Formalisation de Familles d'Architectures Logicielles Coopératives : Démarches, Modèles et Outils.

Mhamed Saidane

Université Joseph Fourier – Grenoble I

Thèse

pour obtenir le grade de

Docteur de l'Université Joseph Fourier

Spécialité : Systèmes d'Information

préparée au Laboratoire Logiciels, Systèmes et Réseaux

dans le cadre de l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique

Présentée et soutenue publiquement par

Mhamed Saidane

le 01 décembre 2005

Formalisation de Familles d'Architectures Logicielles Coopératives : Démarches, Modèles et Outils

Jury :

Pr. Farid Ouabdesselam	Président
Pr. Corine Cauvet	Rapporteur
Pr. Mourad Chabane-Oussalah	Rapporteur
Pr. Danielle Boulanger	Examinateur
Pr. Jean-Pierre Giraudin	Directeur de thèse
Pr. Dominique Rieu	Co-Directeur de thèse

La fin d’une longue aventure n’est que le début d’une autre

… encore plus longue.

Remerciements

La rédaction de cette thèse venant boucler cette belle histoire que constitue ces années de recherche, je tiens donc à utiliser le fait que cette page soit la première à être lue pour exprimer mes sincères remerciements à tous ceux qui ont joué un rôle dans la concrétisation de ce travail.

Je ne peux commencer sans évoquer mes directeurs de thèse Jean-Pierre Giraudin et Dominique Rieu qui malgré leur temps si précieux, ont rendu mes quatre années de Doctorat agréables grâce à leur immense amabilité et leur persistante bonne humeur. Merci Dominique de m'avoir supporté et soutenu depuis le DEA : ces cinq ans m'ont merveilleusement marqués à jamais !...et pardon pour toutes les lectures et toutes les bêtises que tu as eues à corriger pendant ton infinitésimal temps libre !

Je souhaite aussi remercier le Professeur Farid Ouabdesselam, pour m'avoir si chaleureusement accueilli dans son laboratoire et offert tous les moyens, matériels et moraux, pour mener à bien ce travail de longue haleine et en présider le jury !

Je tiens également à exprimer mes sincères remerciements à Mme Corine Cauvet, Professeur à l'Université d'Aix-Marseille, et Mr Mourad Chabane-Oussalah, Professeur à l'Université de Nantes pour l'intérêt qu'ils ont porté à ces travaux en acceptant de les rapporter, malgré la charge de travail dont ils ont la responsabilité.

Un grand merci et une profonde reconnaissance à Danielle Boulanger, Professeur à l'Université de Lyon 3 pour avoir accepté, malgré son emploi du temps très chargé, de faire partie du jury.

Je remercie vivement Pierre Berlioux pour sa collaboration, son amabilité et son dévouement pour la relecture de cette thèse et pour les conseils qu'il m'a prodigués. Je n'oublie pas David Garlan et Bradley Schmerl de l'Université Carnegie Mellon pour leurs réponses très rapides à tous les échanges « électroniques » et leur précieuse aide concernant l'ADL Acme.

Je n'oublierai pas et serai toujours reconnaissant pour la bonne humeur, les aides et interventions diverses, professionnelles et personnelles, reçues : merci Dominique Decouchant ! Merci à tout le personnel administratif : Liliane Di Giacomo, Pascale Poulet, Martine Pernice, Christiane Plumeré et Gilles Thieblemont !

Walid, mon colocataire de bureau que j'ai dû supporter pendant ces longues années...devenues éternelles en sa présence ; Ramzi, Mehdi, Ouafa, Emmanuel, Nicolas : merci pour tous les moments passés ensemble, et bonne chance pour la suite !

J'exprime aussi ma profonde gratitude à mes parents qui se sont tant sacrifiés pour me voir un jour leur offrir cette petite récompense ! Merci Chekib pour ton soutien remarquable, ta générosité et ton dévouement envers moi !

Je ne peux terminer sans remercier mes beaux-parents qui m'ont soutenu par leur patience et m'ont toujours encouragé.

Enfin, mon petit rayon de soleil Leyla, qui, malgré la distance, m'a porté de toutes ses forces et a beaucoup donné pour que je puisse terminer cette petite aventure…et enfin la rejoindre !

Table des matières

Table des illustrations

Table des tableaux

CHAPITRE I

INTRODUCTION GENERALE

1. Problématique

Le domaine de l'informatique et plus particulièrement celui des systèmes d'information a connu une profonde mutation depuis une dizaine d'années. En particulier, les besoins d'ouverture et de communication entre les Systèmes d'Information (SI) sont devenus importants, voire vitaux. Les nouvelles applications telles que les systèmes d'aide à la décision, les systèmes d'aide au commandement, les applications de simulation et le commerce électronique nécessitent une communication entre systèmes d'information. Les applications ont besoin d'accès coordonnés à différentes sources de données afin de synthétiser les informations provenant de ces sources.

Le partage des informations présente des intérêts économiques non négligeables puisqu'il permet de réduire les coûts d'acquisition et de mise à jour des bases de données. Ce partage nécessite également une communication inter-systèmes. Par conséquent, les SI actuels doivent désormais être coopératifs mais leur répartition et leurs hétérogénéités constituent des obstacles majeurs à leur coopération. L'interopérabilité fournit des solutions à un niveau technique pour répondre aux besoins de coopération des SI. Tout d'abord, elle traduit la capacité d'un SI à fonctionner avec d'autres systèmes de natures différentes. De plus, elle a pour objectif second de permettre à un utilisateur d'utiliser, de façon transparente, des données issues d'un ensemble de systèmes d'information autonomes, répartis et hétérogènes. De ce fait, elle vise à développer des architectures et des outils pour le partage, l'échange et le contrôle des données. L'interopérabilité technique et la consultation des bases de données réparties hétérogènes sont des solutions intéressantes mais insuffisantes si elles ne sont pas incluses dans une vision de SI coopérants.

Cette nouvelle génération de SI est appelée *Systèmes d'Information Coopératifs* (SICo). Aussi, dès lors que nous parlons de SICo, un SI d'une organisation ne repose plus sur des choix conceptuels définitifs et exhaustifs. Sa conception devient alors un processus complexe visant à coordonner des sources d'information et des applications préexistantes, hétérogènes et évolutives. Cette diversité des sources d'information ou d'application rend nécessaire la définition de méthodes, de modèles et d'outils aptes à prendre en compte l'hétérogénéité qui en découle et la nécessaire interprétation globale. Cependant, seuls les niveaux "plates–

formes" et "réseaux" semblent être résolus par l'emploi de techniques de communication modernes qui facilitent l'échange et le traitement des données et des services distribués (TCP/IP, http, Internet, Corba, Dcom, Java, etc.). D'autres travaux de recherche ont abouti à l'émergence de différentes techniques supportant l'intégration de SI hétérogènes et autonomes (les bases de données fédérées, les agents informationnels, etc.). Néanmoins, de gros efforts restent aujourd'hui nécessaires afin d'appréhender conceptuellement les besoins des concepteurs de SICo et de les guider vers des solutions techniques adéquates.

Ainsi, les langages de modélisation des SI semblent ne pas suivre cette évolution. Les techniques de modélisation proposées par les langages actuels ne donnent pas de réponses directes aux différents problèmes posés dans la modélisation des SICo. L'un des problèmes majeur dans la modélisation de tels systèmes est de pouvoir spécifier de manière riche et explicite des architectures de coopération afin de favoriser leur réutilisation. En effet, les méthodes de développement actuelles sont essentiellement destinées au développement de Systèmes d'Information "from scratch". Bien qu'elles commencent à intégrer la notion de composants réutilisables, aucune de ces méthodes ne propose des solutions directes pour la réutilisation des architectures de coopération existantes.

Pour mieux contrôler la complexité des SICo, il est nécessaire d'avoir un niveau d'abstraction élevé et de disposer de modèles qui s'approchent du modèle de raisonnement du développeur. Une solution possible consiste à opter pour la définition d'une architecture du système. Les architectures logicielles ont pour origine, à la fois les difficultés rencontrées par les concepteurs de gros systèmes et également les besoins de réutilisation de logiciels. La plupart du temps, l'architecture logicielle d'un système est spécifiée de manière informelle et intuitive par un diagramme de type box-and-line sans sémantique associée. Ce manque de sémantique est fréquent lorsque la structure du système à concevoir est simple mais aussi induit par la "localité" des spécifications orientées objets. Cependant, avec l'apparition de SICo, la définition d'une architecture logicielle devient incontournable. Une architecture logicielle décrit l'ensemble des composants du système ainsi que leurs interactions. Elle permet la conception d'applications en se détachant des détails techniques propres à l'environnement et en respectant les conditions fixées par les futurs utilisateurs.

La définition des architectures logicielles dans le cadre de l'ingénierie des SICo paraît ainsi l'une des pistes les plus prometteuses pour :

- proposer au concepteur des techniques et des langages de modélisation permettant d'exprimer les propriétés des systèmes composants et de leurs interactions,
- capitaliser et réutiliser des modèles d'architectures logicielles adaptés aux SICo,
- promouvoir des guides de conception propres aux SICo.

La notion de *modèle* est la composante principale dans une méthode d'ingénierie. La deuxième composante primordiale est la notion de *démarche*. Cette dernière est essentielle pour aider le concepteur dans l'ingénierie de nouveaux Systèmes Coopératifs.

Ces deux aspects, modèles et démarches, constituent les intentions principales de cette thèse et sont présentées ci-après.

1.1. Définition des modèles

Une des composantes principales d'une méthode d'ingénierie de SI est la notion de *modèles*. Un modèle est un ensemble de concepts et de règles pour les utiliser, destinés soit à expliquer et construire la représentation des phénomènes conceptuels, organisationels ou techniques, soit à expliquer et représenter les éléments qui composent le SI et leurs relations. [ROLLAND & AL., 88]

La notion de modèle ne peut être séparée de la notion de *langage*, lui aussi une composante essentielle d'une méthode. Un langage est défini par un ensemble de constructions qui permettent de décrire les spécifications du SI élaborées aux différents stades du processus de conception en s'appuyant éventuellement sur les modèles de la méthode. [ROLLAND & AL., 88]

Dans la littérature, deux approches de modélisation se sont intéressées à la modélisation des architectures logicielles : les méthodes formelles et les méthodes semi-formelles.

Les modèles et langages formels visent à améliorer le coût et la qualité du logiciel. La rigueur du formalisme, la sémantique et le pouvoir d'abstraction en sont les principales caractéristiques. Ils permettent de définir un vocabulaire précis et commun pour les différents acteurs et d'identifier les incohérences de ce qui a été spécifié. Les ADL (Architectural Description Language) spécifient les composants de l'architecture de manière abstraite sans entrer dans les détails d'implantation. De plus, ils définissent de manière explicite les interactions entre composants d'un système. Ils fournissent enfin un support de modélisation pour aider les concepteurs à structurer les différents éléments. Ils offrent des facilités de réutilisation des composants et des moyens de description de la composition par description des dépendances entre composants et des règles de communication à respecter. Leur utilisation permet ainsi une meilleure compréhension du domaine d'application mais montre néanmoins des limites. Les spécifications formelles sont en effet connues pour leurs difficultés d'écriture mais aussi d'utilisation, dues notamment au manque de formation et de guides méthodologiques.

Les méthodes semi-formelles actuelles s'appuient sur des langages de modélisation orientés objet. Ces derniers peuvent être utilisés pour décrire des architectures logicielles. Les avantages des techniques à objet sont maintenant reconnus du fait qu'elles offrent des mécanismes de structuration très riches. Avec l'utilisation des modèles semi-formels, une application est structurée en composants indépendants qui favorisent la réutilisation. Cependant, leur utilisation montre des limites. Les carences des modèles semi-formels à objets proviennent essentiellement de l'absence d'une sémantique précise des notations utilisées. La modélisation des systèmes complexes peut parfois conduire à des modèles peu compréhensibles et provoquer par conséquent des difficultés d'interprétation.

Le mariage des modèles formels et semi-formels présente un centre d'intérêt pour la recherche afin de proposer des langages et des techniques pour la modélisation des architectures logicielles. Cet intérêt est notamment justifié par les aspects complémentaires et les apports croisés de ces deux techniques :

- D'une part, les techniques à objets de modélisation semi-formelle ont besoin des langages formels pour définir un cadre sémantique pour les modèles à objets. Ainsi, les composants et connecteurs définis dans un tel cadre peuvent être réutilisés de manière sûre et rigoureuse.

- D'autre part, les langages formels tirent profit des techniques à objet pour leur représentation intuitive et synthétique. La modélisation à objet favorise la compréhension du modèle et la communication entre les différents acteurs du développement.

Cependant, concevoir un modèle d'architecture logicielle réutilisable reste un problème difficile. La solution que nous proposons est inspirée des méthodes à objets. Elle consiste à utiliser des schémas de conception connus sous le noms de *patrons*. Les patrons mémorisent les savoir-faire des experts du domaine en matière de modélisation orientée objets. Ils permettent la réutilisation de la connaissance acquise par des développeurs et facilitent la communication entre les différents acteurs du développement. De nombreux patrons ont été décrits dans la littérature, ont fait leur preuve et sont largement utilisés de nos jours.

1.2. Définition d'une démarche

Une démarche est le processus grâce auquel s'effectue le travail de modélisation. Ce processus est décomposable en processus élémentaires et chaque processus possède des modèles / produits en entrée et des modèles / produits au résultat. Il s'agit ici de mettre en place une démarche de développement par réutilisation de patrons.

L'utilisation des techniques à base de patrons pour décrire des processus ou des fragments de processus nous semble une technique de formalisation de démarches de développement particulièrement prometteuse. Elle permet une formalisation des processus en fonction des buts à atteindre (exprimés dans l'intention des patrons) et nécessite l'expression des liens inter-patrons exprimant par exemple un ordre d'application entre eux [RIEU, 99]. Dans le cadre de cette thèse, nous optons pour les patrons processus pour décrire notre démarche. Les patrons processus peuvent être utilisés quel que soit le domaine d'ingénierie. Ils permettent en particulier de guider une activité d'ingénierie en organisant hiérarchiquement et fonctionnellement les problèmes et leurs solutions [CONTE, 97].

Patrons produits et patrons processus constitueront alors un catalogue de patrons pour aider le concepteur dans la conception des SICo. Pour assurer une utilisation efficace d'un catalogue de patrons lors de la spécification de systèmes, il est nécessaire de disposer d'opérateurs de sélection de patrons en fonction des problèmes à résoudre. Il s'agit ici de mettre en place des critères de classification des différents SICo pour aider le concepteur et orienter son choix vers le patron le plus adéquat.

2. Synthèse

Le travail présenté dans ce manuscrit, basé sur les précédentes hypothèses, a pour objectif de mettre en place un guide méthodologique pour l'ingénierie des SICo en abordant quatre aspects complémentaires :

❶ Proposer des patrons produit pour promouvoir la réutilisation des architectures de coopération.

❷ Exprimer graphiquement et textuellement la solution modèle en utilisant conjointement un langage semi-formel et un langage formel.

❸ Classifier les SICo en se basant sur les différents modes de coopération entre les systèmes d'information.

❹ Proposer des patrons processus pour la formalisation de notre démarche.

Ce travail a été initialisé dans le cadre du projet **OSCAR** (**O**rganisation des **S**imulations en **C**onception par la **C**apitalisation et la **R**éutilisation) soutenu par le pôle productique de la région Rhône-Alpes et dont l'objectif est de faciliter l'utilisation des modèles et des démarches de simulation numérique dans le processus de développement des produits. Le projet initial posé peut être résumé comme étant l'apport des techniques de modélisation à l'interfaçage de deux systèmes d'information industriels : "Le Système d'Information de Calcul" (SIC) qui capitalise l'ensemble des modèles utilisés ou produits par la simulation numérique, et le "Système d'Information Produit" (SIP) qui a pour objectif de gérer l'ensemble des informations techniques du produit.

Ce projet a été élargi pour la prise en compte de l'apport complémentaire des approches de type Systèmes d'Information Coopératifs (SICo).

3. Plan du document

Le corps de cette thèse est organisé comme suit :

Ce **premier chapitre** introduit à la problématique du sujet afin de comprendre les problèmes existants et le but visé par cette thèse.

Dans le **deuxième chapitre**, nous recensons les différentes architectures de coopération existantes. Nous abordons plus précisément les techniques de patrons et des styles architecturaux. Ces derniers présentent, en effet, des solutions à la réutilisation des architectures logicielles. Les différentes architectures des Systèmes d'Information Coopératifs (SICo) sont, elles aussi, basées sur certains styles d'architectures ou sur une combinaison de styles d'architectures. La deuxième partie de ce chapitre sera consacrée à la présentation de ces différentes techniques de coopération.

Le **troisième chapitre** constitue une investigation des différents langages de modélisation et nous permettra de déceler le ou les langage(s) le(s) plus adapté(s) pour décrire des architectures de coopération cohérentes et réutilisables. Nous abordons deux approches de modélisation : la première se base sur les langages formels, et la deuxième sur les langages semi-formels.

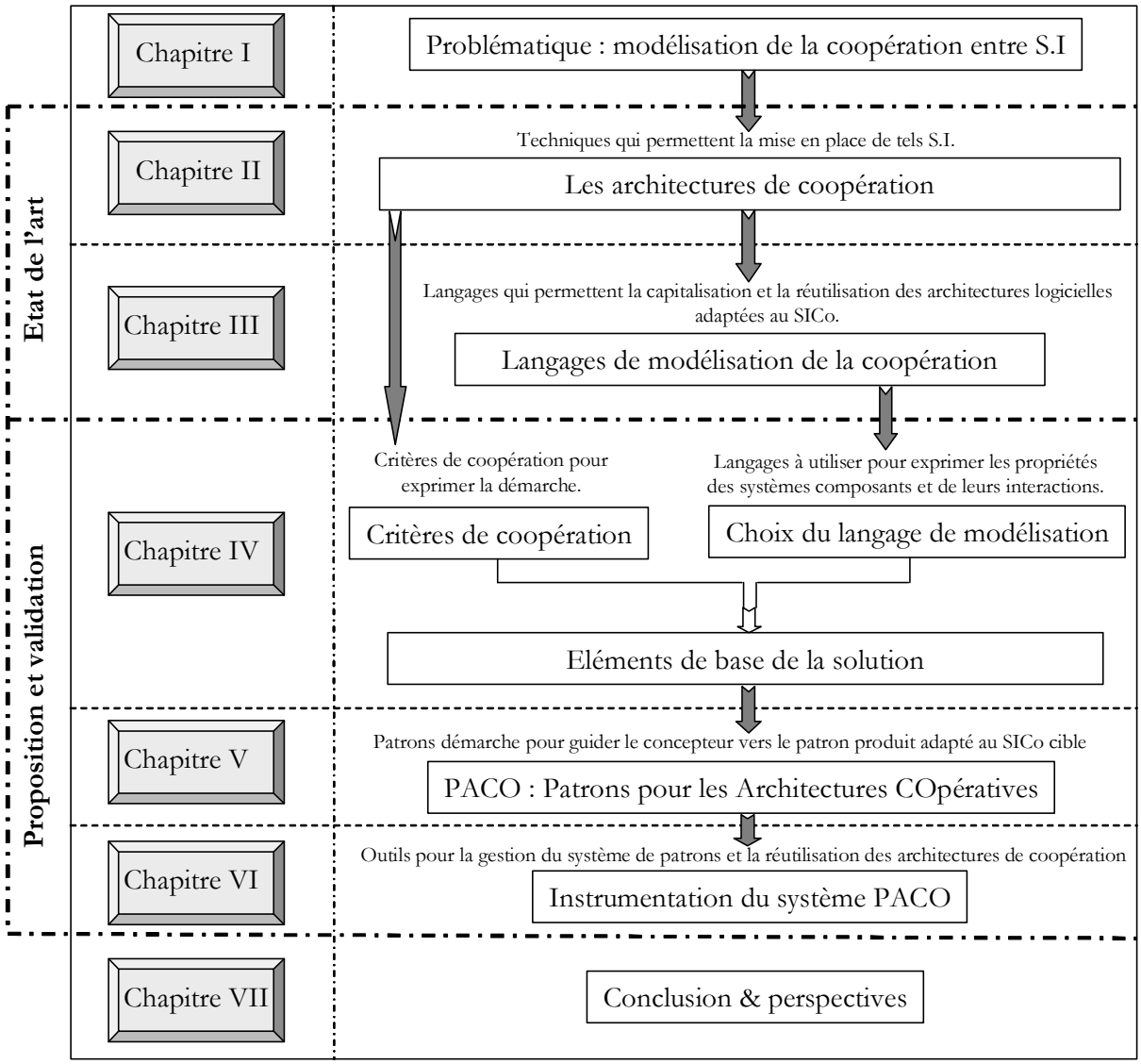
Le **quatrième chapitre** est dédié aux choix des différents éléments sur lesquels nous fondons notre proposition.

Le **cinquième chapitre** présente le système de patrons proposé pour l'ingénierie des Systèmes d'Information Coopératifs, composé de patrons produit et de patrons processus. Les premiers permettent la capitalisation et la réutilisation des architectures de coopération existantes. Les seconds proposent une démarche pour guider le concepteur dans la panoplie des patrons produit proposés et l'aider à choisir celui qui répond le mieux à ses besoins.

Nous consacrons le **dernier chapitre** à l'instrumentation du système de patrons. Nous présentons l'ensemble des outils que nous utilisons pour, d'une part, créer et gérer le système de patrons, et d'autre part, imiter les architectures de coopération proposées.

Nous terminons le document par une **conclusion** sur les résultats que nous aurons montrés, puis nous proposons des perspectives de recherches vers d'autres axes.

Nous résumons la structuration de cette thèse dans le schéma ci-contre.



CHAPITRE II

LES ARCHITECTURES DE COOPERATION

1. Introduction

Les mutations technologiques de ces dernières décennies ont bouleversé profondément la vie des organisations. Progressivement, l'idée de coopération dans les systèmes d'information est devenue une nécessité afin de répondre à des nouvelles exigences organisationnelles. De nouvelles applications ont émergé nécessitant des accès coordonnés à des systèmes d'information distants et hétérogènes. Au niveau informatique, des solutions techniques sont basées sur la notion d'interopérabilité des systèmes qui repose sur des concepts facilitant la transparence de l'échange des données et des services inter-systèmes. Cette nouvelle tendance des systèmes d'information a donné naissance à une nouvelle génération de systèmes d'information appelés les Systèmes d'Information Coopératifs (SICo).

De nombreux travaux menés dans plusieurs domaines de recherche ont pour objectif de proposer des solutions pour la conception et le développement de tels systèmes d'information. Ces recherches ont abouti à l'émergence de différentes techniques supportant l'intégration de systèmes d'information hétérogènes et autonomes.

Dans le cadre de ce chapitre, nous présentons différentes approches utilisées dans l'ingénierie des SICo. Certaines sont issues du domaine du génie logiciel, et d'autres proviennent des techniques de bases de données et de l'intelligence artificielle. Nous commençons tout d'abord par définir la notion de SICo ainsi que ses différentes propriétés. Nous nous intéressons ensuite aux techniques de capitalisation et de réutilisation des architectures logicielles qui peuvent être utilisées dans l'ingénierie des SICo. Nous présentons ainsi en premier lieu la technique des patrons (cf. section 2), et en second lieu les styles d'architectures (cf. section 3). La dernière partie de ce chapitre est consacrée à l'étude de deux techniques de coopération : les Systèmes d'Information Fédérés (cf. section 4.1) et les Systèmes Multi-Agent (cf. section 4.2).

1.1. Définitions

Selon les domaines de recherches, plusieurs définitions ont été attribuées aux SICo. Pour l'instant, nous nous limiterons de détailler deux points de terminologie :

Le terme *Système d'Information* peut se définir au niveau structurel comme : une base d'informations partagées et persistantes, une collection de traitements réalisant des actions sur la base [KORICHE, 95] et un ensemble de ressources matérielles, humaines et organisationnelles. Comme le propose [BRODIE & AL., 92], cette définition générale inclut les systèmes de bases de données, les systèmes de bases de connaissances, etc.

Le terme de *coopération* peut se définir comme la capacité pour un ensemble de systèmes d'information autonomes, coexistant dans un environnement commun, à accomplir une tâche [BRODIE & AL., 92]. Chaque système possède sa propre existence, indépendamment de la tâche globale à résoudre. Cette dernière est un but commun, dont l'accomplissement nécessite plus de ressources que n'en possède chaque système séparément, ou peut être réalisé plus efficacement par la collaboration de plusieurs systèmes [KORICHE, 95].

1.2. Propriétés des Systèmes d'Information Coopératifs

Différentes propriétés des SICo ont été évoquées dans différents articles [SCIORE & AL., 94] [LIU & AL., 95]. Nous nous limiterons à trois propriétés essentielles : distribution, autonomie et hétérogénéité.

1.2.1. Distribution

La question de la distribution physique des sources de données est orthogonale à l'autonomie et à l'hétérogénéité des systèmes. Depuis déjà quelques années, il est naturel de penser que les données et les traitements ne soient pas physiquement sur un même lieu ou machine mais, au contraire, répartis sur un réseau. Des techniques comme CORBA de l'OMG, Java/RMI, COM/DCOM, etc. permettent cette distribution.

1.2.2. Autonomie

L'autonomie fait référence au contrôle indépendant et séparé des systèmes composants. Ainsi, l'intégration ou l'abandon d'un composant ne doit pas perturber le fonctionnement global du SICo. En général, l'autonomie apparaît sous différentes formes. Scheuermann [SCHEUERMANN & AL., 90] a identifié trois niveaux d'autonomie :

1. **L'autonomie de conception :** indique que chaque système composant dans le SICo est libre de choisir sa propre conception. Cependant, il doit respecter certaines contraintes comme le modèle de données, le langage de requêtes, la représentation des données et les contraintes d'intégrité et d'implémentation.
2. **L'autonomie de communication :** se réfère à la capacité d'un système composant de décider de la façon de communiquer avec les autres systèmes. Un système serait capable de décider quand et comment il pourrait répondre aux requêtes provenant des autres systèmes composants.

3. **L'autonomie d'exécution :** permet à un système d'exécuter ses opérations locales sans interférence avec les opérations soumises par d'autres systèmes composants. Ceci implique qu'un système ne peut pas imposer un ordre d'exécution à un autre système ayant ce type d'autonomie.

1.2.3. Hétérogénéité

L'hétérogénéité peut se manifester sous plusieurs formes. Elle peut être provoquée par les différences technologiques au niveau des plates-formes matérielles et/ou logicielles (systèmes d'exploitation, réseau de communication, etc.). On peut distinguer également l'hétérogénéité induite par les différences des systèmes d'information composants. Ces différences se situent en général au niveau des modèles de données pour l'expression des structures et des contraintes et au niveau des langages de requêtes.

Les différentes formes d'hétérogénéités engendrent plusieurs types de conflits lors de la coopération des systèmes d'information. L'état de l'art de [BATINI & AL., 86] basé sur l'étude de douze méthodes d'intégration a identifié cinq types de conflits et propose des techniques pour les résoudre ; ceci dans le cadre d'intégration de vues de bases de données. On trouvera par ailleurs, d'autres classifications [SPACCAPIETRA & AL., 92] [PARENT & AL., 96] qui proposent essentiellement quatre types de conflits portant sur les aspects structurels et sémantiques de schémas (informations stockées) et classant tous les problèmes d'ordre technique (la façon d'enregistrer ces informations, en particulier les valeurs des attributs) sous la notion de conflits descriptifs [DUPONT, 95].

Dans ce qui suit, nous présentons la classification proposée par Busse S. [BUSSE & AL., 99]. Dans cette classification, nous distinguons trois types d'hétérogénéités (cf. figure 1) : l'hétérogénéité syntaxique, du modèle de données et logique, que nous détaillons ci-après.

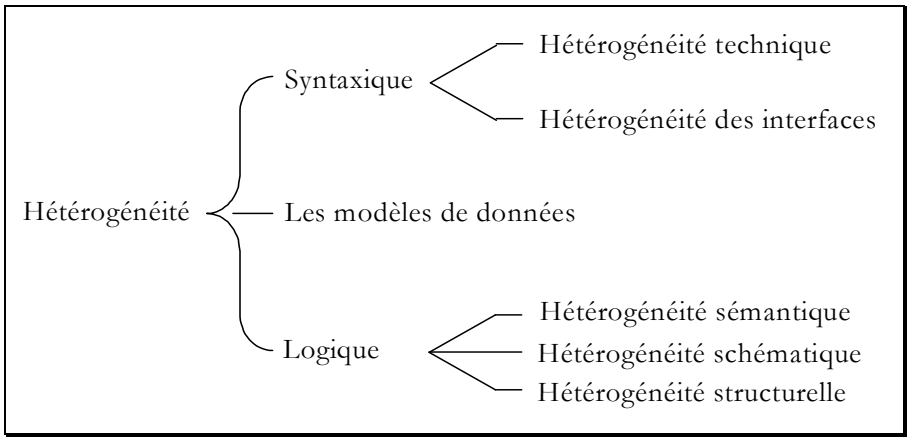


Figure 1 : Classification de l'hétérogénéité [BUSSE & AL., 99]

1.2.3.1. Hétérogénéité syntaxique

L'hétérogénéité syntaxique peut être technique ou liée aux interfaces :

1. **L'hétérogénéité technique :** couvre différents aspects techniques comme les plates-formes d'exécution et les systèmes d'exploitation, les méthodes d'accès (protocoles, méthodes de connexion, etc.).

2. **L'hétérogénéité des interfaces :** existe si différents composants sont sollicités au travers des langages différents. Il ne s'agit pas dans ce cas des choix techniques mais plutôt des choix des méthodes d'accès :

   - Hétérogénéité des langages : différents langages de requêtes avec des restrictions potentielles ;
   - Restrictions de requêtes : requêtes autorisées ou non, expression d'un nombre restreint de conditions, jointures limitées, etc.

1.2.3.2. Hétérogénéité des modèles de données

L'hétérogénéité des modèles de données exprime le fait que différents modèles de données associent des contenus sémantiques différents à leurs concepts. Par exemple, un modèle relationnel n'a pas de concept d'héritage contrairement à un modèle orienté-objet. Il y a alors des concepts qui sont différents et qui ne sont pas couverts par l'ensemble des modèles de données des différents composants. Bien que l'hétérogéniété des modèles des données soit un cas particulier d'hétérogéniété sémantique et structurelle, elle est souvent mise à part dans un nombre important de systèmes.

1.2.3.3. Hétérogénéité logique

L'hétérogénéité logique peut être sémantique, schématique ou structurelle :

1. **L'hétérogénéité sémantique :** traduit le fait que des données dans différents systèmes sont sujettes à des interprétations différentes, même si les schémas qui leur correspondent sont identiques. A l'origine, ce problème provient de l'écart séparant le monde réel du monde conceptuel (le second est une représentation du premier). Cela signifie que si l'on considère un objet dans le monde réel on peut le représenter de diverses manières dans le monde conceptuel. Deux types de conflits sémantiques ont été mis en évidence [SHETH & AL., 92] [BENSLIMANE, 99] :

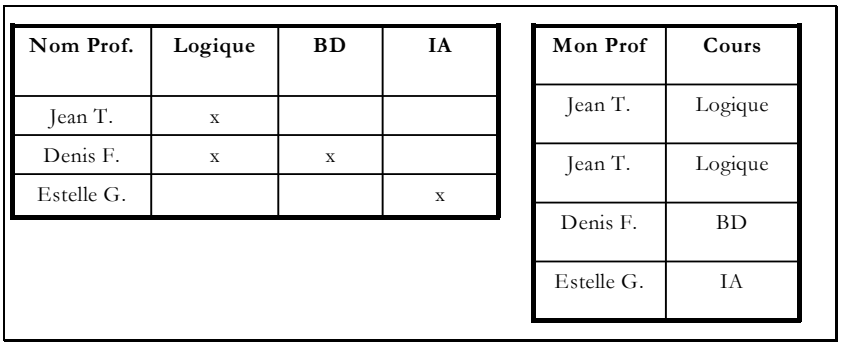
   - Hétérogénéité de nommage : consiste en des relations de synonymie (un même concept est décrit par des noms différents) ou d'homonymie (un même nom peut être utilisé pour dénoter deux concepts différents) entre les valeurs des attributs. Par exemple, un attribut "route" peut avoir comme domaine de valeurs l'ensemble {nationale, départementale} dans un système et l'ensemble {Nat., Dép.} dans un autre.
   - Hétérogénéité d'échelle ou d'unité : concerne l'utilisation d'unités ou d'échelles différentes pour mesurer les valeurs. Par exemple, le prix d'un livre peut être exprimé en Dollar dans un système, et en Euro dans un autre.

2. **L'hétérogénéité schématique :** concerne l'encodage des concepts par différents éléments du modèle de données. Par exemple, dans le modèle relationnel, trois types de conflits ont été recensés [KRISHNAMURTHY & AL., 91] :

   - Relation ↔ nom de l'attribut,
   - Nom de l'attribut ↔ valeur de l'attribut,
   - Relation ↔ valeur de l'attribut.

Un exemple pour illustrer le conflit "nom de l'attribut ↔ valeur de l'attribut" est représenté par la figure 2. Alors que la table de gauche représente les cours enseignés par un professeur

comme des noms d'attributs, la seconde table (de droite) représente les cours comme des valeurs de l'attributs "Cours".



Nom Prof.	Logique	BD	IA
Jean T.	x		
Denis F.	x	x	
Estelle G.			x

Mon Prof	Cours
Jean T.	Logique
Jean T.	Logique
Denis F.	BD
Estelle G.	IA

Figure 2 : Exemple de conflit schématique

3. **L'hétérogénéité structurelle :** elle apparaît lorsque des éléments identiques ayant le même sens (contenu sémantique identique) et modélisés avec le même modèle de données sont schématiquement homogènes mais structurés de différentes façons. Par exemple un concepteur peut définir deux concepts pour distinguer les salariés des personnes non salariés alors qu'un autre concepteur peut utiliser un seul concept pour représenter toutes les personnes.

2. Les Patrons

Le terme patron n'est pas spécifique à un domaine donné. Il est couramment utilisé en couture, en décoration, en aviation mais aussi en informatique. Ainsi, J. Rumbaugh définit un patron comme une tentative pour la représentation de l'expérience personnelle des développeurs de manière uniforme. D'après P. Coad [COAD, 95], un patron est une forme entièrement réalisée, originale ou un modèle accepté ou proposé pour une imitation, quelque chose qui est vue comme exemple normatif pouvant être copié ou utilisé.

2.1. Définition des patrons

Dans la littérature, on trouve plusieurs définitions de patrons. Ainsi, P. Coad définit un patron comme *une abstraction d'un groupe de classes réutilisables plusieurs fois dans un développement orienté objet* [COAD, 95]. Selon C. Schmidt, un patron *offre une solution récurrente à un problème standard dans un contexte donné* [SCHMIDT, 99]. M. Fowler ajoute qu'un patron est *une idée qui semble être utile dans un contexte pratique et sera probablement utile dans d'autres* [FOWLER, 97].

2.2. Description d'un Patron

Un patron est souvent présenté comme *une solution à un problème dans un contexte*. Dans cette phrase, chaque mot a un sens. *Contexte* réfère à un ensemble de situations récurrentes dans lesquelles les patrons sont appliqués. *Problème* exprime un ensemble de forces (buts et contraintes) qui ont lieu dans ce contexte. Enfin, *Solution* réfère à un modèle pour la conception que l'on peut appliquer pour résoudre ces forces. En général un patron possède quatre éléments essentiels (cf. figure 3) [GAMMA & AL., 95] :

1. **Le nom du patron :** c'est un moyen de décrire un problème de conception, ses solutions, et leurs conséquences. Donner un nom à un patron accroît immédiatement le vocabulaire de conception. Cela permet de travailler à un degré d'abstraction plus élevé.
2. **Le problème :** il décrit les situations sur lesquelles le patron s'applique. Il expose le sujet à traiter et son contexte. Il peut s'agir de problèmes spécifiques de conception. Il peut décrire des structures de classes ou d'objets typiques d'une conception immuable. Le problème peut comporter parfois une liste de conditions à satisfaire pour que la solution s'applique correctement.
3. **La solution :** elle décrit les éléments qui la constituent, ainsi que les relations entre eux, et leur coopération. La solution ne décrit pas un modèle impératif ni une implémentation, puisqu'un patron s'applique à diverses situations. Le patron fournit plutôt la description générique d'un problème de conception, et indique comment un agencement d'éléments à adapter peut le résoudre.
4. **Les conséquences :** ce sont les conséquences de la mise en œuvre du patron. Bien que ces conséquences soient rarement évoquées lors de la description des choix de conception, elles sont déterminantes pour l'évaluation des alternatives de conception et pour l'appréciation des avantages et des inconvénients de l'application du modèle.

Nom du patron	
Problème	: Description des conditions d'application. On y décrit aussi le contexte d'utilisation.
Solution	: Description des éléments (objets, relations, responsabilités et collaborations) permettant de concevoir la solution du problème.
Conséquences	: Description des résultats de l'application du patron sur le système. Contient la description des effets induits par cette application.

Figure 3 : Représentation simplifiée d'un patron [GAMMA & AL., 95]

2.3. Classification des patrons dans l'ingénierie des SI

Les patrons sont classifiés selon leur portée, c'est-à-dire l'étape à laquelle ils s'adressent. La plupart des méthodes de développement des SI distinguent traditionnellement trois étapes de développement : l'analyse, la conception et l'implémentation (cf. figure 4).

L'*analyse* permet de modéliser l'application et son domaine. Elle permet de proposer une vue d'ensemble des concepts du système proposé. Elle produit un modèle formel qui exprime à la fois les objets, leurs relations, le flux de contrôle et les contraintes sur le fonctionnement.

La *conception* est basée sur le raffinement de l'analyse. Elle est plus concrète que l'analyse. C'est à ce niveau qu'apparaissent les problèmes liés aux concepts informatiques.

L'*implémentation* consiste en la transformation du modèle de conception sous forme de code exécutable.

On trouve des patrons aux différents niveaux d'abstraction.

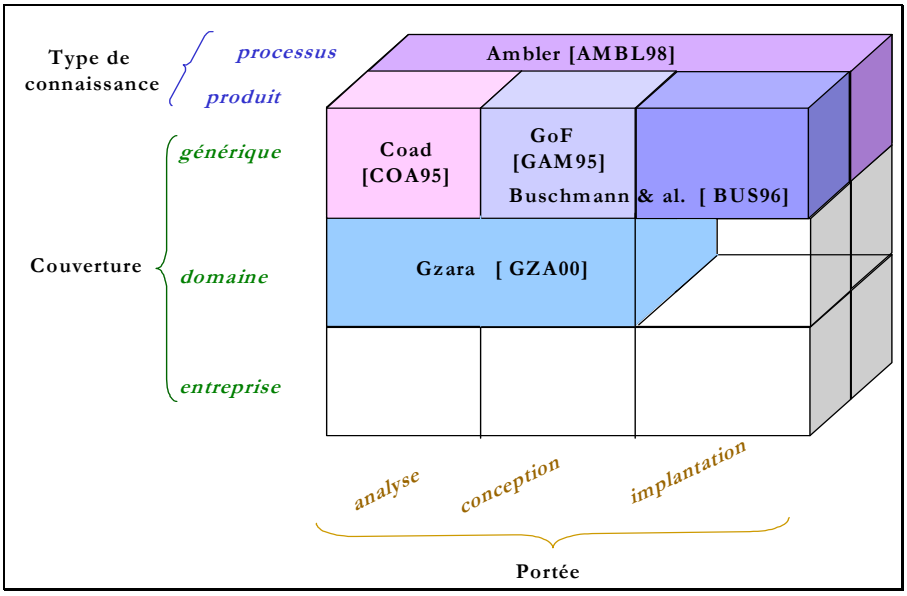


Figure 4 : Différents types de patrons [CONTE & AL., 01]

Les patrons peuvent également être classifiés en fonction de leur couverture qui peut être plus ou moins générale. C'est par exemple le cas des patrons proposés par L. Gzara [GZARA, 00] qui sont spécifiques au *domaine* des Systèmes d'Information Produit (SIP). Ils couvrent les étapes d'analyse et de conception des SIP en combinant *patrons produit* et *patrons processus*.

La classification des patrons ne repose pas sur un consensus. Tout dépend des critères de classification que l'auteur des patrons juge important à prendre en compte. Dans cette partie nous présentons deux catalogues de patrons :

- Le catalogue des patrons de conception de Gamma qui s'adresse à une seule phase de développement, la conception.
- Le catalogue des patron de Buschmann qui est constitué de patrons de conception, d'architecture et d'implantation.

Nous nous intéressons ici aux patrons de ces deux catalogues utilisables dans un contexte d'ingénierie de SICo.

2.3.1. Patrons de conception de Gamma

Les patrons de conception orientés objet, également appelés *design patterns*, sont de loin les patrons les plus connus et sans doute les plus utilisés. Ils ont été introduits par Gamma [GAMMA & AL., 95]. Ces patrons décrivent les objets coopératifs et les classes que l'on a spécialisés pour résoudre un problème général de conception.

Un patron de conception donne un nom, isole et identifie les principes fondamentaux d'une structure générale, pour en faire un moyen utile à l'élaboration d'une conception orientée objet réutilisable [GAMMA & AL., 95].

Les patrons de conception de Gamma [GAMMA & AL., 95] correspondent à des micro-architectures. Ils identifient, nomment et établissent des abstractions de thèmes communs en conception orientée objet. Ces patrons capturent l'expression et la connaissance liées à la conception en identifiant les classes et objets intervenants, leurs rôles et leurs collaborations.

Ils proposent une solution efficace et de bonne qualité (évolutive et réutilisable) à un problème de conception. Ils décrivent les circonstances où ils s'appliquent, leurs adéquations éventuelles à d'autres types de contraintes de conception, ainsi que les effets résultants de leur mise en œuvre [CONTE & AL., 01].

Gamma classifie les patrons suivant deux dimensions orthogonales (cf. tableau 1):

La première dimension *Rôle* différencie les patrons de création, structuraux et comportementaux. La deuxième dimension *Domaine* distingue si un patron s'applique principalement aux classes ou aux objets.

		Rôle		
		Créateur	**Structurel**	**Comportement**
Domaine	**Classe**	Délégation de la création des objets à des sous-classes. Ex : « Fabrication ».	Composition des classes par héritage. Ex : « Adaptateur ».	Description des algorithmes et des flux de contrôle. Ex : « Interpréteur », « patron de méthode ».
	Objet	Délégation de la création des objets à d'autres objets. Ex : « Monteur », « Prototype », etc.	Assemblage des objets. Ex : « Pont », « Composite », « Façade », etc.	Description de coopération de groupes d'objets pour accomplir une tâche. Ex : « Chaîne de responsabilité », « Médiateur », « Observateur », etc.

Tableau 1 : Classification des patrons de Gamma

GOF[1] propose un catalogue de vingt-trois patrons de conception. Ces derniers facilitent la réutilisation de solutions de conception et d'architectures efficaces. Ils aident à choisir parmi les alternatives de conception, celles qui favorisent la réutilisabilité, et à éviter celles qui la compromettent [GAMMA & AL., 95]. Ces patrons offrent des modèles qui ont été appliqués plusieurs fois dans différents systèmes.

Notons que certains patrons de Gamma peuvent être appliqués pour la modélisation de la coopération. Par exemple, le patron « façade » fournit une interface unique pour une collection d'objets ; le patron « médiateur » formalise une coopération entre objets, le patron « observateur » maintient la cohérence d'objets en relation, etc. [GAMMA & AL., 95]

Dans la suite nous présentons le patron « Façade ». Ce patron fournit une interface unifiée à l'ensemble des interfaces de sous-systèmes (figure 5).

[1] GOF est l'abréviation de Gang Of Four, qui est le surnom donné aux quatre auteurs du livre qui ont promu les patrons de conception [GAMMA & AL., 95].

Exemple d'un patron de conception de Gamma sous le formalisme P-Sigma[2] [GAMMA &AL., 95]

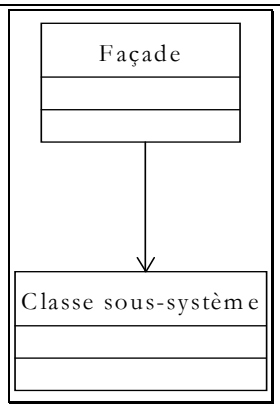
Identifiant	Façade
Classification	Structurel – Objet
Contexte	Ce patron ne nécessite aucun autre patron ou modèle pour être appliqué.
Problème	▪ On dispose d'un sous-système complexe (flux difficiles à gérer). ▪ Plusieurs relations de dépendance entre les clients et les classes d'implémentation d'une abstraction. ▪ Pour structurer un sous-système en niveau.
Force	Ce patron fournit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.
Solution modèle	 ▪ Façade : connaît les classes du sous-système compétentes pour une requête, ▪ Classes sous-systèmes : ne connaissent pas la façade, implémentent les fonctionnalités du sous-système, gèrent les travaux assignés par l'objet façade.
Conséquences d'application	▪ Masquer aux clients des composants du sous-système donc réduire le nombre d'objets. ▪ Favoriser le couplage faible entre le sous-système et ses clients.

Figure 5 : Exemple d'un patron de conception de Gamma

2.3.2. Patrons de Buschmann

Le catalogue proposé par Buschmann [BUSCHMANN & AL., 96] couvre trois phases dans le cycle de développement logiciel. Il propose trois catégories de patrons : les patrons d'architectures, les patrons de conception et les idiomes. La différence entre ces trois types de

[2] Le formalisme P-Sigma est décrit dans l'annexe 1

patrons réside dans leurs niveaux correspondants d'abstraction et de détail. Dans la suite nous nous intéressons uniquement aux patrons d'architecture et de conception.

2.3.2.1. Patrons de conception

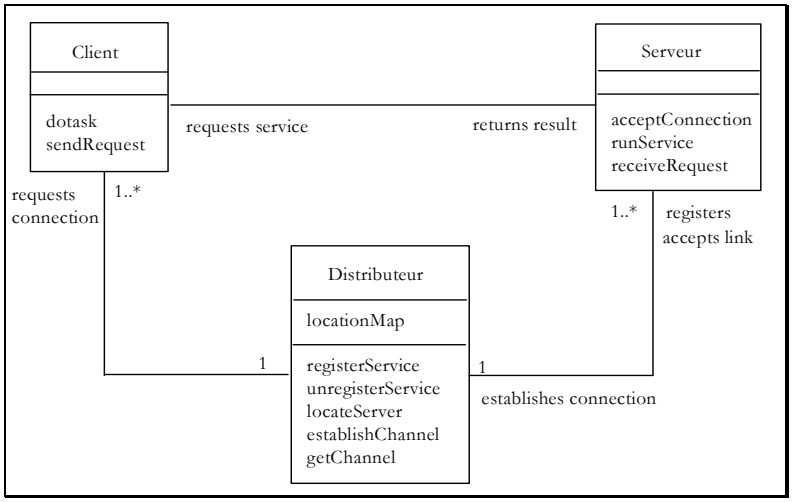
Le tableau suivant résume les différentes catégories de patrons de conception proposées par Buschmann et al (cf. tableau 2).

	Patron de conception
Décomposition structurelle	Découper des sous-systèmes et des composants complexes pour obtenir des composants indépendants plus petits. Ex : « Whole-Part ».
Organisation de travail	Cette catégorie comprend seulement le patron « Master-Slave ». Ce dernier est composé d'un composant maître qui distribue le travail à des composants esclaves identiques et calcule le résultat final à partir des résultats partiels retournés par ses esclaves.
Contrôle d'accès	Autoriser un client à accéder à un composant ou un sous système. Cette catégorie comprend seulement le patron « Proxy ». Ce dernier offre une interface qui permet de contrôler l'accès à un composant distant.
Gestion	Les patrons dans cette catégorie permettent de manipuler une collection homogène d'objets, de services ou bien de composants complexes. Ex : « Command Processor », « View Handler ».
Communication	Faciliter la communication entre des sous-systèmes distribués qui doivent collaborer ensemble. Ex: « Forwarder-Receiver », « Client-distributeur -Serveur », etc.

Tableau 2 : Classification des patrons de conception de Buschmann

Exemple d'un patron de conception de Buschmann sous le formalisme P-Sigma [BUSCHMANN & AL., 96]

Identifiant	Client-Distributeur-Serveur
Classification	Patron de conception - Communication
Contexte	Ce patron ne nécessite aucun autre patron ou modèle pour être appliqué.
Problème	Quand un système logiciel utilise des composants distribués sur un réseau, il doit fournir un moyen pour assurer la communication entre eux. Généralement, une connexion entre les composants doit être établie avant

<table>
<tr><td></td><td>que la communication puisse avoir lieu selon les moyens de communication disponibles. Dans ce cas de figure, il faut tenir compte de deux points :
▪ les principales fonctionnalités des composants doivent être séparées des détails des mécanismes de communication.
▪ les clients n'ont pas besoin de connaître la localisation des serveurs.</td></tr>
<tr><td>Force</td><td>Ce patron introduit une couche intermédiaire entre clients et serveurs : le distributeur (dispatcher). Il assure la transparence de l'emplacement et des détails de l'établissement de la connexion pour la communication entre client et serveur.</td></tr>
<tr><td>Solution modèle</td><td> 

▪ Le distributeur agit comme une couche intermédiaire entre clients et serveurs. Il implémente un nom de service qui permet aux clients de se référer aux serveurs par leurs noms sans se soucier de leurs emplacements physiques.
▪ Serveurs : Chaque serveur est identifié uniquement par son nom et est connecté aux clients par l'intermédiaire du distributeur. Le distributeur peut ajouter à l'application des serveurs qui fournissent des services aux autres composants.
▪ Les clients s'appuient sur le distributeur pour localiser un serveur particulier et établir une communication avec ce dernier. Contrairement au modèle client/serveur classique, les rôles des clients et des serveurs peuvent changer dynamiquement.</td></tr>
<tr><td>Conséquences d'application</td><td>▪ "Echangeabilité" des serveurs.
▪ Transparence de la localisation.</td></tr>
</table>

Figure 6 : Exemple de patron de conception de Buschmann

2.3.2.2. Patrons d'architecture

Un patron d'architecture exprime l'organisation et le schéma d'une application. Il décompose l'application en un ensemble de sous-systèmes tout en spécifiant pour chacun ses responsabilités, ses interactions avec les autres sous-systèmes [BUSCHMANN & AL., 96].

