port de sortie),
- des composants : les filtres,
- des connecteurs : les pipes,
- un patron pour les structures linéaires : *linear*.

```
Pipe_Filter is style extending Component where {
     styles {
          InputPort is style extending Port …
          OutputPort is style extending Port …
          Filter is style extending Component …
          Pipe is style extending Connector …
          linear is style … }
```

Les contraintes du style *Pipe_Filter* sont les suivantes :

- les élements sont soit dans le style *Pipe* soit dans le style *Filter*,
- deux éléments dans le style *Pipe* ne peuvent pas être attachés (c'est implicite pour les filtres qui sont des composants),
- un port ne ne peut être attaché qu'à un seul autre port,
- un port dans le style InputPort ne peut être attaché qu'à un port de style OutputPort et inversement.

[52] Nous utilisons le vocabulaire anglais, pipe, qui signifie tube en français.

```
constraints    {
     only_Pipes_or_Filters is constraint {
          to styleInstance.connectors apply
               forall(c |      c in style Pipe )  and
          to styleInstance.components apply
               forall(c |      c in style Filter )
     },
     pipes_cannot_be_connected is constraint {
          to styleInstance.connectors apply
               forall(p1,p2 | p1 in style Pipe and p2 in style Pipe
          implies not attached(p1,p2)),
     },
     one_to_one_port_connection is constraint {
          to styleInstance.connectors apply
               forall(f |
                    to f.ports apply
                         exists(fp | to styleInstance.connectors apply
                              exists([0..1]p | to p.ports apply{
                                   exists(pp | pp attached to fp)}
               }
          }
     }
}
```

Le style *Pipe_Filter* fournit des analyses :

- *source_of* vérifie si un filtre est la source d'un pipe,
- *sink_of* vérifie si un filtre est en aval d'un pipe,
- *reachable* vérifie si un filtre est atteignable par un autre via un chemin du flux de communication,
- *hasCycle* vérifie si la structure présente des cycles.

```
analysis {
     source_of is analysis {
     -- checks if a filter is a source for a pipe
          kind AAL
          input { filter:Component, pipe:Component }
          output {  Boolean   }
          body{
               to pipe.ports apply
                    exists(pp | pp in style Source and
                         to f.ports apply
                              exists(fp | fp attached to pp)
                    )
          }
     },
     sink_of is analysis {
     -- checks if a filter is a sink for a pipe
          kind AAL
          input { filter:Component, pipe:Component }
          output {  Boolean   }
          body{
               to p.ports apply
                    exists(pp | pp in style Sink and
                         to f.ports apply
                         exists(fp | fp attached to pp)
               )
          }
```

```
      }
      reachable is analysis {
      -- checks is a filter is reachable from another
           kind AAL
           input { filter1:Component, filter2:Component }
           output {  Boolean   }
           body{
                to styleInstance.connectors apply
                     exists(p| source_of(f1,p) and sink_of(f2,p))
                     or
                to styleInstance.components apply
                     exists(f | reachable(f1,f)and
                     to styleInstance.connectors apply
                          exists(p| source_of(f,p) and sink_of(f2,p))
                     )
           }
      },
      hasCycle is is analysis {
      -- hasCycle checks for cycles in the system graph
           kind AAL
           output {  Boolean   }
           body{
                to styleInstance.components apply
                     exists(f | reachable(f,f))
           }
      }
}
}
```

Ports

La partie suivant montre la description des styles de ports.

Le style *InputPort* représente des ports ne possédant que des connections pour recevoir des données.

```
 InputPort is style extending Port where {
      constructors {
           InputPort is constructor(connections, protocol, configuration);
           {Port(connections, protocol, configuration)}
           as { input port with
                     connections { $connections }
                     protocol {$protocol }
                     configuration { $configuration }
           }
      }
      constraints {
           to styleInstance.connections apply{
                forall(c |
                     every sequence{true*.via c send any}
                     leads to state{false})
           }
      }
 }
```

Le style *OutputPort* représente des ports ne possédant que des connections pour envoyer des données.

```
OutputPort is style extending Port where {
     constructors {
          OutputPort is constructor(connections, protocol, configuration);
          {Port(connections, protocol, configuration)}
          as { output port with
               connections { $connections }
               protocol {$protocol }
               configuration { $configuration }
          }
     }
     constraints {
          to styleInstance.connections apply{
               forall(c |
               every sequence{true*.via c receive any}
                    leads to state{false})
          }
     }
}
```

Eléments

Nous présentons ici les styles correspondant aux éléments filtre et pipe.