Un patron d'architecture décrit les règles d'organisation et de fonctionnement. Il décrit les stratégies de haut niveau, les propriétés et les mécanismes globaux d'une application. Un tel procédé de conception permet de créer des architectures extensibles et inter-opérables.

Buschmann et al. distingue quatre catégories de patrons d'architecture que nous synthétisons dans le tableau suivant (cf. tableau 3).

	Patrons d'architecture
Système décomposé	Supportent la décomposition d'une tâche en des sous tâches coopératives. Ex : « Layers », « Pipes & Filters » et « Blackboard ».
Systèmes distribués	Structurent les systèmes logiciels distribués sous forme de composants découplés qui interagissent par des invocations de services distants. Ex : le « Broker », responsable de la coordination de la communication : transmettre une requête, un résultat ou une exception.
Systèmes interactifs	Structurent les systèmes logiciels qui décrivent les interactions homme-machine. Ex : « Model-View-Controller » et « Presentation-Abstraction-Control ».
Systèmes adaptables	Supportent fortement l'extension des applications et permettent une adaptation dynamique de la structure et du comportement de l'application suite aux éventuelles modifications des besoins fonctionnels. Ex : « Microkernel » et « Reflection ».

Tableau 3 : Classification des patrons d'architecture de Buschmann

Notons que certains patrons d'architecture de Bushmann, tels que les patrons « Layers », « Pipes & Filters » et « Blackboard », sont aussi connus sous le nom de *styles d'architecture*. La notion de patron d'architecture est similaire à celle de style d'architecture. Généralement, dans la littérature les deux termes sont utilisés de manière analogue. Buschmann [BUSCHMANN & AL., 96] précise que n'importe quel style architectural peut être décrit sous forme d'un patron d'architecture. Cependant, il met en évidence quelques différences :

- Les styles décrivent la structure globale de l'application (niveau d'abstraction élevé), alors que les patrons pour l'architecture logicielle portent sur différents niveaux d'abstraction.
- Les styles architecturaux sont indépendants l'un de l'autre alors que les patrons sont souvent décrits en relation.
- Les styles d'architecture expriment des techniques de réutilisation indépendamment du contexte de modélisation alors qu'un patron fournit une solution à un problème de modélisation dans un contexte bien déterminé.

Dans le paragraphe suivant nous étudions de plus près des styles d'architecture.

3. Styles architecturaux

3.1. Introduction générale

Dans la pratique, les conceptions architecturales ont été codifiées et réutilisées principalement à travers la transmission informelle des idiomes architecturaux. Par exemple, un système pourrait être défini comme un "système client-serveur", "un système de boîte noire", "un interprète", etc. Une famille importante des idiomes architecturaux constitue ce qu'on appelle des *styles architecturaux*. Dans [MONROE & AL., 97], un style architectural caractérise une famille de systèmes de propriétés sémantiques et syntaxiques similaires. Un style fournit un vocabulaire basé sur des types de composants et de connecteurs (pipe, filtre, analyseur, base de données par exemple), des règles de conception (ou contraintes), une interprétation sémantique et des analyses.

De nombreuses recherches ont permis d'introduire une dimension nouvelle à la notion d'architecture. La définition généralement admise est : *une architecture spécifie les modules du système (appelés composants du système) et l'interaction entre ces composants afin de satisfaire les besoins d'un système* [LUCKHAM & AL., 95]. Cette notion est essentielle puisqu'elle offre une vision globale de l'état et du comportement de l'application et facilite, par la même, sa construction et son administration. Plus particulièrement, elle permet l'évolution de l'application et la définition de contraintes sur le système. En outre, la structure à un haut niveau d'abstraction, exposant un ensemble de composants interagissant, permet la spécification de propriétés importantes du système (protocoles d'interaction, localisation de données).

3.2. Qu'est-ce qu'un style ?

D'après [MEDVIDOVIC & AL., 97], un style architectural est composé de quatre parties :

1. **Composants et connecteurs :** les composants sont des unités de calcul ou de données. Leurs interfaces sont les points d'interaction avec les connecteurs ou avec l'environnement. Les connecteurs sont utilisés pour modéliser les interactions entre les composants. L'interface d'un connecteur est l'ensemble des points d'interaction avec les composants auquel il est rattaché. Composants et connecteurs sont décrits à l'aide de schémas paramétrés.
2. **Configuration globale :** elle décrit la structure architecturale caractéristique du style, c'est-à-dire la manière dont doivent interagir les composants et les connecteurs. Elle est définie par un schéma décrivant le comportement général d'une spécification, la composition des composants et des connecteurs.
3. **Caractéristiques :** il s'agit de l'ensemble des paramètres du style, c'est-à-dire l'ensemble des informations nécessaires à la définition de la spécification par instanciation des schémas. Ces informations sont dénotées par des variables typées et des contraintes qui paramétrisent les différents schémas.
4. **Contraintes architecturales :** ce sont les propriétés qu'une spécification doit vérifier afin d'être conforme au style, et ce, indépendamment du développement suivi, de

l'utilisation ou non des schémas proposés. Ce sont des contraintes structurelles concernant la configuration globale ou comportementale concernant les composants et les connecteurs.

Pour mieux caractériser cette notion de style d'architecture, R. T. Monroe [MONROE & AL., 97] a identifié quatre aspects communs :

1. **Un vocabulaire des éléments de conception :** les types de composants et de connecteurs comme les pipes, les filtres, les clients, les serveurs, les bases de données, etc.
2. **Des règles de conception ou des contraintes** qui déterminent quelles compositions de ces éléments sont autorisées. Par exemple, les règles pourraient interdire des cycles dans un style particulier de pipe-filtre ou spécifier qu'une organisation client/serveur doit être une relation n-à-un.
3. **L'interprétation sémantique**, par laquelle les compositions des éléments de conception, convenablement contraintes par des règles de conception, ont des significations bien définies.
4. **Les analyses** qui peuvent être exécutées sur des systèmes construits dans ce style. Par exemple, l'analyse de l'ordonnançabilité pour un style orienté vers le traitement en temps réel [VESTAL, 94].

3.3. Avantages des styles architecturaux

L'utilisation des styles architecturaux a un certain nombre d'avantages significatifs [MONROE & AL., 97] :

- Elle permet la *réutilisation de la conception architecturale* : des solutions courantes avec des propriétés bien comprises peuvent être appliquées à de nouveaux problèmes.
- L'utilisation des styles architecturaux peut conduire à *réutiliser du code* significatif : les aspects invariants d'un style architectural permettent souvent une implémentation partagée.
- Il est plus facile pour les autres de *comprendre* l'organisation du système si des structures conventionnelles sont utilisées. Par exemple, caractériser un système comme une organisation client/serveur donne immédiatement une image forte des types de composants et connecteurs présents et de leur relation.
- L'utilisation de styles standardisés supporte *l'interopérabilité* (CORBA).
- En contraignant la liberté de conception, un style architectural permet souvent la *spécialisation* et l'analyse d'un style spécifique. Nous pouvons ainsi formuler des propriétés propres au style, ce qui ne pourrait pas être possible si un système était construit avec différents styles.

3.4. Typologie des styles architecturaux

Dans cette section nous allons examiner les styles d'architectures les plus récurrents. La figure 7, présente une classification des styles architecturaux recensés par D. Garlan [SHAW & AL., 96A]. Cependant, dans le cadre de notre travail, nous nous intéressons particulièrement aux styles d'architectures qui offrent des solutions ou des alternatives pour la coopération des systèmes d'information. Ces styles sont présentés en gras dans la figure 7.

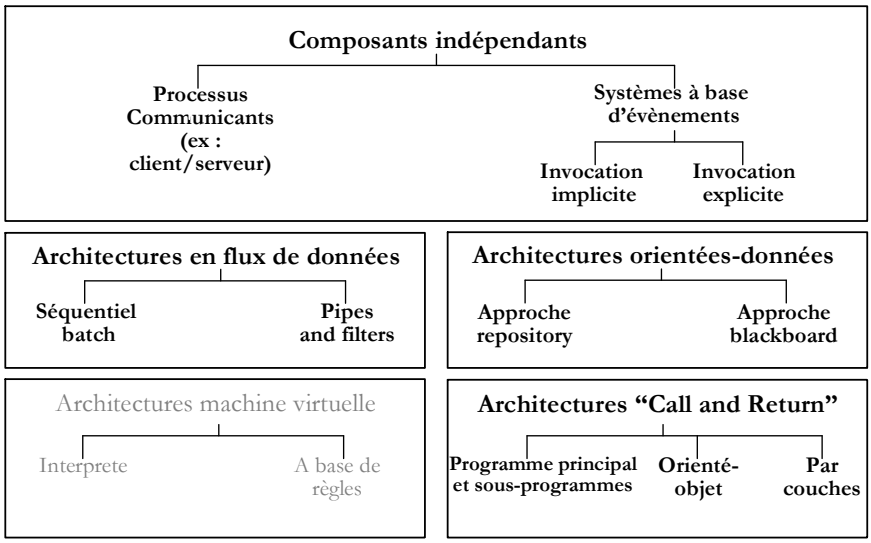


Figure 7 : Typologie des styles d'architectures

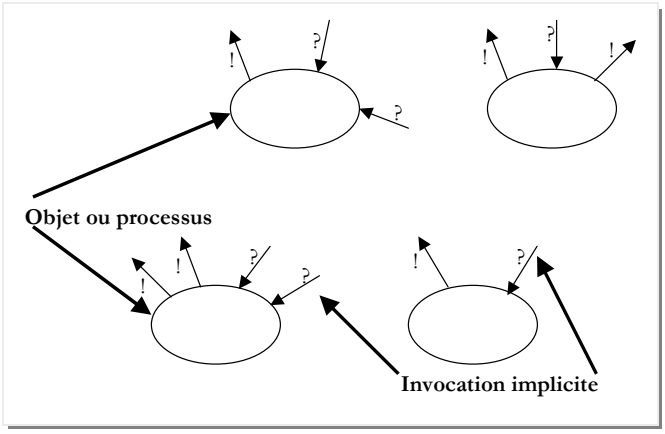
Nous allons aborder en détail les styles : « Invocation implicite », « Pipes & filtres », « Approche Repository », « Architecture Orientée Objet » et « Architecture en Couches ». Les autres styles non traités correspondent à des variantes de ces styles (ex : « l'approche blackboard » est une alternative de « l'approche repository ») ou bien à des styles très connus dans la littérature (ex : « client/serveur »).

Comme nous l'avons vu dans la section 2.3.2, un style architectural peut être documenté sous forme de patrons. Dans le cadre de ce travail, nous adoptons le formalisme de patrons P-Sigma [CONTE & AL., 01] (voir annexe 1).

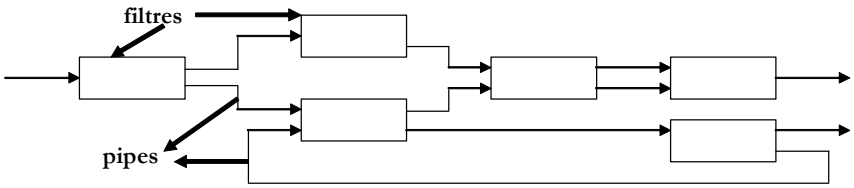
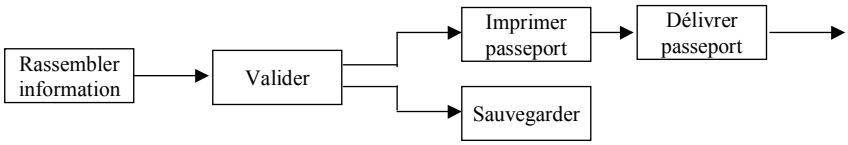
Pour ce chapitre, nous nous contenterons de décrire la *solution modèle* des différents styles architecturaux à l'aide de schémas intuitifs. Dans la rubrique *classification*, nous reprendrons la classification présentée dans la figure 4.

3.4.1. Le style « Invocation implicite »

Identifiant	**Invocation implicite**
Classification	Composants indépendants – Systèmes à base d'événements
Problème	L'utilisation de ce patron est adaptée pour les applications qui impliquent l'utilisation d'un ensemble de composants faiblement couplés. Certains composants diffusent des événements et d'autres s'inscrivent pour des événements. L'utilisation de ce patron est particulièrement utile pour les applications qui ont besoin d'être reconfigurées en cours d'exécution.
Force	L'idée sous-jacente à ce patron est que, plutôt que d'invoquer une procédure directement, un composant peut publier ou diffuser un ou plusieurs événements. D'autres composants du système peuvent

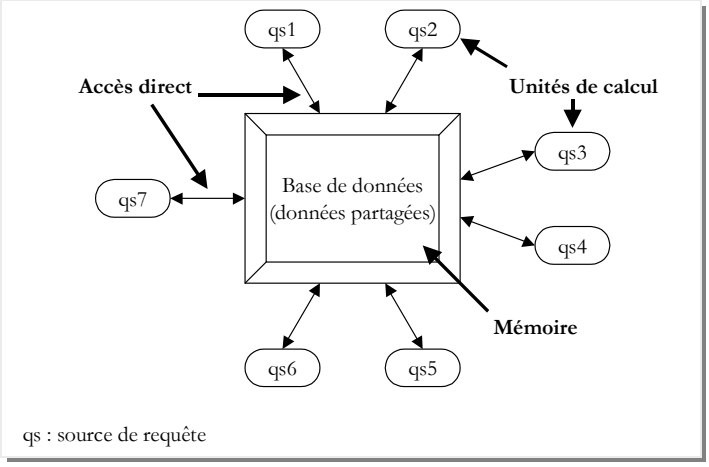
	enregistrer un intérêt pour un événement dans un gestionnaire d'événements, en lui associant une procédure. Lorsque l'événement est émis, le système invoque toutes les procédures qui ont été enregistrées pour l'événement en question. Ainsi, une annonce implicite d'événement d'un composant implique l'invocation de procédures dans d'autres composants.
Solution modèle	Composants : Objet, processus, agent, etc. Connecteurs : Appels de procédures. 
Cas d'application	Exemple d'architectures utilisant ce patron : 1- Les environnements qui fédèrent plusieurs outils tel que les environnements de programmation : ▪ le débogueur détecte une erreur et annonce un événement, ▪ l'éditeur prend en compte l'événement en surlignant la ligne erronée dans le code source. 2- Dans les systèmes de gestion de bases de données pour vérifier les contraintes d'intégrité.
Conséquences d'application	Avantages : 1. Facilite le changement et la réutilisation. 2. Permet l'ajout de services dynamiquement. Inconvénients : 1. Le composant ne sait pas ce que devient l'événement qu'il a créé. 2. Nécessite (encourage) l'utilisation d'un gestionnaire d'événements.

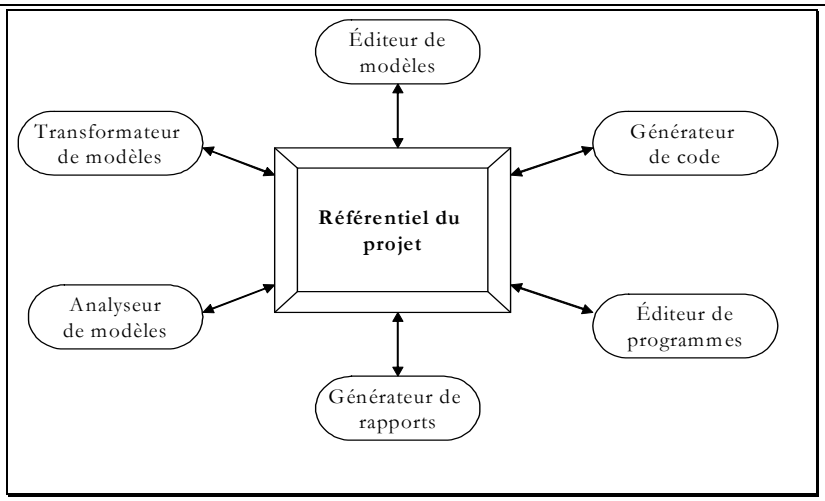
3.4.2. Le style « Pipes & Filters »

Identifiant	Pipes & Filters
Classification	Architecture à flux de données
Problème	L'utilisation de ce patron est appropriée pour les applications qui requièrent des séries prédéfinies de composants pour les exécuter dans un ordre déterminé. Les composants peuvent aussi être exécutés en parallèle, ce qui permet de réduire le temps d'attente du système. Les composants sont appelés des filtres. Les connecteurs de ce style servent de "tuyaux" pour les flots de données, transmettant les sorties d'un filtre vers les entrées d'un autre. Ces connecteurs sont appelés pipes. Les filtres sont des entités indépendantes : ils ne tiennent pas compte des états des autres filtres.
Force	Ce patron est basé sur la capacité de décomposer un système en un ensemble de composants, ou filtres, qui transforment un ou plusieurs flots d'entrée en un ou plusieurs flots de sortie de manière incrémentale.
Solution modèle	 Composants : filtres. Connecteurs : pipes ou flux de données.
Cas d'application	L'exemple suivant décrit le processus de délivrance d'un passeport. 
Conséquence d'application	Avantages : 1. Réutilisation et maintenance accrues. 2. Potentiel d'exécution en parallèle. Inconvénients : 1. Coût d'assemblage / désassemblage. 2. Mal adapté à des systèmes interactifs.

	3. Ne facilite pas l'ajout de composants.
Alternative	Le patron Séquentiel Batch : dans ce cas, le filtre ne peut pas commencer le traitement avant la terminaison du flot d'entrée.

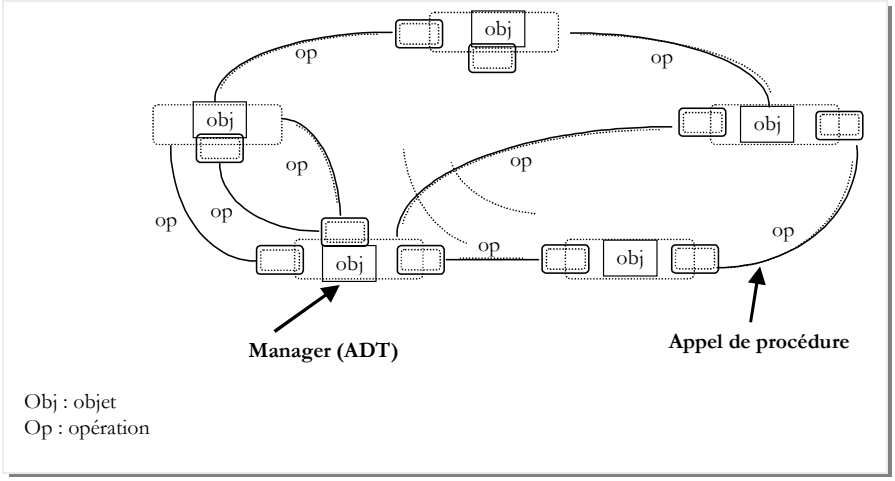
3.4.3. Le style « Repository »

Identifiant	**Approche repository**
Classification	Architectures orientées données.
Problème	L'utilisation de ce patron est appropriée pour les applications dont la vocation principale est d'établir et de maintenir un entrepôt central de données. Souvent, la persistance à long terme des données est exigée.
Force	Ce patron est basé sur un composant qui joue le rôle central et qui gère la structure des données. Les autres composants n'interagissent qu'avec lui.
Solution modèle	Deux types de composants : 1) un composant (repository) centralisé où se trouvent les données partagées, 2) un ensemble de composants actifs quasi autonomes bâtis autour du repository. Connecteurs : Accès aux données ou appels de procédures. 
Cas d'application	Architecture type d'un outil CASE basé sur un référentiel de projet.

	
Conséquences d'application	Avantages : 1. Simplicité. 2. Centralisation de l'accès aux données. 3. Efficacité pour de grosses quantités de données 4. Quasi-indépendance des composants. Inconvénients : 1. Caractère crucial du repository (risque accru de pannes)
Alternative	Architecture à base de repository actif (Blackboard) : chaque changement d'état du composant central (blackboard) peut provoquer l'invocation d'autres composants.

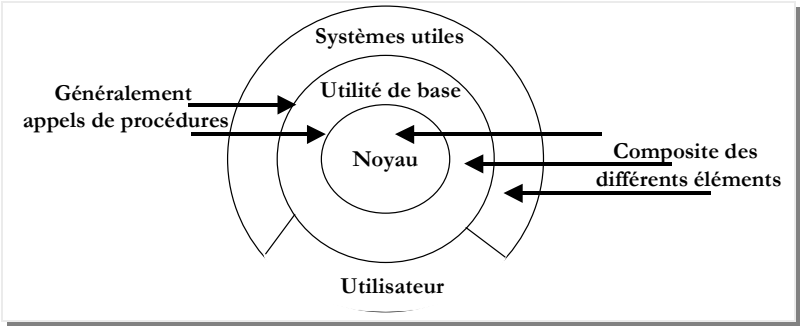
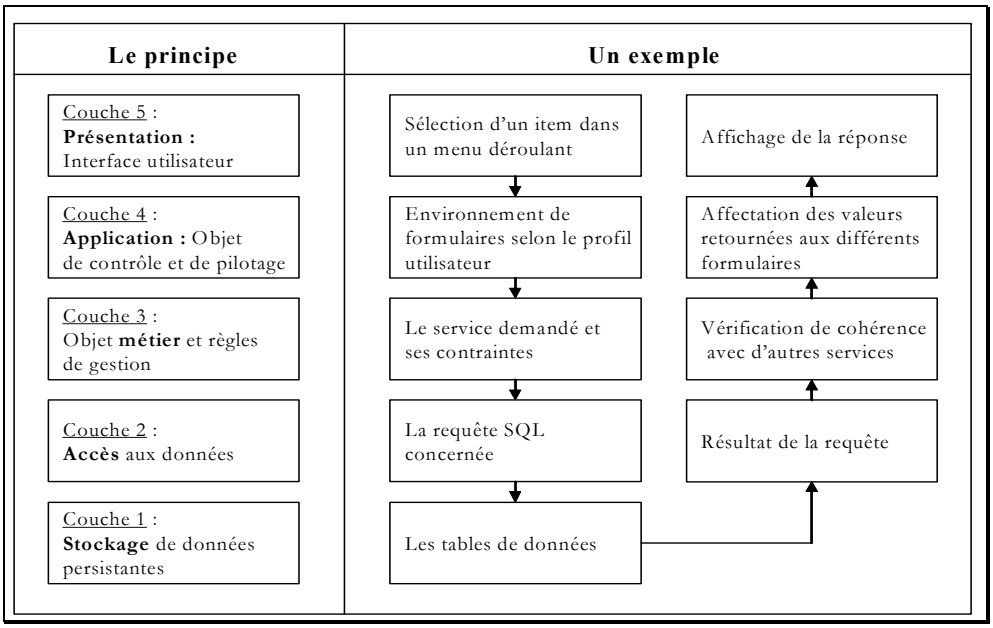
3.4.4. Le style « Organisation orientée objets »

Identifiant	**Oganisation orientée objets**
Classification	Architecture Call and Return.
Problème	L'utilisation de ce patron est appropriée pour structurer un système en différents éléments ou composants. Ces derniers sont assemblés en vue de réaliser le ou les services que le système doit rendre. Les composants encapsulent les données, les opérations, ainsi que les contraintes d'intégrité définies sur les données et les opérations.
Force	Dans ce modèle, les composants sont des objets qui communiquent entre eux par appel de fonctions ou de procédures. Chaque objet est responsable du respect de l'intégrité de sa représentation. D'autre part, la représentation de cet objet est inconnue et invisible pour les autres objets.
	Composants : objets. Connecteurs : appels de procédures.

Solution modèle	
Cas d'application	Tout système orienté objets.
Conséquence d'application	Avantages : 1. Réutilisation et maintenance. 2. Tous les avantages de la programmation orientée objets. Inconvénients : 1. Un composant doit connaître l'identité des modules avec qui il est en interaction.

3.4.5. Le style « Par couches »

Identifiant	**Par couches**
Classification	Architecture Call and Return.
Problème	L'utilisation de ce patron est appropriée pour les applications qui impliquent des composants qui peuvent être organisés hiérarchiquement. Il existe souvent des couches pour les services de base du niveau système, pour les services appropriés aux applications et pour les tâches spécifiques à l'application.
Force	Les modèles en couches sont organisés hiérarchiquement, chaque couche fournissant un service pour la couche supérieure et agissant comme client pour la couche inférieure. Une couche peut être vue comme un composant possédant deux interfaces et n'est autorisé à interagir qu'avec les deux composants adjacents.
Solution modèle	Composants : généralement des composites, le plus souvent composés de collections de procédures.

<table>
<tr><td></td><td>Connecteurs : ils dépendent de la structure des composants. Le plus souvent ils sont représentés sous forme d'appels de procédure avec une visibilité restreinte, et peuvent aussi être sous forme de communication client-serveur.
 
</td></tr>
<tr><td>Cas d'application</td><td>L'exemple suivant présente un cas d'utilisation courant en génie logiciel :
 
</td></tr>
<tr><td>Conséquences d'application</td><td>Avantages :
1. La maintenance : le changement d'un niveau n'en affecte que deux autres (amont, aval).
2. La réutilisation des couches basses.
3. Conception incrémentale.
Inconvénient :
1. Difficulté de définir le contenu de chaque couche et ses interfaces.</td></tr>
</table>

3.4.6. Les styles hétérogènes

Les styles d'architecture que nous avons énumérés dans les paragraphes précedents sont des styles élémentaires. Cependant, dans la pratique, les systèmes reposent souvent sur une

combinaison de plusieurs styles. Ce sont des styles d'architecture hétérogènes (cf. figure 8). Par exemple, [SHAW & AL., 96A] a identifié trois types d'hétérogénéité :

1. **Hétérogénéité hiérarchique :** L'architecture interne d'un composant dans un système organisé selon un style d'architecture bien déterminé peut être développée dans un autre style. Au même titre que les composants, les connecteurs peuvent être décomposés hiérarchiquement.
2. **Hétérogénéité locale :** Un composant peut utiliser des connecteurs différents. Par exemple un composant peut accéder à un repository en utilisant un connecteur de type appel de procédure, et communiquer avec d'autres composants en utilisant un connecteur de type pipe.
3. **Hétérogénéité simultanée :** Chaque niveau dans l'architecture est élaboré dans un style différent de l'autre.

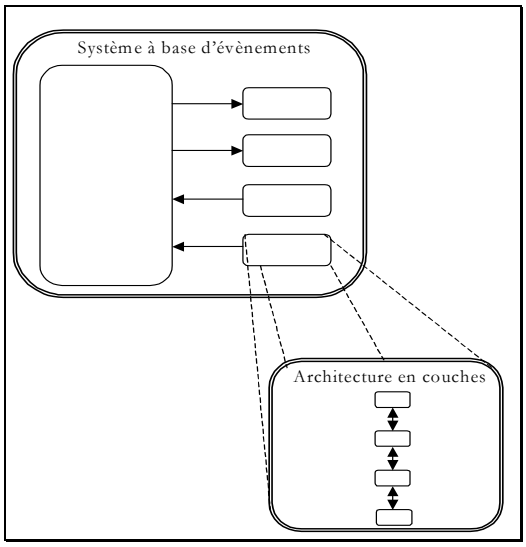


Figure 8 : Exemple de styles hétérogènes : hétérogénéité hiérarchique

3.4.7. Conclusion

Dans cette section, nous avons présenté les styles d'architecture les plus connus et qui peuvent répondre à des situations de coopération entre systèmes composants. La description de ces différents styles a été abordée telle qu'elle a été décrite dans la littérature. Nous pouvons remarquer que les critères utilisés dans la rubrique *classification* ne permettent pas de différencier les multiples styles architecturaux et orienter ainsi le concepteur vers le style le plus adéquat. Le critère "Architecture Call and return" ne permet pas par exemple de différencier le style « Organisation orientée objets » et le style « Par couches ». Dans le chapitre V nous reviendrons sur ces styles pour les formaliser et les classifier dans le contexte des Systèmes d'Information Coopératifs.

Les styles architecturaux ainsi que les patrons de conception peuvent être considérés comme deux techniques de modélisation complémentaires permettant l'encapsulation et la réutilisation des savoir-faire des expériences antérieures. D'une part, un style architectural fournit un ensemble de règles et de contraintes pour décrire les interactions entre les composants du système. D'autre part, les patrons aident les concepteurs à mieux structurer la représentation d'un environnement, les guident sur la perception du monde réel et permettent

d'en obtenir une description à un niveau d'abstraction élevé. Les patrons aident à aborder des systèmes complexes d'une manière simple sans qu'il soit nécessaire d'en étudier les détails d'implantation.

Les patrons constituent une base d'expériences pour construire des composants réutilisables, et agissent comme des blocs de construction à partir desquels des systèmes plus complexes peuvent être construits [CONTE, 97]. Ils sont indépendants des langages de programmation et des plates-formes d'implantation du logiciel, aspect qui accentue la réutilisation des composants. En résumé, ils permettent de capitaliser une connaissance partageable en matière d'étude et de développement logiciel, recensant et diffusant ainsi les bonnes pratiques.

Les styles d'architectures peuvent aussi être vus aussi comme des types de patrons [SHAW & AL., 96A] [BUSCHMANN & AL., 96] ou mieux encore, comme des langages de patrons. Etant donné un style architectural, il existe un jeu d'utilisations typiques qui agissent comme "des micro-architectures", ou des patrons de conception architecturaux qui sont conçus pour travailler dans un style architectural spécifique.

Comme nous l'avons évoqué au début de cette section, les styles architecturaux sont généralement décrits par des noms ou des phrases simples (ex : nous avons opté pour une architecture en couches…) mais évoquent une architecture souvent complexe. Les architectures des Systèmes d'Informations Coopératifs peuvent être basées sur certains styles architecturaux ou sur une combinaison de styles.

Dans la section suivante nous présentons trois techniques de coopération. Chacune d'elles repose sur un style architectural et se présente ainsi comme un exemple d'utilisation de ce style. Dans un premier temps, nous présentons la technique des bases de données fédérées qui repose sur le style « Blackboard ». Ensuite, nous exposons les systèmes de médiation qui sont basés sur le style « par couches ». Enfin, nous décrivons les Systèmes Multi-Agents (SMA) qui reposent, d'une manière générale, sur le style « Invocation implicite ».

4. Les techniques de coopération

Dans la première partie de cette section, nous présentons les techniques de coopération selon deux approches : Les Systèmes d'Information Fédérés (SIF) et les Systèmes Multi-Agents (SMA). Dans la première approche la coopération est assurée par l'interopérabilité des sources d'information autonomes, distribuées et hétérogènes. Dans le cadre des SMA, la coopération présuppose des agents qui ont des buts et qui peuvent agir entre eux. Plus généralement, un agent est coopératif s'il partage des buts avec d'autres agents dans son environnement et agit pour l'accomplissement de ces buts communs [MICHELIS & AL., 97].

4.1. Les Systèmes d'Information Fédérés

Les systèmes d'information peuvent être classifiés selon les critères de la *distribution* et de l'*hétérogénéité* [BUSSE & AL., 99] comme suit :

- Un système unique (monolithique, centralisé) s'exécute comme une application monolithique, sur une seule machine. Un système d'information monolithique peut être soit structuré soit semi structuré.

- Un système d'information distribué est un système pour lequel les données sont physiquement distribuées sur plusieurs sites interconnectés.
- Un système d'information hétérogène est une collection de systèmes qui diffèrent selon des aspects logiques ou syntaxiques (plate-forme, système d'exploitation, modèle de données, etc.).

Si nous ajoutons la dimension de l'autonomie, nous abordons la définition d'un Système d'Information Fédéré (SIF), vu comme un ensemble de composants (ou systèmes) distincts et indépendants : les participants de la fédération.

4.1.1. Critères de classification

Dans la communauté des bases de données, différentes approches de SIF ont été proposées. Leur classification peut être faite selon différents critères comme par exemple l'autonomie des systèmes, les architectures logicielles adoptées ou les outils de résolution de conflits de données, etc. Dans ce contexte nous présentons les critères de classification suivants [BUSSE & AL., 99] :

4.1.1.1. Types de systèmes composants

Les SIF peuvent être classés selon les types des systèmes composants qu'ils peuvent intégrer dans la coopération. Ils peuvent ainsi permettre ou pas l'intégration des systèmes d'information structurés (ex : bases de données relationnelles, ...) ou semi-structurés (ex : bases de données XML, ...).

4.1.1.2. Transparence

Un SIF "parfait" doit faire illusion à l'utilisateur qui doit le percevoir comme un seul système d'information centralisé, homogène et consistant. On distingue les types de transparence suivants :

- Transparence de localisation : l'utilisateur n'a pas besoin de connaître l'emplacement physique de l'information.
- Transparence de schéma : L'utilisateur n'a pas besoin de connaître sous quel modèle de données sont décrits les différents systèmes composants. Autrement dit, tous les conflits logiques sont masqués. Cela nous amène à dire que la transparence de schéma n'est possible que si un schéma fédérateur existe.
- Transparence de langage : l'utilisateur n'a pas à s'occuper du langage de requêtes utilisé ainsi que des mécanismes d'accès.

4.1.1.3. Fédération forte versus lâche

Les SIF se définissent par l'absence ou la présence d'un schéma global. On parle alors respectivement de SIF faiblement couplés ou fortement couplés. Dans le cas des SIF fortement couplés, l'utilisateur interagit avec un seul schéma et il n'a pas à se préoccuper des différentes sources d'information ni de leurs structures. Ce genre de fédération offre ainsi à l'utilisateur la transparence de localisation, de schéma et de langage. Au contraire, les SIF faiblement couplés n'offrent que la transparence de langage. L'utilisateur doit tout de même connaître les schémas des différentes sources d'information (non transparence des schémas).

4.1.1.4. Le modèle de données du SIF

La couche de fédération d'un SIF doit se baser sur un modèle de données spécifique, appelé *modèle de données canonique* ou modèle commun de données. Les systèmes d'information fédérés fortement couplés sont schématisés dans ce modèle. Dans le cas de fédérations faiblement couplées, seul le langage de requêtes est basé sur ce modèle ; il existe, cependant, un schéma de dépendance entre les composants de la fédération.

Cependant, le modèle de données limite les genres de composants qui peuvent être intégrés dans une fédération vue l'impossibilité de traduction entre quelques modèles. Dans le tableau 4, nous donnons les incompatibilités qui peuvent surgir entre le modèle de données objet, le modèle de données relationnel, et le modèle de données semi-structuré. Nous remarquons que l'incompatibilité apparaît si des composants semi-structurés sont intégrés dans des données structurées (indiqué par un «-» dans le tableau 4). D'autres problèmes se produisent également si des modèles sémantiquement plus riches sont intégrés dans un modèle sémantiquement plus faible. Notons que le « + » indique que l'intégration du modèle de données du composant dans le modèle canonique des données est possible sans perte de sémantique. Le « / » indique la possibilité d'une perte potentielle de sémantique. [BUSSE & AL., 99]

		Modèle canonique		
		Orienté objet	**Relationnel**	**Semi-structuré**
Modèle du composant	**Orienté objet**	+	/	/
	Relationnel	+	+	+
	Semi-structuré	-	-	+

Tableau 4 : Compatibilité des modèles de données

4.1.1.5. Paradigme de requête

Les systèmes d'information peuvent être classifiés aussi par le type de requête qu'ils autorisent : les requêtes structurées et les requêtes de recherche d'information (IR : Information Retrieval). Les dernières sont typiquement utilisées par les moteurs de recherche d'information sur le Web, alors que les premières sont typiquement utilisées dans les systèmes de gestion de bases de données.

4.1.1.6. Accès en lecture ou en lecture / écriture

Ce critère fait la distinction entre les systèmes d'information fédérés qui autorisent l'insertion (ou la mise à jour) des données dans le système composant à travers la couche de fédération, et ceux qui ne l'autorisent pas.

4.1.2. Classification des systèmes d'information fédérés

En se basant sur les critères cités ci-dessus, nous distinguons trois types de SIF : les bases de données faiblement couplées, les bases de données fortement couplées et les systèmes de médiation (cf. figure 9). Ces trois types de SIF sont décrits par la suite.

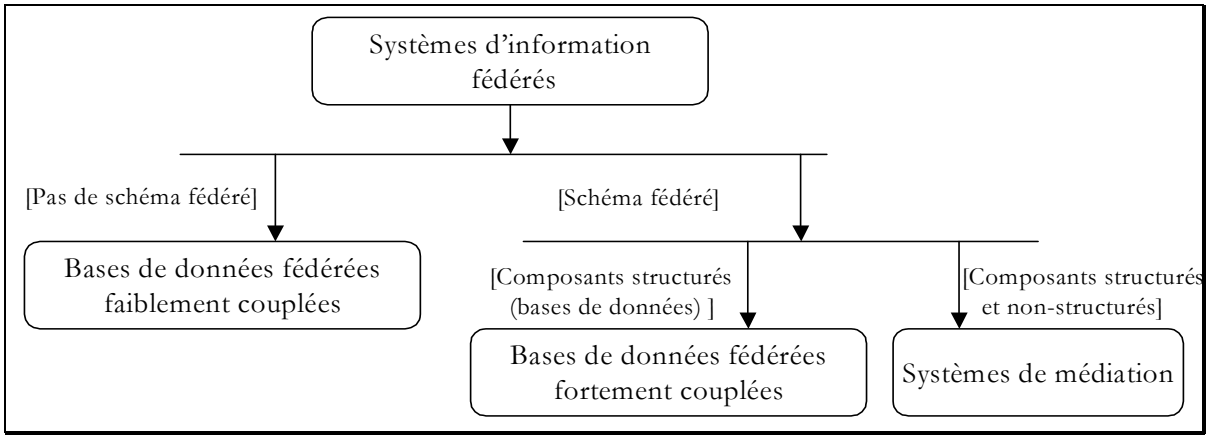


Figure 9 : Classification des approches des systèmes d'information fédérés [BUSSE & AL., 99]

4.1.2.1. Bases de données fédérées faiblement couplées

Cette approche se caractérise par l'absence d'un schéma global. Elle propose de réaliser la fédération non pas à travers une intégration des bases de données mais plutôt via un langage multi-bases de haut niveau (MDBQL : Multi-Database Query Language) permettant de manipuler plusieurs bases simultanément et explicitement [LITWIN & AL., 86]. L'objectif d'un tel langage est de résoudre les hétérogénéités techniques et celles des langages en permettant de formuler des opérations sur plusieurs bases à la fois.

Les systèmes à couplage faible (ou couplage lâche [BOULANGER & AL., 97]) se caractérisent par une grande autonomie et un faible degré d'intégration. Leur autonomie peut atteindre le point où la présence des autres membres peut être ignorée. Le choix d'une faible intégration peut engendrer des problèmes dans la mise en œuvre des applications portant sur les SGBD multiples, puisque ce type de systèmes est caractérisé par l'absence d'un schéma global unique, son contrôle et son maintien sont réservés aux utilisateurs de la fédération.

Les bases de données faiblement couplées présentent deux inconvénients :

- La non transparence de la localisation des données : l'utilisateur doit indiquer explicitement les bases de données qui l'intéressent.
- La résolution des conflits est à la charge de l'utilisateur : ce dernier doit donc avoir assez de renseignements sur la sémantique des bases qu'il utilise pour pouvoir les interroger.

Il est à noter que les systèmes d'information faiblement couplés sont parfois appelés "systèmes multi-bases" [LITWIN & AL., 82] [DADAM, 96] ou classés comme un type particulier de bases de données fédérées. Dans le cadre de ce travail, nous avons distingué les systèmes d'information faiblement couplés des systèmes d'information fortement couplés du fait de leurs conséquences sur l'autonomie et l'évolution des systèmes.

4.1.2.2. Bases de données fédérées fortement couplées

Le fait de choisir une fédération à couplage fort (ou serré) [BOULANGER & AL., 97] implique des restrictions sur l'autonomie des systèmes de gestion de bases de données (SGBD) locaux. Ainsi ces derniers perdent une partie de leur autonomie au profit du système fédéré. En revanche, le système fédéré n'aura pas le droit d'accès aux couches de bases des systèmes locaux. Les tâches de création et de maintien de la fédération sont à la charge des administrateurs de la fédération, qui décident également la construction d'un seul schéma global ou de plusieurs schémas fédérés. En conséquence, le système fédéré est construit par une intégration sélective des systèmes locaux [SHETH & AL., 90]. Un SGBD fédéré fortement couplé offre à l'utilisateur de la fédération une tranparence des conflits des données.

L'architecture de référence d'un SGBD fédéré est structurée en cinq niveaux (cinq types de schémas) définis dans [SHETH & AL., 90] (figure 10).

1. **Le schéma local** correspond au schéma conceptuel d'un SGBD local, et est exprimé dans le modèle de données natif du SGBD local.
2. **Le schéma composant** est obtenu par la traduction du schéma local dans un modèle de données appelé modèle pivot ou modèle de données commun (MDC). La définition du schéma composant dans le modèle pivot a un double intérêt :
   - il est plus simple de détecter les divergences des schémas locaux, lorsque les structures locales sont exprimées dans le même modèle de représentation.
   - les sémantiques oubliées dans les schémas locaux peuvent être ajoutées à leurs schémas composants (enrichissement de schémas).
3. **Le schéma d'export :** la totalité des données locales peut ne pas être disponible pour la fédération. Un schéma d'export représente une portion d'un schéma composant qui est mis à la disposition de la fédération. Il peut également contenir des informations portant sur le contrôle d'accès et leurs utilisations par des usagers spécifiques de la fédération.
4. **Le schéma fédéré** est une intégration multiple des schémas d'export. Il inclut aussi des informations sur la distribution des données. Un concept similaire à celui du schéma fédéré est représenté par les termes schémas d'importation [HEIMBIGNER & AL., 85] ou schéma conceptuel global [LITWIN & AL., 86].
5. **Le schéma externe** correspond à une partie du schéma fédéré. Il reflète une vue destinée à un ou plusieurs utilisateurs (ou groupes d'utilisateurs). Le schéma externe permet de renforcer les contrôles de contraintes d'intégrité et d'accès. La majorité des prototypes des systèmes fédérés existants supporte un seul modèle de données pour tous les schémas externes et une seule interface pour un langage de requêtes.

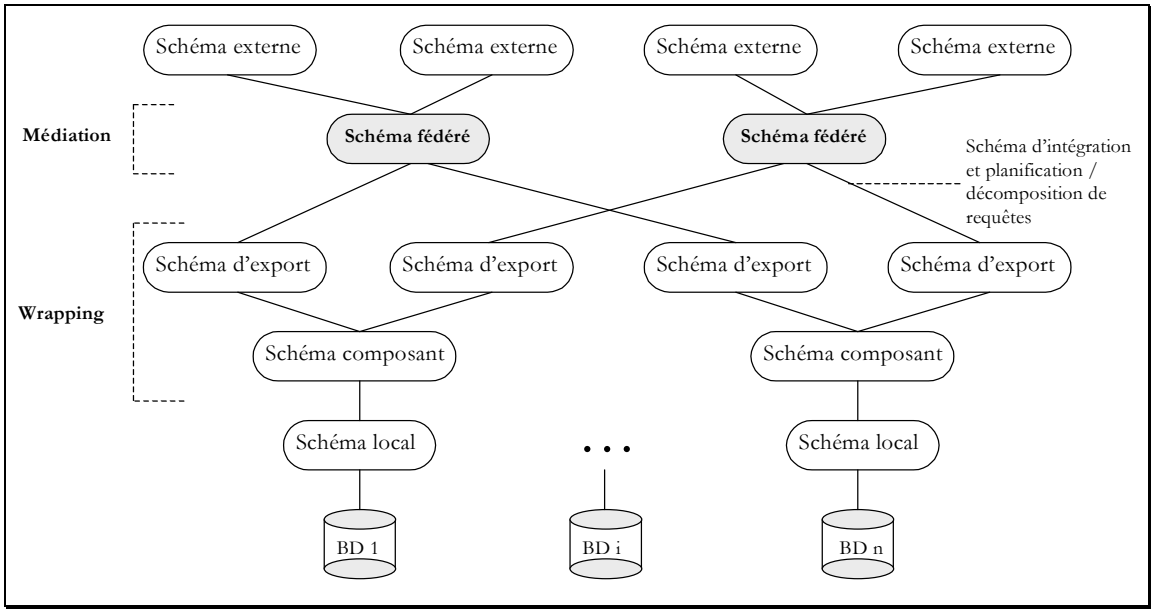


Figure 10 : Architecture à cinq niveaux d'une fédération

4.1.2.3. Systèmes de médiation

Les systèmes de médiation fournissent des composants logiciels pour accéder à l'information là où elle est dans son format d'origine et la transformer dans un format exploitable par les applications. Ces composants sont complémentaires et interopérables, ce qui permet de développer des solutions d'accès et d'intégration d'information pouvant évoluer de manière flexible selon les besoins des entreprises et des organisations. L'une des caractéristiques principales des systèmes de médiation qui peuvent les distinguer des bases de données fédérées est la possibilité de mise à jours des différentes sources de données.

Les systèmes de médiation sont bâtis sur des architectures à deux niveaux et utilisent deux composants spécialisés que sont les adaptateurs et les médiateurs, comme l'illustre la figure 11.

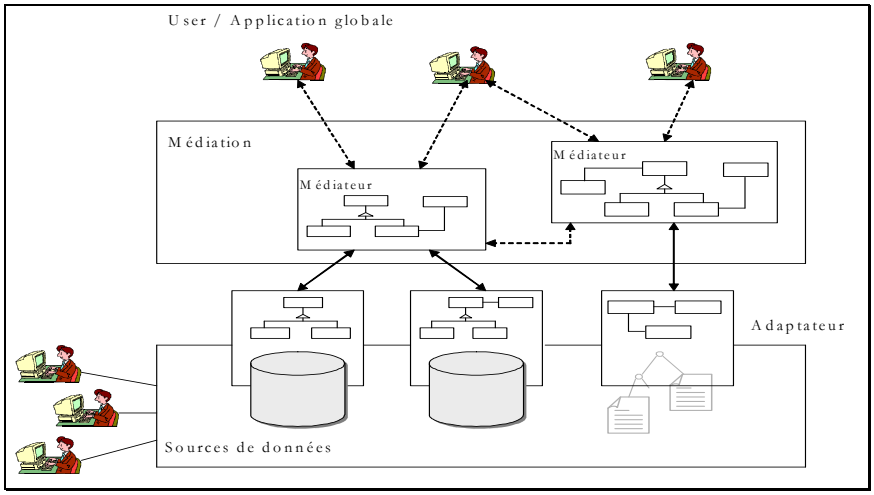


Figure 11 : Architecture générale des systèmes de médiation

L'adaptateur est un composant logiciel fournissant un ensemble de services d'accès uniformes à différents types de données qui peuvent être structurées (relationnel, objet,...), semi structurées (email,...) ou non structurées (image, son,...). Il résout ainsi le problème des conflits syntaxiques des sources des données locales en les présentant dans un même modèle commun mais indépendamment les unes des autres. La plupart des systèmes utilisent le modèle objet standard ODMG [OMG, 91] pour la richesse de ses concepts et son adéquation avec les architectures distribuées. Un adaptateur joue un rôle d'interface entre les médiateurs et les sources de données. Il assure principalement les fonctions suivantes :

- Traduction des requêtes (ou sous-requêtes) du langage commun vers le langage local.
- Mise en forme pour le médiateur des résultats de l'exécution locale des sous-requêtes.

Le médiateur est un composant logiciel fournissant un accès transparent et homogène aux données représentées dans plusieurs adaptateurs. Chaque médiateur possède son propre schéma fédérateur et peut utiliser d'autres médiateurs comme source de données (réseaux de médiateurs). Le médiateur se charge de :

- Cacher la distribution des données.
- Résoudre les conflits schématiques et / ou sémantiques entre les données des différentes sources.
- Traiter les requêtes globales en les décomposant en sous-requêtes élémentaires chacune portant sur les données d'un seul site.

4.1.3. Conclusion

Dans cette section nous avons présenté trois approches pour la coopération des systèmes d'information : les bases de données faiblement couplées, fortement couplées et les systèmes à base de médiation. Nous avons aussi défini des critères pour la classification de ces différentes approches. Dans le tableau suivant nous synthétisons ces trois approches en nous basant sur ces différents critères.

	Bases de données faiblement couplés	**Bases de données fortement couplées**	**Systèmes de médiation**
Types d'hétérogénéité traités	Hétérogénéité technique et des langages	Tous sauf les restrictions de requêtes	Tous
Types de composant	Structuré	Structuré	Tous
Transparence	Langage	Localisation, schéma, et une partie du langage	Localisation, schéma, et langage
Fédération forte vs faible	Faible	Forte	Faible

Langage de requête	Structuré	Structuré	Tous
Type d'accès	Lecture et écriture	Lecture et écriture	Lecture

Tableau 5 : Synthèse sur les approches de coopération

Suite à la description que nous venons de donner des approches de fédération des systèmes d'information, nous pouvons déduire deux stratégies de développement principales :

1. **La stratégie "Top-Down" :** c'est la stratégie de développement des systèmes de médiation. Dans ce cas on part d'un schéma global conçu indépendamment auquel seront reliées les différentes sources de données.
2. **La stratégie "Bottom-up" :** c'est la stratégie de développement des bases de données fortement couplées. Dans ce cas, on part d'un ensemble prédéfini de schémas de sources de données qui doivent être intégrés dans un schéma global.

Ces différentes approches d'intégration de sources de données présentent chacune une alternative différente pour le développement des Systèmes d'Information Coopératifs (SICo). Dans la section suivante nous étudions une autre approche qui se focalise principalement sur les aspects interactifs entre composants, celle des Systèmes Multi-Agents.

4.2. Les Systèmes Multi-Agents

4.2.1. Introduction

Les Systèmes Multi-Agents (SMA) constituent un des trois axes de l'Intelligence Artificielle Distribuée (IAD). A la différence de l'IA classique qui modélise le comportement intelligent d'un seul agent (aspect individuel), l'IAD s'intéresse à des comportements intelligents qui sont le produit de l'activité coopérative de plusieurs agents (aspect collectif). Les deux autres axes de l'IAD sont :

1. **La résolution distribuée des problèmes (RDP)** [Lesser & al., 87] : elle s'intéresse à la manière de diviser un problème particulier sur un ensemble d'entités distribuées et coopératives. Elle s'intéresse aussi à la manière de partager la connaissance du problème et d'en obtenir la solution.
2. **L'intelligence artificielle parallèle (IAP)** [DAVIS, 80] [FEHLING & AL., 83] : elle concerne le développement de langages et d'algorithmes parallèles pour l'IAD. L'IAP vise l'amélioration des performances des systèmes d'IA sans toutefois s'intéresser à la nature du raisonnement ou au comportement intelligent d'un groupe d'agents.

On définit généralement un Système Multi-Agents (SMA) comme un modèle informatique composé d'entités de base, les *agents*. Ces derniers possèdent une autonomie en terme de décisions et d'actions et sont *organisés* en société au sein d'un *environnement* dans lequel et avec lequel ils *interagissent*. Ils doivent, de ce point de vue, être capable de *percevoir*, de *décider*, d'*agir* et de *communiquer* [FERBER, 95] [HERIN & AL., 01]. Les SMA possèdent ainsi l'avantage de faire intervenir des schémas d'interaction sophistiqués. Les types courants d'interaction incluent :

- *La coopération* : travailler ensemble à la résolution d'un but commun.
- *La coordination* : organiser la résolution d'un problème de telle sorte que les interactions nuisibles soient évitées ou que les interactions bénéfiques soient exploitées.
- *La négociation* : parvenir à un accord acceptable pour les parties concernées.

Avant de présenter en détails ces trois schémas d'interaction, nous allons présenter dans ce qui suit une définition du concept d'agent et examiner ensuite ses caractéristiques.

4.2.2. Concept d'agent

Plusieurs définitions ont été attribuées à ce terme, chacune orientée vers le domaine particulier de la recherche. D'après [FERBER, 97] et [WOOLDRIDGE & AL., 95] nous pouvons distinguer deux types de définitions :

Les définitions, dites "faibles", orientées vers une conception très "informaticienne". L'agent est défini comme un objet informatique, dont le comportement peut être décrit par un certain "script". L'agent dispose alors de ses propres moyens de calcul et peut communiquer avec les autres agents.

Les définitions, dites "fortes", où le concept d'agent prend son importance. Dans ce cas Ferber [FERBER, 95] considère l'agent comme une entité (physique ou abstraite) capable d'agir sur elle-même et son environnement, disposant d'une représentation partielle de celui-ci, pouvant communiquer avec d'autres agents et dont le comportement est la conséquence de ses observations, de sa connaissance et des interactions avec les autres agents.

Jennings et al. [JENNINGS & AL., 98] ont proposé la définition suivante pour un agent : c'est un système informatique, *situé* dans un environnement, et qui agit d'une façon *autonome* et *flexible* pour atteindre les objectifs pour lesquels il a été conçu.

A partir de cette définition nous pouvons tirer les caractéristiques suivantes d'un agent : [Anglerot & al., 00]

- *Situé :* l'agent est capable d'agir sur son environnement à partir des entrées qu'il reçoit de ce même environnement. Nous citerons à titre d'exemple les systèmes de contrôle de processus et les systèmes embarqués.
- *Autonomie :* l'agent est capable d'agir sans l'intervention d'un tiers (personne ou agent) et contrôle ses propres actions ainsi que son état interne.
- *Flexible :* l'agent dans ce cas est :
  - capable de répondre à temps : l'agent doit être capable de percevoir son environnement et apporter une réponse dans les temps requis.
  - proactif : l'agent doit exhiber un comportement proactif et opportuniste, tout en étant capable de prendre l'initiative au "bon" moment.
  - social : l'agent doit être capable d'agir avec les autres agents (logiciels et humains) quand la situation l'exige afin de compléter ses tâches ou aider ces agents à accomplir les leurs.

4.2.2.1. Les architectures d'agents

La représentation d'agent en tant que boîte noire (black-box) est considérée comme une architecture minimale, commune à tous les domaines de recherche d'I.A [MULLER, 96] [BRENNER & AL., 98]. Cette architecture est composée de trois modules (figure 12) : le module de perception, le module d'action et celui de traitement intelligent. Ce dernier détermine le comportement d'agent. Il peut ainsi analyser les données reçues et construire la réponse sous la forme d'action ou de message.

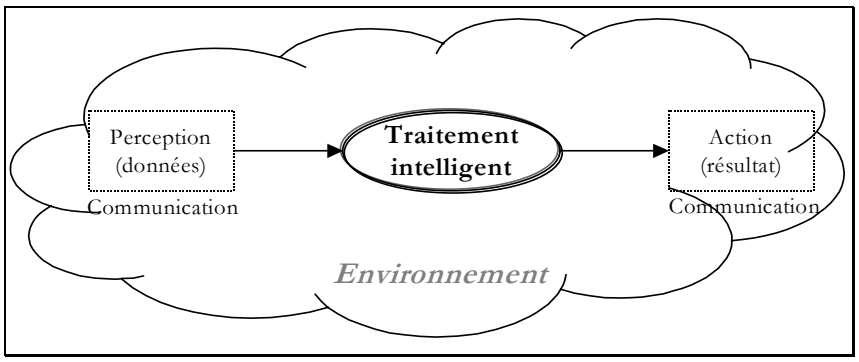


Figure 12 : Architecture d'un Agent

Tous les agents ne disposent pas des mêmes aptitudes. On les classe généralement en deux grandes familles en fonction de leurs capacités de résolution de problèmes : les agents réactifs et les agents cognitifs. Suivant le type d'agent, on parlera de systèmes cognitifs ou de systèmes réactifs.

4.2.2.2. Agents Cognitifs

Les agents cognitifs sont capables d'effectuer seuls des opérations relativement complexes et coopérent les uns avec les autres pour atteindre un but commun (résolution d'un problème, tâche complexe, etc.). Ils possèdent un ensemble de représentations explicites (sur l'environnement, sur les autres agents et sur eux-mêmes) décrits dans une base de connaissances sur laquelle ils peuvent raisonner. La base de connaissances comprend l'ensemble des informations nécessaires à la réalisation de la tâche de l'agent ainsi que la gestion des interactions avec les autres agents. Un système cognitif est composé d'un petit nombre d'agents "intelligents". Dans ce genre de système, les problèmes vont être résolus grâce aux compétences de chaque agent indépendamment des autres et par leur aptitude à coordonner leurs actions ainsi qu'à leur coopération. On parle alors d'agents à forte granularité.

Quatre sous-classes d'agents cognitifs sont distinguées :

- *Les agents intentionnels* sont guidés par leurs buts. Ils possèdent une représentation explicite de leurs plans et de leurs buts.
- *Les agents rationnels* disposent de critères d'évaluation de leurs actions et sélectionnent selon ces critères les meilleurs actions qui leur permettent d'atteindre le but.
- *Les agents coopératifs* planifient leurs actions par coordination et négociation avec les autres agents. En construisant un plan pour atteindre un but, les agents se donnent les moyens d'y parvenir et s'engagent donc à accomplir les actions qui satisfont leurs buts.

↳ Les agents *adaptatifs* sont capables de contrôler leurs aptitudes (à communiquer, comportementales, etc.) selon les agents avec lesquels ils interagissent.

L'architecture d'un agent cognitif peut être divisée de la façon suivante :

- *Le savoir-faire :* c'est une interface permettant la déclaration des connaissances et des compétences de l'agent.
- *La croyance :* dans un univers multi-agents, chaque agent possède des connaissances qui ne sont pas nécessairement objectives. Ce sont ses croyances.
- *L'expertise :* c'est la connaissance sur la résolution de problèmes.
- *La communication :* l'agent doit posséder un protocole de communication lui permettant d'interagir avec les autres agents pour une bonne coopération et une bonne coordination d'actions.

4.2.2.3. Agents réactifs

La deuxième approche, celle des agents réactifs, se base sur le principe suivant : dans un système multi-agents, il n'est pas nécessaire que chaque agent soit individuellement "intelligent" pour parvenir à un comportement global intelligent. En effet, des mécanismes simples de réactions aux événements peuvent faire émerger des comportements correspondant aux objectifs poursuivis. Ils n'ont pas de connaissances explicites ni de l'environnement, ni des autres agents. Ils possèdent une fonction de stimulus/action sur l'environnement qui constitue l'unique protocole de communication avec les autres agents [FERBER, 95]. On parle ainsi d'agents de faible granularité.

4.2.2.4. Conclusion

Nous n'allons pas discuter davantage des différences entre agents cognitifs et agents réactifs. Notons, toutefois, que les agents cognitifs, du fait de leur autonomie et de leur intelligence, peuvent résoudre des problèmes beaucoup plus complexes que les agents réactifs. En outre, les agents cognitifs sont capables d'anticiper sur des actions et de prévoir des événements, contrairement aux agents réactifs qui ne possèdent pas de représentation explicite de leur environnement. Le tableau 6 synthétise les deux approches :

Systèmes d'agents cognitifs	**Systèmes d'agents réactifs**
Représentation explicite de l'environnement	Pas de représentation explicite
Agents complexes	Fonctionnement stimulus/réponse
Petit nombre d'agents	Grand nombre d'agents

Tableau 6 : Synthèse sur les Systèmes d'agents

4.2.3. Société d'agents

Dans le cadre de notre recherche, nous nous intéressons plus particulièrement à des collections d'agents ou sociétés d'agents afin d'identifier leur organisation et leur mode de coopération.

4.2.3.1. Organisation

Deux types d'architectures d'organisation de sociétés d'agents existent [BOND, 90] :

1. **Structure horizontale :** tous les agents sont au même niveau, c'est-à-dire qu'il n'y a pas d'agents maîtres ni esclaves. Ce type d'architecture correspond totalement aux systèmes multi-agents réactifs.
2. **Structure verticale :** les agents sont structurés par niveaux. Au sein d'un même niveau, il existe localement une structure horizontale. Un agent de cette structure a alors le comportement suivant :
   - Il reçoit le problème à résoudre d'un autre agent hiérarchiquement supérieur ;
   - Il décompose ce problème en sous-problèmes qu'il peut résoudre en coopérant avec d'autres agents de son niveau, ou qu'il transmet à des agents de niveaux inférieurs.

4.2.3.2. Coopération

Un SMA se distingue d'une collection d'agents indépendants par le fait que les agents interagissent en vue de réaliser conjointement une tâche ou d'atteindre conjointement un but particulier. En ce qui concerne l'utilisateur d'un SMA, la coopération est simplement le fait que le comportement du groupe permet d'atteindre un certain objectif. En ce sens, on peut définir la coopération, du point de vue de l'utilisateur, comme étant le simple fait que le problème posé est bien résolu collectivement. Du point de vue des agents, la coopération consiste à se mettre d'accord sur les actions que chacun doit effectuer pour que la combinaison de ces actions permette d'atteindre un but commun. Dans une telle dynamique collective, un agent doit disposer, en plus de la connaissance reflétant son degré d'implication, d'un certain nombre de compétences nécessaires pour la coopération. Ainsi il doit pouvoir intégrer des informations provenant d'autres agents, interrompre un plan pour aider d'autres agents et déléguer la tâche qu'il ne sait pas résoudre à un agent dont il connaît les compétences. Ces caractéristiques constituent les qualités essentielles d'un agent coopératif.

Selon [SCHMIDT, 91], trois formes de coopération peuvent être distinguées :

1. **La coopération confrontative** (cf. figure 13), selon laquelle une tâche est exécutée par plusieurs agents de spécialités différentes œuvrant de manière concurrente sur le même ensemble de données. Le résultat est obtenu par "fusion" de ces tâches.

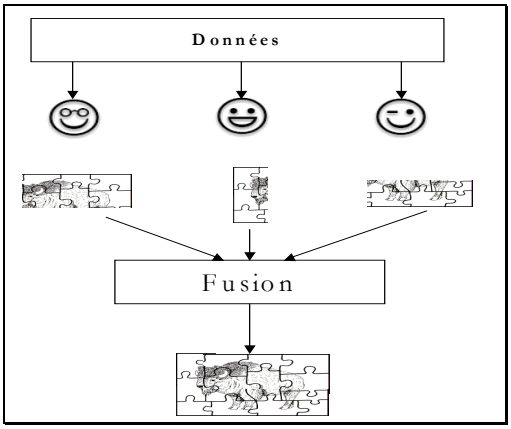


Figure 13 : Coopération confrontative

2. **La coopération augmentative** (cf. figure 14), selon laquelle une tâche est répartie sur une collection d'agents similaires, œuvrant de manière concurrente sur des sous-ensembles disjoints de données. La solution est obtenue sous la forme d'un ensemble de solutions locales.

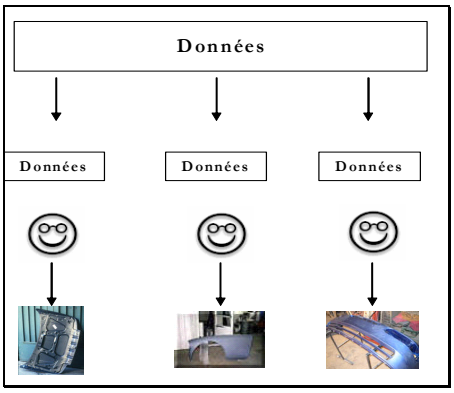


Figure 14 : Coopération augmentative

3. **La coopération intégrative** (cf. figure 15) selon laquelle une tâche est décomposée en sous-tâches accomplies par des agents de spécialités différentes et œuvrant de manière coordonnée. La solution est obtenue au terme de leur exécution.

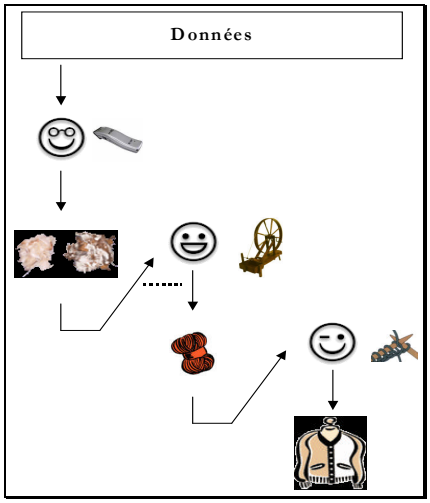


Figure 15 : Coopération intégrative

Pour résoudre un problème, les agents coopératifs ont besoin d'éviter autant que possible les situations conflictuelles. Pour résoudre les conflits, les agents peuvent être amenés à *coordonner* leurs activités et à *négocier* leurs actions pour arriver à une solution.

Nous expliquons dans ce qui suit les aspects de coordination et de négociation.

4.2.3.3. Coordination

Dans le cadre de la coopération, la coordination des actions est la jointure des actions individuelles accomplies par chacun des agents de manière à ce que l'ensemble aboutisse à un résultat cohérent. La coordination peut être nécessaire pour améliorer le fonctionnement global du système. Les agents peuvent travailler sur le même lieu, utiliser les mêmes ressources, ou résoudre des problèmes qui ne sont pas complètement indépendants. De ce fait, ils doivent accomplir, en plus de leurs tâches de base, des tâches de coordination qui ont pour principal atout d'améliorer les tâches productives. La coordination peut être accomplie soit :

- directement par les agents concernés quand les ressources sont relativement rares et qu'elles n'engagent pas un grand nombre d'agents en même temps,
- ou par des agents spécialisés qui recueillent les demandes et fixent les ordres de priorité ou d'autres contraintes.

4.2.3.4. Négociation

Les activités des agents dans un Système Multi-Agent sont souvent interdépendantes et conflictuelles. Dans une situation conflictuelle concernant l'objectif ou la ressource, les agents peuvent chercher eux-mêmes un accord mutuel pendant le processus de la négociation. Cette dernière assure la coopération constructive à l'intérieur d'un groupe des agents indépendants, ayant leurs propres buts [BRENNER & AL., 98]. D'après [BOURON, 92], la négociation définit une stratégie de résolution qui utilise le dialogue pour obtenir un accord concernant les conflits de croyances ou les conflits de buts.

Les conflits de croyances sont les résultats de contradictions entre les croyances des différents agents, possédant des connaissances incomplètes. En ce qui concerne les conflits de buts, ils apparaissent à cause de l'incompatibilité des buts des différents agents. La négociation est caractérisée par [BOUGHZALA, 01] :

- un faible nombre d'agents impliqués dans le processus,
- un protocole minimal d'actions : proposer, évaluer, modifier et accepter ou refuser une solution.

Le processus de négociation ne consiste pas forcément à trouver un compromis mais il peut s'étendre à la modification des croyances d'autres agents pour faire mettre en valeur un point de vue.

D'une manière générale, la structure d'une négociation entre deux agents A et B est la suivante :

➲ A fait une proposition ;

↳ B évalue sa proposition et détermine la satisfaction qui en résulte ;

⇨ Si B est satisfait on s'arrête là

⇨ sinon B élabore une contre-proposition en fonction de ses propres buts et des contraintes ;

➲ Si A prend en compte les arguments de B alors reprendre le schéma dès le debut en inter-changeant A et B, sinon faire intervenir un troisième agent.

4.2.3.5. Communication

La communication entre les agents est indispensable à leur coopération [FERBER, 95]. Elle permet de synchroniser les actions des agents et de résoudre les conflits des ressources par la négociation. D'après [BOURON, 92], la communication définit l'ensemble des processus physiques et psychologiques par lesquels s'effectue l'opération de mise en relation d'un agent émetteur avec un ou plusieurs agents récepteurs, dans l'intention d'atteindre les objectifs prévus. Les processus physiques décrivent les mécanismes d'exécution des actions ; par exemple, l'envoi et la réception de messages. Les processus psychologiques désignent les changements opérés par la communication sur les buts et les croyances des agents.

En général, les actions de communication entre les agents sont considérées comme les actions d'échange d'information, effectuées suivant deux manières : par le partage d'information et par l'envoi de messages.

4.2.3.5.1. Communication par envoi de messages

Ce type de communication permet aux agents d'envoyer leurs messages directement aux destinataires. Il existe trois types de messages : les questions, les réponses et les informations. Au niveau protocolaire, un envoi de message peut être synchrone (un agent émetteur attend la réponse de son récepteur) et asynchrone (un agent émetteur peut agir immédiatement après avoir placé son message dans une file d'attente).

La communication par envoi de messages peut être organisée suivant trois formes différentes (cf. figure 16) :

1. **La communication point à point :** l'agent émetteur connaît l'agent destinataire et lui transmet directement son message. L'agent destinataire est le seul à recevoir le message envoyé. Dans ce mode de communication, l'agent émetteur peut ou non demander un accusé de réception.
2. **La diffusion généralisée :** il s'agit de l'envoi d'un même message à tous les agents du système multi-agent. L'agent émetteur ne connaît pas forcement les destinataires du message envoyé.
3. **La diffusion restreinte :** Il s'agit de l'envoi d'un même message à un certain groupe d'agents du système multi-agent. L'agent émetteur ne connaît pas forcement tous les destinataires, mais il doit être capable de les atteindre en s'appuyant soit sur leurs caractéristiques, soit sur la notion de groupe auquel ils appartiennent.

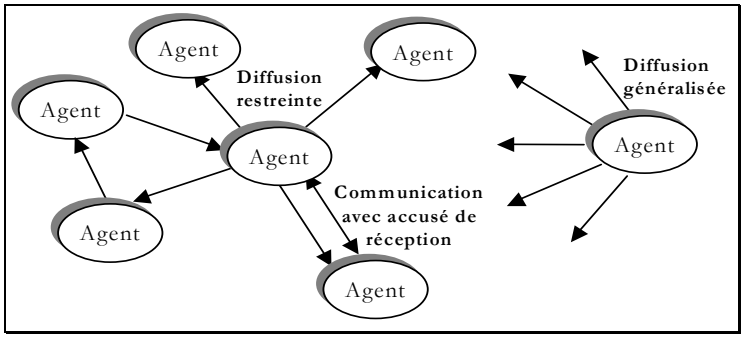


Figure 16 : Communication par envoi de messages

4.2.3.5.2. Communication par partage d'informations

Les agents peuvent communiquer via une structure de données (par exemple, une base de données, un tableau noir (blackboard), etc.), représentant l'état courant du problème. Chaque agent agit sur cette structure afin de résoudre le problème.

L'exemple parfait est l'architecture du tableau noir (cf. figure 17), où les agents (ou les sources de connaissance) sont des modules interdépendants contenant les parties de la connaissance de l'application traitée [BRENNER & AL., 98]. L'information, nécessaire pour résoudre un problème, est mémorisée dans une structure de données (le tableau noir). Cette dernière est généralement divisée en plusieurs parties où chacune n'est accessible qu'aux sources de connaissance (SC) précises. Chaque source de connaissance a le format *condition – action*. La condition décrit les situations dans lesquelles la SC peut être activée. La partie action, spécifie, quant à elle, la contribution de la SC activée.

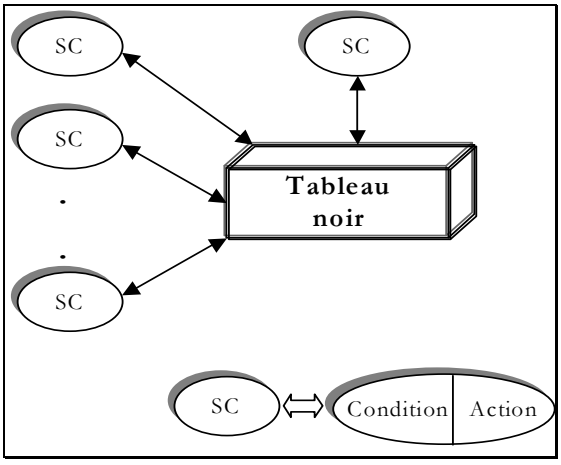


Figure 17 : Communication par partage d'informations

4.2.4. Conclusion sur les agents

Les SMA ont permis d'étudier et d'apporter différents paradigmes de coopération qui sont utilisés ou qui servent de base à la coopération des systèmes d'information.

Cependant, les approches dans le domaine des SMA ont pour principale caractéristique d'être intrusive, c'est-à-dire qu'elles ne font aucune (ou peu) hypothèse sur les agents : pour la plupart des systèmes, les agents sont développés (ou doivent être modifiés) pour s'adapter à

l'environnement dans lequel ils évoluent, particulièrement aux protocoles mis en place. Il résulte de ces remarques que l'organisation des systèmes est basé soit sur la capacité des agents eux-mêmes, soit sur la capacité des agents intermédiaires à « raisonner » pour s'adapter aux demandes de services des uns et des autres [NODINE & AL., 00].

Pour modéliser la coopération entre les agents, le concepteur se trouve face à une alternative :

1. **Prévoir toutes les interactions possibles :** en SMA, le concepteur doit écrire les agents, leurs connaissances, les interactions possibles. Il faut par conséquent, définir des plans permettant de répondre à un ensemble de situations. Le concepteur rencontre des problèmes quand il y a beaucoup d'agents car il doit gérer un grand nombre de données et de paramètres. De plus, dans le cas d'un ajout ou d'un retrait d'agent, il doit reprendre la majorité de son analyse et de son travail de programmation.
2. **Ne pas figer les interactions possibles :** dans les SMA, cela revient à programmer les agents de façon indépendante et doter les systèmes de capacités à traiter les interactions durant l'exécution. Toute la difficulté de ce type d'application réside dans la programmation de tel agent, suffisamment "intelligent" pour résoudre les requêtes et tous les problèmes qui en découlent (coopération, coordination, négociation,...).

5. Résumé du chapitre

Dans ce chapitre, nous avons présenté des solutions qui peuvent être utilisées pour capitaliser et réutiliser des connaissnaces liées aux Systèmes d'Information Coopératifs (SICo). Les patrons de conception [GAMMA & AL., 95] ainsi que les patrons architecturaux [BUSCHMANN & AL.,96] présentent, en effet, des éléments de solution pour la modélisation de la coopération entre systèmes d'information. Cependant, aucune de ces deux techniques ne propose des solutions directes pour l'ingénierie des SICo.

Les patrons architecturaux, connus aussi sous le nom de styles d'architecture [SHAW & AL., 96] ont été largement réutilisés par les techniques de coopération. Dans la pratique, tout SICo repose sur un style d'architecture ou, ce qui est souvent le cas, une combinaison de plusieurs styles architecturaux. Dans la dernière partie de ce chapitre, nous avons abordé les techniques de coopération que nous avons classé en deux catégories selon le domaine de recherche: les Systèmes d'Information Fédérés (S.I.F) et les Systèmes Multi-Agents (SMA).

Les différents travaux étudiés dans ce chapitre ont permis de recenser les techniques et les architectures de coopération existantes. Cette étude nous permettra par la suite de proposer des solutions conceptuelles qui répondent aux besoins spécifiques du domaine de SICo. Dans le chapitre suivant nous abordons un autre point primordial dans l'ingénierie des systèmes d'information : les langages de modélisation permettant de modéliser des architectures complexes telles que les architectures logicielles coopératives.

CHAPITRE III

LANGAGES DE MODELISATION DE LA COOPERATION

1. Introduction générale

Etant donnée l'augmentation de la taille et de la complexité des logiciels, il devient urgent de modifier nos habitudes de conception et d'analyse des produits logiciels. L'idée est donc de fournir des outils permettant une représentation rigoureuse des applications que l'on souhaite produire.

La modélisation de systèmes complexes fait apparaître de nouveaux problèmes : comment concevoir des architectures complexes ? Comment spécifier en terme de protocoles de communications la coopération entre les différents éléments d'un logiciel ou d'une architecture ? D'autre part, il faut tenir compte des tendances actuelles qui visent à favoriser la coopération entre les applications. Mais, pour cela, il faut des outils permettant de spécifier la connexion entre les composants. Il faut également des outils pour tester la validité des applications et éventuellement évaluer le niveau de performance. Le concept d'architecture logicielle a été créé pour combler cette lacune.

Le rôle de l'architecture logicielle est d'offrir au programmeur des primitives de haut niveau pour faciliter la conception d'architectures. Il est donc important d'identifier des composants et des connecteurs génériques utilisés à de nombreuses reprises pour les développer une fois pour toute et les fournir à l'ensemble des concepteurs. De plus, le choix de la bonne architecture est souvent crucial pour le succès du projet, et une erreur d'architecture peut se révéler coûteuse. Une bonne conception architecturale permet, au contraire, des diagnostics précoces et facilite les éventuelles modifications. Aussi, la compréhension détaillée des architectures logicielles permet aux ingénieurs de justifier plus facilement leurs choix. La représentation de l'architecture sous forme graphique est plus compréhensible et elle permet également un transfert plus rapide des compétences lorsqu'un membre de l'équipe quitte le projet. Le nouvel arrivant aura à sa disposition une vue immédiate d'un projet qui lui est encore inconnu.

L'architecture logicielle est un concept de haut niveau : il s'agit d'atteindre un certain niveau d'abstraction pour décrire le comportement d'une application. Comme nous le verrons plus en détail dans la suite, l'architecture logicielle se base sur la notion de composants et de connecteurs. Il est intéressant alors de proposer au concepteur des techniques et langages de modélisation permettant d'exprimer les propriétés des systèmes composants et de leurs interactions ainsi que de capitaliser et réutiliser des modèles d'architectures logicielles adaptés aux SICo. Ce chapitre, traite des différentes techniques d'expression des architectures logicielles. Ces techniques sont classifiées en trois catégories : les langages d'interconnexion, les langages de description d'architecture et les langages de modélisation orientés objet.

De nombreuses recherches [ISSARNY & AL., 98] [ALLEN & AL., 98] [OREIZY & AL., 98] ont permis d'introduire une dimension nouvelle à la notion d'architecture. La définition généralement admise est qu'une *architecture spécifie les modules du système (appelés composants du système) et l'interaction entre ces composants afin de satisfaire les besoins d'un système* [LUCKHAM & AL., 95]. Cette notion est essentielle puisqu'elle offre une vision globale de l'état et du comportement de l'application et facilite, par la même, sa construction et son administration. Plus particulièrement, elle permet l'évolution de l'application et la définition de contraintes sur le système. En outre, la structure à un haut niveau d'abstraction, exposant un ensemble de composants interagissant, permet la spécification de propriétés importantes du système (protocoles d'interaction, localisation de données).

Des notations formelles pour représenter et analyser ces concepts architecturaux ont été introduites. Elles ont abouti à l'élaboration des langages exprimant les interconnexions entre les différents composants (cf. section 2), les langages de configuration (cf. section 3) et des langages de description d'architectures (cf. section 4).

2. Langages d'interconnexion

Comme la taille et la complexité des systèmes logiciels sont croissantes, l'intégration de sous-systèmes développés de manière indépendante devient de plus en plus difficile. Dans les années 1970, les intégrations manuelles de sous-systèmes ont abouti à l'apparition des langages d'interconnexion de modules (MIL : Module Interconnection Language). Le premier MIL, MIL75, fut décrit par DeRemer et Kron [DEREMER & AL., 76] qui ont discuté avec des développeurs et des intégrateurs à propos des applications complexes où un MIL est requis pour exprimer les relations inter-modules. Les MIL fournissent des expressions formelles pour identifier les modules du logiciel et pour spécifier les interconnexions nécessaires pour assembler le programme complet.

Un Langage d'Interconnexion de Modules identifie les différents modules du système et les états selon lesquels ils peuvent évoluer. Une spécification MIL peut être utilisée pour :

- Renforcer l'intégrité du système et la compatibilité entre les sous-modules.
- Supporter des modifications incrémentales. Les modules peuvent être compilés de manière indépendante et une recompilation du système entier n'est pas nécessaire.
- Renforcer le contrôle des versions. Différentes versions d'un module peuvent être identifiées et utilisées lors de la construction du système. Cette idée a été généralisée pour permettre de définir différentes versions de sous-systèmes en termes de versions de modules.

Un MIL se charge d'identifier des chemins indépendants de communication entre modules. Il peut vérifier la cohérence de ces chemins de communications, mais comme les MIL sont indépendants du langage choisi pour réaliser l'implantation des modules, cette vérification se limitera à l'aspect syntaxique (et non sémantique). Récemment, les MIL ont été étendus pour les notions de protocoles de communication et pour construire la définition des propriétés sémantiques des fonctions. Ces MIL sont donc devenus des ADL (Architecture Description Language) que nous allons étudier en détail par la suite.

Bien que les MIL courants basés sur la définition/utilisation aient de nombreux avantages, ils ont aussi des inconvénients. Dans [ALLEN & AL., 94], Allen et Garlen soulèvent un problème significatif : les MIL ne font pas la distinction entre "implémentation" et "interaction" entre les modules. L'implémentation sert à comprendre comment un module est construit pour fournir des services aux autres. L'interaction, quant à elle, permet de décrire les relations architecturales comme la nature de la communication entre les composants. Alors que la relation d'implémentation est concernée par la façon dont un composant réalise son exécution, la relation d'interaction est utilisée pour comprendre comment cette exécution est combinée avec les autres composants.

Un autre problème des MIL est qu'ils ne permettent qu'un seul type d'interconnexion (l'appel de procédure) entre les composants. Pour prendre en compte la multiplicité des interactions possibles, les ADL de leur côté donnent autant d'importance aux connecteurs qu'aux composants et permettent d'en définir différentes sortes (pipe, appel de procédure, client / serveur,...).

En effet, les ADL peuvent être vus comme une extension des MIL au sens où les ADL apportent des informations supplémentaires aux MIL, notamment en terme de protocoles de communication et de comportement du système. Les ADL sont donc aussi utilisés pour définir la structure d'un système et les recherches qui sont menées dans ce domaine font penser que l'usage des MIL sera remplacé à terme par celui des ADL.

3. Langages de configuration

Les langages de configuration fournissent un modèle de notations pour spécifier des configurations d'application. Une configuration est la description de l'ensemble des composants logiciels nécessaires pour le fonctionnement d'une application ainsi que celles de leurs communications et de leur contrôle. La principale différence entre les langages de configuration et les Langages d'Interconnexions de Modules (MIL) [BELLISSARD, 97] est qu'un composant est considéré comme une entité instanciable : la description d'un composant au niveau du langage permet de créer de multiples instances d'un composant lors de l'exécution. La configuration contient, en plus, la spécification du schéma d'instanciation des composants et de leurs emplacements sur les différents sites du système. Les langages de configuration introduisent aussi une certaine structuration générique des modules logiciels de l'application accentuant ainsi leur intérêt pour la définition d'architecture réutilisable.

Les deux langages de configuration les plus connus sont Darwin [ALLEN & AL., 98] et Olan [BELLISSARD, 97]. Dans ces langages, la description de l'architecture vise à produire une image exécutable de l'application, qui est déployée sur différents sites répartis. Ils ont pour cela introduit des éléments dynamiques, permettant de rester plus proche de l'exécution de l'application. Pour cette raison, ces langages et leurs compilateurs associés permettent de générer une image exécutable de l'application mise à la disposition de l'utilisateur.

Cependant, il n'est pas possible de configurer la communication entre composants en terme de mécanismes ou de schémas d'exécution depuis le langage. Il faut toujours effectuer un travail au sein du code des composants. Par exemple dans Darwin, la distinction entre appel synchrone ou appel asynchrone se fait dans le code des composants, aucune information n'étant abstraite dans le langage de configuration. Cette limitation induit un moindre degré de réutilisation des composants et une impossibilité de vérifier avant le déploiement de l'application la validité de l'architecture.

Dans le cadre de cette thèse nous nous intéressons aux langages de description d'architectures qui tentent de combler ces inconvénients. Leur particularité est qu'ils fournissent aux concepteurs un ensemble d'abstractions qui correspondent à celles les plus fréquemment utilisées en génie logiciel. Ces langages portent un effort particulier sur la spécification des interconnexions entre composants. C'est pourquoi le concept de *connecteur* est introduit. La section suivante est consacrée à la présentation de différents langages de description d'architectures.

4. Langages de Description d'Architectures (ADL)

4.1. Concepts de base

L'intégration d'entités logicielles requiert un langage commun indépendant des langages de programmation. Ces langages fournissent une syntaxe concrète pour la caractérisation d'architectures logicielles et permettent de fournir une vision de l'architecture en terme d'entités logicielles nécessaires au fonctionnement de l'application, et une description des interconnexions entre ces différentes entités. L'intégration de ces entités logicielles requiert un langage commun, indépendant des langages de programmation. Sans un tel langage, le développeur d'un système coopératif doit rendre lui-même les entités logicielles inter-opérables au sein d'un environnement hétérogène. Les Langages de Description d'Architectures (ADL : Architectural Description Languages) permettent de spécifier des architectures d'applications en offrant un moyen pratique et abstrait pour une description compréhensible d'un système complexe.

Une autre particularité des ADL est de permettre la description formelle du comportement de l'application à travers ses composants et connecteurs. Pour réaliser ces descriptions formelles, les ADL se sont basés sur différents formalismes : *CHAM* [INVERADI & AL., 95], *CSP* [ALLEN & AL., 97] [HOARE, 78], *LOTOS* [HEISEL & AL., 97], *poset* (partially ordered event set) [LUCKHAM & AL., 95] [PRATT, 86], *StateChart* de Harel [HAREL, 87] [NG & AL., 96], *Z* [SPIVEY, 89]. Les ADL disposent généralement d'outils associés qui exploitent ces descriptions formelles pour réaliser des analyses architecturales, en faisant par exemple des simulations. Ceci est très précieux, car les erreurs peuvent être corrigées à un stade où cela a peu de conséquences.

Les travaux sur les ADL ont été un axe de recherche important et de nombreuses propositions ont rapidement émergé [GARLAN & AL., 94] [MEDVIDOVIC & AL., 00]. Chacun de ces langages fournit une manière de décrire l'architecture d'un logiciel selon son utilisation finale. Certains ADL ont le mérite d'aider à la vérification formelle de propriétés, de permettre la simulation d'une architecture, ou alors d'aider de manière directe à la mise en œuvre et l'exécution du programme.

Généralement les ADL permettent de décrire une architecture par une *configuration* à base de *composants* et de *connecteurs*.

Les composants

Un composant est une unité de calcul ou de stockage. Il peut être simple ou composite. Dans un composant, deux parties sont définies : son interface et son implantation. Une description explicite de son *interface* est indispensable pour décrire les fonctionnalités fournies et nécessaires par le composant. La seconde partie, l'*implantation*, permet de décrire le fonctionnement interne du composant.

Les connecteurs

Dans les ADL, les connecteurs sont des blocs de construction utilisés pour modéliser l'interaction entre composants ainsi que les règles qui gouvernent cette interaction. Un des apports des ADL est d'abstraire le concept de connecteur au même niveau que celui des composants. Il a ainsi été possible de formaliser de manière précise un ensemble de protocoles bien connus comme les pipelines, le protocole client-serveur, etc. [MEDVIDOVIC & AL., 00]

La configuration

Les composants et les connecteurs sont assemblés à partir de leurs interfaces (ports et rôles) pour former une configuration particulière. Une configuration est un agencement, une topologie ; en d'autres termes, il s'agit d'un graphe de composants et de connecteurs qui décrit une structure architecturale pour déterminer si les composants et les connecteurs sont composés correctement.

Dans [MEDVIDOVIC & AL., 00], Medvidovic et Taylor ont proposé un ensemble de caractéristiques globales pour les concepts de composant, connecteur et configuration (cf. section 4.2 et section 4.3). Ces caractéristiques ont été définies dans un framework fournissant un cadre commun favorisant l'évaluation des ADL ainsi que la comparaison entre les différents ADL.

Un quatrième concept est toutefois abordé dans [MEDVIDOVIC & AL., 00] , il s'agit des outils associés aux ADL. Sans en donner tous les détails, nous pouvons indiquer que leurs caractéristiques permettent aux concepteurs d'applications de travailler sur la base des trois concepts précédents. Les outils fournissent donc des aides à la conception, des vues multiples de l'architecture, et des fonctionnalités d'analyse, de raffinement ou de compilation.

4.2. Caractéristiques des composants et des connecteurs

La spécification exhaustive d'un composant ou d'un connecteur dépend de six caractéristiques :

- l'interface
- le type
- la sémantique
- les contraintes

- la maintenance évolutive
- les propriétés non fonctionnelles.

Dans la suite, nous détaillons chacune de ses caractéristiques en précisant à chaque fois leur signification dans le cas du composant ou du connecteur.

L'interface

L'interface consiste en un ensemble de points d'interactions entre le composant / connecteur et le monde extérieur qui permettent l'invocation des services. Un composant / connecteur ne peut être accessible qu'à travers son ou ses interfaces

Composant	**Connecteur**
Services fonctionnels du composant : les services offerts et requis par le celui-ci. Interface appelée aussi **Port**.	Mécanismes de connexion entre composants. Interface appelée aussi **rôle**.

Le type

La notion de type offre un moyen pour décrire explicitement les propriétés communes à un ensemble d'instances d'un même type de composant / connecteur. Ce concept représente des abstractions des fonctionnalités en vue de leur réutilisation soit dans une même architecture, soit dans des architectures différentes.

Composant	**Connecteur**
Abstraction des fonctionnalités fournies par le composant.	Abstraction des mécanismes de communication, de coordination ou de médiation entre composants.

La sémantique

La sémantique du composant / connecteur permet de disposer d'un modèle plus complet et plus abstrait de son comportement et permet de spécificier les aspects dynamiques et les contraintes liées à l'architecture. Le modèle sémantique assure la cohérence entre les différents niveaux d'abstraction de l'architecture.

Composant	**Connecteur**
Abstraction des fonctionnalités du composant (et qui seront utilisées par l'application).	Abstraction des protocoles d'interaction.

Les contraintes

Une contrainte est une propriété qui se doit d'être vérifiée sur l'une ou toutes les parties d'un système. Sa violation conduit à une incohérence du système. Les contraintes peuvent être définies dans le langage de l'ADL en question ou bien dans un autre langage de contraintes.

Composant	Connecteur
Limites d'utilisation du composant et des dépendances intra composants.	Limites d'utilisation du protocole d'interaction mis en œuvre par le connecteur et des dépendances intra connecteurs.

L'évolution

Un ADL doit permettre l'évolution des éléments de l'architecture, donc permettre la modification de leurs propriétés sans perturber leur utilisation ni leur intégration au sein des applications. En général, l'évolution est assurée par des techniques de sous-typage ou de raffinement.

Composant	Connecteur
Evolution de l'interface, du comportement, de l'implantation du composant.	Evolution de l'interface et du comportement du connecteur. Permet notamment de faire évoluer les protocoles d'interaction.

Propriétés non fonctionnelles

Les propriétés non fonctionnelles regroupent les aspectes non fonctionnels liés à la sécurité, la performance, la portabilité, etc. Elles ont le même sens pour les composants et les connecteurs.

4.3. Caractéristiques de la configuration

Un ADL doit fournir les possibilités de configuration suivantes :

- spécification compréhensible
- composition hiérarchique
- raffinement et la traçabilité
- hétérogénéité
- passage à l'échelle
- évolution de la configuration
- dynamique d'une application
- contraintes
- propriétés non fonctionnelles.

Spécification compréhensible

Dans un ADL, la syntaxe du modèle topologique doit être simple et intuitive. Dans ce cas, la lecture seule de la configuration doit suffire à la compréhension du système sans entrer dans les détails des composants et des connecteurs.

Composition hiérarchique

Un ADL doit supporter le fait qu'une architecture entière peut être représentée comme un seul composant dans une autre architecture plus large. Ainsi, il est crucial qu'un ADL supporte la propriété de composition hiérarchique. On parle de composition hiérarchique, dans laquelle le composant primitif est une unité non décomposable, et le composant composite est composé de composants (composites ou primitifs).

Raffinement et traçabilité

Un ADL doit offrir la possibilité de raffiner une configuration à chaque étape du processus de développement. La traçabilité permet de garder la trace des changements successifs entre les différents niveaux d'abstraction.

Hétérogénéité

L'un des buts des architectures est de faciliter le développement de grands systèmes avec des composants et des connecteurs ayant différents degrés de granularités, implémentés par différents développeurs dans des langages (de programmation ou de modélisation) différents et sur des systèmes d'exploitation différents. Un ADL doit alors permettre de réutiliser l'existant et de spécifier une architecture indépendamment des supports techniques utilisés.

Passage à l'échelle

Un ADL doit permettre de réaliser des applications complexes et dont la taille peut devenir importante.

Evolution

Un ADL doit permettre l'évolution de la configuration pour qu'elle puisse prendre en compte des nouvelles fonctionnalités. Cela se traduit essentiellement par la possibilité d'ajouter, de retirer ou de remplacer des composants ou des connecteurs.

Dynamique d'une application

La dynamique de l'application se traduit par les changements qu'elle subit lors de son exécution tels la création ou la suppression d'instances de composants, au contraire de l'évolution où les changements sont effectués en atelier (off-line).

Contraintes

Les contraintes qui décrivent les dépendances entre les composants et les connecteurs dans une configuration sont aussi importantes que celles spécifiées dans les composants et les connecteurs. Elles complètent les contraintes définies pour les composants et les connecteurs.

Le concepteur spécifie ces contraintes, cela revient à définir des contraintes globales, c'est-à-dire des contraintes qui s'appliquent à tous les éléments d'une application.

Propriétés non fonctionnelles

Les propriétés non fonctionnelles qui ne concernent ni les connecteurs ni les composants doivent être spécifiées au niveau de la configuration. Par conséquent, un ADL doit pouvoir définir les contraintes liées à l'environnement d'exécution au niveau de la configuration.

4.4. Conclusion

Dans cette partie, nous avons défini les différents concepts que nous retrouvons dans les différents langages de description d'architectures. Cette étude qui a été menée par Medvidovic et Taylor [MEDVIDOVIC & AL., 00], consiste d'une part à définir les éléments communs d'un grand nombre d'ADL (composant, connecteur, configuration) et, d'autre part, à proposer un ensemble de caractéristiques globales pour chaque élément. Ces caractéristiques sont considérées comme un ensemble de critères que doit prendre en compte un langage pour être considéré comme un ADL, et fournissent un cadre commun favorisant la comparaison entre plusieurs ADL. Nous reprenons ces critères pour évaluer et comparer les ADL que nous présentons par la suite.

5. Quelques ADL existants

5.1. UniCon

UniCon (**Uni**versal **Con**nector support) [SHAW & AL. 96B] développé par G. Zelesnik au département Informatique de l'Université Carnegie Mellon de Pittsburg aux Etats-Unis est un langage de description d'architectures dont les principaux concepts sont les composants et les connecteurs. Une application sera le résultat de l'interconnexion de composants et de connecteurs.

5.1.1. Les composants

Un composant est avant tout une entité d'encapsulation de fonctions d'un module logiciel ou de données. Un composant est caractérisé par :

- une interface définissant les services requis et fournis par le composant,
- une partie décrivant sa réalisation, i.e. le lien entre l'interface et le code logiciel,
- un type correspondant au mécanisme ou au mode de mise en œuvre du composant.

La notion de type dans UniCon est importante parce qu'il permet aux outils de génération de l'application de produire automatiquement du code utilisant le mécanisme qu'il représente. Par exemple, un composant peut être de type *Process* si tout ce qu'il encapsule est contenu dans un processus particulier à l'exécution. Il sera de type *Module* si son implémentation correspond à des ensembles de fonctions contenues dans une bibliothèque de fonctions. Il sera de type *SeqFile* s'il représente un fichier, etc. De plus, le type d'un composant impose un ensemble d'opérations possibles pour ce type de composant. Ces opérations sont définies par

des types de *Players* ; par exemple, le type *SeqFile* représentant un accès séquentiel n'est associé qu'au type de Player Readnext et WriteNext correspondant respectivement à la lecture ou à l'écriture dans un fichier. Ainsi un type de Player permet de spécifier un mécanisme d'accès à un service fourni ou requis du composant.

Le tableau suivant (tableau 7) [BELLISSARD, 97] montre l'ensemble des types de composants existant dans UniCon ainsi que les types de Players acceptés par chacun d'eux.

Type de composant	**Signification du type**	**Types de Players autorisés**
Module	Correspond à la notion de bibliothèque logicielle ou binaire.	RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, PLBundle, ReadFile, WriteFile.
Computation	Correspond à la notion de bibliothèque de fonctions. Elle se distingue de la précédente par l'absence de liens avec les fichiers.	RoutineDef, RoutineCall, GlobalDataUse, PLBundle.
SharedData	Zone de données partagées. Le mécanisme de partage est défini par les connecteurs d'accès.	GlobalDataDef, GlobalDataUse, PLBundle.
SeqFile	Fichiers à accès séquentiel.	ReadNext, WriteNext.
Filter	Flux de données.	StreamIn, StreamOut.
Process	Processus indépendant.	RPCDef, RPCCall.
SchedProcess	Processus avec contrôle temps réel.	RPCDef, RPCCall, RTLoad.
General	Tout	Tous les types de Players sont admis.

Tableau 7 : Types de composants dans UniCon

La réalisation d'un composant peut être de deux types : *Primitive* ou *Composite*. La première, fait directement référence à du code source ou binaire. La deuxième, à savoir les composants composites, ont une implémentation qui consiste en une liste de composants et de connecteurs, des instructions de composition et des connexions avec le composite englobant. Un composant composite permet ainsi de structurer l'application dans le sens où il présente la hiérarchisation du système en termes d'entités plus petites, sous-composants primitifs ou composites.

La déclaration syntaxique d'un composant se fait de la manière suivante :

COMPONENT *Name*
INTERFACE IS
TYPE *ComponentType*
< Liste des propriétés du composant >
< Liste des Players du composant >
End INTERFACE
IMPLEMENTATION IS
< Liste de propriétés >
Implémentation Primitive | Implémentation Composite
END IMPLEMENTATION
END *Name*

5.1.2. Les connecteurs

Le connecteur a un rôle majeur dans UniCon car il contient les informations concernant les règles d'interconnexion de composants. La définition d'un connecteur ressemble à celle d'un composant. Chaque connecteur est spécifié par un protocole (l'équivalent de l'interface d'un composant) qui spécifie les points de branchements autorisés pour les composants. Ces points d'entrées ou de sorties sont définis par des *Rôles* auxquels les *Players* d'un composant peuvent être interconnectés. Un protocole est défini par un type qui définit le moyen d'accès aux informations contenues dans des composants. Par exemple, le type de connecteur *RemoteProcCall* possède deux rôles, le rôle *Definer* (l'appelé) et le rôle *Caller* (l'appelant) qui permettent l'interconnexion des Players *RPCDef* et *RPCCall*. Un Rôle est défini par un nom, un type et des attributs supplémentaires tels qu'une signature, une spécification fonctionnelle ou des contraintes d'utilisation. De plus, le type de Rôle n'autorise que le branchement d'un type de Player, comme le montre le tableau suivant. Par exemple, le rôle *Definer* du connecteur *RemoteProcCall* n'autorise que le branchement du Player *RPCDef*.

Type de connecteur	Type des Rôles et Players autorisés
Pipe	Source (accepte StreamOut de Filter, ReadNext de SeqFile) Sink (accepte StreamIn de Filter, WriteNext de SeqFile)
FileIO	Reader (accepte ReadFile de Module) Readee (accepte ReadNext de SeqFile) Writer (accepte WriteFile de Module) Writee (accepte WriteNext de SeqFile)
ProcedureCall	Definer (accepte RoutineDef de Computation ou Module) Caller (accepte RoutineCall de Computation ou Module)
DataAccess	Definer (accepte GlobalDataDef de SharedData ou Module) Caller (accepte GlobalDataUse de SharedData ou Module)
PLBundler	Participant (accepte PLBundle, RoutineDef, RoutineCall, GlobalDataUse, GlobalDataDef de Computation ou Module)

RTScheduler	Load (accepte RTLoad de SchedProcess)
RemoteProcCall	Definer (accepte RPCDef de Process ou SchedProcess) Caller (accepte RPCCall de Process ou SchedProcess)

Tableau 8 : Types de connecteurs dans UniCon

La version actuelle de UniCon ne permet pas la définition de l'implantation de connecteurs formés de plusieurs connecteurs (connecteur composite). De plus, les éléments faisant partie de l'implantation d'un connecteur primitif sont tous définis et générés par UniCon (mot clé *BUILTIN*). Ainsi, il n'est pas possible, pour l'instant, d'associer au connecteur UniCon une implantation créée sans l'aide de cet ADL. De même, il est impossible de définir de nouveaux types de connecteurs, il faut obligatoirement choisir parmi l'ensemble prédéfini.