Un *Filter* est un composant vérifiant la contrainte suivante :

- Il a seulement des ports suivant les styles *InputPort* et *OutputPort*.

```
Filter is style extending Component where {
     constraints {
          only_Input_Output_port is constraint {
               to styleInstance.ports {
                    forall(p | p in style InputPort xor
                           p in style OutputPort)
               }
          }
     }
}
```

Un *Pipe* est un connecteur vérifiant la contrainte suivante :

- il a un port dans le style *Source* (*InputPort*) et un dans le style *Sink* port (*OutputPort*), et c'est tout.

```
Pipe is style extending Connector where {
     styles {
          Source is style extending InputPort where {…},
          Sink is style extending OutputPort where {…}
     }
     constraints {
          one_Source_one_Sink is constraint {
               to styleInstance.ports apply {
                    forall(p | p in style Source or
                           p in style Sink),
                    exists([1]p | p in style Source),
                    exists([1]p | p in style Sink)}
     }
}
```

5.2. Data Indirection

Le style Data Indirection tel qu'il est nommé dans [Kle&Kaz 99] est aussi appelé Tableau Noir (Blackboard en anglais).

5.2.1. Concepts

Le style Data Indirection est caractérisé par le fait que des composants producteurs et des composants consommateurs de données partagées n'ont pas connaissance les uns des autres. Ceci est accompli en interposant un composant intermédiaire entre ces derniers. Ce principe permet d'améliorer la modifiablité du système en réduisant les couplages de données et de contrôle entre les composants distincts. Les couplages sont réduits par la présence d'un dépôt de données partagées.
Ce style est efficace si on anticipe les changements des producteurs et des consommateurs de données ainsi que l'ajout de ces éléments, et que ces changements sont assez fréquents et dominants pour avoir un fort impact sur les coûts de maintenance.

Le diagramme suivant illustre le style Data Indirection.