La déclaration syntaxique d'un connecteur se fait de la manière suivante:

CONNECTOR *Name*
PROTOCOL IS
TYPE *ConnectorType*
< Liste de propriétés >
ROLE NomRole **IS** *RoleType*
< Liste de Propriétés de définition du rôle tel que le nombre de composants connectés,…>
End ROLE *NomRole*
ROLE …
END ROLE …
End PROTOCOL
IMPLEMENTATION IS
BUILTIN // On ne peut pas redéfinir des implémentations différentes de connecteur
END IMPLEMENTATION

5.1.3. Configuration

Une configuration est la représentation d'une architecture d'une application définie par un composant composite dont l'implantation spécifie les interfaces des composants utilisés dans l'architecture, les connecteurs utilisés et les interactions entre les composants de l'architecture.

L'interface du composant composite existe mais ne spécifie rien. C'est une interface de type général (*General*) utilisée lors de la définition d'une architecture qui permet de définir un nouveau composant correspondant au système.

5.1.4. Avantages

UniCon utilise un système de typage qui s'avère efficace pour trois raisons :

1. tout d'abord, ce système vérifie statiquement la validité des interconnexions et leur conformité au typage des entrées et des sorties des composants et des connecteurs ;
2. le typage impose des restrictions d'accès aux entités du système ;
3. chaque type de player et de rôle correspond à un mécanisme d'accès aux composants et aux connecteurs. Cette caractéristique offre, aux outils de génération, le moyen de produire le code associé.

Même s'il n'est pas inclus dans le code des composants, le mécanisme de communication est un aspect important du langage de description d'architecture. Ainsi, une seule définition d'un connecteur peut gérer des échanges entre plusieurs composants s'ils sont autorisés à interagir à travers ce type de communication.

Dans UniCon, les sources, les objets et les exécutables peuvent être encapsulés dans les composants, ce qui permet d'offrir des mécanismes suffisants pour la conception d'applications. Citons à titre d'exemple les processus ou les filtres qui sont des primitives de construction disponibles, parmi tant d'autres.

UniCon offre la simplicité de la conception des architectures puisqu'il permet de définir les composants et les connecteurs d'une manière uniforme.

Finalement, UniCon est un langage qui est associé à un environnement complet possédant :

- un éditeur graphique pour la définition des composants et des connecteurs, mais aussi pour la description de la structure architecturale sous forme d'un graphe de connexion entre composants et connecteurs
- un analyseur syntaxique pour vérifier la bonne utilisation des éléments du modèle
- un compilateur et un générateur de code (en langage C)

5.1.5. Inconvénients

En dépit des avantages que présente UniCon, ce langage montre tout de même certaines faiblesses que nous énumérons dans ce paragraphe.

Tout d'abord, UniCon fournit une description statique de l'architecture ; aucun ajout ni suppression ne sont autorisés après l'instanciation initiale des composants.

D'autre part, UniCon est un langage semi-formel qui reste par conséquent limité pour l'expression de la sémantique. Ainsi, la signature des services est dépendante du langage de programmation, notamment pour la description des paramètres d'entrée et de sortie d'un service d'un composant. De plus, UniCon est restreint en terme de langages puisque le langage C est actuellement le seul accepté.

Une troisième restriction d'UniCon concerne la limitation des types pour les composants et les connecteurs.

De plus, en dépit de ses avantages, le fort typage du langage présente l'inconvénient de restreindre les concepteurs à l'utilisation d'abstractions par défaut ou implicites ; ceux-ci ne peuvent donc pas effectuer de choix délibéré.

Quant aux éléments composites, le langage est capable de supporter les composants composites mais pas encore les connecteurs composites.

5.2. Rapide

Rapide [LUCKHAM & AL., 95] a pour but initial de vérifier par la simulation la validité d'une architecture logicielle donnée. Une application est construite sous la forme de modules ou composants communiquant par échange de messages ou évènements. Le simulateur associé à Rapide permet ensuite de vérifier la validité de l'architecture.

Les concepts de base du langage Rapide sont les suivants : la notion d'événement, de composant, et d'architecture.

5.2.1. Evénements

Le concept de base de Rapide est l'événement qui est une information transmise entre composants. L'événement permet de construire des expressions appelées *event patterns*. Ces expressions permettent de caractériser les évènements circulant entre les composants. Par exemple, si A est un événement, A > B signifie que B sera envoyé après A. La construction de ces expressions se fait avec l'utilisation d'opérateurs qui définissent les dépendances entre événements. L'ensemble de ces opérateurs est répertorié dans le tableau suivant :

Opérateur	Sémantique
A > B	B est envoyé après A
A -> B	B dépend causalement de A
A || B	A et B ne sont pas causalement dépendants
A~B	A et B sont différents
A and B	A et B sont vérifiés simultanément

Tableau 9 : Les expressions d'événements dans Rapide

5.2.2. Composants

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

1. les **Provides** peuvent être appelés de manière synchrone par les composants,
2. les **Requiers** sont les services que le composant demande de manière synchrone à d'autres composants. Un composant peut donc communiquer de manière synchrone avec un autre composant si leurs services requiers et provides sont connectés,
3. les **Actions** qui correspondent à des appels asynchrones entre composants. Deux types d'actions existent : les actions *in* et *out* qui correspondent respectivement à des événements acceptés ou envoyés par un composant.

L'interface contient également une section de description du comportement (*clause behavior*) du composant. Cette dernière correspond au fonctionnement observable du composant, par

exemple l'ordonnancement des événements ou des appels aux services. C'est grâce à cette description que Rapide est en mesure de simuler le fonctionnement de l'application.

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

Une application est représentée par son architecture. Une architecture consiste en des déclarations d'instances de composants, des connexions entre ces instances et des contraintes sur le comportement de l'architecture. Voici la structure d'une architecture :

```
Architecture Name is
        // Déclarations
        Connections
        // Connexions
        [constraints]
        // Contraintes optionnelles
End Name
```

Les connexions entre les instances de composants sont régies par des règles. Une règle d'interconnexion est composée de deux parties. La première est la partie gauche qui contient une expression d'événements qui doit être vérifiée. La seconde est la partie droite qui contient également une expression d'événements qui doivent être déclenchés après la vérification de l'expression de la partie gauche. L'exemple suivant illustre ces règles :

```
With Client, serveur ;
…
// Déclaration des instances de composants de l'application susceptible d'exister
?s : Client ; // Fait référence à une instance de Client
!r : Serveur ; // Fait référence à toutes les instances de Serveur
?d : Data ; // Fait référence à un bloc de paramètres d'un certain type Data
…
// Une règle d'interconnexion
?s.Send( ?d) => !r.Receive( ?d) ;
// Si un client transmet un événement de type Send avec ce type de paramètres, alors l'événement
est transmis à tous les serveurs de l'application avec ces paramètres.
```

Les parties gauches et droites peuvent être connectées par trois types d'opérateurs :

1. **L'opérateur « to »** connecte deux expressions d'événements simples (ne définissant qu'un événement possible vers un composant). Si la partie gauche est vérifiée alors l'expression de la partie droite permet le déclenchement de l'événement vers l'unique composant désigné par cette expression.

2. **L'opérateur « ‖ > »** connecte deux expressions quelconques. Dès que la partie gauche est vérifiée, tous les événements contenus dans la partie droite sont déclenchés. Ils sont envoyés vers l'ensemble des destinataires désignés dans cette expression. L'ordre d'évaluation de cette règle de connexion est quelconque, c'est-à-dire qu'un déclenchement de cette règle de connexion est indépendante des autres déclenchements antérieurs ou postérieurs. L'ordre d'observation de ces déclenchements n'est pas significatif.
3. **L'opérateur « => »** possède le même rôle que l'opérateur précédent mais ici l'ordre d'évaluation des règles est contrôlé. Un déclenchement de cette règle est causalement dépendant des déclenchements antérieurs de cette règle. Cet opérateur de connexion est appelé opérateur pipe-line.

5.2.3. Avantages

Rapide prend totalement en compte différents modèles d'exécution. Pour cela, il offre, au niveau de l'interface des composants, des appels de services synchrones et asynchrones, et utilise plusieurs opérateurs d'interconnexion.

De plus, Rapide offre le moyen d'éviter l'installation et le déploiement d'une application de taille importante pour réaliser les tests de validité de l'architecture. En effet, sous Rapide, la simulation d'un modèle génère un ensemble d'événements qui apparaissent à l'exécution avec des relations causales et temporelles.

Rapide permet d'exprimer une caractéristique importante et intéressante : la dynamique d'une application, grâce à l'utilisation de règles d'interconnexion déclenchées par le comportement des composants logiciels au moment d'un changement d'état par exemple. Cette dynamique est exprimée sous la forme d'ensembles d'instances de composants ou de règles d'interconnexion évoluant selon le comportement des composants.

5.2.4. Inconvénients

La langage Rapide ne prend pas en compte la notion de connecteur "statique" et ne permet pas ainsi de représenter un connecteur comme une entité de premier degré.

Avec Rapide, il n'est pas possible de reconfigurer une application du fait de l'indisponibilité d'opérateurs de création, de suppression, de migration de composants et de modification d'interconnexions.

De plus, Rapide est un langage qui ne propose pas d'éléments de structuration de l'application.

Enfin, Rapide ayant pour but principal de vérifier la validité des architectures, il ne permet pas de générer des applications.

5.3. Wright

Wright est un langage de description orienté vers la vérification des protocoles entre les composants, plutôt que sur la correction fonctionnelle de l'architecture globale [ALLEN & AL.,

98]. Ce langage n'est pas dédié à la production d'une image exécutable de l'application, il porte effectivement son accent sur la vérification formelle du système.

Wright privilégie les interactions entre les entités d'une architecture et propose d'exprimer des architectures complexes telles que les systèmes à base d'événements, les structures de flots de données et les protocoles client / serveur. Il repose sur quatre concepts qui sont le composant, le connecteur, la configuration et le style.

Pour illustrer le langage Wright, nous allons utiliser une architecture relativement simple (figure 18). C'est un système de type pipe-filtre utilisant trois composants :

- le composant « Eclateur » permettant de récupérer le flot de caractères en entrée et de le diviser en deux flots (en prenant un caractère sur deux) ;
- le composant « Majuscule » met en majuscule les caractères qu'il reçoit ;
- le composant « Assembleur » permettant de fusionner les deux flots. Par exemple, si on avait le mot "composant" à l'entrée de « Eclateur », on aura le mot "CoMpOsAnT" à la sortie de « Assembleur ».

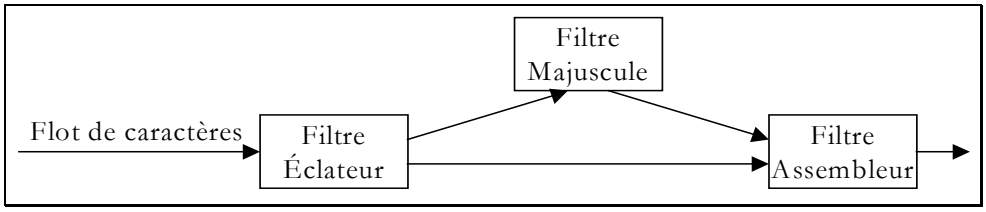


Figure 18 : Exemple d'architecture de type « Pipe & filtre » [ALLEN, 97]

5.3.1. Composants

Un composant en Wright est une entité abstraite et indépendante. La description d'un composant possède deux parties qui sont l'interface (*Interface*) et la partie calcul (*Computation*). Une interface est constituée d'un ensemble de ports. Chaque port représente une interaction dans laquelle le composant participe. A chaque port est associée une description formelle exprimée en langage CSP spécifiant son comportement par rapport à l'environnement. La partie calcul, quant à elle, consiste à décrire le comportement du composant en indiquant comment celui-ci utilise les ports. Elle reprend les interactions décrites au sein des ports et montre les relations entre les ports ainsi que leur lien avec le comportement du composant dans le calcul.

Voici la description du composant « Eclateur » de type « Filtre » dans le langage Wright :

Component Eclateur

Port Input (lire les données jusqu'à la fin des données)

Port Left (sortir les données de manière permanente)

Port Right (sortir les données de manière permanente)

Computation (lire continuellement les données à partir du port Input
puis les sortir à la fois dans le port Left et le port Right)

// Les spécifications du comportement des ports et de la partie calcul sont écrites ici de manière informelle pour plus de compréhension. Elles sont normalement écrites avec le langage CSP.

5.3.2. Connecteurs

Un connecteur détermine les interactions parmi une collection de composants. Il spécifie le patron d'une interaction de manière explicite et abstraite. Ce patron peut être réutilisé dans différentes architectures. Une description Wright d'un connecteur est divisée en un ensemble de *rôles* et une spécification *GLUE*. Chaque rôle indique comment se comporte un composant qui participe à l'interaction. La glue d'un connecteur, quant à elle, décrit la manière avec laquelle les participants (c'est-à-dire les rôles) interagissent entre eux pour former une interaction. Par exemple, la glue d'un connecteur « appel de procédure » indiquera que l'appelant doit initialiser l'appel et que l'appelé doit envoyer une réponse en retour.

Pour illustrer la description d'un connecteur dans le langage Wright, nous allons reprendre le même exemple illustré par la figure 18. Dans cette architecture, les composants de type filtre sont liés entre eux par des connecteurs de type pipe. Dans notre cas, tous les pipes fonctionnent de la même manière et obéissent aux mêmes règles. L'exemple suivant illustre la spécification d'un connecteur de type pipe dans le langage Wright.

Connector Pipe

Role Source (délivrer les données continuellement, signaler la fin et fermer)

Role Sink (lire les données continuellement, fermer avant ou au moment de la signalisation de la fin des données)

Glue (le rôle Sink reçoit les données dans le même ordre que celui utilisé par le rôle Source pour fournir ces données)

5.3.3. Configuration

Pour décrire l'architecture complète d'un système, les instances des composants et des connecteurs d'une description Wright sont combinées dans une configuration. Une configuration est une collection d'instances de composants et d'instances de connecteurs. La configuration du système est décrite en trois parties. La première est la déclaration de l'ensemble composants et de connecteurs utilisés dans l'architecture. Ensuite, la déclaration des instances de composants et des connecteurs. Dans la phase finale, les instances des composants et connecteurs sont combinées pour décrire quels sont les ports de composants attachés aux rôles des connecteurs.

Si on reprend le même exemple illustré par la figure 18, la configuration correspondant à l'architecture du système Capitale est la suivante :

```
Configuration Capitale
        Component Majuscule
        …                                  } déclaration des composants et connecteurs
        Connector Pipe
        …
Instances
        Split : Eclateur
        Upper : Majuscule                  } déclaration des instances de connecteurs et composants
        Merge : Assembleur
        P1, P2, P3 : Pipe
Attachments
        Split.Left as P1.Source
        Upper.Input as P1.Sink
        Split.Right as P2.Source           } définition des liens entre les instances
        Merge.Right as P2.Sink             (mot clé : Attachments)
        Upper.Output as P3.Source
        Merge.Left as P3.Sink
End Capitale
```

Il est à noter que Wright supporte la composition hiérarchique (figure 19). Ainsi, un composant peut être composé d'un ensemble de composants (de même pour un connecteur). Lorsqu'un composant représente un sous-ensemble de l'architecture, ce sous-ensemble est décrit sous forme de configuration dans la partie calcul du composant.

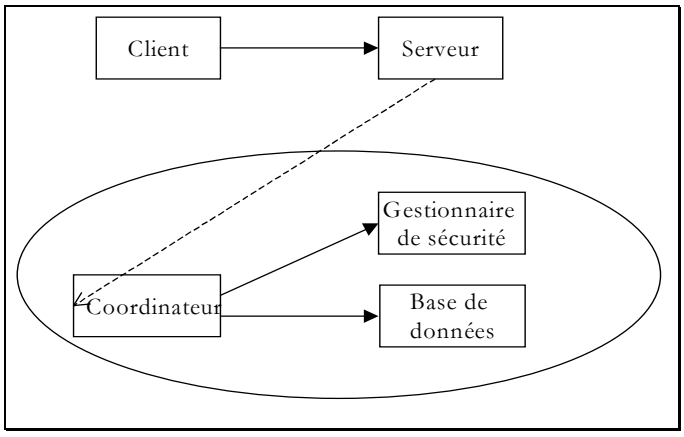


Figure 19 : Exemple d'un serveur composé de plusieurs composants

```
Configuration ArchServer
        Component …
        Connector …
        Component ServeurType
                Port Service …
                Computation
```

```
                    Configuration SécuritéDonnée
                            Component Coordinateur
                            Component GestionSecurité
                            Component BaseDeDonnées
                            Component CSConn
                            …
                    Instances
                            C : Coordinateur
                            Sec : GestionSecurité
                            BD : BaseDeDonnées
                            Sl : CSConn
                    Attachments
                            C.Secure as Sl.Client
                            Sec.Service as Sl.Service
                            …
                    End SécuritéDonnée
                    Bindings
                            C.combine = Service
                    End Bindings
            Instances
            …
            Attachments
            …
End ArchServer
```

5.3.4. Styles

Le langage permet également de définir une famille d'architectures avec la notion de style d'architecture. Il permet de décrire un vocabulaire commun en définissant un ensemble de types de connecteurs et de composants et un ensemble de propriétés et de contraintes partagées par toutes les configurations appartenant à ce style. Les propriétés et les contraintes communes à une architecture peuvent être définies selon trois caractéristiques :

- Les types d'interfaces pour caractériser des familles de rôle et de port.
- Les paramètres pour fournir plus de flexibilité lors de la définition de types. Cela permet de distinguer des traitement différents (pour une même interface) ou de spécifier plusieurs ports pour chaque instance dérivant du même composant.
- Les contraintes qui sont des prédicats logiques de premier ordre qui doivent être satisfaits pour tous les éléments appartenant au style.

L'exemple suivant décrit le style « Pipe & Filtre » avec le langage Wright :

```
Style PipeFilter
        InterfaceType DataInput   (lire les données jusqu'à la fin des données)
        InterfaceType DataOutput   (envoyer les données de manière permanente)
        Connector Pipe
                Role Source = DataOutput
                Role Sink = DataInput
                Glue = … (Sink va recevoir les données dans l'ordre délivré par Source)
        Constraints
                ∀ c : Connectors.Type(c) = Pipe
```

$$\forall c : Components ; p : Port \mid p \in Ports(c).Type(p) = DataInput \vee Type(p) = DataOutput$$

End style

5.3.5. Avantages

L'aspect important apporté par Wright concerne la spécification des connecteurs et des composants en fournissant un langage formel (CSP).

Le second avantage concerne la séparation de la notion de composant et de connecteur. Ainsi, le type de connecteur est considéré comme un patron d'interconnexion et peut être réutilisé plusieurs fois dans une même archichtecture ou dans des architectures différentes.

Enfin, Wright permet de spécifier de manière formelle et abstraite le comportement des composants, des connecteurs et de l'architecture. Il propose en effet des outils formels pour la vérification d'architecture d'applications et un formalisme permettant de vérifier la compatibilité des ports et des rôles du point de vue échange de paramètres et modèle d'exécution de la communication.

5.3.6. Inconvénients

En dépit de ses avantages remarquables, Wright présente quelques limites que nous évoquons dans ce paragraphe.

Le premier inconvénient de Wright est qu'il est difficile à comprendre. En effet, l'outil de spécification qui lui est associé (CSP) n'est pas facile à assimiler pour un non spécialiste.

De plus, Wright n'est pas un langage dédié à la construction d'applications ; il est uniquement consacré à la vérification d'applications. Il ne possède pas d'environnement d'utilisation ou d'exécution comme certains ADL. Il ne possède pas, par exemple, de générateur de code.

Enfin, Wright est sensible à la complexité des applications : pour des architectures simples, le langage se base sur un modèle de processus compréhensible mais qui se complexifie avec les applications complexes.

5.4. Acme

Le projet Acme [GARLAN & AL., 97A] [GARLAN & AL., 97B] propose de combiner les fonctionnalités de multiples ADL. Son but principal est de fournir un langage pivot qui prend en compte les caractéristiques communes de l'ensemble des ADL, qui soit compatible avec leurs terminologies et qui propose un langage permettant d'intégrer facilement de nouveaux ADL.

Il a été développé pour prendre en compte les différents points communs qui existent entre les ADL dans leur manière d'analyser une architecture. En effet, la plupart des langages fournissent des notions similaires comme celles de composant et de connecteur. Acme est

perçu donc comme un langage fédérateur. Les éléments pour la description de la structure de l'architecture (cf. figure 20) sont présentés dans la suite.

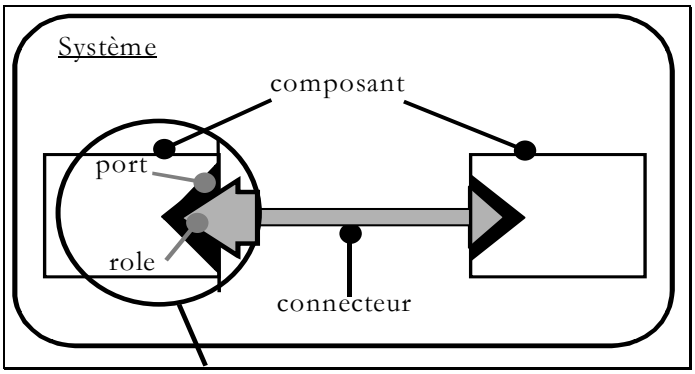


Figure 20 : Eléments d'une description Acme

5.4.1. Composants

Un composant représente un élément primitif de calcul et d'enregistrement de données du système. Des exemples typiques de composants sont les processus, les serveurs, les bases de données, les flots de données, etc. Un composant est spécifié par une interface composée d'un ensemble de *ports*. Chaque port identifie un point d'interaction entre le composant et son environnement. Un port peut présenter une interface aussi simple qu'une signature de procédure ou d'un ensemble de procédures qui doivent être invoquées dans un ordre défini.

5.4.2. Connecteurs

Un connecteur représente une interaction entre composants. Il s'agit d'un médiateur de communication qui coordonne les connexions entre les composants. Les pipes, appels de procédures, etc. sont des exemples de connecteurs simples. Cependant, des interactions plus complexes peuvent être représentées, comme une requête SQL entre une base de données et une application. Un connecteur est spécifié par une interface composée d'un ensemble de *rôles*. Chaque rôle du connecteur décrit les participants de l'interaction. La plupart des connecteurs sont binaires, ils possèdent deux rôles comme appelé et appelant d'une communication RPC ou émetteur et récepteur pour un envoi de messages.

5.4.3. Système

Un système représente la configuration d'une application, c'est-à-dire un assemblage structurel entre des composants et des connecteurs. La structure d'un système est indiquée par un ensemble de composants, un ensemble de connecteurs, et un ensemble d'attachements. Un *attachement* lie un port de composant à un rôle de connecteur. L'exemple suivant décrit l'architecture Client / Serveur en Acme :

```
System Simple_ClientServeur = {
        Component client = { Port send-request }
        Component serveur = { Port receive-request }
        Connector rpc = { Role {caller,callee }
```

Attachments : {
Client.send-request **to** rpc.caller ;
Serveur.reveive-request **to** rpc.callee}

5.4.4. Représentation

La représentation (cf. figure 21) permet à Acme de supporter la description hiérarchique d'une architecture. Ainsi, un composant ou un connecteur peut être décrit d'un niveau général à un niveau plus détaillé et peut donc être raffiné. Chaque nouvelle description (sous-élément) d'un élément est appelée une représentation. Une représentation de composant décrit un système et un ensemble de *Binding*. Chaque *Binding* relie un port interne à un port externe. Ainsi un composant peut être décomposé hiérarchiquement. Les connecteurs peuvent également être décomposés d'une manière semblable.

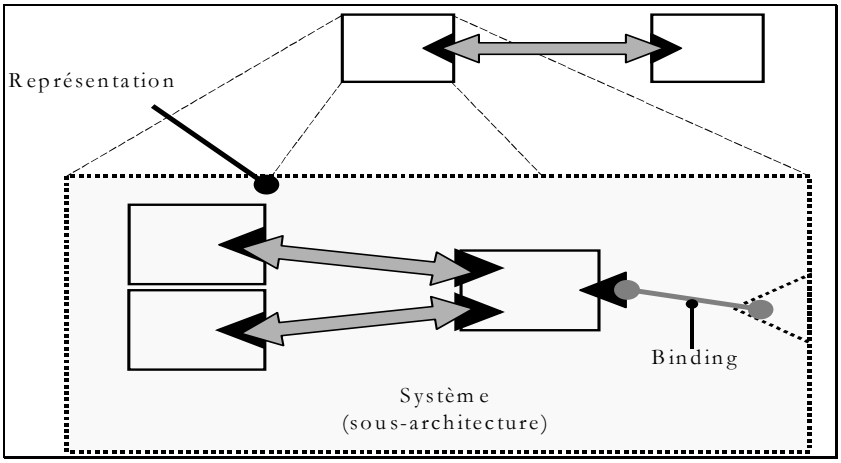


Figure 21 : Représentation sous Acme

5.4.5. Propriétés

Les concepts présentés ci-dessus permettent de décrire l'aspect structurel de l'architecture du logiciel. Pour améliorer la description des composants, des connecteurs, et des systèmes, chaque ADL offre son propre ensemble d'informations supplémentaires pour définir la sémantique à l'exécution, les types (exemple : les types de données communiquées entre les composants), les protocoles d'interaction, etc. Au lieu de définir un ensemble fixe de caractéristiques, Acme propose un système d'annotations basé sur des listes de propriétés. Chaque propriété a un *nom* qui l'identifie, un *type* optionnel et une *valeur*. Chaque entité (composant, connecteur) peut avoir une ou plusieurs propriétés.

Le type optionnel de la propriété Acme peut indiquer :

- Un type simple comme un entier, une chaîne de caractères, ou un booléen.
- Un type indiquant une propriété d'un sous-langage ADL comme Wright, Unicon, Rapide ; dans ce cas, les outils utilisent les attributs *nom* et *type* pour voir si la propriété les concerne. Si l'outil ne comprend pas une propriété donnée, il la laisse non interprétée pour les autres outils.
- Un type indiquant un lien externe (type « *external* ») avec une implantation comme par exemple un programme.

L'architecture d'un système client/serveur est décrite avec des propriétés comme suit :

```
System simple_cs = {
        Component client = {
            Port send-request ;
            Properties {Wright-style : style-id = client-server ;
            //indique que le composant client dans Acme est défini comme
            //un style dans Wright
                    source-code : external =  "CODE-LIB/client.c" }}
Component server = {
            Port receive-request ;
            Properties {max-concurrent : integer = 1;
                    source-code : external =  "CODE-LIB/server.c" }}
        Connector rpc = {
            Role {caller,callee }
            Properties {synchronous : boolean = true ;
                    max-roles : integer = 2;
                    protocol : Wright = "…"}}
Attachments : {
            Client.send-request to rpc.caller ;
            Server.reveive-request to rpc.callee}
}
```

5.4.6. Styles

Acme fournit un moyen de décrire des gabarits de conception : les f*amilies*. Cette notion est équivalente à la notion de style d'architecture que l'on peut trouver dans la plupart des ADL. Le langage permet de créer des gabarits de conception paramétrables et réutilisables permettant de spécifier des patrons d'architecture. Cette particularité permet de réutiliser n'importe quel style architectural. Le style d'architecture « Pipe & Filter » est décrit comme suit :

```
Family PipeFilterFam = {
        Component Type FilterT = {
            Port {stdin; stdout; };
            Property throughput : int;
        };
        Component Type UnixFilterT extend FilterT with = {
            Port stderr;
            Property implementationFile : String ;
        };
        Connector Type PipeT = {
```

```
            Roles {source; sink; };
            Property bufferSize = int;
        };
        Property Type StringMsgFormatT : Record [ size:int; msg:String; ];
}
System simplePF : PipeFilterFam = {

        Component smooth : FilterT = new FilterT
        Component detectErrors : FilterT = new FilterT;
        Component showTracks : UnixFilterT = new UnixFilterT extend with {
            Property implementationFile : String = "IMPL_HOME/showtracks.c";
        };
        Connector firstPipe : PipeT;
        Connector secondPipe : PipeT;
        Attachments { smooth.stdout to firstPipe.source;
                      DetectErrors.stdin to firstPipe.sink;
                      ShowTracks.stdin to secondPipe.sink; }
}
```

5.4.7. Traduction d'une description d'architecture

Acme est un langage qui fournit le noyau structurel exigé par n'importe quel ADL. Il permet de représenter une description d'architecture donnée sous la forme d'un ADL neutre. De ce fait toute description d'architecture peut être convertie d'un ADL à un autre en utilisant Acme comme langage intermédiaire.

Actuellement, Acme permet de faire des correspondances entre Aesop, C2, Meta-H, Wright et Rapide. D'autres langages sont sur le point d'être intégrés (tableau 10).

Actuellement	En cours
Aesop à Acme	Acme à Meta-H
C2 à Acme	Acme à UML
Meta-H à Acme	
Wright à Acme	
Rapide à Acme	

Tableau 10 : Correspondances entre les ADL dans Acme

5.4.8. Avantages

Acme présente un atout en offrant un langage et une boîte à outils servant de base à l'élaboration de nouveaux outils de construction. C'est un langage qui offre un double

avantage aux constructeurs : il leur fournit une base solide et extensible, ainsi qu'une infrastructure qui leur évite d'en reconstruire une de type standard.

Par ailleurs, Acme étant un langage d'échange générique, les outils développés sous celui-ci ont l'avantage d'être compatibles avec une large panoplie de langages.

5.4.9. Inconvénients

Malgré ses atouts, le langage Acme est sujet à quelques limites que nous citons dans ce paragraphe.

Acme n'offre pas la possibilité d'avoir une intégration complète des différents langages de description. Ne prévoyant pas dans sa conception toutes les structures architecturales, certains systèmes décrits dans un ADL ne pourront pas, à cet effet, être transcrits dans un autre.

De plus, les aspects évolution et dynamicité ne sont pas plus développés dans Acme que dans d'autres langages. Il se contente simplement d'utiliser ceux des autres ADL, sans y apporter de nouveautés. Des recherches sont néanmoins en cours pour s'ouvrir vers des modèles sémantiques plus riches, tels les contraintes logiques temporelles permettant d'exprimer les aspects dynamiques [GARLAN & AL., 97A].

6. Synthèse sur les ADL

Les langages de description d'architecture (ADL) que nous venons de présenter ont tous la même vocation qui est la formalisation, la vérification et la validation d'architectures. Leur but est de raisonner sur un niveau plus abstrait que l'implémentation. Cette abstraction permet au concepteur de mieux appréhender la complexité de l'application et de mieux concevoir le fonctionnement et le découpage de celle-ci ainsi que les connexions entre les différents élements la constituant. La définition d'une architecture d'un système revient donc à la définition des différents composants et connecteurs utilisés dans le système ainsi que la topologie de leur inter-connexion, à l'exception de Rapide où les connections sont seulement spécifiées dans les composants comme des comportements de communication.

6.1. Principaux avantages et inconvénients

Bien qu'ils aient beaucoup de points en commun, les buts recherchés par les ADL ne sont pas toujours les mêmes. Par exemple, certains s'intéressent plus particulièrement à la sémantique des composants et connecteurs alors que d'autres s'attachent plutôt à définir les inter-connexions entre composants et connecteurs. Chaque ADL possède alors ses atoûts et le choix d'un langage plutôt qu'un autre est guidé essentiellement par les besoins et les attentes du concepteur du système. Dans le tableau suivant (cf. tableau 11) nous résumons les principaux avantages et invonvénients des ADL que nous avons présentés dans cette section.

ADL	Principaux avantages	Principaux inconvénients
	▪ UniCon est simple à utiliser car il fournit un modèle de types de	▪ La description des interfaces se fait par le langage UniCon + un langage

UniCon	composants et de types de connecteurs prédéfinis. ▪ Il caractérise les interactions entre composants de manière explicite (existence d'un modèle de connecteurs). ▪ Il facilite l'intégration de logiciels existants (exécutables, sources, objets). ▪ Il fournit un environnement de conception complet : - un éditeur graphique pour décrire les composants et les connecteurs et l'architecture statique d'une application. - un analyseur syntaxique et un compilateur. ▪ Un générateur de code C est disponible.	de programmation. ▪ Les modèles de types de composants et de connecteurs sont des modèles qui ne sont pas extensibles et qui sont trop proches de l'implantation. ▪ La spécification de l'architecture est une spécification statique uniquement. ▪ Il y a peu de moyens pour spécifier les propriétés non fonctionnelles.
Rapide	▪ Rapide permet d'exprimer la dynamique d'une application de manière précise et détaillée ▪ Il est possible de simuler une application grâce à la création d'événements causals et grâce à l'environnement d'exécution intégrés à l'ADL.	Il n'y a pas de représentation explicite de connecteur.
Wright	▪ Wright permet de spécifier une architecture logicielle de manière formelle et totalement abstraite (composant, connecteur, configuration). ▪ Il fournit un modèle abstrait de composant et d'un modèle abstrait de connecteur. Les deux modèles sont indépendants.	▪ Il est difficile à assimiler. ▪ Wright ne possède pas d'environnement d'utilisation ou d'exécution (outils de modélisation, génération de code). ▪ Il ne possède pas de moyen de projeter l'architecture logicielle vers un système concret (le passage de l'abstrait au concret est difficile). ▪ Il propose peu de moyens pour séparer les spécifications

		fonctionnelles et non fonctionnelles.
Acme	▪ Acme est un langage pivot et fédérateur de l'ensemble des concepts utilisés par les ADL. ▪ Il s'agit d'un véritable langage d'intégration des ADL existants.	▪ Acme est un langage orienté vers la spécification structurelle d'une architecture logicielle : très peu de moyens pour exprimer la dynamique d'un système. ▪ Le langage ne permet pas une séparation claire entre les propriétés fonctionnelles et non fonctionnelles.

Tableau 11 : Comparaison générale de différents ADL

Dans ce paragraphe nous avons établi une synthèse globale sur les quatre ADL que nous avons étudiés. Les paragraphes suivants fournissent une comparaison plus détaillée des concepts de base de ces ADL, à savoir les composants, les connecteurs et la configuration. Pour ce faire, nous nous basons sur les caractéristiques définies par Medvidovic et Taylor (cf. section 4.2 et section 4.3).

6.2. Composants

Tous les ADL que nous avons présentés fournissent un support compréhensif pour modéliser les composants. Ils décrivent un composant comme une entité de premier degré et permettent de spécifier ses interfaces et de distinguer les types des instances de composants. Une synthèse plus détaillée est présentée dans le tableau suivant.

	Acme	**Rapide**	**Unicon**	**Wright**
Interface	Les points d'interface sont des ports	Les points d'interface sont des constituants (*provides*, *requires*, *action*)	Les points d'interface sont des « players»	Les points d'interface sont des ports
Type	Système de type extensible	Système de type extensible; contient un sous-langage de types	Ensemble de types énumérés et prédéfinis	Système de type extensible; nombre de ports paramétrable
Sémantique	Utilisation des modèles de sémantiques d'autres ADL	Ensemble d'événements partiellement ordonnés	Traces d'événements dans la liste de propriétés	la sémantique d'interaction de port est spécifiée en CSP

	dans la liste des propriétés	(posets)		
Contraintes	Langage de contraintes basé sur la logique de premier ordre	Via des patrons d'évènements (clause constraint)	Via les interfaces; restriction sur les "players" qui peuvent être fournis par les types de composants	Protocoles d'interaction pour chaque port dans CSP
Evolutions	Sous-typage structurel (extends)	Héritage (sous-typage structurel)	Aucune	Via différentes instanciations de paramètres

Tableau 12 : Synthèse détaillée sur les composants dans des ADL

6.3. Connecteurs

Le support fourni pour modéliser les connecteurs, est pour la plupart des ADL plus restreint que celui des composants. Parmi les quatre langages présentés, Rapide est le seul ADL qui ne considère pas le connecteur comme une entité à part au même niveau que celle du composant. Pour ce dernier, les connections sont seulement spécifiées dans les composants comme des comportements de communication. A l'opposé, les trois autres ADL permettent de modéliser explicitement la notion de connecteur, de décrire ses interfaces et de distinguer entre types et instances de connecteurs. Pour Acme et Wright, la richesse des descriptions des connecteurs est assez proche de celle des composants. Une synthèse plus détaillée est présentée dans le tableau suivant.

	Acme	**Rapide**	**Unicon**	**Wright**
Interface	Les points d'interface sont des rôles	Il n'y a pas d'interface explicite; la connexion de composants est interne	Les points d'interface sont des rôles	Les points d'interface sont des rôles
Type	Système de type extensible basé sur des protocoles	Aucun	Les types sont prédéfinis; ensemble de types énumérés	Système de type extensible basé sur des protocoles
Sémantique	Utilisation des modèles de	Règles de connexion	Implicite dans le type de	La sémantique d'interaction des

	sémantiques d'autres ADL dans la liste des propriétés	(Posets)	connecteurs. L'information de la sémantique peut être donnée dans la liste des propriétés	rôles est spécifiée dans CSP : "glue"
Contraintes	Langage de contraintes basé sur la logique de premier ordre	Aucune	Via des interfaces; Restreint le type de "players" qui peut être utilisé dans un rôle donné.	Via des interfaces et des sémantiques; des protocoles d'interaction pour chaque rôle dans CSP
Evolutions	Sous-typage structurel via la caractéristique "extends"	Aucune	Aucune	Via les différentes instanciations de paramètres

Tableau 13 : Synthèse détaillée sur les connecteurs dans des ADL

6.4. Configuration

Une représentation explicite de la configuration architecturale facilite la communication entre les différents acteurs du système (concepteurs, développeurs, testeurs, etc.). De ce fait, il est important de faire abstraction des détails de chaque composant et connecteur et de représenter le système à un niveau d'abstraction plus élevé. Dans le tableau suivant (cf. tableau 14) nous fournissons une synthèse détaillée sur les solutions offertes par chaque ADL pour décrire une configuration architecturale. Nous introduisons le critère "*Adaptabilité*" pour juger la capacité d'un langage à prendre en compte les changements qui peuvent surgir sur un système avant ou durant son exécution.

	Acme	**Rapide**	**Unicon**	**Wright**
Formalisme commun	Spécification textuelle et graphique concise et explicite	Spécification textuelle et graphique explicite	Spécification textuelle et graphique explicite	Spécification textuelle concise et explicite
Composition	Via les représentations	Les « mappings » pour relier une	Composants composites	Représentation de composants et de connecteurs

		architecture à une interface		composites
Hétérogénéité	"Open property lists" acceptant n'importe quelle notation de modélisation ; permet la traduction d'un ADL vers un autre	Supporte le développement des systèmes spécifiés en VHDL, C/C++, Ada et Rapide	Supporte uniquement les composants et types de connecteurs prédéfinis	Supporte à la fois des éléments de fine et de forte granularité
Adaptabilité	Permet de spécifier explicitement une famille d'architectures comme une classe de premier ordre ce qui facilite son évolution	Permet d'exprimer une création, une suppression ou une description plus fine du schéma d'instanciation du composant	Pas d'ajout ni de suppression n'est autorisé après l'instanciation initiale des composants	Permet l'ajout et la suppression des entités du système durant son exécution
Contraintes	Les ports peuvent seulement être rattachés aux rôles et vice-versa	langage de contraintes pour restreindre le comportement de l'architecture	Les « players » peuvent seulement être rattachés aux rôles et vice-versa	Les ports peuvent seulement être rattachés aux rôles et vice-versa

Tableau 14 : Synthèse détaillée sur la configuration dans des ADL

7. Langages orientés objet

Le paradigme commun permettant la modélisation des systèmes informatisés est le paradigme orienté objet qui s'est généralisé et standardisé dans les méthodes de conception depuis une dizaine d'années. UML (Unified Modeling Language), dont le développement a débuté en 1994, constitue un nouveau standard de modélisation objet. G. Booch et J. Rumbaugh [BOOCH & AL., 99] de *Rational Software Corporation* ont décidé de travailler ensemble pour réaliser une unification des principales méthodes à objet : OOD (Object Oriented Design) et OMT (Object Modeling Technique). UML n'est pas une approche entièrement nouvelle : elle résulte de l'évolution de ces dernières. Il est donc très facile de travailler avec UML si l'on

dispose déjà d'une expérience dans les autres méthodes. Un des ingrédients du succès d'UML est le fait d'être un langage générique. L'ensemble des concepts et des vues qui composent la sémantique UML peut s'adapter à tous les domaines et problèmes de conception. De plus, UML est très intuitif, plus simple, plus homogène et plus cohérent que les autres méthodes objet proposées.

Le paragraphe 7.1 constitue un rappel des concepts de base d'UML, le paragraphe suivant (cf. section 7.2) montre leur utilisation dans le cadre des architectures logicielles.

7.1. Eléments de base d'UML Version 1.x

7.1.1. Classes et objets

Une classe décrit un ensemble d'objets (cf. figure 22) de même structure définie par un ensemble d'attributs et de relations, et de même comportement défini par un ensemble d'opérations. Les classes identifient les abstractions du domaine du problème et/ou de la solution. Les instances es classes sont appelées des objets. Les diagrammes d'objets illustrent un cas concret à un instant donné en montrant des objets (instances des classes) et des liens (instances des associations) existants entre ces objets.

Le concept d'objet en UML peut être un candidat pour représenter un composant logiciel ou matériel de l'architecture. Une classe peut donc représenter le concept de composant générique. La spécialisation d'un composant générique dans un composant particulier est réalisée lors de l'instanciation d'une classe en objet (instanciation des attributs, signature de l'objet, etc.).

Les classes sont le concept fondamental utilisé dans les diagrammes de classes. Ces derniers décrivent la vue statique du système en faisant abstraction de son implémentation et précisent la topologie du système (association) et les relations de hiérarchie (agrégation, composition). Au sens donné dans les ADL, un diagramme de classes avec ses relations peut représenter un style architectural.

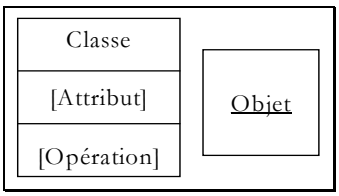


Figure 22 : Classe et objet

7.1.2. Interfaces

L'interface est la vue externe d'un objet. Elle définit les services accessibles (offerts) aux utilisateurs de l'objet [PIECHOKI]. En d'autres termes, une interface (cf. figure 23) représente une collection d'opérations qui spécifient les services offerts par une classe, un composant ou un sous-système.

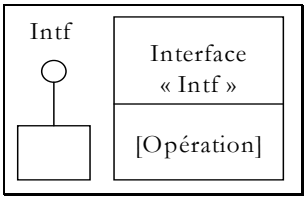


Figure 23 : Interface

7.1.3. Composants et instances de composants

Un composant (cf. figure 24) est une partie physique du système. Il peut s'agir d'un fichier code source. Les composants dans UML exposent leurs fonctionnalités au travers de leurs interfaces de la même manière que les classes. Ils sont typiquement liés par des relations de dépendance. Les composants peuvent être organisés en paquetages qui définissent des sous-systèmes.

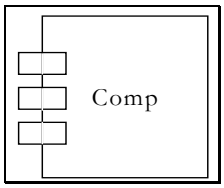


Figure 24 : Composant

7.1.4. Paquetages et instances de paquetage

Un paquetage regroupe des éléments de diagrammes qui entretiennent entre eux des relations étroites (ou fortement couplées). Cette notion est utilisée pour décrire les composants logiques d'une architecture logicielle [BOOCH & AL., 99].

Grâce aux paquetages (cf. figure 25), il est possible de manipuler un ensemble d'éléments (des classes, des relations, …) comme un groupe. Le regroupement des éléments de modélisation est fait d'une manière purement logique. On peut contrôler la visibilité des éléments au sein du paquetage, de façon à ce que certains éléments soient visibles à l'extérieur du paquetage et d'autres soient cachés. On peut de même utiliser les paquetages pour structurer un système en catégories (vue logique) et en sous systèmes (vue des composants). On peut, de plus, accéder à un paquetage à partir de sa classe interface. La classe interface est une classe publique qui peut être identifiée par un nom symbolique. On peut enfin établir des relations entre les classes interfaces des différents paquetages.

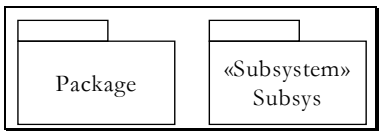


Figure 25 : Paquetage et sous-système

7.1.5. Les relations

Les relations permettent de lier les concepts d'UML. Ce dernier offre quatre types de relations : les dépendances, les associations, les généralisations et les réalisations :

1. **La dépendance** (cf. figure 26.a) est une relation sémantique entre deux éléments, telle que le changement apporté à l'un des éléments (élément source) affecte le second (élément cible). Elle peut contenir une étiquette, comme par exemple «utilise».
2. **L'association** (cf. figure 26.b) est une relation structurelle entre les classes (plus généralement les classificateurs) et possède des rôles qui décrivent la manière avec laquelle une classe voit l'autre classe au travers de l'association. L'agrégation, par exemple, représente une relation d'association. Les associations possèdent des instances appelées *Liens*, qui connectent les objets dans les diagrammes d'objets.
3. **La généralisation** (cf. figure 26.c) est une relation hiérarchique selon laquelle le parent représente une généralisation de l'enfant, et, inversement, l'enfant une spécialisation du parent.
4. **La réalisation** (cf. figure 26.d) est une relation sémantique entre classificateurs (interface, classe, composant, etc.). Un classificateur propose un contrat dont l'autre classificateur en garantit la réalisation.

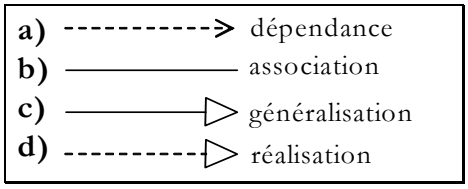


Figure 26 : Relations

7.1.6. Stéréotypes, étiquettes et contraintes

Les stéréotypes (cf. figure 27) permettent de définir de nouvelles classes d'éléments de modélisation. Ils étendent le vocabulaire d'UML en spécialisant les classes du méta-modèle. UML possède des stéréotypes standard tels que « Subsystem », « System », etc. Les étiquettes (tagged values) permettent d'ajouter de nouvelles propriétés aux classes du méta-modèle ou à leur spécialisation par stéréotypes. Les contraintes précisent la sémantique des diagrammes UML. Elles peuvent être exprimées en langue naturelle ou en utilisant le langage de contraintes d'UML : OCL (Object Constraint Language).

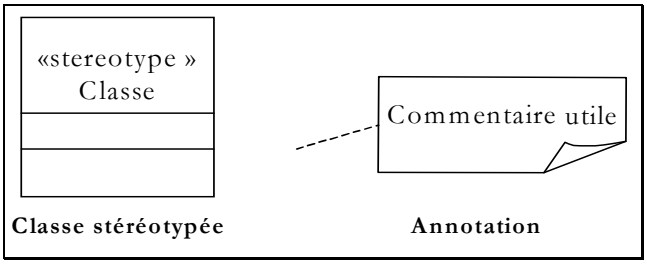


Figure 27 : Stéréotype et annotation

7.1.7. Collaborations

Une collaboration (cf. figure 28) définit une interaction entre divers éléments (classes, interfaces, etc.) qui coopèrent (par envoi de message) pour fournir un comportement global non réalisable par utilisation des éléments de manière indépendante.

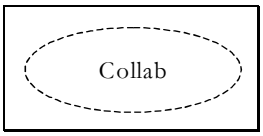


Figure 28 : Collaboration

7.2. Modélisation d'une architecture logicielle avec UML 1.x

Plusieurs travaux ont montré qu'il est possible d'utiliser UML pour la description des architectures logicielles [ROBBINS & AL., 98]. La technique utilisée à travers les diverses approches est souvent similaire. Les auteurs utilisent les diagrammes proposés par UML pour spécifier les différents aspects et structures d'une architecture : les diagrammes de classes permettent de représenter les aspects structurels (composants, connecteurs, modules, processus,…). Les diagrammes d'objets décrivent les instances de ces entités. Les associations, agrégations et envois de messages matérialisent des connecteurs élémentaires. Les comportements des composants et des connecteurs peuvent être décrits par des diagrammes d'états. Les diagrammes de déploiement sont plus appropriés pour les aspects matériels (machines, réseaux…). Les paquetages UML offrent la possibilité de regrouper plusieurs entités et de réaliser des descriptions hiérarchiques et modulaires. Le langage de contraintes OCL [WARMER & AL., 98] associé à UML permet, entre autres, d'exprimer les contraintes architecturales à vérifier.

Nous pouvons classer les différentes solutions proposées pour décrire les architectures logicielles avec UML en deux grandes catégories :

- Utilisation du modèle UML pour modéliser les différents aspects d'architecture logicielle.
- Utilisation d'un profil du modèle UML pour enrichir la notation et l'adapter aux besoins du domaine d'application.

7.2.1. Utilisation du modèle UML

Dans cette partie nous nous basons sur les travaux de D. Garlan [GARLAN & AL., 02] qui a présenté quatre stratégies pour décrire des architectures logicielles en utilisant UML. Les stratégies sont organisées selon le choix fait pour représenter les types et les instances de composants puisque le composant est l'élément conceptuel central pour toute description architecturale. Pour chaque stratégie, D. Garlan considère des sous-stratégies pour représenter les autres éléments architecturaux : port, connecteur, système et représentation. Dans le cadre de ce travail, nous nous intéressons seulement au port et au connecteur. Nous utilisons la notion de paquetage pour représenter un système.

Dans la suite de ce paragraphe nous présentons trois des quatres stratégies évoquées dans [GARLAN & AL., 02] et nous les illustrons par un exemple simple (voir p.62) basé sur le style « pipe & filter ».

7.2.1.1. Classes & objets

Dans cette stratégie (cf. figure 29), les types de composants sont représentés par les classes (Filter) et les instances des composants par des objets (ex : Eclateur).

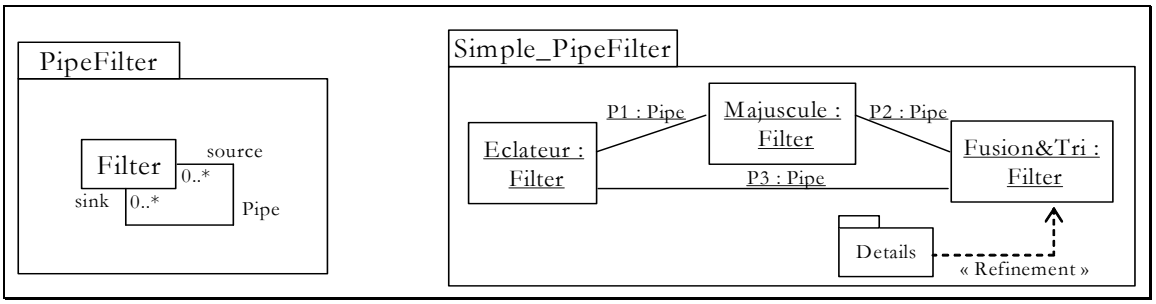


Figure 29 : Représentation des concepts d'une architecture en UML

Cependant, les instances de composants dans les ADL peuvent avoir une structure et un comportement différents les uns des autres. Dans l'exemple présenté ci-dessus, les instances « Eclateur » et « Fusion&Tri » sont toutes les deux de types « Filter » mais elles ont chacune un comportement différent et par suite doivent avoir une structure interne et des attributs différents. En effet, « Fusion&Tri », à la différence de « Eclateur », possède une sous-structure détaillée dans le paquetage "Details".

Pour décrire les ports des composants (interfaces), nous pouvons énumérer cinq options (cf. figure 30) :

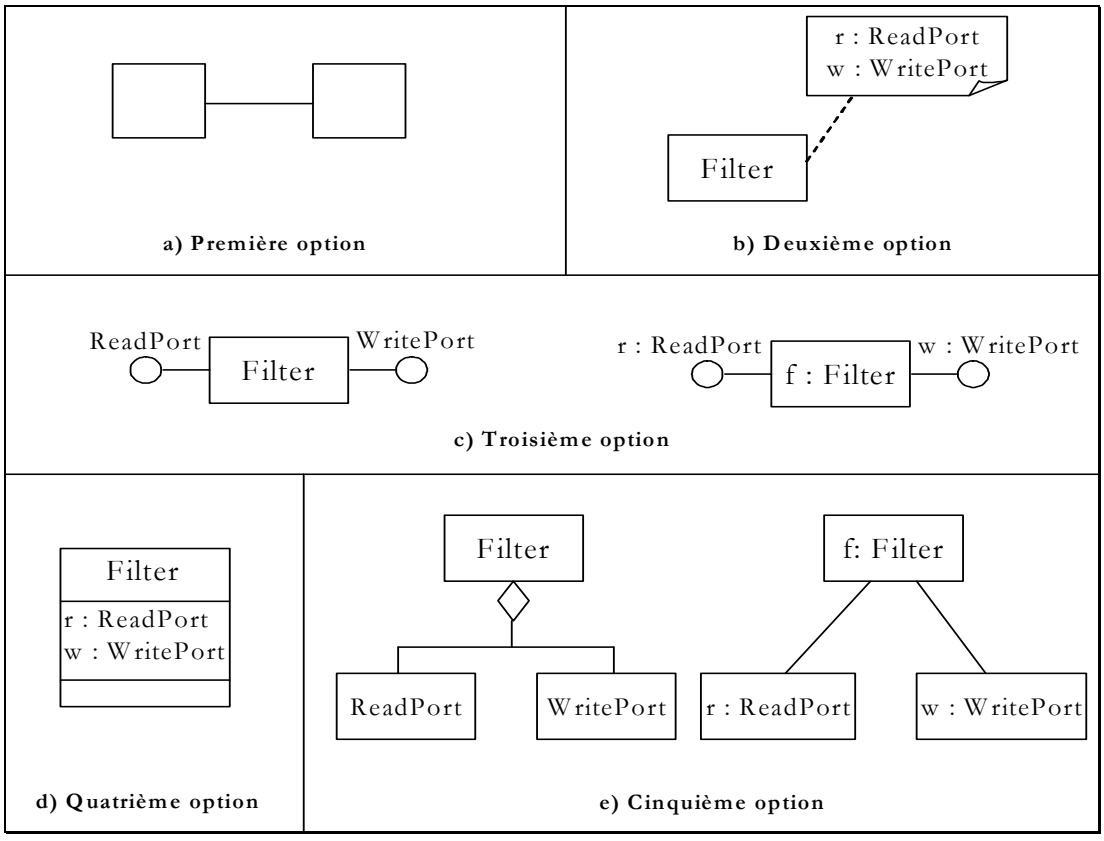


Figure 30 : Représentation des ports de composants en UML

1. **Aucune représentation explicite** (cf. figure 30-a) : C'est l'option adoptée dans la figure présentée ci-dessus. Ce choix peut être raisonnable quand il s'agit d'un diagramme simple ou bien quand le composant n'a qu'un seul port. Cependant, il n'y a pas de moyen pour décrire les propriétés des ports.
2. **Représentation avec les annotations** (cf. figure 30-b) : Les ports peuvent être représentés par des annotations. Cette approche peut être utilisée pour spécifier des informations arbitraires concernant les ports. Comme la première option, cette approche peut être utilisée si la description des propriétés des ports n'est pas d'un intérêt majeur.
3. **Représentations avec les interfaces** (cf. figure 30-c) : La solution la plus évidente est d'utiliser les interfaces pour décrire les ports. Cette manière de faire a l'avantage de fournir une description visuelle du port. Le concept d'interface et de port ont aussi la même intention, qui est celle de caractériser la manière avec laquelle une entité interagit avec son environnement. Néanmoins, si les deux concepts sont similaires, ils ne sont pas identiques. La différence majeure entre les deux est qu'en UML, une interface ne peut pas avoir des attributs ou de structure interne. A l'inverse, beaucoup d'ADL permettent d'associer aux ports des propriétés et des représentations plus détaillées.
4. **Représentation avec les attributs** (cf. figure 30-d) : On peut décrire les ports comme des attributs de la classe/objet. Dans ce cas, les ports ne peuvent avoir qu'une description simple, essentiellement un nom et un type.
5. **Représentation avec les classes** (cf. figure 30-e) : Une autre alternative consiste à décrire les ports comme des sous-structures qui indiquent qu'un type de composant peut avoir plusieurs ports. L'instance d'un composant est modélisée par un objet associé à un ensemble d'objets ports. Cependant, en représentant les ports par des classes, non seulement le diagramme sera plus complexe, mais en plus, il n'y aura plus de distinction visuelle entre ports et composants.

Dans la stratégie classes et objets, on peut distinguer trois possibilités pour représenter les connecteurs :

1. Les connecteurs sont des **associations** dans le diagramme de classes, et des liens dans le diagramme d'objets. Néanmoins, les connecteurs dans les ADL ont une signification différente que celle des associations dans UML. La relation d'association ne permet pas de représenter un connecteur comme une entité à part au même degré que celle du composant. Ainsi, il est impossible de représenter des types de connecteurs ce qui empêche leur réutilisation.
2. Une deuxième solution pour décrire les connecteurs consiste à représenter les types de connecteurs comme des **classes d'association**. Dans ce cas, les attributs des types et des instances de connecteurs sont considérés comme des attributs de la classe/objet d'association correspondante. Toutefois, avec cette approche il n'y pas de moyen de représenter explicitement les interfaces des connecteurs (ie. ses rôles).
3. Une troisième alternative consiste à modéliser les types de connecteurs comme une **classe**, et les instances de connecteurs comme des **objets**. L'avantage est de représenter un connecteur comme une entité de premier degré au même titre que celui d'une classe. Dans ce cas, pour représenter les rôles d'un connecteur, on peut choisir l'une des options que nous avons présentées ci-dessus pour les ports. Le rattachement des ports et des rôles peut être représenté par une relation d'association ou de dépendance.

L'inconvénient de cette approche est l'absence d'une distinction visuelle entre les composants et les connecteurs.

7.2.1.2. Classes & classes

Cette seconde stratégie consiste à représenter les types et les instances comme des classes (cf. figure 31). Cette solution permet de palier aux inconvénients de la première stratégie. Il est ainsi possible de décrire le fait que les instances du même type de composant peuvent avoir des comportements, des propriétés et des sous-structures différent(e)s. Cependant, une telle approche peut engendrer une confusion pour distinguer les types et les instances de composants.

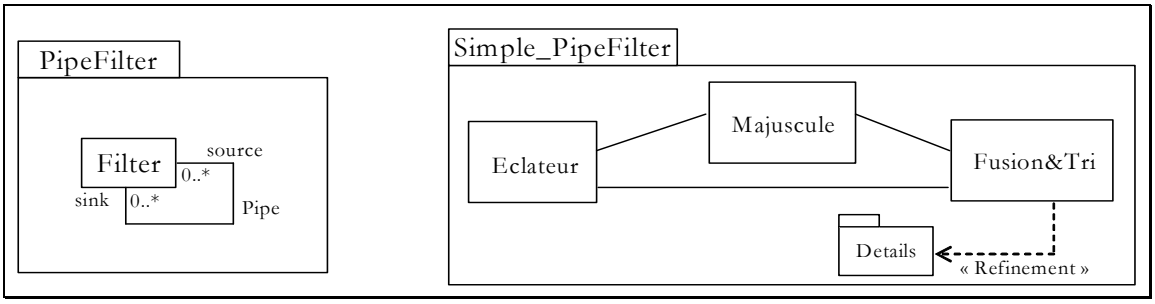


Figure 31 : Composants architecturaux et classes UML

Les alternatives pour représenter les ports et les connecteurs sont les mêmes que celles présentées dans la première stratégie.

7.2.1.3. Composants

D'une certaine manière, les composants en UML peuvent être des "candidats" naturels pour représenter les composants architecturaux. Les composants interagissent avec leur environnement à travers leurs interfaces. Les interfaces peuvent alors représenter les ports exposés par un composant architectural. Cette stratégie (cf. figure 32) est similaire à celle qui se base sur les classes et les objets (cf. section 7.2.1.1). Cependant, dans la dernière stratégie, nous disposions de plus de possibilités permettant de fournir une description riche des ports des composants et des connecteurs. En fait, cette stratégie limite le choix de description des connecteurs au nombre de deux : soit comme une relation de dépendance entre les ports / interfaces des composants ; soit comme un composant UML. Si on représente les instances de connecteurs en tant que relations de dépendance entre les composants, on peut représenter les types de connecteurs par des relations de dépendance stéréotypées. Néanmoins, la notion de dépendance dans UML ne fournit pas une description adéquate de la notion de connecteurs architecturaux. Pour remédier à cela, il est possible de représenter un connecteur comme un composant UML. Cette solution ne permet pas de distinguer entre composants et connecteurs.

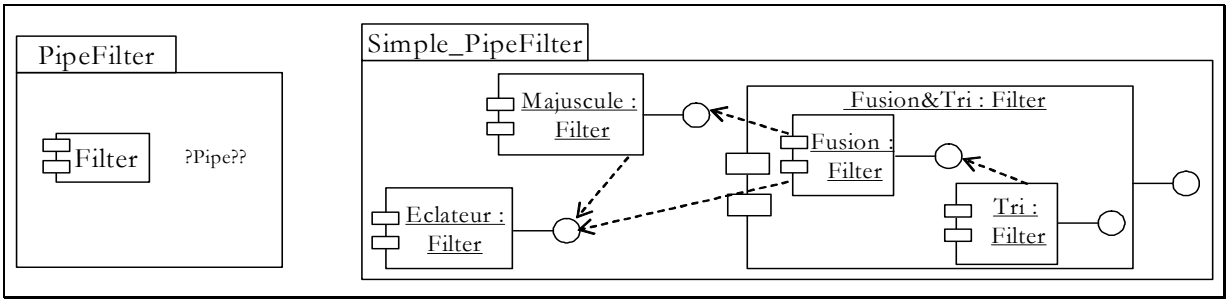


Figure 32 : Composants architecturaux et composants UML

7.2.1.4. Sous-systèmes

Dans cette alternative (cf. figure 33), un composant est décrit comme un sous-système. Un sous-système est un paquetage stéréotypé qui permet de regrouper ou d'encapsuler des éléments de modélisation. Cette solution ne permet pas de décrire avec précision des composants architecturaux.

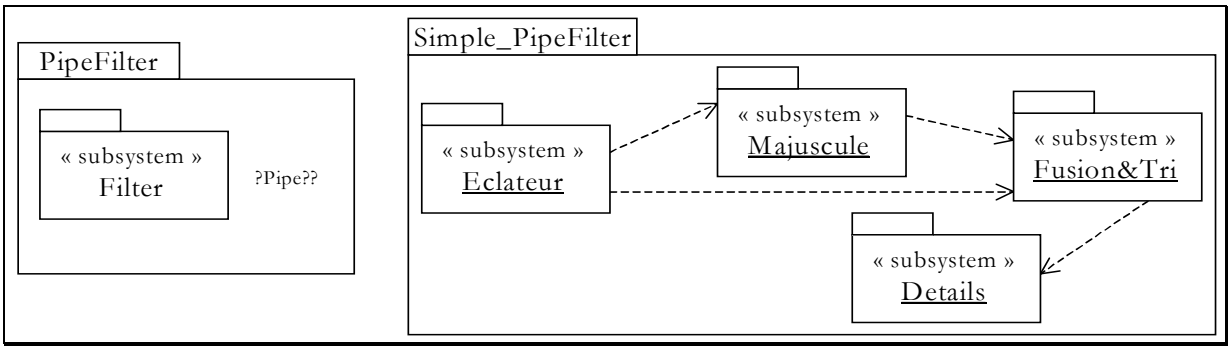


Figure 33 : Composants architecturaux et sous-systèmes UML

7.2.2. Utilisation d'un profil UML

Cette solution emploie des mécanismes d'extensions propres à UML afin d'enrichir la notation et l'adapter aux besoins du domaine d'application. Un ensemble cohérent d'extensions est appelé profil.

Cette approche est celle qui s'est le plus développée dans le domaine de la conception des applications temps réel (UML-RT). Il est intéressant de retrouver dans les profils déjà proposés, des concepts appartenant aux ADL.

UML-RT est supporté par l'outil Rose RT de Rational et intègre les concepts de l'ADL ROOM [SELIC & AL., 98]. Il introduit un stéréotype de classes appelé *Capsule* ainsi que les notions de *port*, *protocole* et *connecteur* qui sont mieux adaptés aux systèmes temps réel que les classes. Pour illustrer ces concepts, nous décrivons l'exemple du « pipe & filtre » avec le diagramme de collaboration d'UML-RT. Dans la figure suivante (cf. figure 34), les filtres sont représentés par des capsules de type *Filter* avec des ports *Input* et *Output*. Dans ce cas, les pipes deviennent des connecteurs conformes au protocole pipe (*ProtPipe*) avec les rôles

source et *sink*. Les ports *Output* et *Input* liés aux connecteurs, sont aussi respectivement les rôles *source* et *sink*.

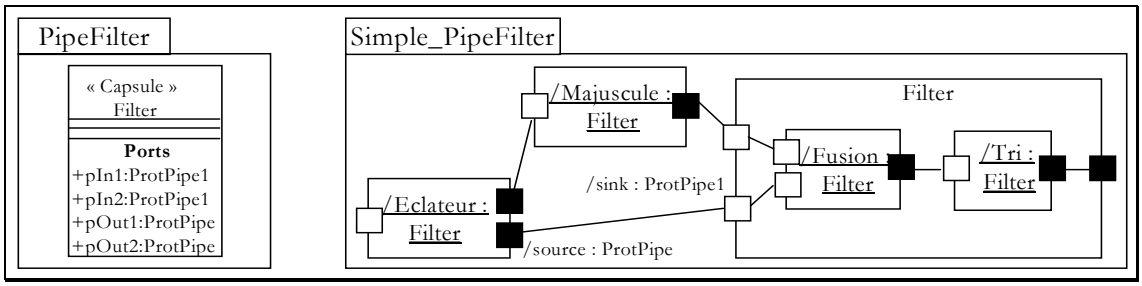


Figure 34 : Diagramme de collaboration d'UML-RT

7.2.3. Conclusion

Les méthodes orientées objet sont devenues très populaires et très utilisées vu l'apport et les avantages qu'elles proposent pour la modélisation des applications. Dans cette partie, nous avons montré qu'elles peuvent être utilisées pour décrire des architectures logicielles. Cependant, elles ont également montré leurs limites lorsqu'il s'agit de décrire et de réutiliser des interactions complexes entre groupes d'objets. Aussi, il est difficile de structurer les études en UML selon une approche reposant sur les trois mots clés : composant, connecteur, architecture. Notons que UML 2.0 fait des propositions pour tenir compte de la notion de composant logique (dans le sens des composants COM, EJB,…) (cf. section 7.3). Néanmoins, un certain nombre de lacunes et de difficultés à utiliser la notation UML ou une extension d'UML ont été identifiées [MEDVIDOVIC & AL., 99] [HOFMEISTER & AL., 99] :

Description de concepts architecturaux : UML ne fournit pas de description explicite pour les concepts architecturaux comme, par exemple, les notions de connecteurs ou de ports. Le lien entre ces concepts architecturaux et leur modélisation doit être défini et maintenu implicitement par le concepteur. Pour pouvoir représenter ces concepts, il faut utiliser les mécanismes d'extension comme les stéréotypes.

Sémantique : l'utilisation intensive de mécanismes d'extension comme les stéréotypes pose des problèmes de définition sémantique. En effet, utiliser une même notation pour des concepts relativement différents peut engendrer des confusions. Ceci ne facilite pas non plus l'exploitation de la description par différents outils.

Représentation des différentes structures : les diagrammes disponibles dans la notation UML répondent à une modélisation objet. Par contre, ils ne sont pas toujours adaptés pour décrire les différentes structures d'une architecture logicielle. On est souvent conduit à utiliser les diagrammes de classes pour plusieurs structures en utilisant les mécanismes d'extension pour les différencier.

7.3. Modélisation d'une architecture logicielle avec UML 2.0

Récemment, avec l'apparition de UML 2.0 [OMG, 03] nous assistons à une convergence des approches formelles et semi-formelles. Cette nouvelle version propose un modèle de composants nettement amélioré par rapport à la version UML 1.x. Dans la suite nous présentons les différents concepts du diagramme de composants d'UML 2.0 (cf. figure 35).

Un composant UML 2.0 est une entité instanciable qui interagit avec son environnement par l'intermédiaire de points d'interactions appelés *ports*. Un port permet de spécifier les points d'interactions d'un composant : soit entre le composant et son environnement, soit entre le composant et sa décomposition interne. Un port est typé par les *Interfaces*. Une interface contient un ensemble d'opérations et/ou de contraintes. Elle définit le comportement du composant en terme de services fournis et requis.

UML 2 introduit aussi la notion de connecteur. Un connecteur (*connector*) est une entité qui relie des ports de composants. Il en existe deux types : d'une part, le connecteur d'assemblage (*AssemblyConnector*) permet d'assembler deux composants en connectant un port fourni d'un composant au port requis de l'autre composant. D'autre part, le connecteur de délégation (*DelegationConnector*) permet de connecter un port externe au port interne d'un sous-composant. L'invocation d'un service sur le port externe est transmise au port interne auquel il est connecté. Le connecteur de délégation permet de modéliser la décomposition hiérarchique d'un composant.

Tout comme ses prédécesseurs, UML 2 ne considère pas le connecteur comme une classe de premier degré. Toutefois, pour contourner cette lacune, le connecteur peut être présenté par un composant stéréotypé. De plus, cette version d'UML souffre d'une carence d'outils pour la réutilisation des styles d'architecture, la vérification et le maintien de la cohérence de la description architecturale.

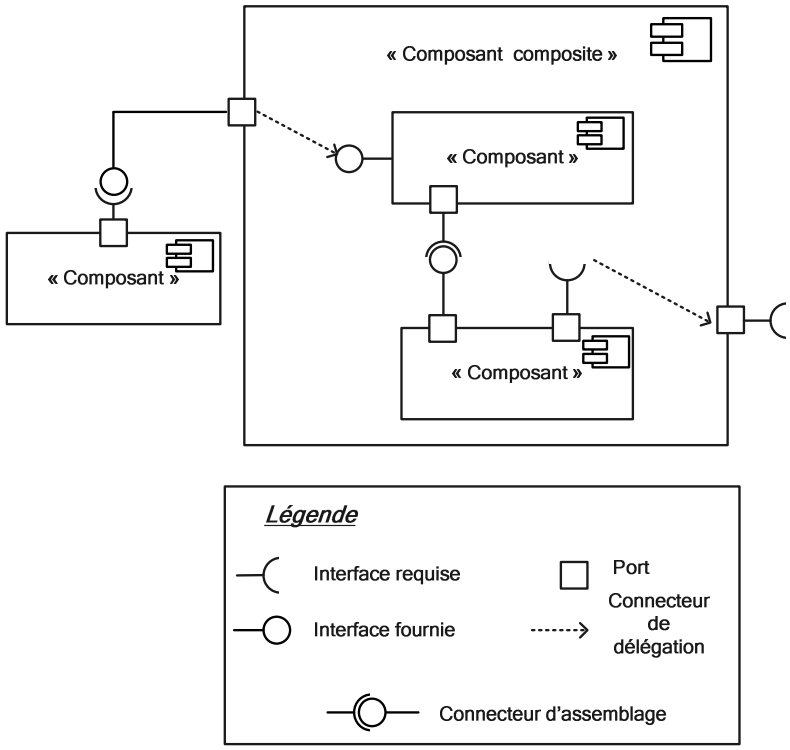


Figure 35 : Notion de composant dans UML 2.0

8. Résumé du chapitre

Dans ce chapitre nous avons présenté deux approches pour la description exhaustive des architectures logicielles (cf. figure 36), les langages de description d'architecture (ADL :

Architectural Description Language) et les méthodes orientées objet (UML : Unified Modelling Language).

Nous avons montré qu'il est possible de décrire une architecture en utilisant un langage formel ou semi-formel. L'inconvénient majeur de la dernière approche est le fait de faire abstraction de tout ce qui pouvait concerner les interactions de haut niveau entre composants et leur manipulation en tant qu'entité de premier ordre. Cette lacune trouve une réponse chez certains ADL qui, quant à eux, se focalisent sur la représentation et la spécification des interactions complexes entre composants.

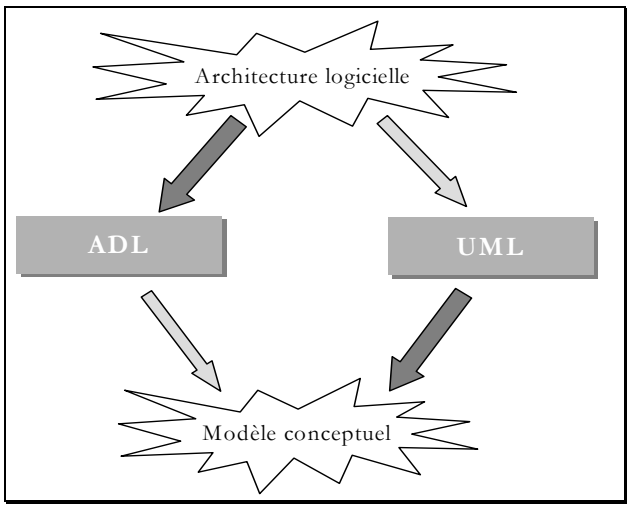


Figure 36 : Description des architectures logicielles

Chaque approche offre ainsi un certain nombre d'avantages mais possède aussi des inconvénients (cf. tableau 15). De ce fait, chacune peut être retenue comme un candidat pour modéliser des Systèmes d'Information Coopératifs. La question qui se pose est alors : quelle est l'approche la mieux adaptée ou la plus adéquate pour décrire les SICo ? A ce stade, nous nous limitons à cette étude comparative sans faire de choix précis. Le chapitre suivant apportera la réponse à cette question en combinant ces deux approches avec des éléments supplémentaires de solution comme les patrons et une caractérisation des modes de coopération.

	Avantages	**Inconvénients**
ADL	▪ Orienté architecture ; ▪ Capable de suivre un système dès la conception et tout au long de sa vie ; ▪ Favorise la conception par assemblage de composants (réutilisation de composants et de connecteurs) ; ▪ Capacité d'analyses multiples (propriété non fonctionnelle) ;	▪ Multiplicité des ADL : ▪ Multiplicité de formalismes ▪ Multiplicité des outils (Chaque ADL possède sa propre gamme d'outils) ▪ Pas toujours mature ; ▪ Long à apprendre, à comprendre et à prendre en main.

	▪ Pour certains ADL, génération du code intermédiaire.	
UML	▪ Orienté objet ; ▪ Normalisé; ▪ Mature et répandu.	▪ Multiplicité et complexité des diagrammes pour décrire toute une architecture ; ▪ Basé sur le graphique : il faut limiter le volume d'information à cause du risque de surcharge ; ▪ Langage semi-formel : peut poser des problèmes d'interprétation ; ▪ Extensibilité réduite (par les annotations ou les stéréotypes) ; ▪ La sémantique de composant, connecteur et interface est dispersée.

Tableau 15 : Synthèse comparative ADL - UML

CHAPITRE IV

ELEMENTS DE BASE DE LA SOLUTION

1. Introduction

Les deux chapitres précédents étaient dédiés à la présentation des architectures de coopération les plus connues et des langages de modélisation permettant de décrire et de spécifier ces architectures. La diversité des architectures et des langages existants rend la tâche difficile à un concepteur pour choisir, adapter ou spécifier l'architecture de son Système d'Information Coopératif (SICo). Nous proposons une méthode (une démarche et des modèles) formalisée sous la forme d'un système de patrons, composé de patrons processus et de patrons produit, dédiée aux SICo. Les patrons produit permettront la capitalisation et la réutilisation des architectures de coopération existantes. Les patrons processus proposent au concepteur une démarche pour naviguer dans la panoplie des patrons produit proposés pour choisir celui qui répond le mieux à ses besoins.

L'objectif de ce chapitre est de présenter les éléments de base sur lesquels nous fondons notre proposition. Nous reviendrons tout d'abord (section 2.1) sur la notion de patrons pour mieux expliquer et mettre en valeur leur apport dans l'ingénierie des SICo. Dans un second temps nous définirons la notion de système de patrons (section 2.2) ainsi que le formalisme que nous avons choisi pour représenter (section 2.3) les différents patrons. La troisième section relatera du choix du langage de modélisation qui nous permettra d'exprimer la solution modèle de nos patrons produit. La quatrième partie de ce chapitre (section 4) sera consacrée à la définition des critères qui nous permettront d'effectuer une classification des différentes architectures de coopération. Ces critères nous serviront de base pour élaborer les patrons processus qui guideront le concepteur dans le choix du patron produit le plus approprié à un problème de coopération particulier. Par la suite (section 6), nous présenterons un exemple de patron produit formalisé. Pour finir (section 7), nous proposerons un patron processus qui décrit la démarche à suivre permettant de tenir compte des nouveaux besoins pour faire évoluer le système de patrons.

2. Système de patrons pour les SICo

2.1. Patrons pour l'ingénierie des SICo

Les patrons permettent au concepteur d'un Système d'Information (SI) de réutiliser des modèles répondant au mieux aux besoins d'une famille de SI. Ils capitalisent ainsi d'un côté les besoins des SI et de l'autre côté les solutions pour spécifier ces besoins.

Deux types de connaissances sont mises en jeu dans les patrons :

1. **les connaissances du domaine** détenues par les experts et les utilisateurs du SI (les besoins : les problèmes),
2. **les connaissances de développement** dont les concepteurs de SI disposent pour spécifier les besoins (les techniques de spécification : les solutions).

Les connaissances du domaine sont des connaissances métiers, détenues par les experts (les acteurs) du métier auquel s'adresse le SI à concevoir. Ces connaissances métiers correspondent aux différents concepts manipulés dans le SI.

Les connaissances de développement sont des connaissances informatiques, détenues par les concepteurs de SI. Elles correspondent à des techniques de spécification et d'implantation des SI. On s'oriente alors vers le savoir et le savoir-faire d'ingénierie de SI. Un patron peut être utilisé pour capitaliser un modèle général répondant à un problème spécifique. Le patron capitalise ainsi des fragments de modèles, c'est-à-dire des fragments de produits de développement. A ce titre, il s'agit d'un patron modèle ou orienté savoir.

Le patron capitalise également des savoir-faire de modélisation, c'est-à-dire des manières pertinentes d'élaboration de modèles. La manière de construire un modèle de composition (composants optionnels, contraintes d'incompatibilité entre composants) en est un exemple. De ce fait, le patron permet alors d'organiser hiérarchiquement et fonctionnellement les problèmes et les manières de les résoudre. Le patron capitalise ainsi des fragments de démarches et par conséquent des fragments de processus de développement. On parle alors de patron de démarche ou orienté savoir-faire.

En résumé, la technologie des patrons favorise, en plus de la capitalisation et de la réutilisation, l'intégration des connaissances de domaine et des connaissances de développement. Un patron permet en effet de spécifier des problèmes liés aux besoins des utilisateurs du domaine (connaissance domaine) à l'aide de solutions de développement adaptées (connaissances de développement).

Dans le cadre de ce travail...
Connaissances du domaine = connaissances détenues par les spécialistes des SICo. Notre objectif est de capitaliser des : savoirs = des modèles représentants des architectures logicielles coopératives. savoir-faire = des fragments de démarche pour l'ingénierie de SICo.

2.2. Système de patrons

Les patrons ne sont pas isolés. Il y a des relations de dépendance entre eux. Un système de patrons relie ses différents patrons constitutifs. Il décrit la manière avec laquelle les patrons sont inter-reliés et la manière avec laquelle ils se complètent pour former un tout cohésif.

Tout comme le langage de patrons, le terme *système de patrons* a été introduit par l'architecte C. Alexander qui définit un système de patrons comme *une collection de patrons formant un vocabulaire qui permet de comprendre et communiquer les idées*. D'après J. Coplien [COPLIEN, 96], un langage de patrons est *une collection structurée de patrons construits l'un sur l'autre pour transformer les besoins et les contraintes dans une architecture*. Un patron étant une solution récurrente à un problème dans un contexte donné, un langage de patrons est un ensemble de solutions qui travaillent ensemble pour résoudre un problème complexe selon une solution ordonnée relativement à un but prédéfini [CONTE, 97].

Dans [BUSCHMANN & AL., 96], un système de patrons pour l'architecture logicielle est *une collection de patrons pour l'architecture logicielle accompagnée d'un guide pour leur combinaison, implémentation, et leur utilisation dans le développement logiciel.* Les patrons doivent être ainsi tissés entre eux dans un tout cohésif qui révèle les structures et les relations inhérentes à chacun de ses composants pour atteindre un objectif commun.

Naturellement, dans le cas d'un système complexe de plusieurs dizaines de patrons, il est nécessaire de disposer de critères de classification, ou de préciser le patron coordonnateur d'une démarche. Dans notre système de patrons, nous proposons un patron de type processus pour coordonner l'usage des différents patrons de type produit. De même, nous énumérons quelques critères pour classifier les différents patrons produit. Ce travail sera présenté dans la dernière partie de ce chapitre.

Dans le cadre de ce travail...
Des critères pour classifier les différents patrons produit. Des patrons processus pour guider le concepteur vers le patron produit le plus adéquat.

Dans la section suivante, nous abordons le choix du formalisme que nous utilisons pour représenter notre système de patrons.

2.3. Choix du formalisme

Un formalisme est la structure adoptée par le concepteur de patrons pour représenter des patrons. Dans la littérature, plusieurs formalismes ont été proposés. On peut distinguer deux classes de formalismes : les formalismes **narratifs** tels que ceux de C. Alexander [ALEXANDER, 79] et ceux de Portland [PORTLAND, 94], et les formalismes **structurés** tels que ceux de M. Fowler [FOWLER, 97] et de P. Coad [COAD, 95] pour les patrons d'analyse, d'E. Gamma [GAMMA & AL., 95] pour les patrons de conception, ou de S.W. Ambler [AMBLER, 98] pour les patrons processus. Par opposition aux formalismes narratifs par nature peu structurés et informels, le second type de formalisme offre une meilleure représentation des patrons. Ces formalismes structurés, composés chacun d'un ensemble de rubriques qui leur

est propre, sont globalement équivalents dans la mesure où ils expriment tous le triplet {problème, solution, contexte}. Ils diffèrent cependant par le nombre de rubriques proposées et le degré de détail de celles-ci.

Les formalismes structurés actuels sont dédiés soit à l'expression des patrons produit en mettant l'accent sur la représentation des solutions modèles (c'est le cas par exemple du formalisme de Gamma [GAMMA & AL., 95]), soit à l'expression des patrons processus en mettant l'accent sur la représentation des solutions démarches (c'est le cas par exemple du formalisme d'Ambler [AMBLER, 98]).

Le formalisme recherché devait permettre de conjuguer des patrons produit et processus au sein du même système de patrons. Le choix du formalisme s'est fait naturellement puisque dans le cadre des activités de notre équipe Sigma du laboratoire LSR, un travail collectif a abouti à la définition d'un formalisme structuré de représentation de patrons, *P-Sigma* (cf. annexe 1). Ce formalisme est destiné à pallier à certaines insuffisances des formalismes de patrons existants dédiés à l'ingénierie des SI. Les principaux objectifs du formalisme P-Sigma sont :

- d'exprimer une sémantique commune à la majorité des formalismes proposés dans la littérature,
- de décrire aussi bien des fragments de modèles que des fragments de démarche réutilisables, ce qui permet d'uniformiser l'expression des patrons produit et des patrons processus,
- d'expliciter l'interface de sélection des patrons afin de faciliter leur classification et leur réutilisation,
- de proposer des relations inter-patrons afin d'améliorer l'organisation des catalogues de patrons et donc d'augmenter leur taux de réutilisation.

Le formalisme est constitué de trois parties : *Interface*, *Réalisation* et *Relation*. Chaque partie regroupe un certains nombres de rubriques :

- les rubriques « Interface » permettent de classifier et de sélectionner le patron dans un catalogue de patrons,
- les rubriques « Réalisation » permettent de résoudre le problème soulevé par le patron,
- les rubriques « Relation » permettent d'organiser les patrons dans un ctalogue en définissant les relations qu'entretient un patron avec d'autres patrons.

Chaque rubrique est composée d'un ou de plusieurs champs de nature différente (texte, expression formelle, diagramme). La sémantique des différentes rubriques est présentée d'une manière détaillée dans l'annexe 1.

Pour représenter les différents patrons produit, nous adaptons la partie « Réalisation » du formalisme P-Sigma. Nous mettons en évidence trois rubriques : une rubrique « Description Comportementale » pour décrire en langage naturel le comportement général de l'architecture de coopération, et deux rubriques « Solution produit formelle » et « Solution produit semi-formelle ». Les deux dernières rubriques seront détaillées dans la section suivante (cf. section 3). La figure 37, présente le méta-modèle du formalisme P-Sigma adapté à nos besoins. Dans ce méta-modèle, nous distinguons les rubriques que seront utilisées pour représenter les patrons produit, de celles qui seront utilisées pour représenter les patrons processus, et celles qui sont communes aux deux types de patrons.

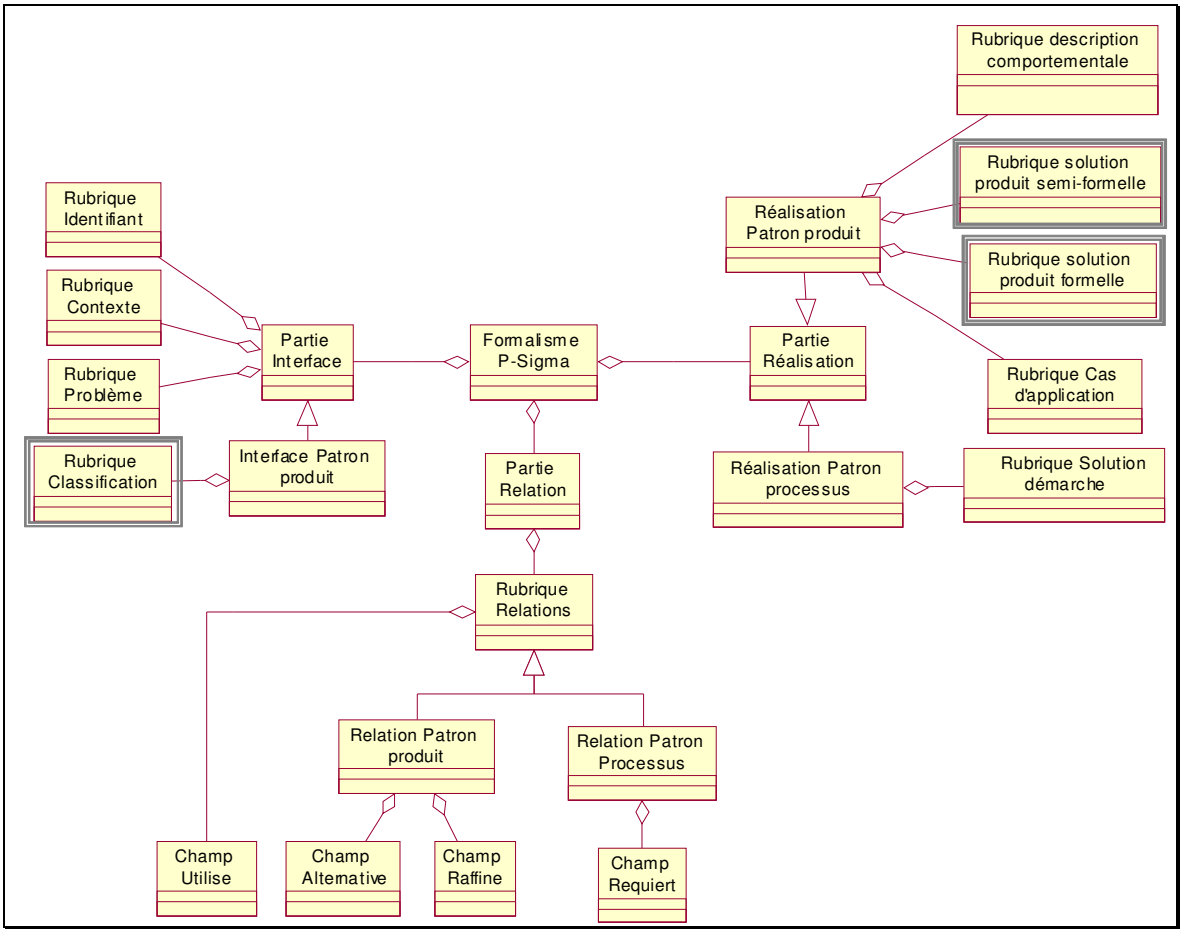


Figure 37 : Formalisme P-Sigma adapté

Dans le cadre de ce travail...
Adaptation du formalisme P-Sigma pour représenter les différents patrons produit et processus.

Dans la suite de ce chapitre, nous expliciterons les rubriques « Solution produit formelle», « Solution produit semi-formelle » et « Classification » (mises en relief dans la figure 37). Ainsi, nous présenterons dans la section suivante (section 3) les langages que nous retenons pour décrire les différents concepts d'une architecture logicielle coopérative (rubriques « Solution produit formelle» et « Solution produit semi-formelle »). La section 4 sera consacrée à l'énumération de certains critères qui permettront de classifier les différents patrons (rubrique « Classification »).

3. Les langages de modélisation choisis

3.1. UML et Acme

3.1.1. Introduction

Les méthodes formelles et semi-formelles offrent des aspects complémentaires pour la spécification des architectures logicielles : rigueur et vérification pour les uns, clarté et validation pour les autres.

Pour ce travail, le choix s'est orienté vers le langage orienté objet UML et l'ADL Acme permettant de prendre en compte des besoins spécifiques à la modélisation des architectures logicielles coopératives. Ces besoins sont au nombre de trois :

- une représentation explicite des interactions entre les systèmes composants (section 3.1.2),
- une description graphique et sémantiquement riche (section 3.1.3),
- un outil qui permet :
  - la vérification de la cohérence de la description architecturale,
  - la réutilisation et l'instanciation des différentes architectures de coopération.

Les différents outils seront présentés dans le chapitre instrumentation et validation (cf. chapitre VI).

3.1.2. Connecteur en tant qu'entité de premier degré

La notion de connecteur recouvre l'ensemble des moyens nécessaires pour assurer l'interaction entre les systèmes composants par la connexion des interfaces requises et offertes. Ces moyens peuvent être perçus à deux niveaux :

1. La spécification des propriétés des interfaces requises et des interfaces offertes peut être utilisée pour définir les connecteurs sur un plan conceptuel. Un connecteur est ainsi vu comme la spécification d'un mode de communication.
2. Dans un second temps, le connecteur est implanté par un ensemble de moyens logiciels et matériels (par exemple pour réaliser un protocole de communication).

Dans le cadre de notre travail, nous ne traitons que le premier niveau, à savoir, la spécification des propriétés des connecteurs et la modélisation des connexions entre les systèmes d'information composants.

Dans la pratique, il est recommandé de ne pas faire apparaître les traitements associés aux fonctions des connecteurs dans les systèmes composants mais d'en faire des entités bien distinctes. Ainsi, plus les interactions ont un niveau d'abstraction élevé, plus le couplage systèmes composants / interactions est faible. Or, un couplage faible favorise la réutilisation, la maintenance et l'évolution des interactions et des systèmes composants interagissants.

Ce premier besoin recensé est satisfait par le langage Acme. En effet, dans ce dernier, un connecteur correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants en définissant les règles qui gouvernent ces

interactions. Il est défini comme une entité de premier degré au même titre qu'un composant. Par exemple, un connecteur peut décrire des interactions simples de type appel de procédure ou accès à une variable partagée, mais aussi des interactions complexes telles que des protocoles d'accès à des bases de données avec gestion des transactions, la diffusion d'événements asynchrones ou encore l'échange de données sous forme de flux. Un connecteur comprend également deux parties. La première correspond à la partie visible du connecteur qui permet la description des rôles des participants à une interaction. La seconde partie correspond à la description de son implantation. Il s'agit de la mise en œuvre du protocole associé à l'interaction.

3.1.3. Un langage graphique et sémantiquement riche

Comme nous l'avons évoqué précédemment, l'utilisation conjointe de spécifications formelles et semi-formelles permet de combiner les avantages de ces deux types d'approches : notations graphiques et lisibilité des approches semi-formelles, haut niveau d'abstraction, expressivité, cohérence et moyens de vérification des approches formelles :

- D'un côté, la notation UML est devenue omniprésente en conception orientée objet et ce principalement grâce à sa convivialité et sa relative souplesse d'emploi. L'utilisation d'UML nous permet donc de profiter de ses notations graphiques compréhensibles par les différents acteurs d'un projet (même les non spécialistes). Nous retenons le diagramme de composants de la version 2.0 d'UML, nettement amélioré par rapport à ses prédécesseurs, pour spécifier "graphiquement" les solutions produit de nos patrons. Cependant, comme nous l'avons évoqué dans le chapitre précédent, UML souffre d'une critique permanente concernant son manque de sémantique formelle. Ceci pose problème pour garantir la cohérence des systèmes décrits à l'aide de plusieurs diagrammes et plus généralement pour les étapes de vérification qui suivent la modélisation de ces systèmes. Pour remédier à cela, le langage OCL a été proposé pour spécifier des invariants de classes, des contraintes, des pré et post conditions d'opérations et des pré-conditions d'activation d'événements. Mais il n'existe pas aujourd'hui d'outils associés permettant d'analyser et de vérifier l'ensemble des spécifications.
- D'un autre côté, les méthodes formelles existent maintenant depuis plusieurs décennies. Leurs principaux intérêts résident dans leur sémantique bien définie et dans leur capacité de vérifier la cohérence des systèmes spécifiés. Les ADL rendent aussi possible la description des systèmes à un haut niveau d'abstraction ce qui permet de contrôler leur complexité. L'utilisation d'Acme nous permet alors de profiter du pouvoir d'expression des ADL. Acme permet en particulier de décrire des types (ou styles) d'architecture et de les instancier.

Dans le cadre de ce travail...
UML 2.0 pour une représentation intuitive et "compréhensible". Acme pour une spécification détaillée et "instanciable".

3.1.4. Conclusion

Dans cette section, nous avons procédé au choix de méthodes et de langages que nous allons utiliser pour spécifier des architectures de coopération. Ce choix était guidé et argumenté par

différents critères qui nous semblent pertinents pour obtenir des spécifications claires, compréhensibles, cohérentes et réutilisables. Par la suite, nous présentons au sein d'un méta-modèle les différents concepts que nous avons recensés pour décrire des architectures logicielles coopératives.

3.2. Méta-modèle de la solution

Le méta-modèle que nous proposons restreint et unifie les concepts d'UML et d'Acme à ceux essentiels pour décrire l'aspect structurel d'une architecture de coopération. Il permet d'avoir un niveau d'abstraction indépendant des langages de modélisation utilisés (UML, Acme, etc.). Etant donné qu'Acme est un ADL pivot [GARLAN & AL., 1997A], nous reprenons ses différents concepts. Le méta-modèle est organisé en deux paquetages qui reflètent les niveaux de réutilisation dans une approche de développement par réutilisation d'architecture logicielle. Le paquetage *Type d'architecture logicielle* et le paquetage *Architecture logicielle.* Ces deux paquetages sont liés par une relation de dépendance. En effet, la classe *Entité* dans *Architecture logicielle* requiert les services de la classe *Type entité* dans *Type architecture logicielle*.

Le méta-modèle que nous proposons permet de décrire les concepts généraux des architectures logicielles dans le but de promouvoir leur réutilisation.

Pour illustrer les différentes notions du méta-modèle, nous prenons comme exemple le patron *ClientServeur_SICo*. Ce dernier formalise la coopération entre deux systèmes d'information qui communiquent en mode client / serveur.

3.2.1. Le paquetage « Type d'architectures logicielles »

Le paquetage *Type d'architectures logicielles* (cf. figure 38) modélise les *types* (de composants, connecteurs, ports et rôles) réutilisables pour la construction des architectures logicielles.

Une famille d'architectures est composée de différents types d'entités ainsi que de contraintes et de propriétés qui portent sur toute l'architecture. Une famille d'architectures peut réutiliser les différents types d'entités définis dans une famille d'architectures existante. Un type d'entités peut être un type de composants, de connecteurs, de ports ou de rôles, et possède elle-même des propriétés et des contraintes. Un type de composants ou de connecteurs est défini par ses interfaces appelées *port* pour les composants et *rôle* pour les connecteurs. Ces ports ou ces rôles modélisent les types d'interactions d'un composant ou connecteur avec son environnement. Tous ces éléments sont regroupés dans le paquetage « Type d'architectures logicielles ».

Dans la figure 39 nous présentons le type d'architectures du patron *ClientServeur_SICo*. Ce dernier est composé de deux types de composants : le composant *TcompClient* qui accède aux services proposés par le composant *TcompServeur*. La communication entre ces deux types de composants est assurée par un connecteur de type *Trpc (Remote Procedure Call)*.

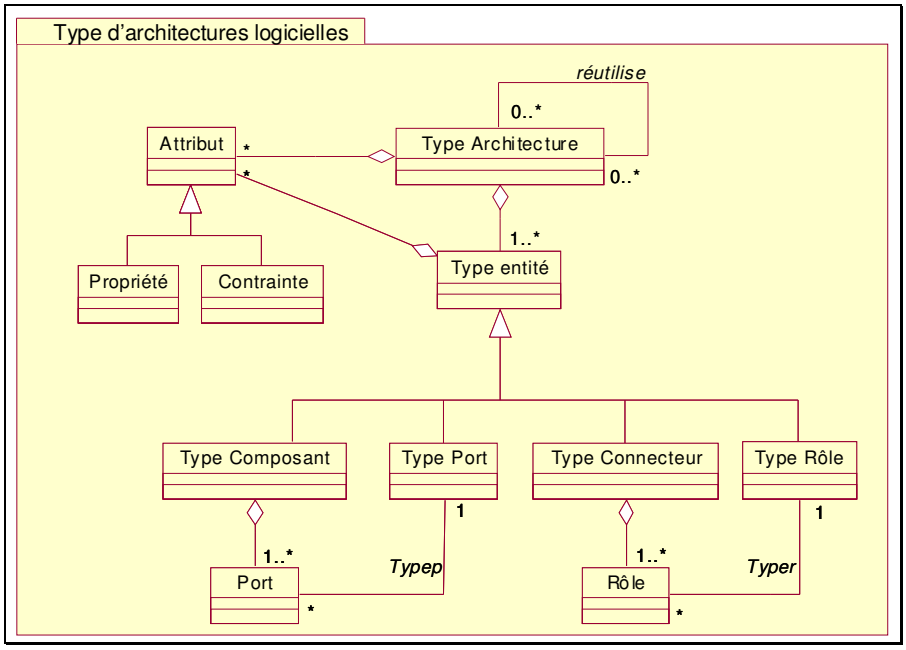


Figure 38. Paquetage « Type d'architectures logicielles »

Pour décrire la solution modèle de notre patron, nous utilisons, comme nous l'avons évoqué auparavant, le diagramme de composants d'UML 2.0 et l'ADL Acme. Toutefois, les propriétés et les contraintes sont exprimées par des notes en language naturel en UML et de manière formelle en Acme. En effet, ce dernier dispose de concepts permettant d'exprimer des propriétés et des contraintes vérifiables par des outils (AcmeStudio). Ces propriétés et contraintes peuvent être exprimées sur chaque entité ou sur le comportement global de l'architecture.