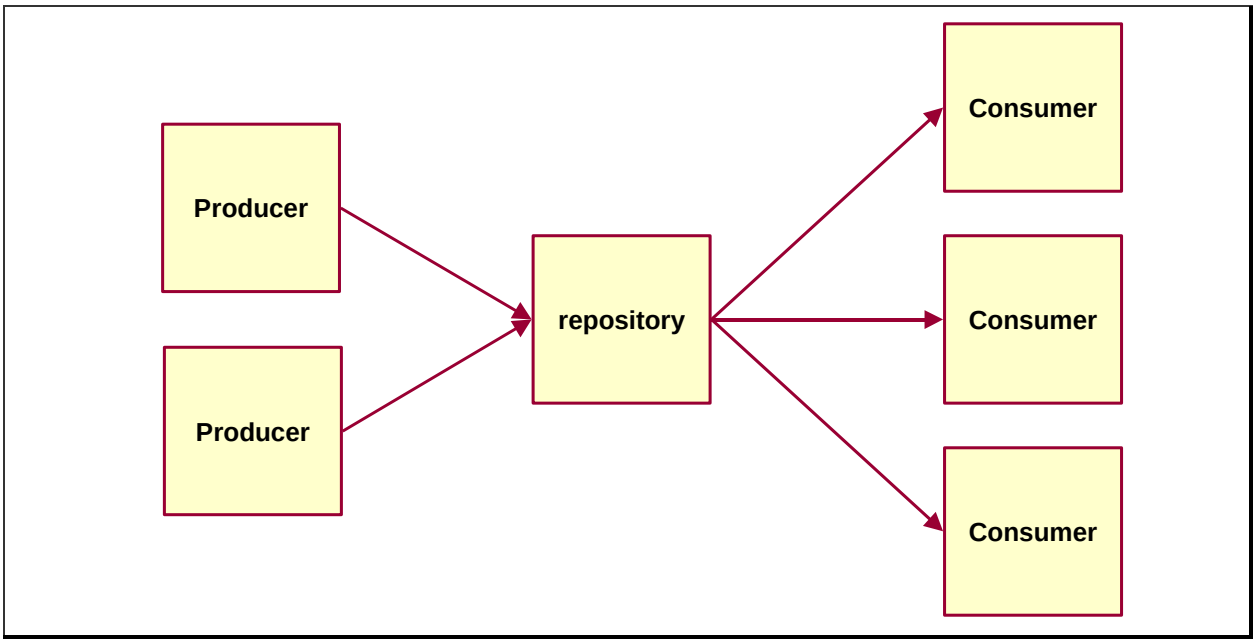


Figure 68 - Data Indirection Architecture

5.2.2. Description

Style Data Indirection

La description est la définition du style *Data_Indirection*. Le support structurel fournit part ce style en termes déléments architecturaux est :

- un composant générique : *Data_Indirection_Component*,
- des composants : les producteurs (*Producer*) et les consommateur (*Consumer*), le depôt (Repository).

```
Data_Indirection is style extending Component where {
     styles {
          Data_Indirection_Component is style extending Component …
          Producer is style extending Data_Indirection_Component …
          Consumer is style extending Data_Indirection_Component …
          Repository is style extending Data_Indirection_Component …}
```

Les contraintes du style *Data_Indirection* sont les suivantes :

- les composants sont soit *Producer*, *Consumer* ou *Repository*,
- il n'y a qu'un *Repository*,
- les producteurs et les consommateurs ne communiquent qu'à travers le *Repository*.

```
constraints    {
     only_Consumers_Producers_Repository is constraint {
          to styleInstance.components apply
               forall(c |      c in style Producer or
                               c in style Consumer or
                               c in style Repository )
     },
     only_one_Repository is constraint {
          to styleInstance.components apply
               exists([1]r | r in style Repository)
     },
     consumers_and_producers_connected_through_repository is constraint {
          to styleInstance.components apply
               forall(e, r | (e in style Producer or
                         e in style Consumer) and
                         not(r in style Repository)   implies
                         not( e connected to r ))
     }
```

Le style *Data_Indirection* défini un patron par un constructeur :

- le constructeur *star* permet de décrire des structures en étoile.

```
constructors {
     star is construtor(  consumers : set[Component],
                          producers : set[Component],
                          repository : Component);{
     iterate consumers by c
          do attach c~read to repository~receiver;
     iterate producers by p
          do attach p~write to repository~sender
}
```

Les analyses du style Data_Indirection sont les suivantes:

- *rippling* permet de calculer le nombre d'éléments qui peuvent être concernés par la modification d'un élément particulier,
- *modifiability_quantification_elt* permet de quantifier la modifiability d'un élément,
- *modifiability_quantification_arch* permet de quantifier la modifiabilité de l'ensemble de l'architecture.

```
analysis {
     rippling is analysis {
     -| permits to compute the number of element that can be
     concerned by the change a particular element |-
          kind Data_Indirection
          input  { element in style component }
          output { number :Integer }
          body { rippling }
     },
     modifiability_quantification_elt is analysis {
     -|permit to quantify the modifiability on an element|-
          kind Data_Indirection
          input {element in style component}
          output { quantification:Real}
```

```
            body{ modifiability_quantification_elt }
      },
      modifiability_quantification_arch is analysis {
      -|permit to quantify the modifiability on the whole architecture|-
            kind Data_Indirection
            output { quantification:Real}
            body{ modifiability_quantification_elt }
      }
}
```

Elements

Data_Indirection_Component est un style générique pour l'ensemble des styles de composants du style *Data_Indirection*.

- Ce style contraints les composants à posséder l'attribut *change_frequency*. Cet attribut représente la fréquence de changement d'un composant spécifique. Il est utilisé par les analyses pour évaluer la modifiabilité du système.

```
Data_Indirection_Component is style extending Component where {
      styles {
            WritePort is style extending Port where {…},
            ReadPort is style extending Port where {…}
      }
      constraints
            have_changing_frequency is constraint{
                  to styleInstance.attributes apply
                        exists (a|a.name="change_frequency" and a.type="Real")
            }
      }
}
```

Le style *Producer* hérite de *Data_Indirection_Component* et définit la contrainte suivante :

- Il a un port dans le style *WritePort* nommé *write*.

```
Producer is style extending Data_Indirection_Component where {
      constraints
            have_write_port is constraint{
                  to styleInstance.ports apply
                        exists (p|p.name="write" and p in style WritePort)
            }
      }
}
```