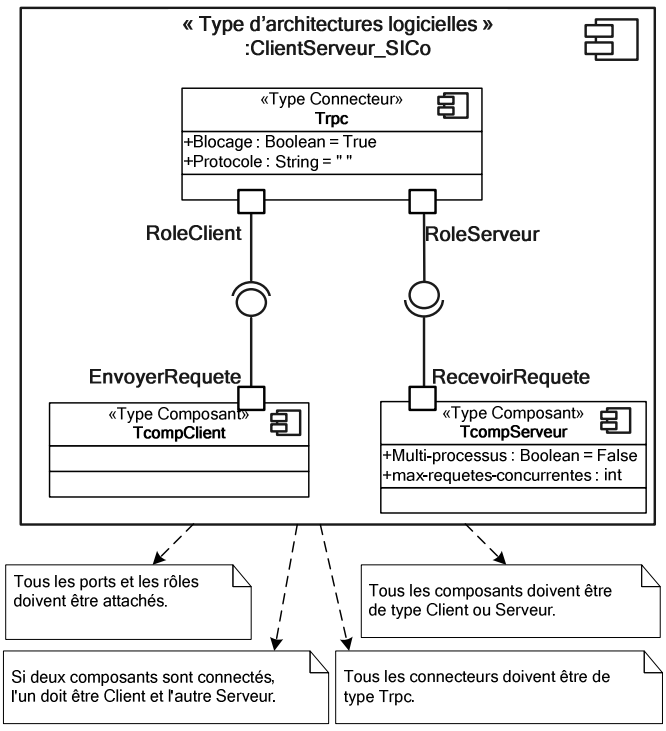


Figure 39 : Diagramme UML du patron « ClientServeur_SICo »

```
Family ClientServeur_SICo = {

   Description Acme du diagramme UML

   // Déclaration des ports et des rôles
   Port Type TportClient = {…}
   Port Type TportServeur = {…}
   Role Type TroleClient = {…}
   Role Type TroleServeur = {…}

   Component Type TcompClient = {
      Port EnvoyerRequete : TportClient =  new TportClient;
   }

   Component Type TcompServeur = {
      Port recevoirRequete : TportServeur =  new TportServeur;
      Property multiProcessus : boolean << default : boolean = false;  >>;
      Property max-requetes-concurrentes : int;
   }

   Connector Type Trpc = {
      Role RoleClient : TroleClient =  new TroleClient;
      Role RoleServeur : TroleServeur =  new TroleServeur;
      Property blocage : boolean << default : boolean = true;  >>;
      Property protocole : string << default : string = "";  >>;
      …
   }

   Contraintes et propriétés exprimées en Acme

   // Si deux composants sont connectés, l'un doit être client et l'autre serveur
   invariant Forall c1 : component in self.components |
      Forall c2 : component in self.components |
         connected(c1, c2) -> (declaresType(c1, TcompClient) AND declaresType(c2,
         TcompServeur)) OR (declaresType(c1, TcompServeur) AND declaresType(c2,
         TcompClient))  <<label : string = "Toutes les pairs de composants connectées sont
         Client et Serveur.";errMsg : string = "Pair(s) de composants détectée(s) qui ne
         respecte(nt) pas la règle Client avec Serveur!";>>;
…

}
```

Figure 40 : Description Acme du patron « ClientServeur_SICo »

3.2.2. Le paquetage « Architecture logicielle »

Le paquetage *Architecture logicielle* (cf. figure 41) permet de décrire des concepts spécifiques à une architecture bien déterminée en s'appuyant sur un ou plusieurs types d'architectures décrits dans le paquetage *Type d'architecture logicielle*. Outre le concept d'entité (composant, connecteur, port et rôle), il décrit aussi le concept d'*attachement*, de *représentation* et de *binding*.

Pour modéliser l'association entre une classe entité et une classe type-entité (ex : entre la classe *composant* et la classe *Typecomposant*), nous avons utilisé le patron *item-description* de P. Coad [COAD, 92]. Ce patron permet la gestion d'informations de différents niveaux. Ce patron comporte deux classes : d'une part, la classe *Classe* qui indique les propriétés et le comportement des objets de la classe ; d'autre part, la classe *Classe-*

Description, où l'on précise les propriétés d'une partition des objets de la classe *Classe*. Dans notre cas, *Classe-Description* correspond à *Type entité* et *Classe* à *Entité.* De même, nous avons utilisé le patron *composite* [GAMMA & AL., 95] pour modéliser la construction abstraite de l'entité. Les instanciations concrètes de la méta-classe *Entité* (composant, connecteur, port et rôle) peuvent être primitives (boîte opaque indivisible) et/ou composites. Chaque entité possède un ensemble d'opérations et de contraintes. Les composants et les connecteurs composites possèdent, en plus, des *représentations*. La relation entre une entité composite (composant ou connecteur) et sa structure interne (sa représentation) est établie par le biais de la notion de *Binding*. Enfin, les ports et les rôles, respectivement, des composants et des connecteurs, sont liés grâce à la notion d'*attachement*.

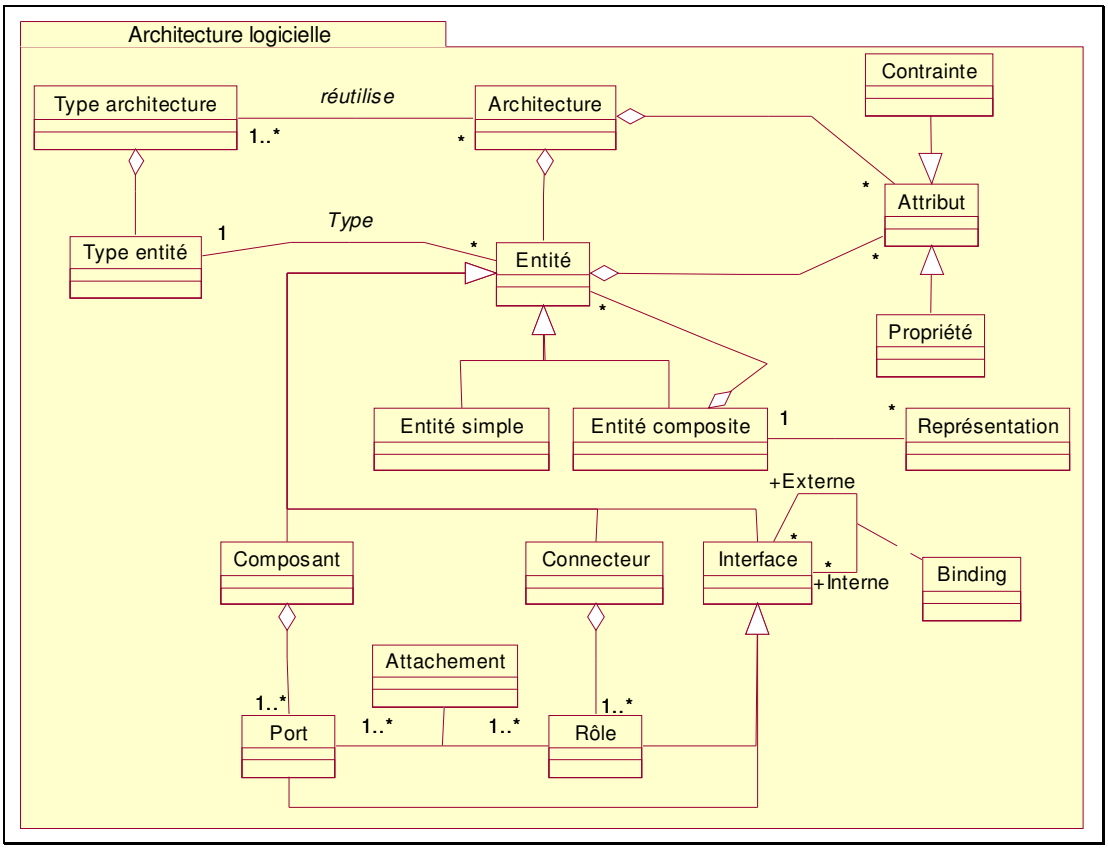


Figure 41. Paquetage « Architecture logicielle»

Dans la figure 42 et la figure 43, nous présentons une instanciation du type d'architectures *ClientServeur_SICo* (cf. figure 41). A partir d'un type d'architectures, l'instanciation d'une architecture est réalisée grâce à l'héritage et au raffinement. En effet, dans un premier temps, les différentes entités héritent les propriétés et les contraintes des types d'entités ; par la suite, il est possible d'en rajouter d'autres qui leur sont spécifiques.

L'instanciation des types d'architectures et des architectures de coopération est un pas vers la réutilisation. Cependant, leur réutilisation peut être améliorée par une bonne documentation des solutions.

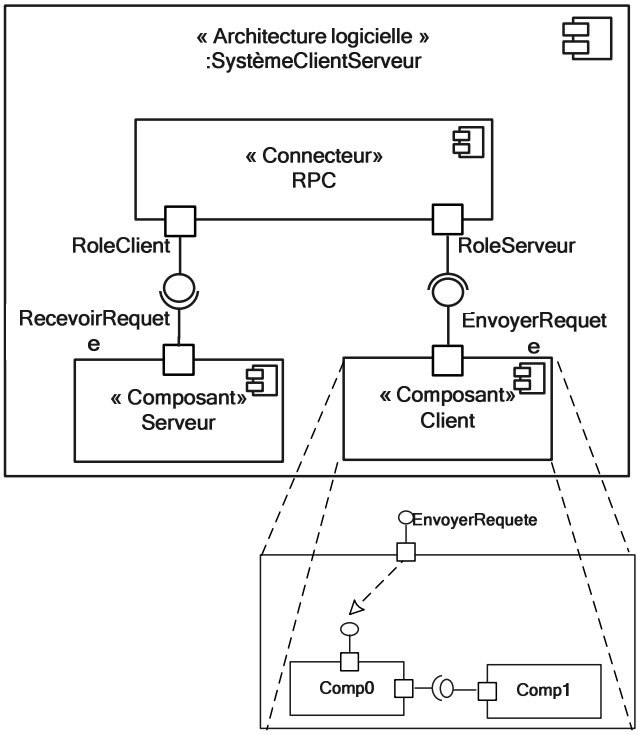


Figure 42. Diagramme UML du « SystèmeClientServeur »

```
import $AS_PROJECT_PATH\families\ClientServeur.acme;
System SystemClientServeur : ClientServeur =  new ClientServeur extended with {
  Component Client : TcompClient =  new TcompClient extended with {
    Representation SysteмeClient = {
      System SysteмeClient = {
        Component comp0 = {
          Port p0;
          Port EnvoyerRequete;
        };
        Component comp1 = {
          Port p0;
        };
        Connector conn0 = {
          Role r1;
          Role r2;
        };
        Attachment comp1.p0 to conn0.r1;
        Attachment comp0.p0 to conn0.r2;
      };

      Bindings {
        comp0.EnvoyerRequete to Client.EnvoyerRequete;
      }
    };
  };

  Component Serveur : TcompServeur =  new TcompServeur extended with {…};
  Connector RPC : Trpc =  new Trpc extended with {…};

  Attachment Serveur.recevoirRequete to RPC.RoleServeur;
```

Héritage

Raffinement

```
    Attachment Client.EnvoyerRequete to RPC.RoleClient;
};
```

Figure 43. Description Acme du « SystèmeClientServeur »

Dans le cadre de ce travail...
Restreindre les concepts d'UML 2.0 et d'Acme par un méta-modèle décrivant les concepts essentiels pour la description structurelle des architectures logicielles en général et des architectures logicielles coopératives en particulier.

3.2.3. Conclusion

Le méta-modèle que nous avons proposé dans cette section permet d'effectuer une description structurelle de n'importe quelle architecture logicielle et entre autre les architectures logicielles coopératives. Telle que nous l'avons présentée, une architecture peut être vue comme un graphe acyclique de types de composites, types de composants et types de ports. La structure de graphe est plus adaptée qu'une structure arborescente car un type de ports peut être réutilisé au sein de plusieurs types de composants. De même un type de composants peut être le fils de plusieurs types de composites.

Pour classifier les différentes architectures de coopération dans notre système de patrons, nous proposons un ensemble de critères pour caractériser les composants et les connecteurs au sein d'une architecture de coopération (section 4).

4. Critères de classification

Dans le paquetage *Type d'architectures logicielles* nous avons défini la méta-classe *Propriété* qui capitalise les propriétés pouvant être définies sur l'architecture globale ou bien sur chaque entité de l'architecture. Dans cette section, nous allons définir une liste de propriétés (ou critères) que nous allons appliquer sur des architectures de coopération. Cette liste permet de caractériser les composants (cf. section 4.1) et les connecteurs (cf. section 4.2) au sein d'une architecture de coopération (cf. figure 44). Ces critères faciliteront par la suite la recherche et la sélection d'une architecture de coopération dans notre système de patrons afin d'aider le concepteur.

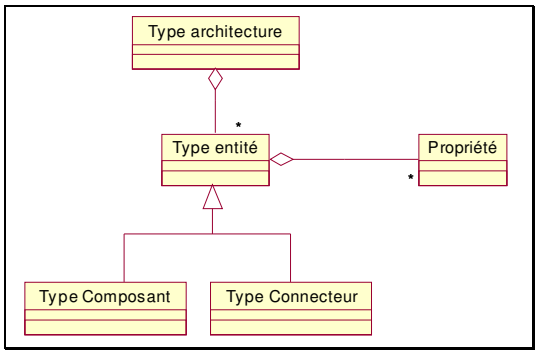


Figure 44 : Vue partielle du paquetage "Type d'architectures logicielles"

Les critères que nous présentons par la suite sont le résultat de l'étude de l'existant que nous avons menée et présentée dans les chapitres deux et trois. Ils constituent les critères de base pour mettre en place un guide méthodologique propre au domaine de la coopération. Ils ne sont, de ce fait, pas exhaustifs et peuvent être enrichis.

4.1. Systèmes composants

4.1.1. Ouverture

L'ouverture caractérise le niveau de transparence du Système Composant (SC), selon que son adaptation entraîne ou non une modification de sa structure interne. Sur le plan conceptuel, nous pouvons différencier les composants *boîte noire* des composants *boîte blanche*.

Les composants *boîte blanche* (cf. figure 45) sont totalement transparents. Leur spécification (et implantation) est accessible et modifiable. L'avantage majeur de la transparence est que l'accès aux différents SC est réalisé de manière ciblée et qu'il est possible d'adapter leur spécification.

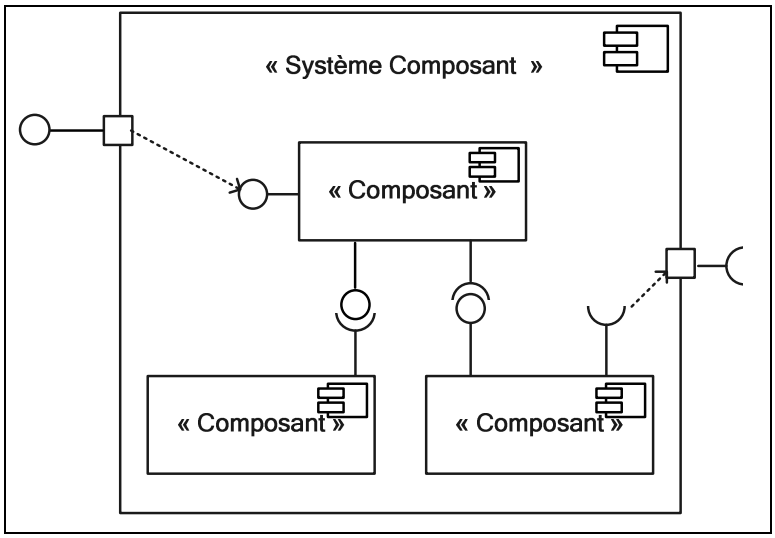


Figure 45 : Système Composant "Boîte blanche"

A l'inverse, la structure interne des composants *boîte noire* (cf. figure 46) n'est ni visible ni modifiable. Au niveau implantation, ce sont des codes compilés ou binaires. Leurs fonctionnalités ne sont acessibles qu'au moyen de leurs interfaces publiques et ne peuvent pas être modifiées directement. Pour étendre ou redéfinir certaines de leurs fonctionnalités, il devient nécessaire d'utiliser des adaptateurs.

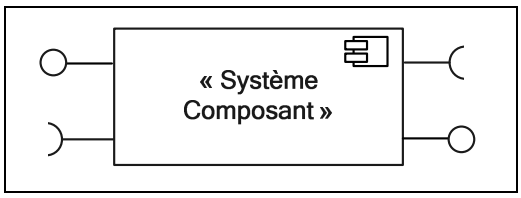


Figure 46 : Système Composant "Boîte noire"

4.1.2. Partage de données

La coopération entre les SC peut être orientée données ou/et processus. Ainsi, selon le cas, les SC peuvent ou non partager ou échanger des données. Les données d'un SC sont alors visibles et/ou accessibles par les autres SC. Nous distinguerons deux approches de coopération centrées sur les données selon que la coopération repose ou non sur un schéma global d'intégration de données.

- Schéma global (figure 47) : Cette architecture repose sur un schéma global qui combine les informations accessibles dans les différentes sources de données. Il assure un accès transparent aux différentes données des SC.

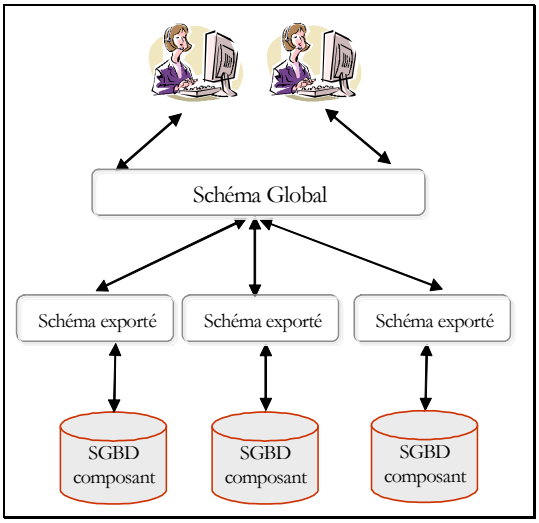


Figure 47 : Architecture avec schéma global

- Absence d'un schéma global (figure 48) : Dans cette architecture, les données des différents SC sont accessibles grâce à un langage de haut niveau qui permet d'interroger directement les bases de données concernées par une requête.

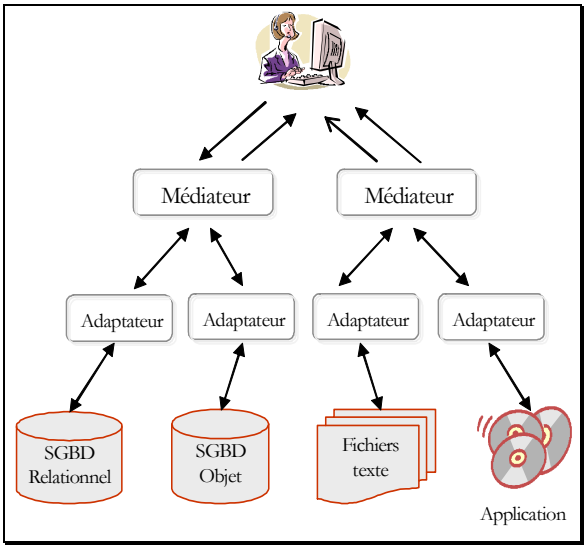


Figure 48 : Architecture sans schéma global

4.2. Les connecteurs

Pour réaliser une tâche coopérative, les systèmes composants (SC) communiquent en échangeant des données et des messages. Les types de connecteurs utilisés pour réaliser cet échange sont variés et peuvent être simples (ex : appel de procédure) ou complexes (ex : protocoles d'accès à des bases de données). De même, les propriétés liées à ces connecteurs sont diversifiées. En effet, la communication entre les SC peut s'effectuer en mode synchrone ou asynchrone, d'une manière implicite ou explicite, etc. C'est sur ce dernier aspect que porte notre travail. Cette section présente différents critères qui caractérisent la communication entre les SC. Ceux-ci sont présentés dans la figure 49 et seront détaillés par la suite.

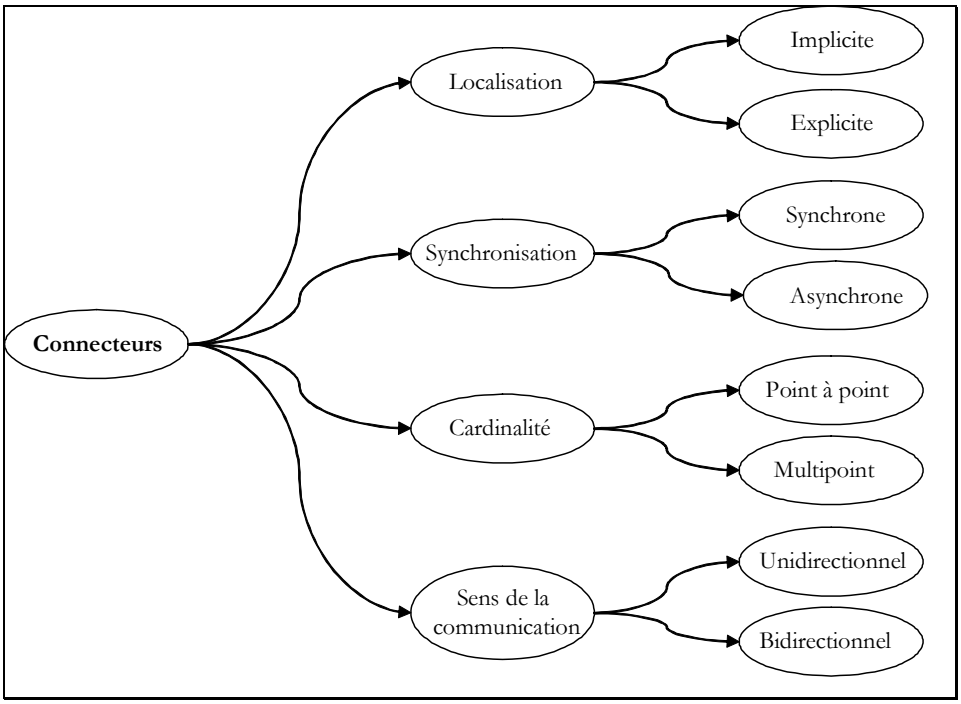


Figure 49 : Critères de classification des connecteurs

4.2.1. Localisation

Les systèmes composants dans un SICo peuvent coopérer tout en gardant leur anonymat. Dans ce cas, le SC émetteur ne connaît pas la localisation (ou l'identité) du SC destinataire et ne peut pas invoquer directement une procédure. La communication, dans ce genre de système, est basée sur l'invocation *implicite* des événements (cf. figure 50). Dans une communication de type invocation implicite, les SC ne s'appellent pas directement. Le SC émetteur doit annoncer (ou émettre) un ou plusieurs événements au gestionnaire des événements. Pour participer à la communication, les autres SC du système doivent s'abonner et enregistrer un intérêt pour un événement en lui associant une procédure. Quand un événement est annoncé, le système appelle toutes les procédures associées.

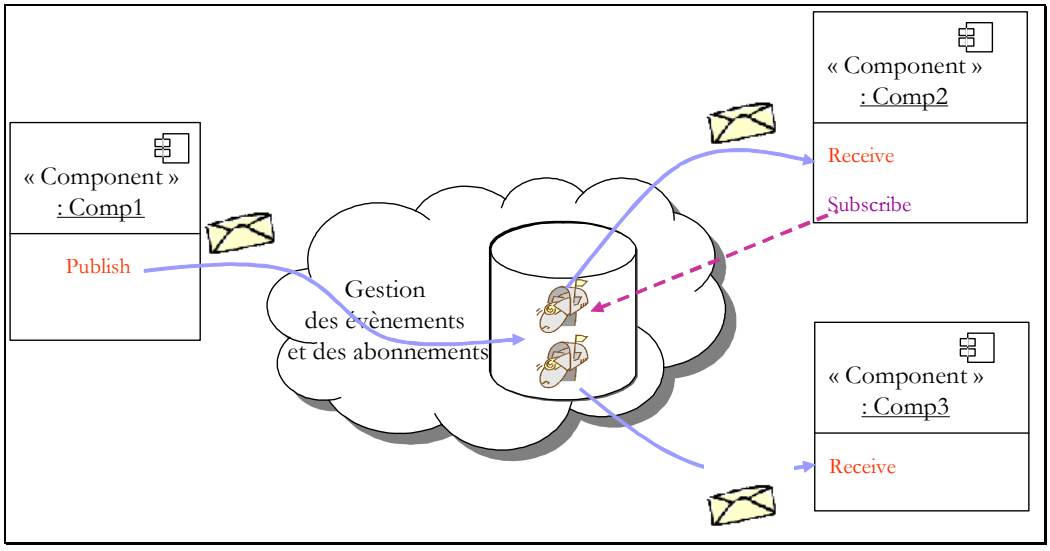


Figure 50 : Communication implicite entre Systèmes Composants

Dans de nombreux cas, les composants appelleront directement d'autres composants, configuration que l'on pourrait qualifier d'appel (ou invocation) *explicite* (cf. figure 51). Dans une communication de type invocation explicite, la communication entre les composants se fait par un appel direct de procédure.

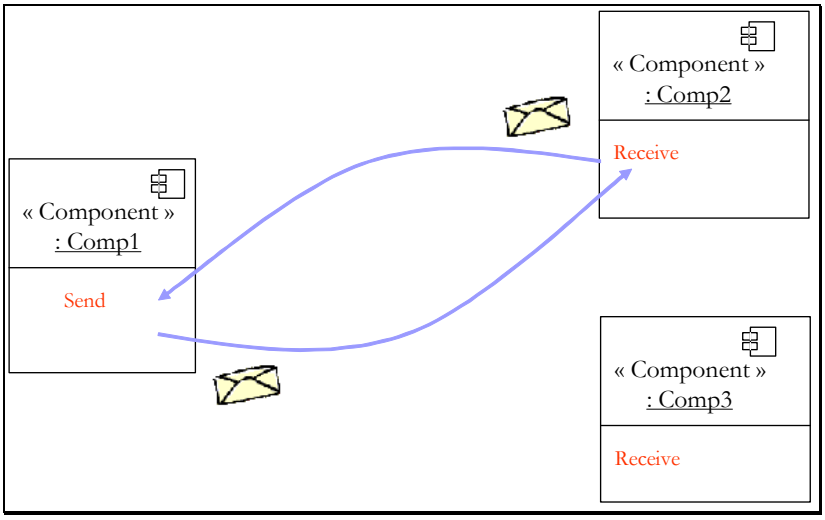


Figure 51 : Communication explicite entre Systèmes Composants

4.2.2. Synchronisation

Le type de communication entre les SC (invocation d'événements, envoi de messages, appel de procédures, etc), peut être réalisé en mode *synchrone* ou *asynchrone*.

Dans une communication synchrone (cf. figure 52), l'émetteur reste bloqué jusqu'à ce qu'une réponse lui soit envoyée.

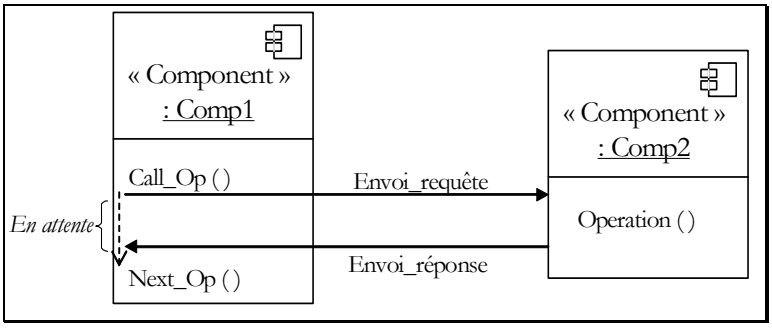


Figure 52 : Communication synchrone

Dans le deuxième cas, à savoir la communication asynchrone (cf. figure 53), le SC émetteur envoie son message ou sa requête, et continue son exécution sans attendre de réponse. Si une réponse doit arriver, elle parviendra en différé.

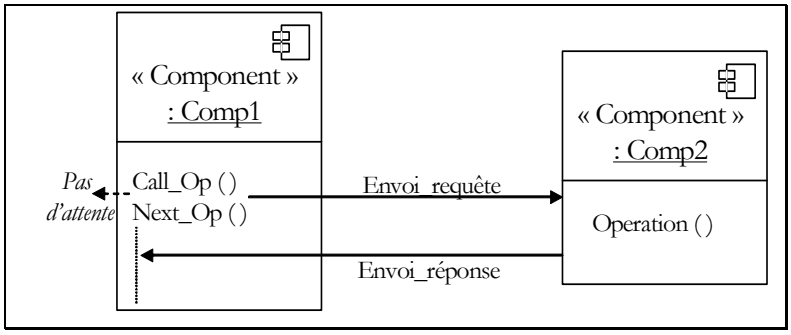


Figure 53 : Communication asynchrone

4.2.3. Cardinalité

Selon le nombre des Systèmes Composants impliqués dans la communication, nous pouvons distinguer deux types de communication : la communication *point à point* s'il y a un émetteur et un récepteur ; sinon il s'agit d'une communication *multi-point*.

Dans une communication en point à point (ou unicast) (cf. figure 54), un SC émetteur connaît la liste des destinataires et envoie le message explicitement à chacun des SC concernés et à aucun autre. La communication ne concerne alors que deux SC en même temps.

Dans le cas de la communication multi-point (ou par diffusin générale), un SC envoie le message simultanément à tous les autres SC participants de la coopération. Ceux qui ne sont pas concernés (qui n'ont pas souscrit) ignorent le message (cf. figure 50).

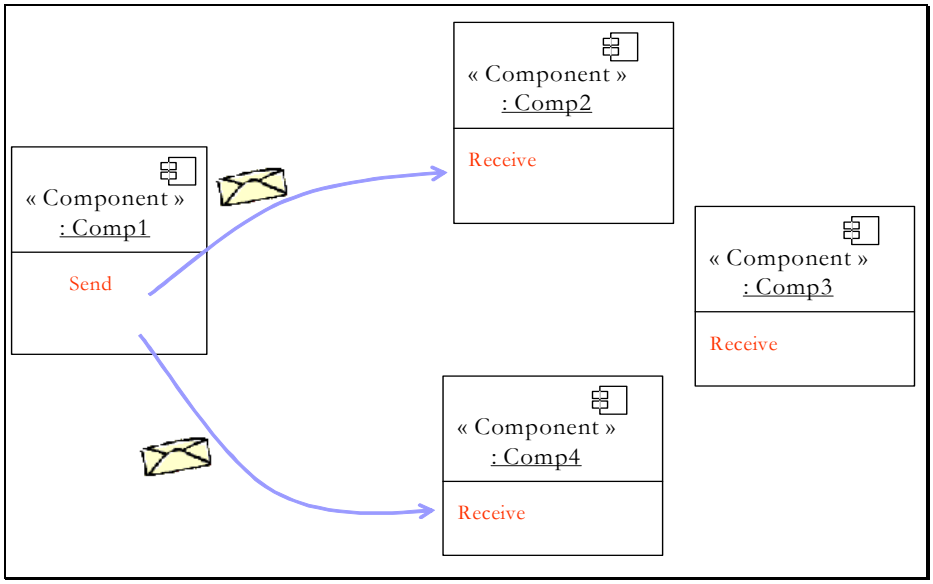


Figure 54 : Communication "point à point" entre Systèmes Composants

4.2.4. Sens de la communication

Lors de l'élaboration d'un système de coopération, il convient d'analyser le sens des flux de données ou de commandes entre les systèmes composants. La communication entre les deux SC peut être *unidirectionnelle* ou *bidirectionnelle*, c'est-à-dire dans un sens ou dans les deux sens.

Dans un SICo basé sur une communication unidirectionnelle (cf. figure 55), certains SC ne peuvent que recevoir des ordres et des données (pour affichage par exemple), d'autres qu'en émettre (exemple d'une application de saisie). Le SC émetteur continue son exécution après avoir envoyé la requête au composant destinataire (mode asynchrone). Ce dernier ne retourne aucune réponse au composant émetteur. Ce mode de communication est inadéquat pour les systèmes demandant un degré de fiabilité élevé parce que le composant émetteur ne peut pas savoir si son appel a réussi ou pas.

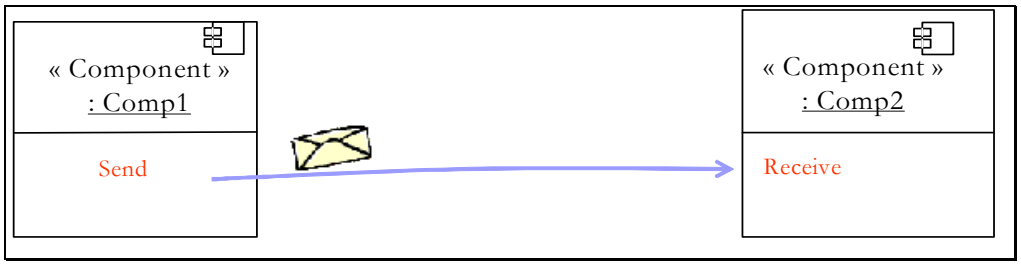


Figure 55 : Communication unidirectionnelle entre Systèmes Composants

Dans le mode bidirectionnel de communication, les SC peuvent émettre et recevoir des données et des traitements (cf. figure 51). La communication peut ainsi être en mode synchrone ou asynchrone.

4.3. Conclusion

L'ensemble des critères ou propriétés présenté n'est pas exaustif et peut être raffiné et augmenté. Les propriétés que nous avons définies ont porté sur les connecteurs et les composants séparément. La combinaison de ces critères permet d'obtenir les propriétés de l'architecture de coopération. Nous appelons cette combinaison mode de coopération. Dans la section suivante nous présentons les différents modes de coopération que nous avons retenus.

5. Modes de coopération

Dans cette section nous proposons une classification de SICo basée sur les différents modes de coopération (cf. figure 58) qui peuvent exister entre les systèmes d'information composants. Le recensement de ces modes est basé sur la combinaison des divers critères de coopération (cf. figure 57) énumérés dans la section précédente. Comme nous l'avons évoqué, ces derniers ne sont pas exaustifs et peuvent être améliorés, entre autre, en ajoutant d'autres critères. Dans ce cas, les modes de coopération qui en résultent doivent évoluer à leur tour.

Cependant, certaines combinaisons de valeurs de critères ne correspondent à aucun mode de coopération possible (cf. figure 56). En effet, la nature intrinsèque d'un critère peut rendre illogique sa combinaison avec un autre critère dont la nature est antagoniste. Par exemple, un connecteur ne peut pas être à la fois *unidirectionnel* et *synchrone* puisque ce dernier nécessite obligatoirement une connexion bidirectionnelle entre système composants.

Néanmoins, si nous tenons compte des différentes contraintes, la combinaison des critères de coopération donne lieu à de nombreuses combinaisons que nous présentons dans la figure 57.

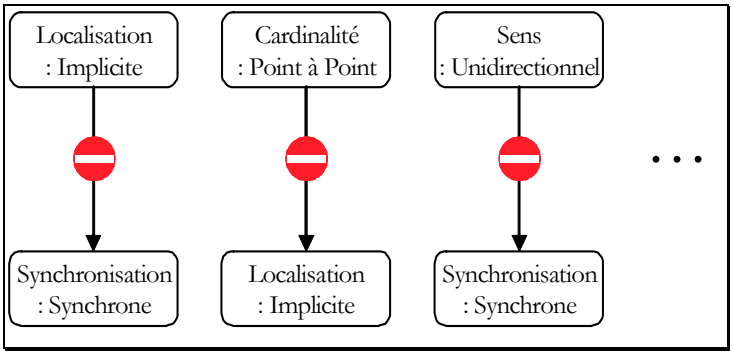


Figure 56 : Quelques combinaisons exclues

Actuellement, chaque mode de coopération peut être mis en place par plusieurs architectures de coopération. Seule la rubrique « Problème » permet de différencier deux architectures voisines. Inversement, une architecture coopérative peut aussi répondre à plusieurs modes de coopération. Pour de futurs travaux, un raffinement de ces derniers peut s'avérer nécessaire pour aboutir à une relation un-à-un entre mode et architecture de coopération.

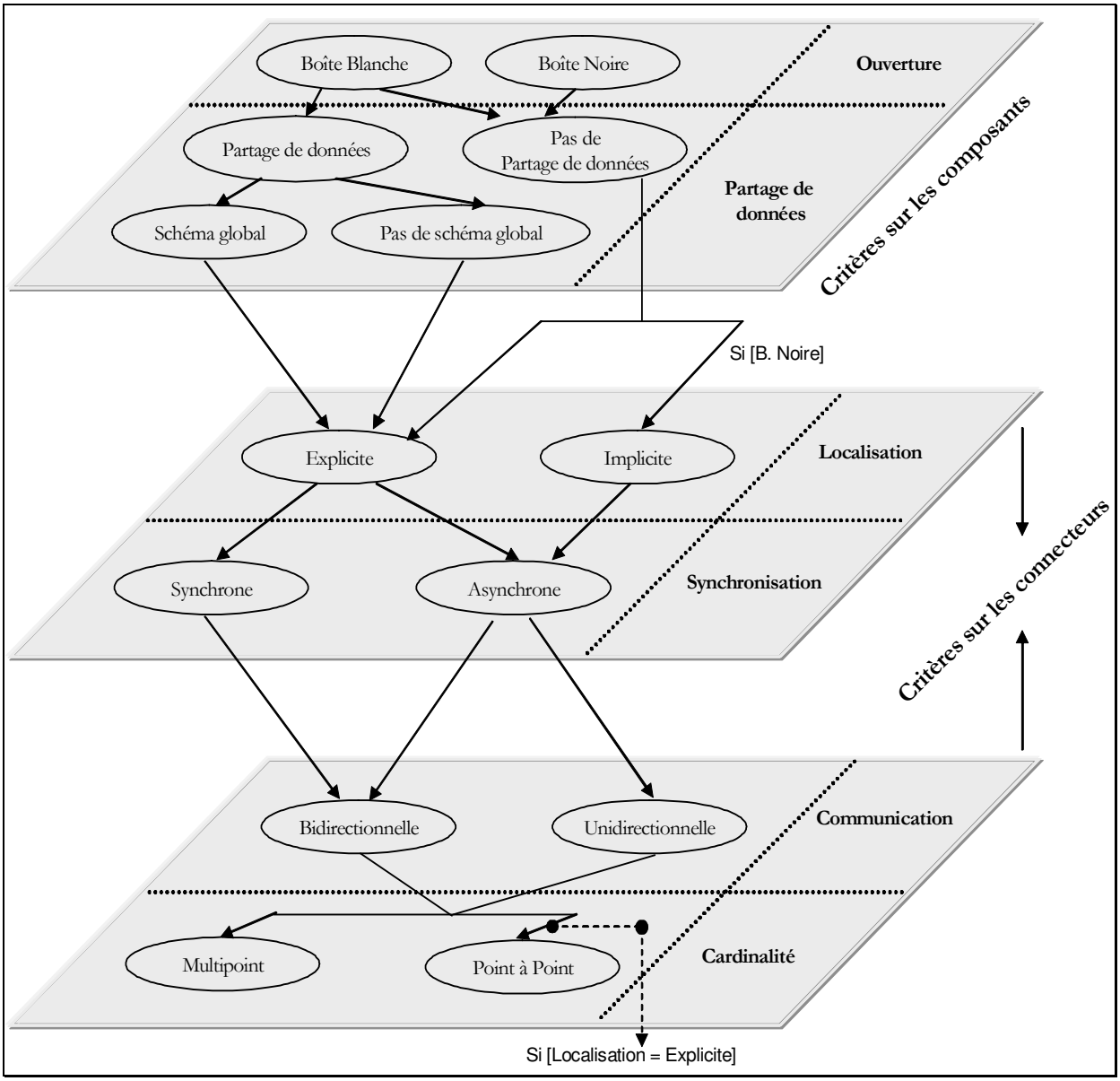


Figure 57 : Graphe des coopérations pertinentes

Les différentes combinaisons que nous avons recensées aboutissent à vingt-huit modes de coopérations types possibles. Ces modes de coopération sont présentés dans la figure 58.

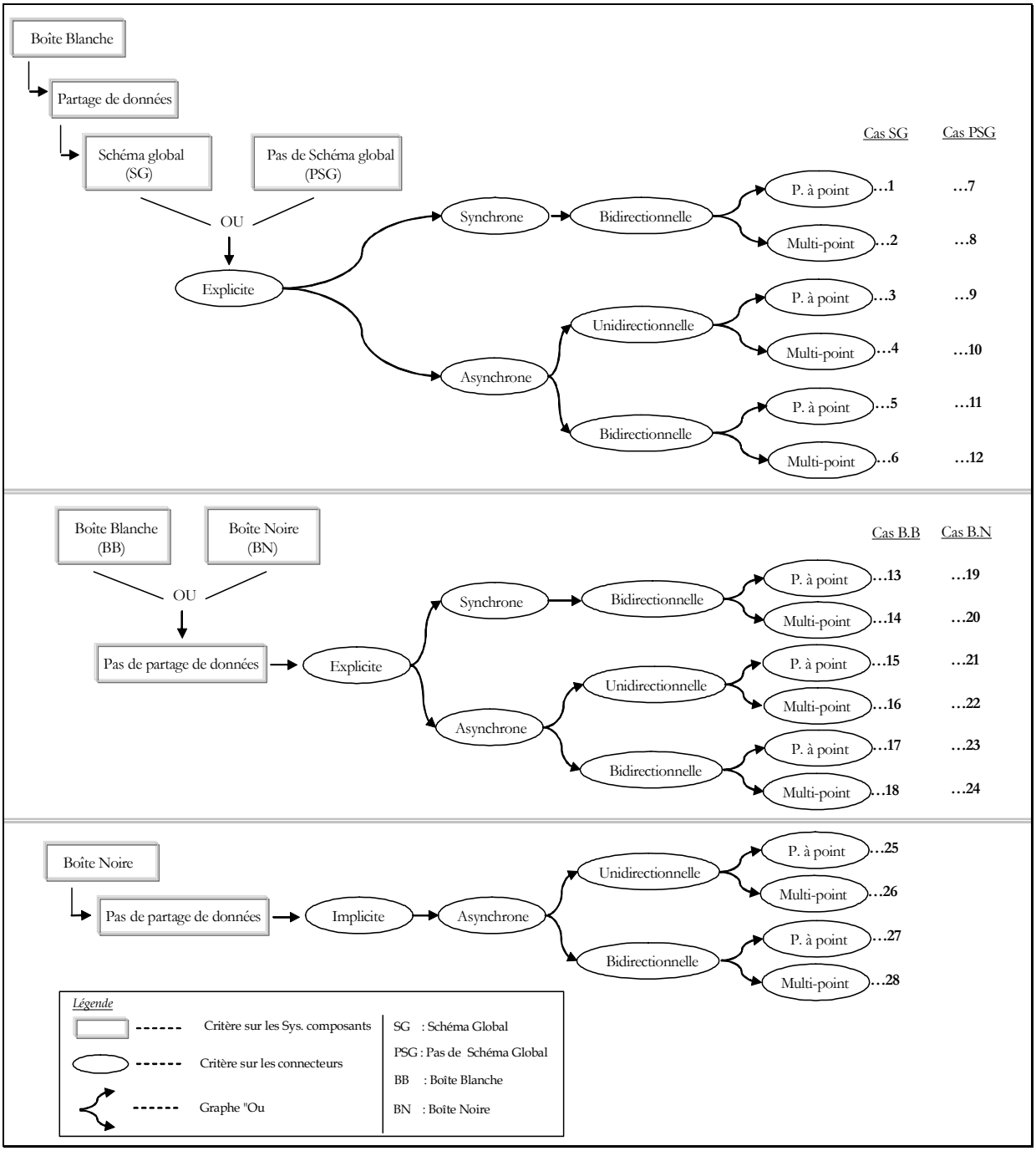


Figure 58 : Modes de coopération types

6. Exemple de patron produit formalisé

Dans ce chapitre, nous avons présenté et argumenté le choix des différents éléments qui composeront la solution. La méthode d'ingénierie que nous proposons repose sur un système de patrons constitué de patrons produit et de patrons processus. La solution modèle des patrons produit sera exprimée conjointement avec le langage semi-formel UML et l'ADL Acme. Les patrons processus reposent sur les différents modes de coopération que nous avons identifiés dans la section précédente. Pour documenter les différents patrons, nous utilisons le formalisme P-Sigma.

Dans cette section, nous présentons un exemple de patron produit formalisé. Pour cela, nous reprenons le type d'architecture « ClientServeur_SICo ». L'intégralité du système de patrons sera présentée dans le chapitre suivant.

Identifiant	**ClientServeur_SICo**
Classification	*Divers modes de coopération possibles :* 1) Boîte Blanche ∩ Pas de Partage de Données ∩ Explicite ∩ Synchrone ∩ Bidirectionnelle ∩ Point à point 2) Boîte Blanche ∩ Pas de Partage de Données ∩ Explicite ∩ Asynchrone ∩ Bidirectionnelle ∩ Point à point 3) Boîte Noire ∩ Pas de Partage de Données ∩ Explicite ∩ Synchrone ∩ Bidirectionnelle ∩ Point à point 4) Boîte Noire ∩ Pas de Partage de Données ∩ Explicite ∩ Asynchrone ∩ Bidirectionnelle ∩ Point à point *Formule simplifiée :* Pas de Partage de Données ∩ Explicite ∩ Bidirectionnelle ∩ Point à point
Contexte	Ce patron peut nécessiter l'utilisation du patron « Adaptateur_SICo »[3] pour adapter les services rendus par le serveur aux besoins des clients.
Problème	L'architecture Client/Serveur réalise le comportement coopératif sans distinction, a priori, entre les systèmes composants. En effet, un système composant peut être, indifféremment, client et/ou serveur. L'interaction est généralement synchrone mais peut être réalisée en mode asynchrone.
Description comportementale	La coopération entre client et serveur se traduit par un échange de services. Le système composant client invoque une procédure distante (processus serveur). La procédure distante s'exécute et renvoie le résultat au client. L'échange est assuré grâce au connecteur de type "Appel de Procédure à Distance". La procédure distante peut être simple ou complexe et est traitée unitairement en mode synchrone ou asynchrone. Pour une demande de service donnée, le client et le serveur ne peuvent pas intervertir leur rôle. Cependant, pour une autre demande de service, le client peut jouer le rôle du serveur et vice-versa.

[3] Le patron « Adaptateur-SICo » sera présenté dans le chapitre suivant qui sera dédié à la présentation du système de patrons.

Solution modèle

Semi-formelle

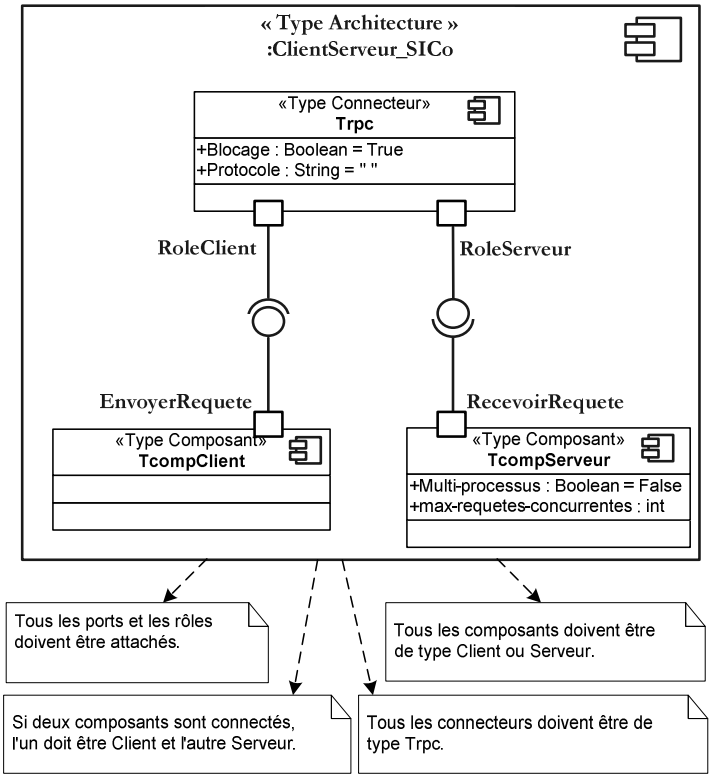
Systèmes composants :

TcompClient : système composant Client qui initie l'échange.

TcompServeur : système composant Serveur qui est à l'écoute d'une requête cliente éventuelle.

Connecteur :

Trpc : connecteur de type RPC (Remote Procedure Call).