Le style *Consumer* hérite de *Data_Indirection_Component* et définit la contrainte suivante :

- Il a un port dans le style *ReadPort* nommé *read*.

```
Producer is style extending Data_Indirection_Component where {
      constraints
            have_read_port is constraint{
                  to styleInstance.ports apply
                        exists (p|p.name="read" and p in style ReadPort)
            }
      }
}
```

Le style *Repository* hérite de *Data_Indirection_Component* et définit les contraintes suivantes :

- avoir un port dans le style *ReadPort* nommé *receiver_port*,
- avoir un port dans le style *WritePort* nommé *sender_port*,

- être passif – ne pas initié de réception ou d'envoi.

```
Repository is style extending Data_Indirection_Element where {
     have_ReadPort is constraint {
          to styleInstance.ports apply
               forall(p |      p in style ReadPort )
     },
     have_WritePort is constraint {
          to styleInstance.ports apply
               forall(p |      p in style WritePort )
     },
     passivity is constraint {
               every sequence {via receiver_port~channel send any}
                    leads to state{false} and
               every sequence {via sender_port~channel receive any}
                    leads to state{false});
     }
}
```

5.3. Style en couche

5.3.1. Concepts

Les architectures en couches représentent des niveaux d'abstraction cumulés autour d'une base fonctionnelle. Les couches cache les détails de l'implémentation et promeuvent la modifiabilité à plusieurs niveaux d'abstraction.

Une architecture en couche, suivant le style en couche (layered style), consiste en ensembles de composants implémentant des catégories distinguables de fonctionnalités. Chacun de ces ensembles sont organisés en couches. Les éléments de chaque couche peuvent seulement intéragir avec les éléments de leur couche et des couches adjacentes.

La figure suivante illustre le style en couche.