Solution modèle

formelle

```
Family ClientServeur_SICo = {

  // Déclaration des ports et des rôles. Chaque port / rôle doit être attaché
  Port Type TportClient = {
    heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg :
    string = "Ce port doit être attaché!";>>;
  }

  Port Type TportServeur = {
    heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg :
    string = "Ce port doit être attaché!";>>;
  }

  // Rôlé côté composant client
  Role Type TroleClient = {
    invariant size(self.attachedPorts) >= 1<<label : string = "Ce rôle est attaché.";errMsg :
    string = "Ce rôle doit être attaché!";>>;
  }

  // Rôle côté composant serveur
  Role Type TroleServeur = {
    invariant size(self.attachedPorts) >= 1<<label : string = "Ce rôle est attaché.";errMsg :
    string = "Ce rôle doit être attaché!";>>;
  }
  Component Type TcompClient = {
```

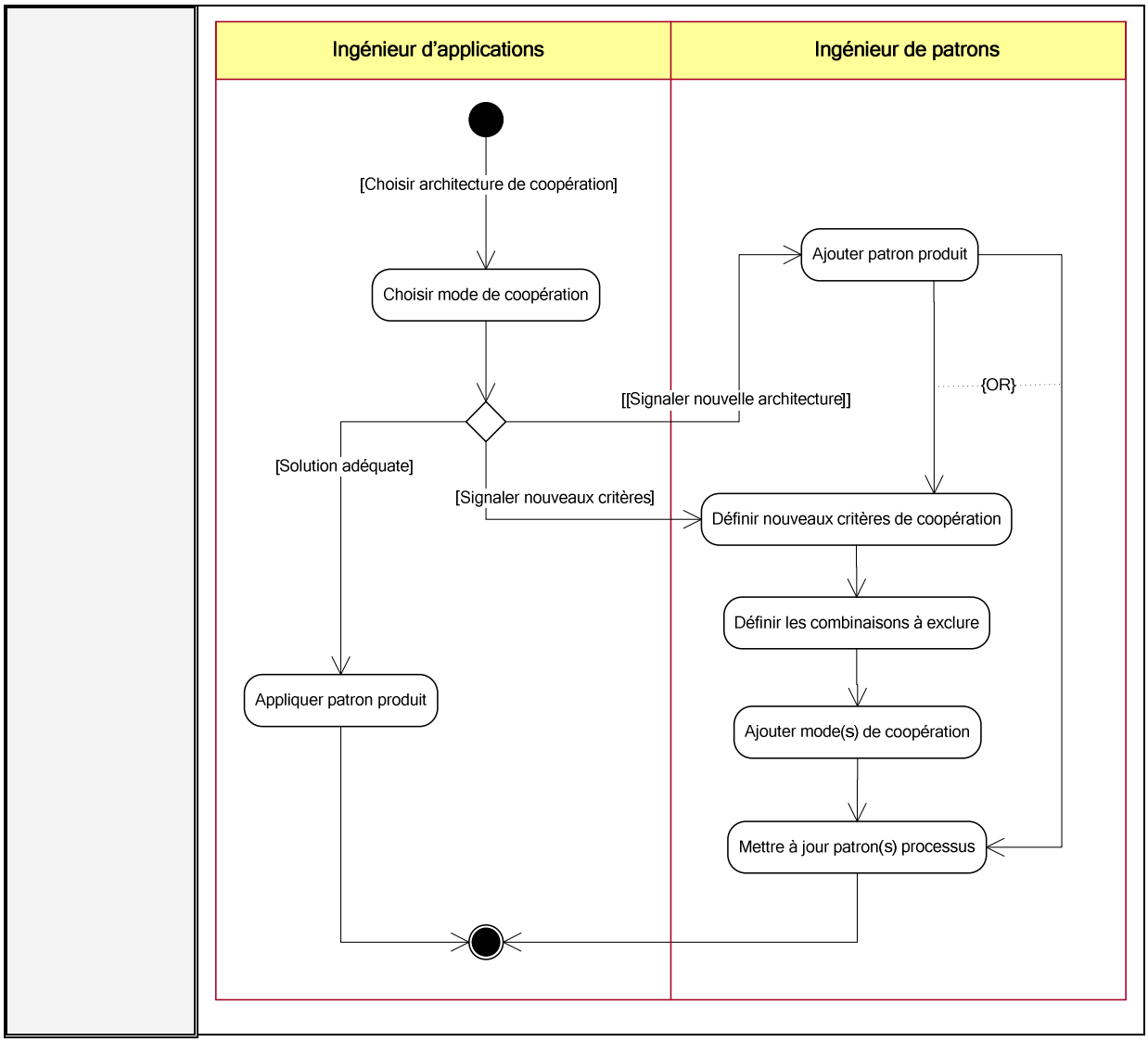
	Port EnvoyerRequete : TportClient = new TportClient;
	}
	
	Component Type TcompServeur = {
	Port recevoirRequete : TportServeur = new TportServeur;
	Property multiProcessus : boolean << default : boolean = false; >>;
	Property max-requetes-concurrentes : int;
	}
	
	Connector Type Trpc = {
	Role RoleClient : TroleClient = new TroleClient;
	Role RoleServeur : TroleServeur = new TroleServeur;
	Property blocage : boolean << default : boolean = true; >>;
	Property protocole : string << default : string = ""; >>;
	invariant size(self.roles) == 2;
	}
	
	// Si deux composants sont connectés, l'un doit être client et l'autre serveur
	invariant Forall c1 : component in self.components |
	Forall c2 : component in self.components |
	connected(c1, c2) -> (declaresType(c1, TcompClient) AND declaresType(c2, TcompServeur)) OR (declaresType(c1, TcompServeur) AND declaresType(c2, TcompClient)) <<label : string = "Toutes les pairs de composants connectées sont Client et Serveur.";errMsg : string = "Pair(s) de composants détectée(s) qui ne respecte(nt) pas la règle Client avec Serveur!";>>;
	
	// Tous les composants doivent être de type Client ou Serveur
	invariant Forall comp : component in self.components |
	(declaresType(comp, TcompClient) AND satisfiesType(comp, TcompClient)) OR (declaresType(comp, TcompServeur) AND satisfiesType(comp, TcompServeur)) <<label : string = "Tous les composants sont soit Client soit Serveur.";errMsg : string = "Composant détecté qui n'est ni Client ni Serveur!";>>;
	
	// Tous les connecteurs doit être de type Trpc
	invariant Forall conn : connector in self.connectors |
	satisfiesType(conn, Trpc) <<label : string = "Tous les connecteurs sont de type Trpc.";errMsg : string = "Connecteur détecté qui n'est pas de type Trpc!";>>;
	
	}

Requiert	« Adaptateur_SICo »

7. Système de patrons extensible

Les différents patrons produit du système de patrons sont organisés selon les modes de coopération présentés dans la figure 58. Ils formalisent des architectures de coopération qui répondent aux critères de coopération que nous avons définis précédemment. Cependant, seul un retour d'expérience pourra nous montrer si ces différents critères sont suffisamment complets pour classer les différentes architectures de coopération. L'ajout d'autres critères de coopération peut ainsi s'avérer nécessaire pour couvrir d'autres modes de coopération non pris en compte. Cet ajout implique la mise à jour des différents patrons processus ainsi que l'ajout de nouveaux patrons produits qui répondent à ces nouveaux modes de coopération. L'objectif du patron processus « Nouveaux besoins » que nous présentons par la suite, est de décrire la démarche à suivre pour faire évoluer le système de patrons en cas d'ajout de nouveaux critères de coopération.

Identifiant	Nouveaux besoins
Problème	Ce patron permet d'ajouter de nouveaux critères de coopération ou de prendre en compte des nouvelles architectures de coopération pour répondre à des nouveaux besoins exprimés par l'ingénieur d'applications.
Solution **Démarche**	Les critères de coopération et les modes de coopération, permettent de guider l'ingénieur d'applications dans son choix du patron produit le plus approprié à son problème. Une fois qu'il a choisi son mode de coopération, l'ingénieur d'applications peut être confronté à deux situations : 1. Le mode de coopération mène vers plusieurs patrons produit. Dans ce cas, l'ingénieur d'applications peut suggérer à l'ingénieur de patron l'ajout d'un ou de plusieurs critères pour différencier les deux architectures voisines. Cela implique l'ajout de nouveaux modes de coopération et la mise à jour des différents patrons processus constituant la démarche. 2. Le patron produit ne correspond pas aux besoins du concepteur. Le mode de coopération conduit alors à deux architectures différentes : celle proposée par le système de patrons et celle souhaitée par l'ingénieur d'applications. Il est alors nécessaire d'élaborer un nouveau patron produit représentant l'architecture spécifique attendue. L'ingénieur d'applications peut oui ou non proposer de nouveaux critères de coopération pour différencier son architecture de celle qui lui a été proposée. La figure suivante illustre l'ensemble du processus que doit suivre l'ingénieur de patrons.



8. Conclusion

Nous avons consacré les chapitres précédents à la présentation des différentes approches traitant la coopération des systèmes d'information. Tout d'abord, les techniques de coopération, tels que les Systèmes d'Information Fédérés et les Systèmes Multi-Agents ont fait l'objet de plusieurs travaux de recherche. Néanmoins, ces techniques présentent un inconvénient relatif à la décentralisation des informations les concernant et rares sont les travaux qui les réunissent et les synthétisent [COUTURIER, 05]. Nous nous sommes intéressés par la suite aux méthodes de modélisation permettant de décrire ces différentes techniques de coopération. Ces méthodes, formelles ou semi-formelles, présentent des limites quant à la description des styles d'architectures dédiés à la coopération des systèmes d'information.

L'ingénierie des Systèmes d'Information Coopératifs reste ainsi un domaine complexe. Les travaux sur la coopération sont encore, généralement, soit difficilement réutilisables, soit insuffisamment documentés pour être facilement compréhensibles par les différents acteurs d'un projet.

Dans ce chapitre, nous avons présenté les éléments de base pour une méthode d'ingénierie dédiée aux SICo, à savoir :

- Un méta-modèle pour la description structurelle des architectures logicielles coopératives. Ce méta-modèle repose sur la combinaison des méthodes formelles et semi-formelles pour une spécification cohérente, compréhensible et réutilisable.
- Des critères de coopération pour classifier les différentes architectures de coopération présentant ainsi une aide au concepteur pour le guider vers l'architecture de coopération la plus appropriée à son problème.
- Un système de patrons combinant patrons processus et patrons produit pour une meilleure documentation favorisant et promouvant la réutilisation des architectures de coopération. Les différents patrons produit proposés formalisent les architectures de coopération les plus répandues. Les patrons processus permettent, quant à eux, de guider le concepteur vers un mode de coopération. A l'état actuel, ce dernier peut correspondre à une ou plusieurs architecture(s) de coopération et inversement.
- Une solution extensible et évolutive par ajout de nouveaux patrons produit (nouvelles architectures coopératives) ou de nouveaux critères de coopération (modes de coopération).

L'ensemble des patrons du système de patrons sont présentés dans le chapitre suivant. L'instrumentation de ce système de patrons sera détaillée dans le chapitre VI.

CHAPITRE V

PACO : un système de PATRONS POUR LES ARCHITECTURES COOPERATIVES

1. Introduction

Ce chapitre est consacré à la présentation de notre système de patrons dédié à l'ingénierie des Systèmes d'Information Coopératifs (SICo). Un système de patrons complet doit offrir des techniques d'organisation et donc de sélection de patrons, par exemple des tableaux et/ou des critères de classification des patrons coordonnateurs d'une démarche. Le système PACO offre cette double possibilité. Il est composé d'un ensemble de patrons processus servant de guide pour la recherche et la sélection des patrons produit proposés. Le patron processus « PACO » représente le point d'entrée du système de patrons et coordonne l'usage des autres patrons processus. Ces derniers permettent de guider le concepteur vers le patron produit qui répond à ses besoins. La navigation dans les différents patrons processus est réalisée grâce à un ensemble de critères de coopération. L'enchaînement entre différents patrons processus constitue un mode de coopération et mène vers un patron produit. Le but visé par ce mode de recherche par navigation est d'assister le concepteur et faciliter sa tâche. Cependant, le choix d'un patron produit peut être fait directement en s'appuyant sur la rubrique « Classification ». Cette rubrique reprend, en effet les différents modes de coopération résolus par le patron produit.

Ce chapitre est ainsi organisé selon les deux types de patrons que nous proposons, à savoir processus (cf. section 2) et produit (cf. section 3). Les premiers formalisent la démarche en se basant sur les différents modes de coopération ; les seconds formalisent les architectures de composants de coopération et offrent une meilleure documentation pour faciliter leur réutilisation [SAIDANE & AL., 02].

2. Patrons processus

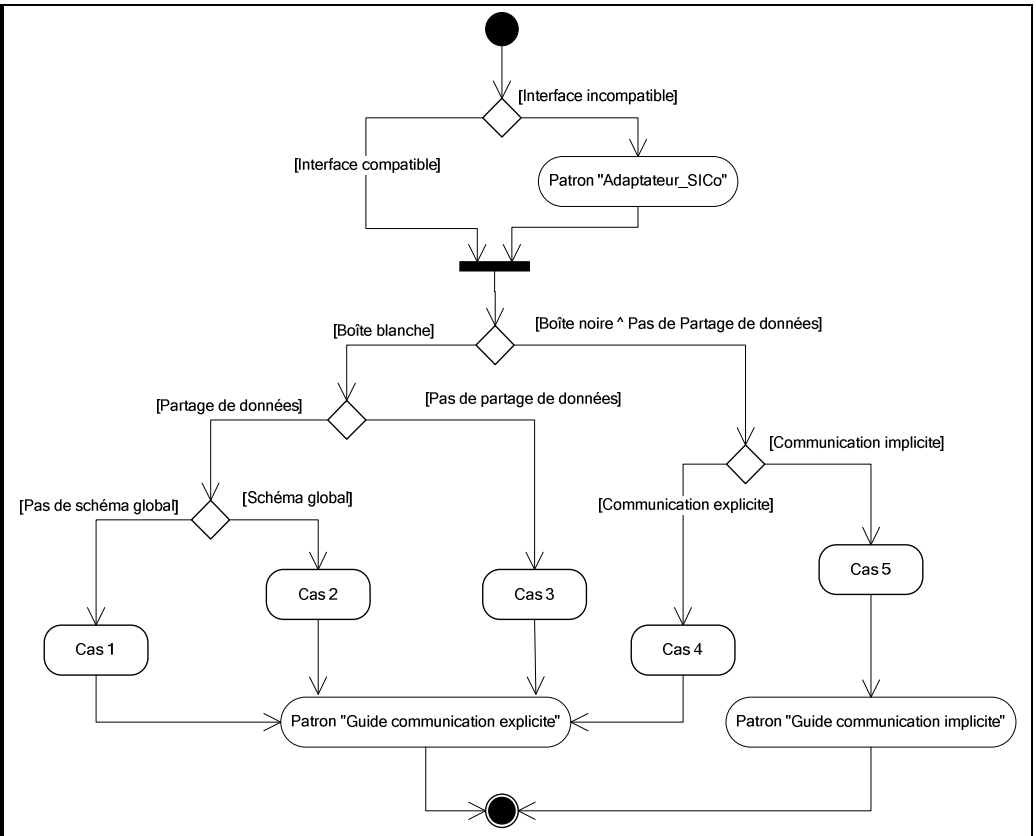
Cette section présente les différents patrons processus qui forment la démarche. Chaque patron processus propose ainsi un fragment de la démarche et oriente le concepteur vers un

patron processus ou produit du système de patrons. Les différents patrons processus sont représentés sous forme de diagrammes d'activités UML.

Les patrons processus conduisent vers un patron produit du système PACO. Dans le cadre de ce travail, nous nous limitons à ne présenter que les patrons produit qui formalisent les architectures de coopération les plus répandues. Les autres patrons produits seront identifiés par « patrons xxx ».

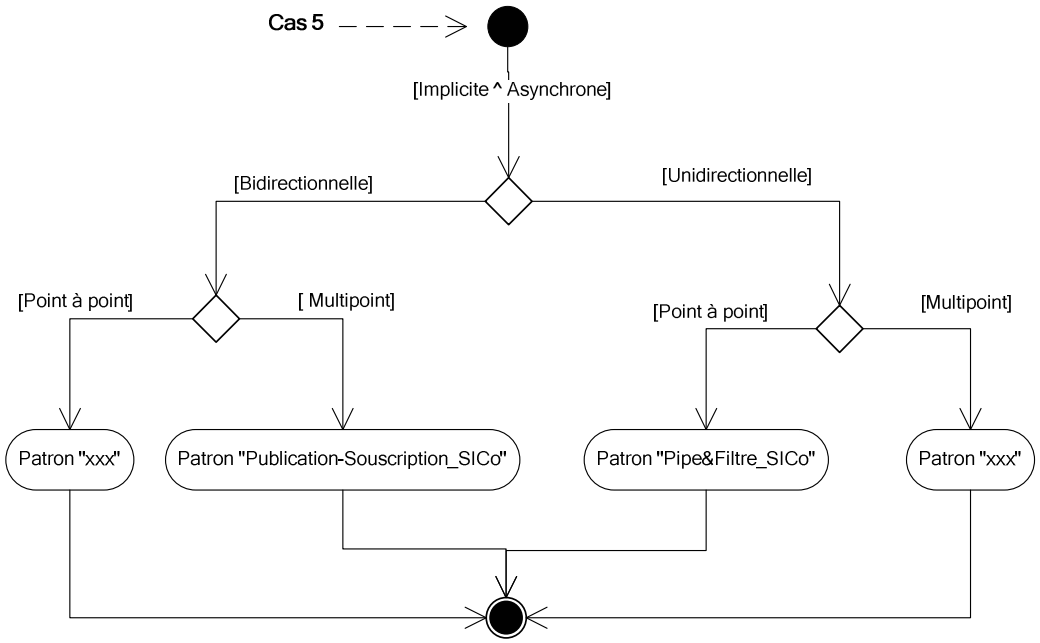
2.1. Patron « PACO »

Identifiant	**PACO**
Contexte	Ce patron ne nécessite aucun autre patron ou modèle pour être appliqué.
Problème	Ce patron constitue le point d'entrée du système de patrons PACO. C'est à partir de ce patron que le concepteur peut raisonner sur les caractéristiques des systèmes composants de la coopération.
Solution Démarche	Les systèmes composants sont souvent préexistants et de nature hétérogène. Pour palier à cette hétérogénéité, l'application du patron « Adaptateur_SICo » est capitale. En effet, ce dernier permet de modifier le(s) service(s) rendu(s) par le système composant pour l'adapter aux besoins des autres systèmes composants. L'application de ce patron est alors indépendante des critères de coopération des systèmes composants mais est liée aux conflits que peuvent engendrer ces derniers, s'ils sont de natures différentes (systèmes hétérogènes). Selon la nature des systèmes composants (type d'ouverture et partage de données), cinq possibilités sont offertes au concepteur. Les quatre premières (cas 1, 2, 3 et 4) renvoient le concepteur vers le patron «Guide communication explicite » ; le dernier cas (cas 5) le dirige vers le patron « Guide communication implicite ». C'est à partir de ces deux patrons que le concepteur va effectuer son choix selon les critères des connecteurs.

	
Utilise	{« Adaptateur_SICo », « Guide communication explicite », « Guide communication implicite » }.

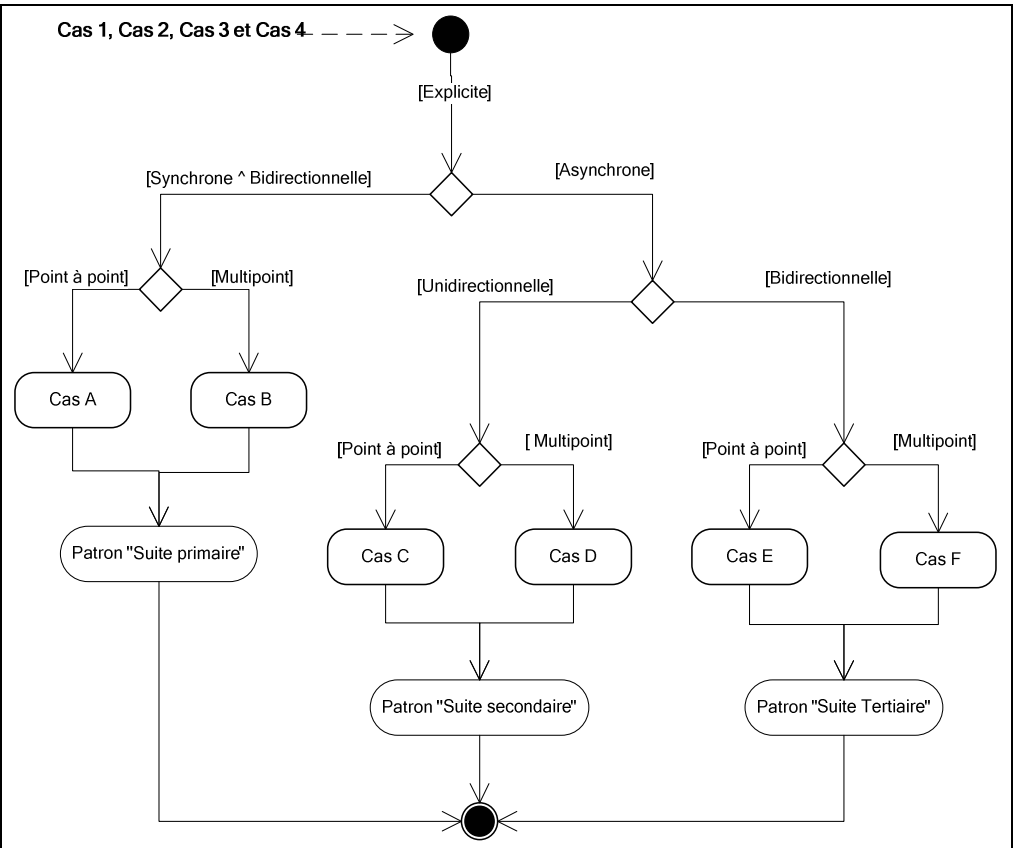
2.2. Patron « Guide Communication Implicite »

Identifiant	**Guide Communication Implicite**
Contexte	Ce patron nécessite l'utilisation préalable du patron «PACO».
Problème	Ce patron permet de guider le concepteur dans le cadre d'une communication implicite entre les systèmes composants.
Solution Démarche	Une communication implicite restreint le choix du concepteur. En effet, un connecteur implicite ne peut être que de type asynchrone. Parmi les quatre modes de coopération envisageables, nous traitons deux cas : le patron « Pipe&Filtre_SICo » et le patron « Publication-Souscription_SICo ».

	
Utilise	{ « Pipe&Filtre_SICo », « Publication-Souscription_SICo »}.
Requiert	« PACO».

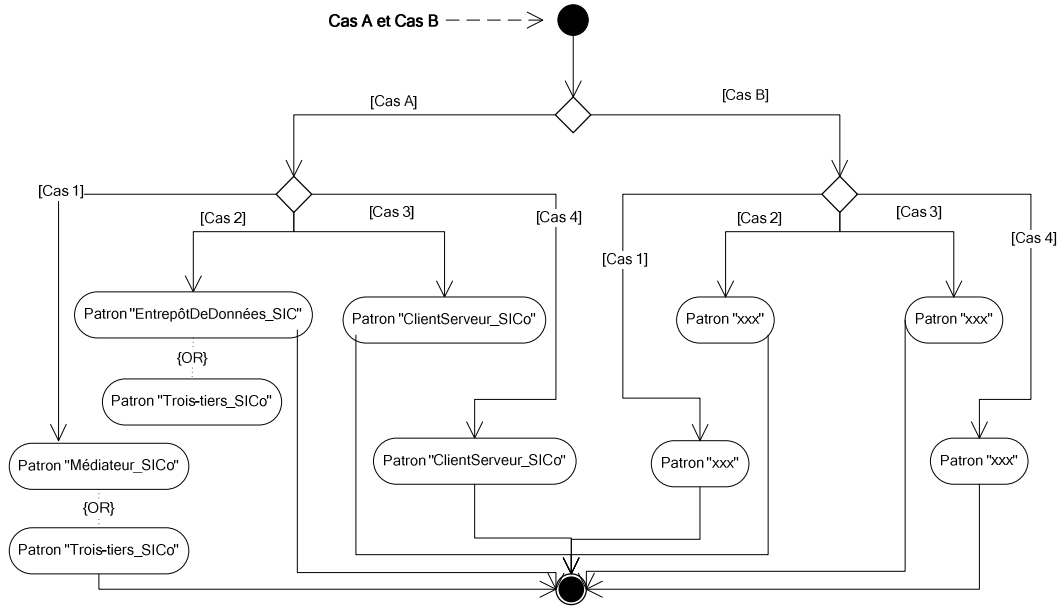
2.3. Patron « Guide Communication Explicite »

Identifiant	**Guide Communication Explicite**
Contexte	Ce patron nécessite l'utilisation préalable du patron « PACO».
Problème	Ce patron permet de guider le concepteur dans le cadre d'une communication explicite entre les systèmes composants.
Solution Démarche	Lorsque la communication est explicite entre systèmes composants, l'échange de données ou de processus peut être réalisé de manière asynchrone ou synchrone. Dans ce dernier cas, la connexion ne peut être que bidirectionnelle. Ce patron mène vers trois patrons processus : « Suite primaire », « Suite secondaire » et « Suite tertiaire ». Ces derniers traitent respectivement le cas d'une communication synchrone et bidirectionnelle, asynchrone et unidirectionnelle, et enfin asynchrone et bidirectionnelle.

	
Utilise	{« Suite primaire », « Suite secondaire », « Suite tertiaire »}.
Requiert	« PACO ».

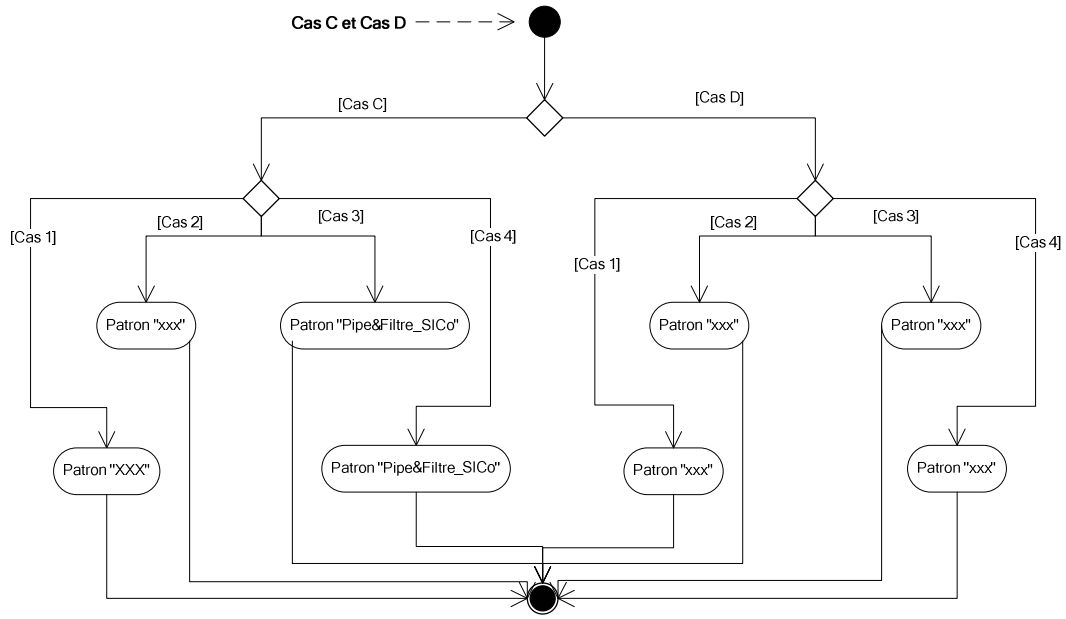
2.4. Patron « Suite primaire »

Identifiant	**Suite primaire**
Contexte	Ce patron nécessite l'utilisation préalable du patron « Guide communication explicite ».
Problème	Ce patron est la suite du patron « Guide communication explicite » lorsque la communication est à la fois synchrone et bidirectionnelle, point à point (cas A) ou multipoint (cas B). Il permet de guider le concepteur vers l'un des patrons produit du système de patrons selon qu'il se positionne dans les cas 1, 2, 3 ou 4.

Solution Démarche	
Utilise	« Médiateur_SICo », « Trois-tiers_SICo », « EntrepôtDeDonnées_SICo », « ClientServeur_SICo ».
Requiert	« Guide communication explicite ».

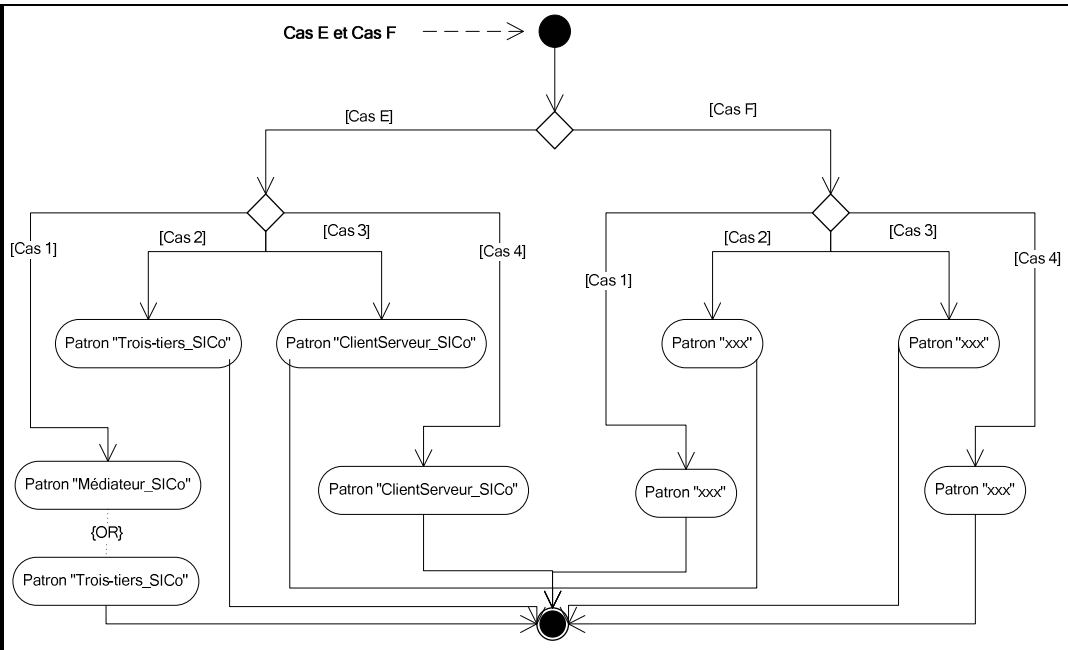
2.5. Patron « Suite secondaire »

Identifiant	**Suite secondaire**
Contexte	Ce patron nécessite l'utilisation préalable du patron « Guide communication explicite ».
Problème	Ce patron est la suite du patron « Guide communication explicite » lorsque la communication est à la fois asynchrone et unidirectionnelle, point à point (cas C) ou multipoint (cas D). Il permet de guider le concepteur vers l'un des patrons produit du système de patrons selon qu'il se positionne dans les cas 1, 2, 3 ou 4.
Solution Démarche	

	
Utilise	« Pipe&Filtre_SICo ».
Requiert	« Guide communication explicite ».

2.6. Patron « Suite tertiaire »

Identifiant	**Suite tertiaire**
Contexte	Ce patron nécessite l'utilisation préalable du patron « Guide communication explicite ».
Problème	Ce patron est la suite du patron « Guide communication explicite » lorsque la communication est à la fois asynchrone et unidirectionnelle, point à point (cas E) ou multipoint (cas F). Il permet de guider le concepteur vers l'un des patrons produit du système de patrons selon qu'il se positionne dans les cas 1, 2, 3 ou 4.
Solution Démarche	

	
Utilise	« Médiateur_SICo », « Trois-tiers_SICo », « EntrepôtDeDonnées_SICo », « ClientServeur_SICo ».
Requiert	« Guide communication explicite ».

3. Patrons produit

Les patrons produit du système PACO formalisent les architectures de coopération les plus répandues. Ils fournissent un niveau d'abstraction qui permet au concepteur d'un Système d'Information Coopératif (SICo) de raisonner sur le comportement global de l'architecture sans en donner les détails d'implémentation. Ils proposent une description qui permet de décrire la structure de l'architecture coopérative en terme de systèmes composants la constituant ainsi que leurs interactions.

L'utilisation conjointe d'UML et d'Acme pour spécifier la solution modèle des patrons produit permet d'augmenter la réutilisation des patrons proposés. Les diagrammes de composants UML et les annotations en langage naturel facilitent la compréhension de l'architecture globale et de ses différentes entités (composants et connecteurs). La spécification Acme permet, d'une part, de décrire des propriétés de l'architecture, et d'autre part, de préciser des contraintes permettant de contrôler sa réutilisation et son adaptation.

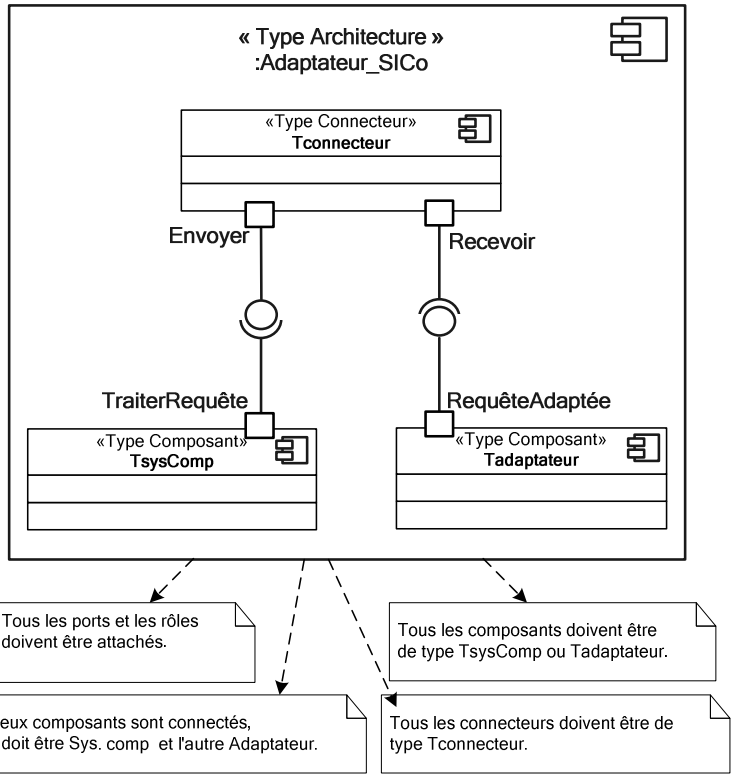
Les différents patrons présentés par la suite formalisent des architectures de coopération connues et largement utilisées. Nous ne présenterons pas d'exemples d'utilisation pour l'ensemble des patrons offerts. Nous reviendrons sur ce point au cours du chapitre suivant (ch. Chapitre VI) dans lequel nous appliquons la démarche sur une application industrielle. Ceci permettra de montrer un exemple d'imitation sur l'un des patrons développés et de tester l'adéquation de notre approche pour la modélisation des SICo.

3.1. Patron « Adaptateur_SICo »

Le patron « Adaptateur_SICo » peut être appliqué à tous les patrons produit du système PACO. Ce premier permet en effet d'homogénéiser les interfaces, a priori hétérogènes, des systèmes composants coopérants. L'adaptateur peut transformer les données d'un système composant d'un modèle à un autre, convertir une requête dans un format compréhensible par le SC destinataire, intercepter un événement émis par un SC pour le transformer dans un format approprié au bus d'événements, etc. A ce niveau d'abstraction, la solution modèle est identique quelque soit la tâche attendue par l'adaptateur. Cependant, l'implantation de ce dernier diffère selon le cas et peut être complexe.

Notons que dans le cas des SC boîtes blanches, il est possible d'accéder à la structure interne du système composant et de modifier ainsi son (ou ses) interface(s) pour les adapter à l'environnement de la coopération. Néanmoins, il est souhaitable d'appliquer la patron « Adaptateur_SICo » pour, d'une part, préserver l'autonomie et la structure interne du SC, et d'autre part, n'exposer que les services que le SC met à la disposition de la coopération.

Identifiant	**Adaptateur_SICo**
Classification	Si interface incompatible.
Problème	Adapter l'interface d'un système composant aux exigences des interfaces des autres systèmes composants participants à la coopération. L'adaptateur permet à des systèmes composants de coopérer en masquant l'incompatibilité de leurs interfaces respectives.
Description comportementale	L'adaptateur intercepte la demande du système composant émetteur et la modifie pour l'adapter aux besoins du système composant cible.
Solution modèle **Semi-formelle**	*Systèmes composants:* TsysComp : Le système composant à adapter. Tadaptateur : Intercepte la requête et la transforme sous un format compréhensible par le système composant. *Connecteur :* Tconnecteur : connecteur acheminant la requête entre les deux composants.



Solution modèle formelle

```
Family Adaptateur_SICo = {

// Déclaration des ports et des rôles. Chaque port / rôle doit être attaché
   Port Type TportAdaptateur = {
      heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg :
      string = "Ce port doit être attaché!";>>;
   }

   Port Type TportSysComp = {
      heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg :
      string = "Ce port doit être attaché!";>>;
   }

   Role Type TroleRecevoir = {
      invariant size(self.attachedPorts) >= 1<<label : string = "Ce rôle est attaché.";errMsg :
      string = "Ce rôle doit être attaché!";>>;
   }

   Role Type TroleEnvoyer = {
      invariant size(self.attachedPorts) >= 1<<label : string = "Ce rôle est attaché.";errMsg :
      string = "Ce rôle doit être attaché!";>>;
   }

   Component Type Tadaptateur = {
      Port RequeteAdaptee : TportAdaptateur =  new TportAdaptateur;
   }

   Component Type TsysComp = {
      Port TraiterRequete : TportSysComp =  new TportSysComp;
   }

   Connector Type Tconnecteur = {
      Role Recevoir : TroleRecevoir =  new TroleAdapter;
      Role Envoyer : TroleEnvoyer =  new TroleIntercepter;
```

```
    }

    // Si deux composants sont connectés, l'un doit être de type Tadaptateur et
    // l'autre TsysComp
    invariant Forall c1 : component in self.components |
        Forall c2 : component in self.components |
            connected(c1, c2) -> (declaresType(c1, Tadaptateur) AND declaresType(c2,
            TsysComp)) OR (declaresType(c1, TsysComp) AND declaresType(c2, Tadaptateur))
            <<label : string = "Toutes les paires de composants connectés sont de type
              Tadaptateur  et  TsysComp.";errMsg : string = "Paire(s) de composants détectée(s)
              qui ne respecte pas la règle Tadaptateur avec TsysComp!";>>;

    // Tous les composants doivent être de type Tadaptateur ou TsysComp
    invariant Forall comp : component in self.components |
        (declaresType(comp, Tadaptateur) AND satisfiesType(comp, Tadaptateur)) OR
        (declaresType(comp, TsysComp) AND satisfiesType(comp, TsysComp)) <<label : string
        = "Tous les composants sont soit Tadaptateur soit TsysComp.";errMsg : string =
        "Composant détecté qui n'est ni Tadaptateur ni TsysComp!";>>;

    // Tous les connecteurs doit être de type Tconnecteur
    invariant Forall conn : connector in self.connectors |
        satisfiesType(conn, Tconnecteur) <<label : string = "Tous les connecteurs sont de type
        Tconnecteur";errMsg : string = "Connecteur détecté qui n'est pas de type
        Tconnecteur!";>>;
}
```

3.2. Patron « Pipe&Filtre_SICo »

L'architecture "pipe & filtre" est utilisée dans la littérature pour permettre à des composants de différentes spécialités de coopérer. Une tâche est décomposée en sous-tâches accomplies par chaque composant. La solution attendue du système est obtenue au terme de leur exécution. Cette architecture est utilisée, par exemple, dans le mode de coopération intégrative des Systèmes Multi-Agents. Dans ce cas, la solution finale est obtenue par l'intégration successive des solutions partielles provenant d'agents de différents métiers. La figure ci-dessous (cf. figure 59) synthétise les modes de coopération réalisés par ce patron.

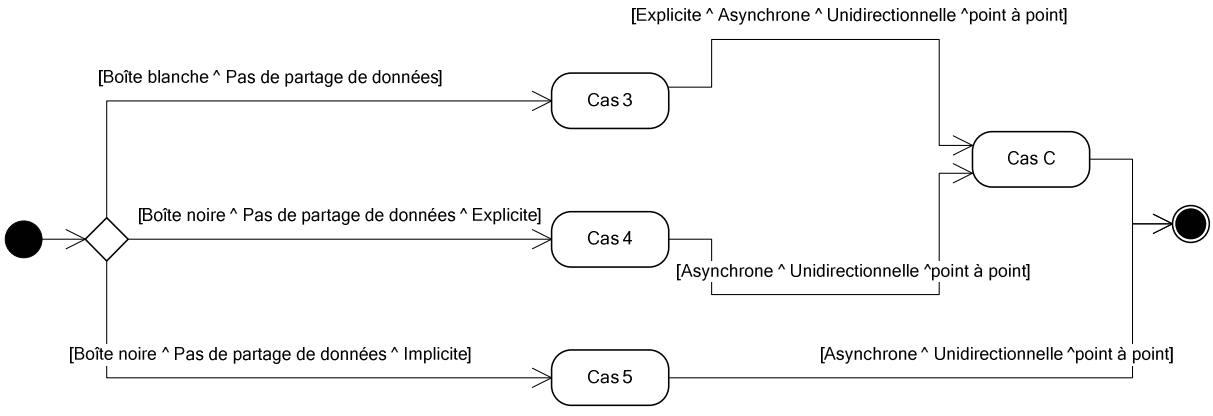


Figure 59 : Modes de coopération du patron "Pipes&Filtre_SICo"

Identifiant	Pipe&filtre_SICo
Classification	*Divers modes de coopération possibles :* 1) Boîte Blanche ∩ Pas de Partage de Données ∩ Explicite ∩ Asynchrone ∩ Unidirectionnelle ∩ Point à point 2) Boîte Noire ∩ Pas de Partage de Données ∩ Explicite ∩ Asynchrone ∩ Unidirectionnelle ∩ Point à point 3) Boîte Noire ∩ Pas de Partage de Données ∩ Implicite ∩ Asynchrone ∩ Unidirectionnelle ∩ Point à point *Formule simplifiée :* Pas de Partage de Données ∩ Asynchrone ∩ Unidirectionnelle ∩ Point à point ∩ ((Boîte Blanche ∩ Explicite) ∪ Boîte Noire)
Contexte	Ce patron peut nécessiter l'utilisation du patron « Adaptateur_SICo ».
Problème	Le patron « Pipe&Filtre_SICo » permet de structurer une architecture de coopération traitant un flot de données. Chaque étape de traitement est encapsulée dans un système composant (composant "filtre"). Les données sont ensuite acheminées entre filtres adjacents à travers les pipes.
Description comportementale	La tâche à réaliser est divisée en plusieurs étapes de traitements séquentiels, chacune étant implémentée par un système composant "filtre". Les différents filtres sont connectés par des pipes qui transmettent les flots de données des sorties d'un filtre vers les entrées d'un autre. Les flots de données à traiter sont fournis en entrée par le composant "Source de données". Le résultat final du processus de traitement est stocké dans le composant "Destination de données".

Solution modèle

Semi-formelle

Systèmes composants :

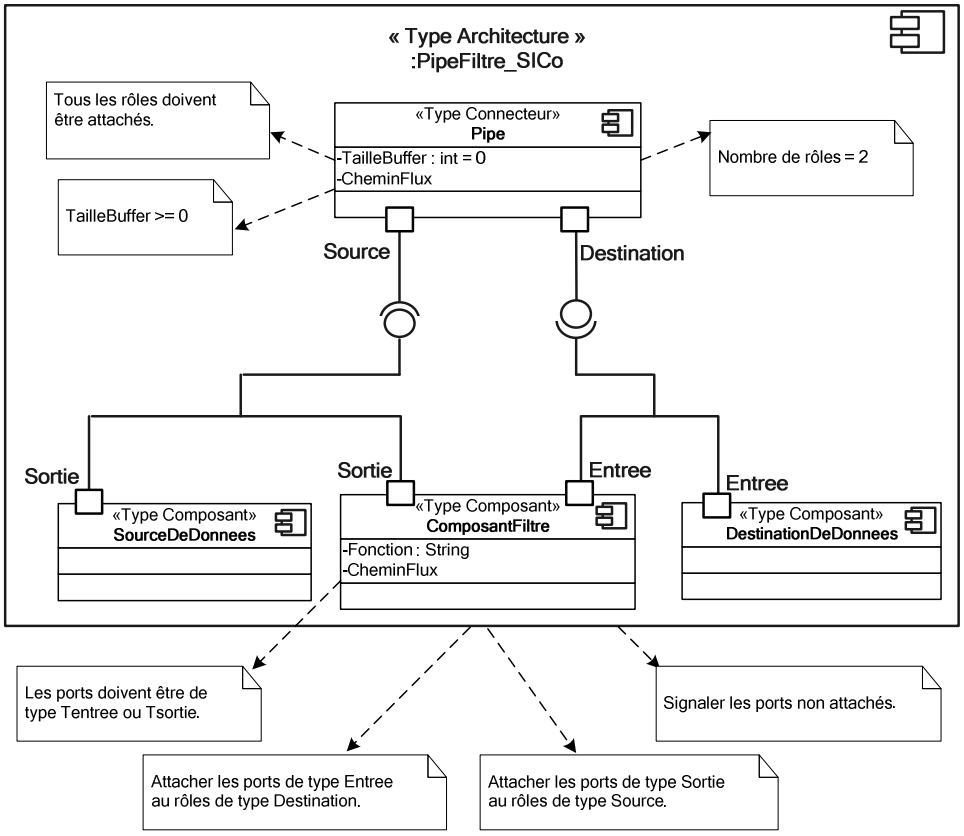
SourceDeDonnées : fournit le flot de données en entrée.

ComposantFiltre : composant indépendant qui transforme un ou plusieurs flots d'entrée en un ou plusieurs flots de sortie de manière incrémentale.

DestinationDeDonnées : stocke le flot de données, résultat des différents traitements.

Connecteur

Pipe : achemine le flot de données entre les différents composants.



Solution modèle

formelle

```
Family PipeFiltre_SICo = {
   Property Type TCheminDuFlux = Record [
      DeP : string;
      AP : string;
   ];

   Port Type Tentree = {
      Property protocole : string = "char entree";
   }

   Port Type Tsortie = {
      Property protocole : string = "char sortie";
   }

   Role Type Tsource = {
      Property protocole : string = "char source";
      invariant size(self.attachedPorts) >= 1<<label : string = "Ce rôle est attaché.";errMsg :
      string = "Ce rôle doit être attaché!";>>;
   }
```

```
Role Type Tdestination = {
    Property protocole : string = "char destination";
    invariant size(self.attachedPorts) >= 1<<label : string = "Ce rôle est attaché.";errMsg :
    string = "Ce rôle doit être attaché!";>>;
}

Component Type ComposantFiltre = {
    Port Entree : Tentree =  new Tentree;
    Port Sortie : Tsortie =  new Tsortie;
    Property fonction : string;
    Property CheminsFlux : Set{TCheminDuFlux} << default : Set{TCheminDuFlux} =
    {[  DePt : string = "Entree";
        AP : string = "Sortie" ]};  >>;
    // Tous les ports doivent être de type Tentree ou Tsortie
    invariant Forall p : port in self.Ports |
        satisfiesType(p, Tentree) OR satisfiesType(p, Tsortie) <<label : string = "Seuls les
        ports de type Tentree et Tsortie sont permis";>>;
}

Component Type SourceDeDonnees = {
    Port Sortie : Tsortie =  new Tsortie;
}

Component Type DestinationDeDonnees = {
    Port Entree : Tentree =  new Tentree;
}

Connector Type Pipe = {
    Role Source : Tsource =  new Tsource;
    Role Destination : Tdestination =  new Tdestination;
    Property TailleBuffer : int << default : int = 0;  >>;
    Property CheminsFlux : Set{TCheminDuFlux} = {[
        DeP : string = "Source";
        AP : string = "Destination" ]};
    invariant size(self.roles) == 2;
    invariant self.TailleBuffer >= 0;
}

// Attacher uniquement les ports de type Tentree / Tsortie aux rôles de type
// Tdestination / Tsource

invariant Forall comp : component in self.components |
    Forall conn : connector in self.connectors |
        Forall p : port in comp.ports |
            Forall r : role in conn.Roles |
                attached(p, r) -> ((satisfiesType(p, Tentree) AND satisfiesType(r, Tdestination))
                OR (satisfiesType(p, Tsortie) AND satisfiesType(r, Tsource)))    <<label : string
                = " Attacher uniquement les ports de type Tentree / Tsortie aux rôles de type
                Tdestination / Tsource  // Tdestination / Tsource";>>;

// Signaler les ports non attachés
heuristic Forall comp : component in self.components |
    Forall p : port in comp.Ports |
        Exists conn : connector in self.Connectors |
            Exists r : role in conn.Roles |
                attached(p, r)    <<label : string = "Tous les ports sont attachés";errMsg : string
                = "Il existe des ports non attachés";>>;
}
```

Requiert	« Adaptateur_SICo »

3.3. Patron « Publication-Souscription_SICo »

Pour décrire l'architecture du patron « Publication-Souscription_SICo », nous définissons deux types d'entités : un type de composant participant qui peut publier ou souscrire à un événement, et un type de connecteur (bus d'événement) qui gère les souscriptions et la notification des événements. Toutefois, la description de l'architecture globale nécessite la prise en compte d'un certain nombre de contraintes et de propriétés. Par exemple, un connecteur ne peut pas avoir plus d'une interface (ou rôle) pour recevoir l'événement à notifier mais un nombre arbitraire de rôles pour publier l'événement. Les différentes entités, propriétés et contraintes sont décrites ci-dessous. Les modes de coopération opérés par ce patron sont rappelés par la figure ci-après (cf. figure 60).

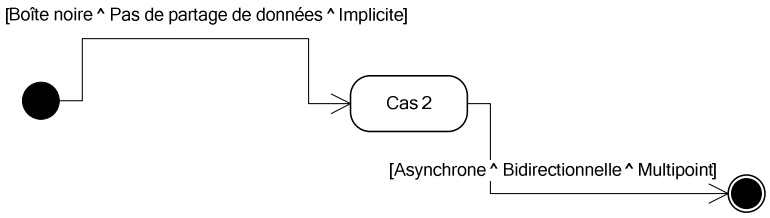
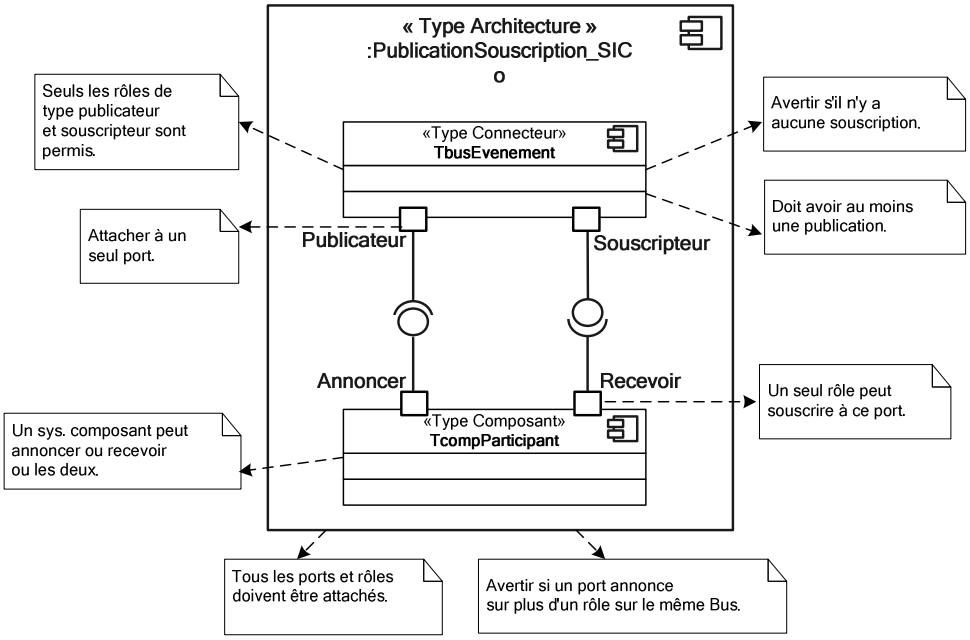


Figure 60 : Mode de coopération du patron "Publication-Souscription_SICo"

Identifiant	**Publication-Souscription_SICo**
Classification	Boîte Noire ∩ Pas de Partage de Données ∩ Implicite ∩ Asynchrone ∩ Bidirectionnelle ∩ Multipoint
Contexte	Ce patron peut nécessiter l'utilisation du patron « Adaptateur_SICo ».
Problème	Permettre à un système composant d'envoyer différents types de messages aux autres systèmes composants de la coopération sans nécessairement connaître leur identité. Ces derniers ne reçoivent que les messages pour lesquels ils ont enregistré un intérêt.
Description comportementale	Ce patron est basé sur la notion de "consommateurs" d'information appelés "souscripteurs", et de "producteurs" d'information appelés "éditeurs" (*publishers*). Le paradigme de coopération du patron « Publication-Souscription_SICo » repose sur un service de notification d'événements qui stocke et gère aussi bien les souscriptions que les envois des notifications. Ce service est assuré par le "Bus d'événements". Quand un système composant publie un événement, ce dernier notifie tous les systèmes

<table>
<tr><td></td><td>composants ayant souscrit un intérêt pour celui-ci. Un système composant ne peut pas être à la fois souscripteur et éditeur pour le même événement. En revanche, il peut être éditeur d'un événement et souscripteur d'un autre.</td></tr>
<tr><td>Solution modèle

Semi-formelle</td><td><u>Systèmes composants :</u>
TcomParticipant : système composant pouvant être éditeur ou souscripteur.
<u>Connecteur :</u>
TbusEvenement : bus d'événements assurant la gestion des souscriptions et la notification des événements.