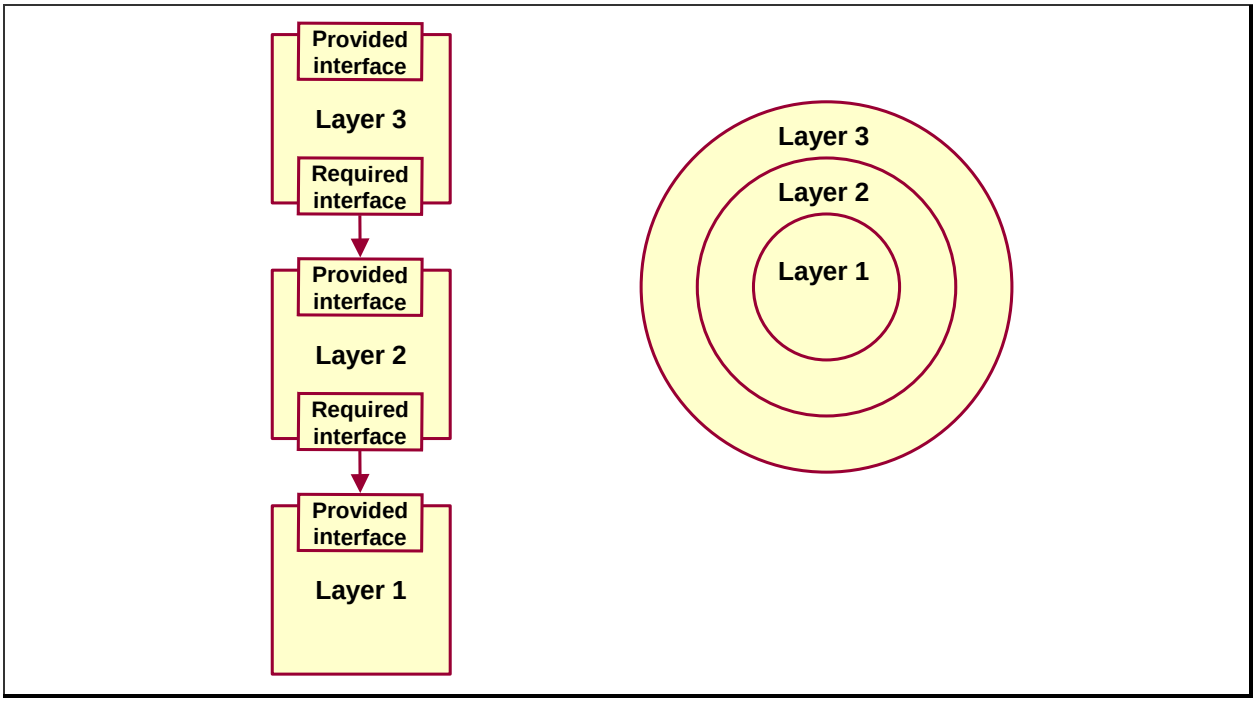


Figure 69 - Layered architecture

5.3.2. Description

Le style *Layered* est un sous-style du style *Client_Server*. Il définit les contraintes supplémentaires suivantes :

- il doit définir un élément *CoreLayer* qui ne possède qu'un port *Provided_Port*,
- Chaque couche, sauf la *CoreLayer*, possède un port *Required_Port* et un port *Provided_Port*,
- Chaque couche, sauf la *CoreLayer*, est connectée avec une sous-couche.

```
Layered is style extending Client_Server where {
     styles {
          CoreLayer is style extending Server where {
               constraints {
                    Have_One_Provided_Port is constraint {
                         to styleInstance.ports apply {
                              forall(p | p in style Provided_Port ),
                              exists([1]p | p in style Provided_Port )
                         }
                    }
               }
          }
     }
```

```
     constraints {
          have_One_CoreLayer is constraint {
               to styleInstance.components apply
                    exists([1]c | c in style CoreLayer )

          },
          layer_have_One_ProvidePort_and_One_RequiredPort is constraint {
               -|every layers but the core layer have one required port and one
provided port |-
               to styleInstance.components apply
               forall(l | not l in style CoreLayer implies
                    l in style Client and l in style Server and
                    to ports apply {
                         forall(p |  p in style Provided_Port or
                              p in style Required_Port  ),
                         exists([1]p | p in style Provided_Port ),
                         exists([1]p | p in style Required_Port )
               )    }
          },
          have_one_sub_layer is constraint {
          -|every layers but the core layer are connected to one sub-layer|-
               to styleInstance.components apply {
                    forall(c | not c in style CoreLayer implies
                    exists([1]l2 | l2!=c and      c attached to l2))
               }
          }

     }
  }
```

ASL : un langage et des outils pour les styles architecturaux.

Contribution à la description d'architectures dynamiques.

Résumé

Dans la dernière décennie, l'architecture logicielle a émergé comme une notion centrale dans le développement logiciel des systèmes complexes. Des modèles architecturaux ont été codifiés et réutilisés de manière informelle à travers des styles architecturaux. Ils capturent l'expérience et les connaissances acquises dans un domaine d'application donné. Des travaux de formalisation ont été entrepris afin de pouvoir donner des définitions précises des styles architecturaux pour transmettre ces concepts sans ambiguïté, et afin de pouvoir étudier et garantir les propriétés architecturales d'une architecture ou d'une famille d'architectures.

On peut identifier deux catégories de systèmes logiciels : ceux dont l'architecture ne change jamais en cours d'exécution (architectures "statiques") et ceux dont l'architecture change afin de répondre à des besoins précis (architectures "dynamiques"). Des langages sont nécessaires pour permettre la description de ces deux catégories d'architectures. Si plusieurs travaux ont traité le problème de la définition de styles architecturaux de systèmes statiques, le problème de la définition de styles architecturaux de systèmes dynamiques reste un problème ouvert. Notre thèse a eu pour objectif de combler ce manque de formalisation pour la définition et l'utilisation des styles architecturaux des systèmes dynamiques.

Nous définissons un langage, ASL, pour la description de styles architecturaux décrivant des familles de systèmes mais également des architectures spécifiques. Alors que d'autres langages de description d'architectures intègrent une vision composant-connecteur figée, le langage ASL est lui basé sur le concept plus générique d'abstraction. La vision composant-connecteur, elle, est introduite comme une spécialisation d'ASL sous la forme d'un style appelé Composant-Connecteur.

Nous avons développé des outils pour la gestion des styles architecturaux. Le langage ASL et ces outils ont été développés et validés dans le cadre du projet européen de recherche ArchWare (IST-2001-32360).

Mots-clés : Architecture logicielle, style architectural, langage de description d'architectures, architecture dynamique, composant-connecteur