 
</td></tr>
<tr><td>Solution modèle

formelle</td><td>Family PublicationSouscription = {
Port Type p_annoncer = {
// Ce port doit être attaché
heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg : string = "Ce port doit être attaché!";>>;

// Attaché seulement au rôle de type r_publicateur
invariant Forall r : role in self.attachedroles |
declaresType(r, r_publicateur) <<label : string = "Le type de (s) rôle (s) attaché (s) est correct.";errMsg : string = "Le rôle attaché n'est pas de type r_publication!";>>;
}

Port Type p_recevoir = {
// Ce port doit être attaché
heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg : string = "Ce port doit être attaché!";>>;

// Attacher seulement au port de type r_souscripteur
invariant Forall r : role in self.attachedroles |
declaresType(r, r_souscripteur) <<label : string = "Le type de rôle attaché est correct.";errMsg : string = "Le rôle attaché n'est pas de type r_souscription!";>>;

// Au plus un rôle peut souscrire à ce port</td></tr>
</table>

```
    invariant size({Select r : role in self.attachedRoles |
        declaresType(r, r_souscripteur) }) <= 1<<label : string = "Pas plus d'un seul rôle peut
        souscrire à ce port.";errMsg : string = "Erreur! Plusieurs rôles ont souscrit à ce
        port!";>>;
}

Role Type r_publicateur = {
    // Ce rôle doit être attaché à un et un seul port
    invariant size(self.attachedPorts) == 1<<label : string = "Ce rôle a un seul
    attachement.";errMsg : string = "Le rôle n'est pas attaché ou est attaché à plus d'un
    port!";>>;

   // Attacher seulement au port de type p_annoncer
    invariant Forall p : port in self.attachedports |
        declaresType(p, p_annoncer) <<label : string = "Le type de port attaché est
        correct.";errMsg : string = "Le port attaché n'est pas de type p_announcer!";>>;
}

Role Type r_souscripteur = {
    // Ce rôle doit être attaché
    invariant size(self.attachedPorts) >= 1<<label : string = "Ce rôle a un seul
    attachement.";errMsg : string = "Le rôle n'est pas attaché ou est attaché à plus d'un
    port!";>>;

    // Attacher seulement au port de type p_recevoir
    invariant Forall p : port in self.attachedports |
        declaresType(p, p_recevoir) <<label : string = "Le type de port attaché est
        correct.";errMsg : string = "Le port attaché n'est pas de type p_recevoir!";>>;
}

Connector Type TbusEvenement = {
    Role Publicateur : r_publicateur =  new r_publicateur;
    Role Souscripteur : r_souscripteur =  new r_souscripteur;

    // Seuls les rôles de type r_publicateur et r_souscripteur sont permis
    invariant Forall r in self.roles |
        Exists t in {r_publicateur, r_souscripteur} |
            declaresType(r, t)  <<label : string = "Les rôles respectent les types requis.";errMsg :
            string = "Au minimum un rôle n'est pas de type r_publication ou r_souscription
            !";>>;

    // Avertir s'il n'y a aucune souscription
    heuristic Exists r : role in self.roles |
        declaresType(r, r_souscripteur) <<label : string = "Il existe au minimum une
        souscription.";errMsg : string = "Warning! Aucune souscription détectée!";>>;

    // Doit avoir au moins une publication
    invariant Exists r : role in self.roles |
        declaresType(r, r_publicateur) <<label : string = "Il existe au minimum une
        publication.";errMsg : string = "Erreur!  Aucune publication détectée!";>>;
}

Component Type TcompParticipant = {
    Port Annoncer : p_annoncer =  new p_annoncer;
    Port Recevoir : p_recevoir =  new p_recevoir;

    // Le participant peut annoncer ou recevoir ou les deux
    heuristic Exists p in self.ports |
        Exists t in {p_annoncer, p_recevoir} |
            declaresType(p, t)  <<label : string = "Il existe un port de type p_annoncer ou
            p_recevoir.";errMsg : string = "Aucun port détecté, ni de type p_announce ou de type
```

	p_recevoir!";>>;
	}
	
	// Avertir si un port annonce sur plus d'un rôle sur le même bus
	heuristic Forall comp : component in self.components |
	Forall p : p_annoncer in comp.ports |
	(!(Exists r1 : role in p.attachedRoles |
	Exists r2 : role in p.attachedRoles |
	r1 != r2 AND (Exists conn : connector in self.connectors |
	isSubset({r1, r2}, conn.roles)))) <<label : string = "Tous les ports annoncent sur un seul rôle du même bus.";errMsg : string = "Warning! Détection d'un port qui annonce sur deux rôles sur le même bus!";>>;
	}

Requiert	« Adaptateur_SICo ».

3.4. Patron « Médiateur_SICo »

L'architecture de ce patron est basée sur deux composants principaux : le médiateur et l'adaptateur. Ces deux composants seront représentés explicitement dans le diagramme de composants UML (solution semi-formelle du patron). Cependant, pour décrire la solution formelle, nous réutilisons la spécification Acme du patron « Adaptateur_SICo ». Rappelons que le langage et l'outil Acme permettent de réutiliser des types d'architectures logicielles déjà prédéfinies. Ainsi, la solution formelle du patron « Médiateur_SICo » reprend les différentes entités, propriétés et contraintes du patron « Adaptateur_SICo » mais les raffine aussi pour prendre en compte la spécification de nouvelles entités et contraintes qui lui sont spécifiques. Nous rappelons dans la figure suivante (cf. figure 61) les modes de coopération considérés par ce patron.

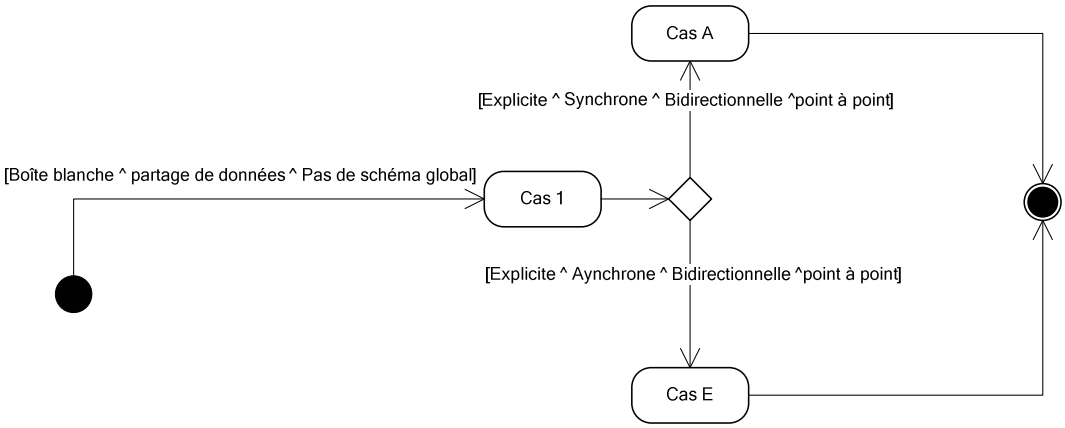
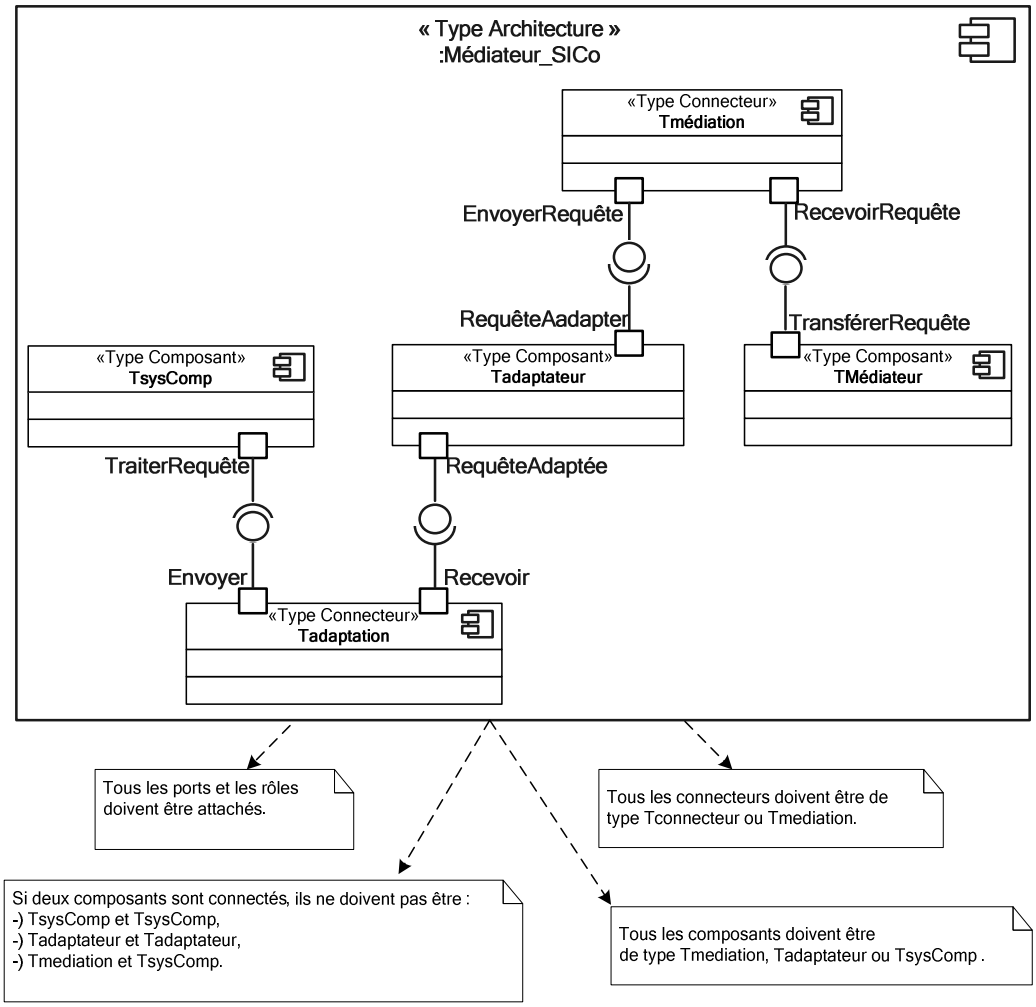


Figure 61 : Modes de coopération du patron "Médiateur_SICo"

Identifiant	Médiateur_SICo
Classification	*Divers modes de coopération possibles :* 1) Boîte Blanche ∩ Partage de Données ∩ Pas de schéma Global ∩ Explicite ∩ Asynchrone ∩ Bidirectionnelle ∩ Point à point 2) Boîte Blanche ∩ Partage de Données ∩ Pas de schéma Global ∩ Explicite ∩ Synchrone ∩ Bidirectionnelle ∩ Point à point *Formule simplifiée :* Boîte Blanche ∩ Partage de Données ∩ Pas de schéma Global ∩ Explicite ∩ Bidirectionnelle ∩ Point à point
Contexte	Ce patron nécessite l'utilisation du patron « Adaptateur_SICo » pour adapter les requêtes envoyées par le médiateur aux interfaces des systèmes composants.
Problème	Offrir à l'utilisateur un accès uniforme et centralisé à un ensemble de bases de données autonomes, distribuées et hétérogènes. Le médiateur assure le transfert de la requête utilisateur vers les systèmes composants. La communication entre les deux peut être en mode synchrone ou asynchrone.
Description Comportementale	Le rôle du médiateur est d'offrir une interface unique et transparente pour accéder aux systèmes composants. Il accepte en entrée une requête, la traite en accédant aux sources de données via les adaptateurs concernés et renvoie la réponse à l'utilisateur. Les adaptateurs permettent un accès uniforme aux systèmes composants en convertissant les modèles de données locaux vers le modèle de données de médiation et en traduisant les requêtes d'un modèle à un autre. Ils offrent ainsi une interface homogène locale d'accès aux données.
Solution modèle Semi-formelle	*Systèmes composants :* TMédiateur : décompose la requête de l'utilisateur et renvoie les sous-requêtes aux adaptateurs concernés. Tadaptateur : transfert la requête reçue du médiateur vers le système composant. TsysComp : sources de données traitant la requête envoyée par l'adaptateur. *Connecteur :* Tmédiation : achemine la requête de l'utilisateur du médiateur vers l'adaptateur. Tadaptation : achemine la requête de l'adaptateur vers le système composant.



Solution modèle formelle

```
import $AS_GLOBAL_PATH\families\Adaptateur_SICo.acme;

Family MediateurSICo extends Adaptateur_SICoo with {
   Port Type TportAadapter = {
      heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg : string =
      "Ce port doit être attaché!";>>;
   }

   Port Type TportMediateur = {
      heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg : string =
      "Ce port doit être attaché!";>>;
   }

   Role Type TroleRecevoirRequete = {
      invariant size(self.attachedPorts) >= 1<<label : string = "Ce rôle est attaché.";errMsg : string =
      "Ce rôle doit être attaché!";>>;
   }

   Role Type TroleEnvoyerRequete = {
      invariant size(self.attachedPorts) >= 1<<label : string = "Ce rôle est attaché.";errMsg : string =
      "Ce rôle doit être attaché!";>>;
   }

   Component Type Adaptateur extends Tadaptateur with {
      Port RequeteAadapter : TportAadapter =  new TportAadapter;
   }
```

```
Component Type TMediateur = {
   Port TransfererRequete : TportMediateur =  new TportMediateur;
}

Connector Type Tmediation = {
   Role RecevoirRequete : TroleRecevoirRequete =  new TroleRecevoirRequete;
   Role EnvoyerRequete : TroleEnvoyerRequete =  new TroleEnvoyerRequete;
}

// Deux composants qui sont attachés ne doivent pas être de type Tmediation et
// TsysComp
invariant Forall c1 : component in self.components |
   Forall c2 : component in self.components |
      !connected(c1, c2) -> (declaresType(c1, Tmediation) AND declaresType(c2, TsysComp)) OR
      (declaresType(c1, TsysComp) AND declaresType(c2, Tmediation))  <<label : string =
      "Aucune paire des composants connectés n'est de type Tmediateur et TsysComp.";errMsg :
      string = "Paire de composants connectés de type Tmediateur et TsysComp !";>>;

// Deux composants qui sont attachés ne doivent pas être de type Tadaptateur et
// Tadaptateur
invariant Forall c1 : component in self.components |
   Forall c2 : component in self.components |
      !connected(c1, c2) -> (declaresType(c1, Tadaptateur) AND declaresType(c2, Tadaptateur))
      <<label : string = "Aucune paire des composants connectés n'est de type Tadaptateur et
      Tadaptateur.";errMsg : string = "Paire de composants connectés de type Tadaptateur et
      Tadaptateur !";>>;

// Deux composants qui sont attachés ne doivent pas être de type TsysComp et
// TsysComp
invariant Forall c1 : component in self.components |
   Forall c2 : component in self.components |
      !connected(c1, c2) -> (declaresType(c1, TsysComp) AND declaresType(c2, TsysComp))
      <<label : string = "Aucune paire des composants connectés n'est de type TsysComp et
      TsysComp.";errMsg : string = "Paire de composants connectés de type TsysComp et
      TsysComp !";>>;

// Tous les composants doivent être de type Tmediateur, Tadaptateur ou TsysComp
invariant Forall comp : component in self.components |
   (declaresType(comp, Tmediateur) AND satisfiesType(comp, Tmediateur)) OR
   (declaresType(comp, Tadaptateur) AND satisfiesType(comp, Tadaptateur)) OR
   (declaresType(comp, TsysComp) AND satisfiesType(comp, TsysComp)) <<label : string =
   "Tous les composants sont soit Tmediateur soit Tadaptateur soit TsysComp.";errMsg : string =
   "Composant détecté qui n'est ni Tmediateur ni Tadaptateur ni TsysComp!";>>;

// Tous les connecteurs doivent être de type Tconnecteur ou Tmédiation
invariant Forall conn : connector in self.connectors |
   declaresType(conn, Tmediation) OR declaresType(conn, Tconnecteur) <<label : string = "Tous
   les connecteurs sont de type Tconnecteur ou Tmediation.";errMsg : string = "Connecteur détecté
   qui n'est pas de type Tconnecteur ou Tmediation!";>>;

}
```

Requiert	« Adaptateur_SICo »

3.5. Patron « EntrepôtDeDonnées_SICo »

Ce patron présente l'architecture d'intégration de plusieurs bases de données dans un schéma global. Notons qu'à ce niveau d'abstraction, certains détails sont supprimés. Nous ne nous préoccupons pas, par exemple, de l'éventuelle différence qui pourrait exister entre le schéma local et le schéma export des sources de données.

Pour décrire la solution formelle de cette architecture, nous réutilisons, comme pour le patron « Médiateur-SICo », la solution formelle du patron « Adaptateur_SICo ». Néanmoins, le rôle de l'adaptateur dans une architecture de médiation est de traduire la requête exprimée par l'utilisateur en une requête spécifique acceptée par le système composant, dans une architecture d'entrepôt de données ; l'adaptateur a pour rôle de convertir les données de la source dans le modèle unique choisi par l'entrepôt. Pour faciliter la compréhension de l'architecture du patron, nous avons renommé les interfaces des composants (ou ports) et des connecteurs (ou rôles). La figure ci-après (cf. figure 62) rappelle les modes de coopération mis en oeuvre par ce patron.

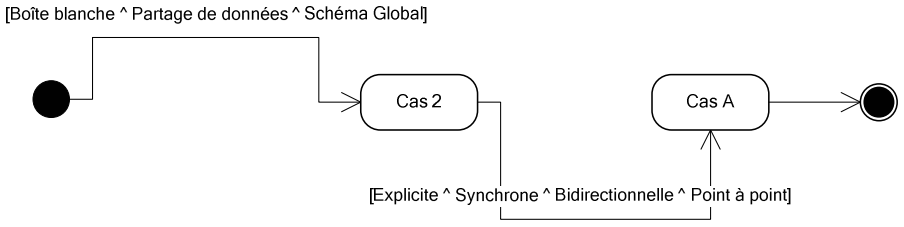
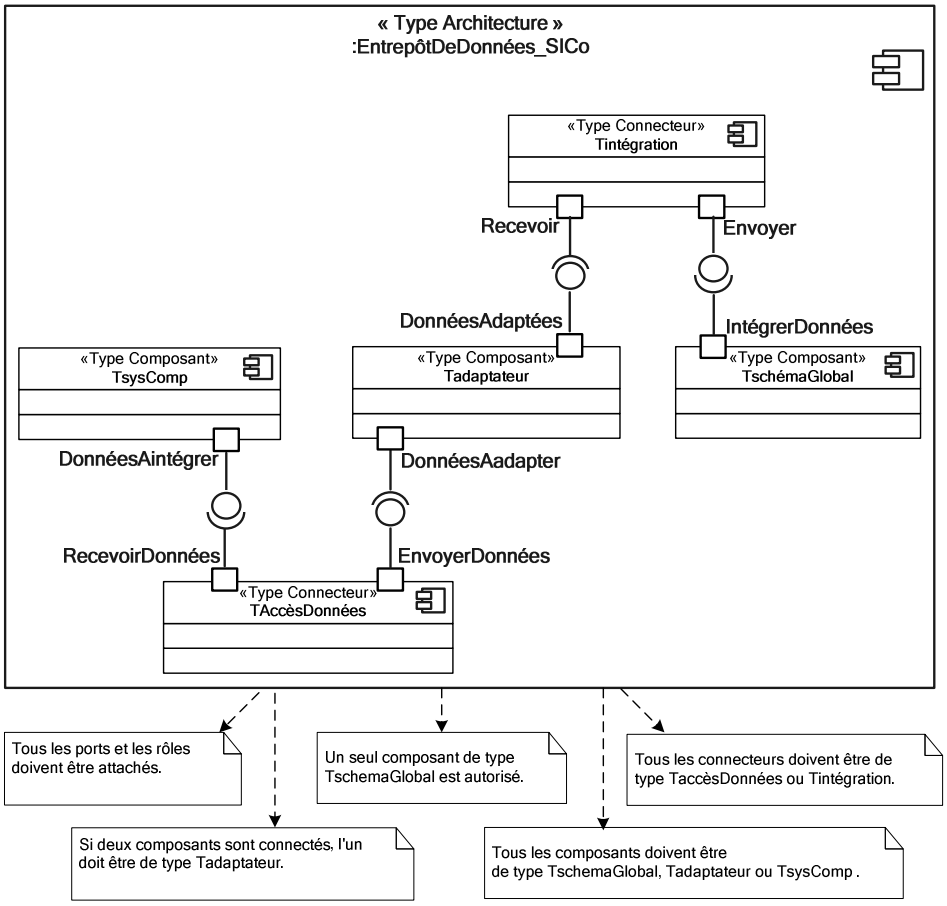


Figure 62 : Mode de coopération du patron "EntrepôtDeDonnées_SICo"

Identifiant	**EntrepôtDeDonnées_SICo**
Classification	Boîte Blanche ∩ Partage de Données ∩ schéma Global ∩ Explicite ∩ Synchrone ∩ Bidirectionnelle ∩ Point à point
Contexte	Ce patron nécessite l'utilisation du patron « Adaptateur_SICo » pour convertir les données de la source dans le modèle unique choisi pour l'entrepôt de données.
Problème	Organiser, coordonner, intégrer et stocker des données provenant de différentes sources dans une vue globale exploitable par l'utilisateur.
Description Comportementale	Intégrer des données de différentes sources de données dans un schéma global. L'intégration des différentes sources de données est fondée sur la construction d'un schéma global intégrateur (entrepôt de données). L'utilisation de l'adaptateur permet de convertir les données des sources dans un modèle de données pivot. Une fois construit, l'entrepôt de données devient indépendant des sources de données. Les données sont alors disponibles même si les ressources ne le sont pas. Cette intégration se fait grâce à certains outils qui résolvent des conflits syntaxiques et quelques conflits sémantiques. Les requêtes de l'utilisateur ne sont pas décomposées

	et renvoyées aux systèmes composants (cas du patron « Médiateur_SICo), mais plutôt traitées directement par l'entrepôt de données. Cette architecture permet alors à l'utilisateur d'interroger d'une manière synchrone une unique base de données qu'il faut maintenir à jour.
Solution modèle **Semi-formelle**	*Systèmes composants :* TschémaGlobal : schéma unifié intégrant les différentes sources de données. Tadaptateur : convertit et transfert les données de la source vers le schéma global. TsysComp : source de données. *Connecteur :* Tintégration : achemine les données de la source de données vers l'adaptateur. TaccèsDonnées : intègre les données envoyées par l'adaptateur dans le schéma global. 

Solution modèle
 	
formelle
	import $AS_GLOBAL_PATH\families\Adaptateur_SICo.acme;
	
	Family EntrepotDonnees extends Adaptateur_SIC with {
	// Déclaration des ports. Chaque port doit être attaché
	Port Type TportAdaptee = {
	heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg : string
	= "Ce port doit être attaché!";>>;

```
    }

    Port Type TintegrerDonnees = {
        heuristic size(self.attachedRoles) >= 1<<label : string = "Ce port est attaché.";errMsg :
string
        = "Ce port doit être attaché!";>>;
    }

    Component Type TschemaGlobal = {
        Port IntegrerDonnees : TintegrerDonnees =  new TintegrerDonnees;
    }

    Component Type Adaptateur extends Tadaptateur with {
        Port DonneesAdaptee : TportAdaptee =  new TportAdaptee;
    }

    Connector Type Tintegration = {
        Role Envoyer : TroleEnvoyer =  new TroleEnvoyer;
        Role Recevoir : TroleRecevoir =  new TroleRecevoir;
    }

    //Un seul composant peut être de type TschemaGlobal
    invariant size({Select c : component in self.components |
        declaresType(c, TschemaGlobal) }) == 1<<label : string = "Un seul composant est de type
        TschemaGlobal.";errMsg : string = "Erreur! Plus d'un composant est de type
        TschemaGlobal!";>>;

    // Si deux composants sont connectés l'un doit être de type Adaptateur
    invariant Forall c1 : component in self.components |
        Forall c2 : component in self.components |
            connected(c1, c2) -> ((declarestype(c1, Adaptateur)) OR (declarestype(c2, Adaptateur)))
            <<label : string = "L'un de chacun des deux composants connectés est de type
            Tadaptateur.";errMsg : string = "Détection de deux composants connectés dont l'un n'est
            pas de type Tadaptateur!";>>;

    // Tous les composants doivent être de type TschemaGlobal, Adaptateur ou  TsysComp
    invariant Forall comp : component in self.components |
        (declaresType(comp, TschemaGlobal) AND satisfiesType(comp, TschemaGlobal)) OR
        (declaresType(comp, Adaptateur) AND satisfiesType(comp, Adaptateur)) OR
        (declaresType(comp, TsysComp) AND satisfiesType(comp, TsysComp)) <<label : string =
        "Tous les composants sont soit TschemaGlobal soit Tadaptateur soit TsysComp.";errMsg :
        string = "Composant détecté qui n'est ni TschemaGlobal ni Tadaptateur ni TsysComp!";>>;

    // Tous les connecteurs doivent être de type TaccesDonnées ou Tintegration
    invariant Forall conn : connector in self.connectors |
        (declaresType(conn, Tintegration) AND satisfiesType(conn, Tintegration)) OR
        (declaresType(conn, TaccesDonnees) AND satisfiesType(conn, TaccesDonnees)) <<label :
        string = "Tous les connecteurs sont de type TaccesDonnees ou Tintegration.";errMsg :
string
        = "Connecteur détecté qui n'est pas de type TaccesDonnees ou Tintegration!";>>;

}
```

Requiert	« Adaptateur_SICo »

3.6. Architecture « Trois-tiers_SICo »

L'architecture à deux niveaux (appelée aussi architecture deux-tiers) caractérise le système client/serveur dans lequel le client demande un service et le serveur le lui fournit directement (cf. patron « ClientServeur_SICo »). Dans l'architecture trois-tiers appelée aussi architecture à trois niveaux, un niveau supplémentaire est ajouté entre le client et le serveur. Cette couche permet de séparer les traitements (couche logique) de l'interface graphique (couche client) et du serveur de base de données (couche de données ou de persistance). Cette architecture est très courante dans les applications où de nombreux utilisateurs doivent accéder à une même source de données. Pour modéliser la solution modèle du patron « Trois-tiers_SICo », nous réutilisons la solution formelle du patron « ClientServeur_SICo ». Une synthèse des modes de coopération traités par ce patron est présentée par la figure suivante (cf. figure 63).

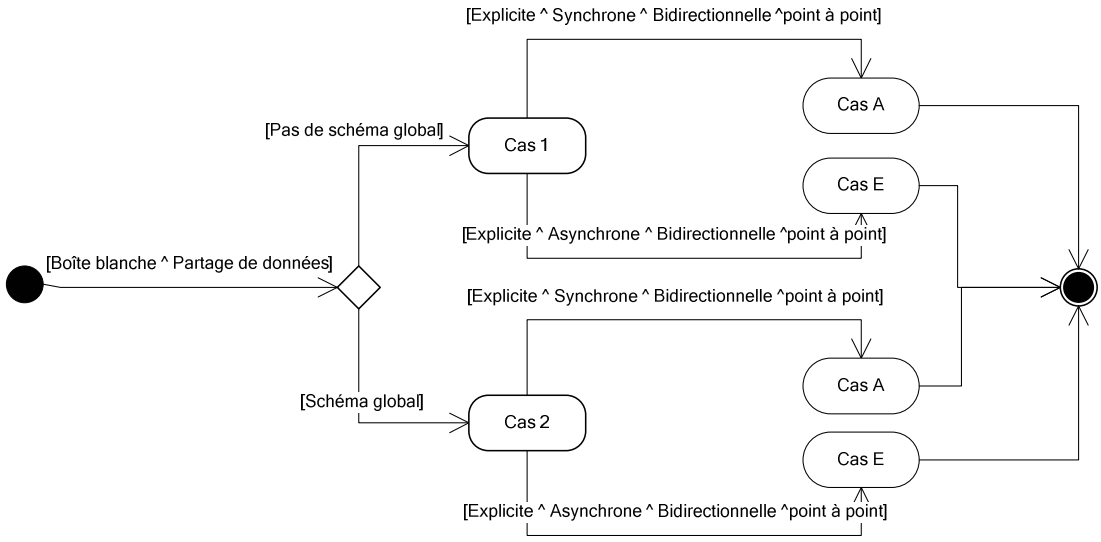
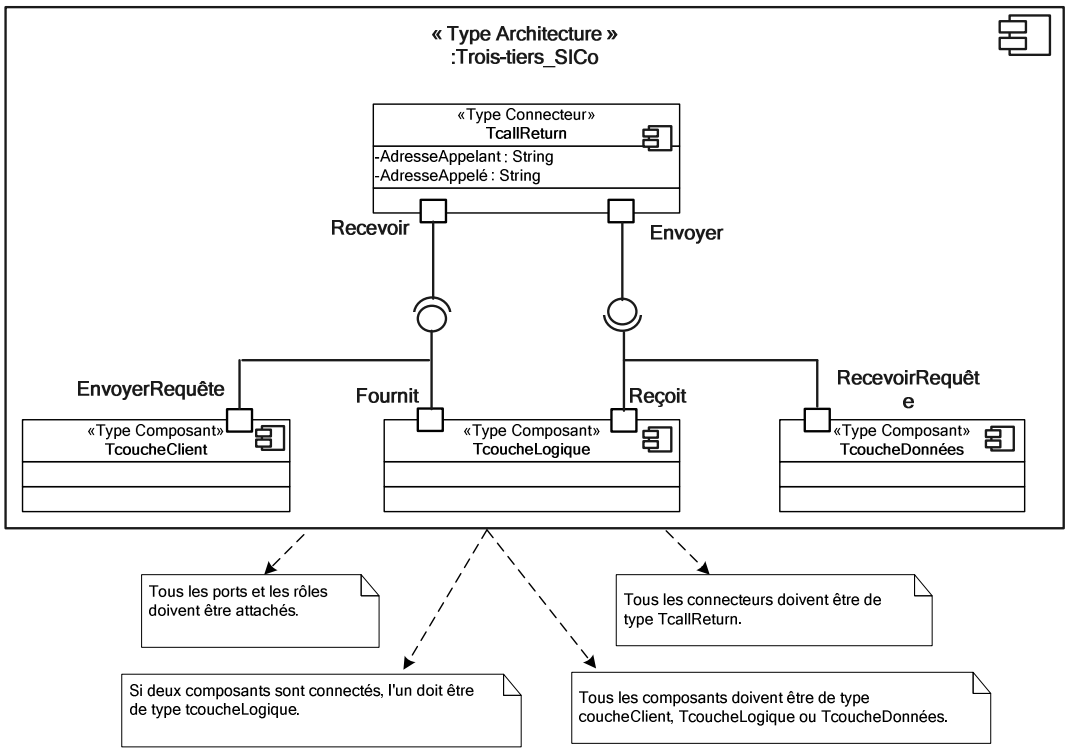


Figure 63 : Mode de coopération du patron "Trois-tiers_SICo"

Identifiant	**Trois-tiers_SICo**
Classification	*Divers modes de coopération possibles :* 1) Boîte Blanche ∩ Partage de Données ∩ Schéma Global ∩ Explicite ∩ Synchrone ∩ Bidirectionnelle ∩ Point à point 2) Boîte Blanche ∩ Partage de Données ∩ Pas de schéma Global ∩ Explicite ∩ Synchrone ∩ Bidirectionnelle ∩ Point à point 1) Boîte Blanche ∩ Partage de Données ∩ Schéma Global ∩ Explicite ∩ Asynchrone ∩ Bidirectionnelle ∩ Point à point 2) Boîte Blanche ∩ Partage de Données ∩ Pas de schéma Global ∩ Explicite ∩ Asynchrone ∩ Bidirectionnelle ∩ Point à point

	Formule simplifiée : Boîte Blanche ∩ Partage de Données ∩ Explicite ∩ Bidirectionnelle ∩ Point à point
Contexte	Ce patron peut nécessiter l'utilisation du patron « Adaptateur_SICo ».
Problème	Séparer la logique applicative et les données dans une architecture de type client/serveur (ou deux-tiers).
Description Comportementale	Cette architecture est, comme son nom l'indique, divisée en trois couches. La première correspond à la couche présentation (c'est-à-dire la couche client) qui ne contient que l'interface de l'utilisateur final. La deuxième couche est la couche logique qui est le cœur même de l'application et fait le lien entre le client et le serveur, formant ce qu'on appelle le serveur d'application. La dernière couche est celle de gestion des données (le serveur de données), qui consiste en une ou plusieurs base(s) de données ou tout système de stockage de données. Chaque couche peut être modélisée, développée et testée individuellement. Cette architecture sépare la logique de présentation, la logique applicative et la logique de persistance (ou de données). La relation de dépendance entre les trois couches siginifie qu'un composant de la couche inférieure ne peut pas accéder à un composant de la couche supérieure. Autrement dit, l'accès à la couche de données n'est effectué que par la couche logique.
Solution modèle **Semi-formelle**	*Systèmes composants :* TcoucheClient : composants permettant aux utilisateurs d'utiliser l'application. TcoucheLogique : serveur d'application. TcoucheDonnées : serveur(s) de base(s) de données. *Connecteur :* TcallReturn : connecteur permettant la communication entre les différentes couches.



Solution modèle formelle

```
import $AS_GLOBAL_PATH\families\ClientServeur_SICo.acme;

Family TroisTiers extends ClientServeur with {
   Property Type TproprieteCouche = Enum { client, logique, donnees };

   Component Type TcoucheClient extends TcompClient with {
      Property Couche : TproprieteCouche = client;
   }

   Component Type TcoucheLogique = {
      Port Fournit : TportClient =  new TportClient;
      Port Utilise : TportServeur =  new TportServeur;
      Property Couche : TproprieteCouche = logique;
   }

   Component Type TcoucheDonnees extends TcompServeur with {
      Property Couche : TproprieteCouche = donnees;
      Property ServeurPrimaire : boolean << Default : boolean = false;  >>;
   }

   Connector Type TcallReturn = {
      //Adresse de l'appelant et de l'appelé
      Property AdresseAppelant : string;
      Property AdresseAppele : string;
   }

   // Si deux composants sont connectés, l'un doit être de type TcoucheLogique
   invariant Forall C1 : component in self.components |
      Forall C2 : component in self.components |
         connected(C1, C2) -> (declaresType(C1, TcoucheLogique) OR declaresType(C2,
         TcoucheLogique))  <<label : string = "Si deux composants sont connectés l'un doit être de
         type TcoucheLogique.";errMsg : string = "Détection de deux composants connectés don't l'un
         n'est pas de type TcoucheLogique !";>>;

   // Tous les composants doivent être de type TcoucheClient, TcoucheLogique ou TcoucheDonnées
```

	invariant Forall comp : component in self.components |
	(declaresType(comp, TschemaGlobal) AND satisfiesType(comp, TschemaGlobal)) OR
	(declaresType(comp, Adaptateur) AND satisfiesType(comp, Adaptateur)) OR
	(declaresType(comp, TsysComp) AND satisfiesType(comp, TsysComp)) <<label : string = "Tous les composants sont soit TcoucheClient soit TcoucheLogique soit TcoucheDonnées.";errMsg : string = "Composant détecté qui n'est ni TcoucheClient ni TcoucheLogique ni TcoucheDonnées!";>>;
	
	// Tous les connecteurs doivent être de type TcallReturn
	invariant Forall conn : connector in self.connectors |
	declaresType(conn, TcallReturn) AND satisfiesType(conn, TcallReturn) <<label : string = "Tous les connecteurs sont de type TcallReturn.";errMsg : string = "Connecteur détecté qui n'est pas de type Tcallreturn!";>>;
	
	}

Requiert	« Adaptateur_SICo ».
Raffine	« ClientServeur_SICo ».

4. Conclusion

Dans ce chapitre, nous avons présenté un système de Patrons pour les Architectures Coopératives (PACO). Les patrons processus proposés dans le système PACO se présentent comme des fragments de démarche pour orienter le concepteur vers un autre patron processus ou vers un patron produit. Pour faciliter leur utilisation, chaque patron processus regroupe un sous-ensemble de critères de coopération. Ces derniers sont utilisés comme des gardes pour représenter des transitions conditionnelles dans les diagrammes d'activités décrivant la solution démarche du patron processus. Pour plus de lisibilité, nous avons nommé la fin de chaque transition (cas 1 à 5 et cas A à F). La combinaison des différents cas réalise un mode de coopération et mène vers un ou plusieurs patrons produit.

Actuellement, un mode de coopération peut conduire vers plusieurs patrons produit. C'est en effet le cas des patrons « Médiateur_SICo » et « Trois-tiers_SICo » qui répondent tous les deux au mode de coopération "Boîte blanche ^ Partage de données ^ Pas de schéma global". Dans ce cas, la rubrique "problème" permet de différencier les deux architectures voisines. Inversement, un patron produit peut répondre à plusieurs modes de coopération. C'est le cas par exemple du patron « ClientServeur_SICo » qui peut réaliser la coopération en mode synchrone ou asynchrone. Dans ce cas, le choix est fait par l'ingénieur d'applications au moment de l'instanciation de la solution modèle.

Le chapitre suivant présente les différents outils pour l'instrumentation du système PACO. La validation de la démarche sera réalisée en se basant sur le projet industriel OSCAR (Organisation de la Simulation en Conception pour la CApitalisation et la Réutilisation). Ceci permettra de montrer un exemple concret d'imitation et de réutilisation d'un des patrons produit.

CHAPITRE VI

INSTRUMENTATION & VALIDATION

1. Introduction

Toute ingénierie de Système d'Information repose sur des méthodes. C. Rolland [ROLLAND & AL., 88], décrit une méthode en définissant quatre composants complémentaires : des modèles, des langages, une démarche et des outils (ou techniques). A ce stade, nous avons abordé les trois premières composantes. Ce chapitre est consacré à la présentation de la dernière composante, à savoir celle des outils. Les outils concernent les trois premières composantes et ont pour objectif d'aider leur mise en œuvre.

Pour mettre en œuvre notre système de patrons, nous avons besoin de deux outils support. Le premier outil est dédié à la création et à la gestion de notre système de patrons : nous optons pour l'atelier de gestion de patrons *AGAP* (**A**telier de **G**estion et d'**A**pplication de **P**atrons (section 2.1). Le deuxième est nécessaire pour décrire les spécifications formelles des architectures de coopération (la solution modèle), pour cela, nous utilisons l'outil support d'Acme : *AcmeStudio* (section 2.2). Toutefois, nous n'associons aucun outil directement aux diagrammes de composants d'UML 2.0 (solution semi-formelle). Ces derniers sont présentés en tant qu'image et leur rôle se réduit, ainsi, à une meilleure compréhension de la solution modèle. L'instanciation effective de la solution est assurée par l'outil AcmeStudio. Ce premier point qui concerne les outils fera l'objet de la première partie de ce chapitre (section 2). Dans la deuxième partie (section 3), nous appliquons le système PACO (Patrons pour les Architectures COopératives) pour spécifier une application industrielle. Ceci nous permet de valider et de vérifier l'adéquation de notre solution pour la modélisation des Systèmes d'Information Coopératifs (SICo) sur un cas réel.

2. Les outils support

Dans cette section, nous décrivons les deux outils support : AGAP (section 2.1) et AcmeStudio (section 2.2). Pour chacun de ces outils, nous décrivons les fonctionnalités destinées à l'ingénieur de patrons, d'une part, et celles allouées à l'ingénieur d'applications d'autre part.

2.1. L'outil support AGAP

L'atelier de gestion de patrons *AGAP* (**A**telier de **G**estion et d'**A**pplication de **P**atrons) est développé au sein de l'équipe SIGMA dans le labortatoire LSR [CONTE & AL., 01] [TASTET, 04] [JAUSSERAN, 05]. AGAP propose deux modules :

1. un module système **pour la réutilisation** (*for reuse*) centré sur l'identification, la spécification et l'organisation des patrons,
2. un module système **par la réutilisation** (*by reuse*) centré sur la sélection, l'imitation, l'adaptation et l'intégration des patrons dans des systèmes d'information.

Ces deux modules sont utilisés par les deux acteurs de l'atelier : **l'ingénieur de patrons** et **l'ingénieur d'applications**.

Trois phases principales composent le cycle de vie d'un système de patrons (cf. figure 64) :

1. **La phase de création :** l'ingénieur de patrons analyse le problème à résoudre et conçoit de nouveaux patrons (organisés en système), avec création d'un nouveau formalisme le cas échéant.
2. **La phase de validation :** pendant cette phase, l'ingénieur de patrons teste la validité des solutions offertes par les patrons et vérifie l'intégrité et la cohérence des interactions entre patrons (relations inter-patrons) au sein du système de patrons.
3. **La phase d'utilisation et de maintenance :** le système de patrons entre dans le processus de production des systèmes d'information. Au cours de cette phase, l'ingénieur d'applications peut détecter des anomalies ou bien il peut être confronté à des problèmes qui n'ont pas été traités dans le système de patrons. Dans ce cas, il peut formuler des demandes vis-à-vis de l'ingénieur de patrons pour lui faire part d'anomalies ou de nouveaux problèmes à résoudre.

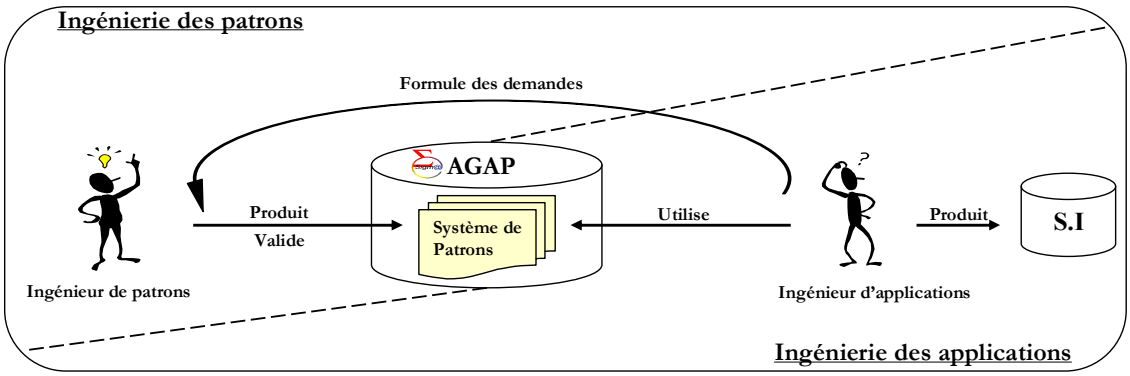


Figure 64 : Cycle de vie d'un système de patrons dans AGAP [JAUSSERAN, 05]

Les différentes fonctionnalités d'AGAP sont présentées dans le schéma suivant (figure 65). Ces fonctionnalités sont destinées à deux types d'acteurs : l'ingénieur de patrons et l'ingénieur d'applications. Dans la suite nous détaillons les fonctionnalités de chacun de ces acteurs.

2.1.1. L'ingénieur de patrons

AGAP fournit à l'ingénieur de patrons toutes les fonctionnalités lui permettant de gérer des formalismes et des systèmes de patrons. Pour créer un système de patrons, l'ingénieur doit déterminer ou créer le formalisme à utiliser (par exemple, le formalisme de Gamma). La création d'un nouveau formalisme peut se faire par adaptation ou à partir de rien. Dans ce cas, l'ingénieur doit vérifier et valider le nouveau formalisme. La deuxième étape consiste à créer, vérifier et valider le système de patrons (en tant que conteneur). La dernière étape consiste à saisir, vérifier et valider les patrons (par exemple, les vingt-trois patrons de Gamma). L'ingénieur a aussi la possibilité d'identifier les relations qui existent entre les patrons et de déterminer les règles d'imitation (pour les patrons produit).

La fonction d'extraction de systèmes de patrons sous AGAP est particulièrement importante. L'extracteur permet d'extraire toute démarche de développement formalisée par un langage de patrons sous AGAP. Il permet de fournir un guide méthodologique autonome pour guider les ingénieurs d'applications dans un processus de développement.

L'un des grands avantages d'AGAP est de considérer les formalismes distinctement des systèmes de patrons. Cela permet ainsi à plusieurs systèmes de patrons de pouvoir utiliser le même formalisme ou bien de construire de nouveaux formalismes à partir de ceux existants.

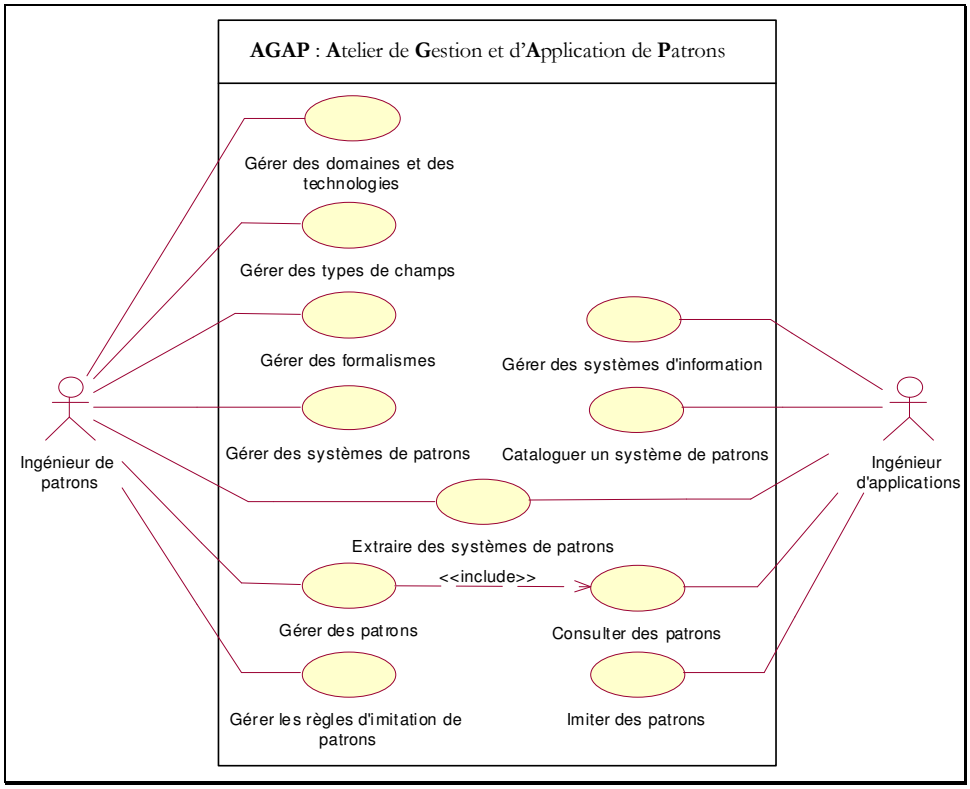


Figure 65 : Diagramme simplifié des cas d'utilisation d'AGAP

2.1.2. L'ingénieur d'applications

L'objectif de l'ingénieur d'applications est de sélectionner et d'imiter les patrons créés et validés par l'ingénieur de patrons.

Actuellement l'atelier propose à l'ingénieur d'applications des moyens pour consulter les systèmes de patrons validés, par l'intermédiaire des fiches de patrons (cas d'utilisation « consulter des patrons ») complètes ou résumées (cas d'application « cataloguer un système de patrons »).

L'imitation consiste en une duplication suivie d'une adaptation de la solution du patron au contexte spécifique du problème. Adapter un patron revient à appliquer les opérateurs de renommage, redéfinition, ajout et suppression sur les attributs, les méthodes et les associations afin d'arriver à une solution adaptée au nouveau contexte.

Pour cette fonctionnalité, des travaux de recherche [ARNAUD & AL., 04] et des développements sont en cours. Il est possible de procéder à des essais d'imitation sur des patrons de conception (patrons de Gamma [GAMMA & AL., 95] par exemple) proposant une solution structurelle à base de diagrammes de classes UML. L'imitation porte sur les classes et les relations de généralisation / spécialisation.

L'opération d'imitation propre à chaque modèle de produit est à mettre en œuvre non pas dans AGAP mais dans des outils spécifiques. C'est donc dans l'environnement de modélisation du langage Acme que se fera l'imitation des solutions modèles offertes par les patrons produit de PACO. Nous consacrons la section suivante à la description de cet outil.

2.2. L'Outil AcmeStudio

Le langage Acme est supporté par l'outil graphique *AcmeStudio* utilisé pour gérer des conceptions architecturales basées sur Acme. Cet outil permet au concepteur de :

- vérifier l'exactitude et la cohérence de la description architecturale,
- éditer graphiquement la description architecturale,
- décrire et instancier des familles (appelées aussi types) d'architectures offrant ainsi de ce fait l'opération d'imitation.

L'environnement d'AcmeStudio offre deux modules :

1. un module pour la description des familles d'architectures favorisant une ingénierie d'architectures logicielles **pour la réutilisation**,
2. un module pour la spécification d'architectures logicielles **par la réutilisation** de famille(s) d'architecture(s) déjà prédéfinie(s).

Ces modules sont respectivement utilisés par l'ingénieur de patrons et l'ingénieur d'applications et sont schématisés par la figure suivante (cf. figure 66). L'ingénieur d'applications peut formuler des demandes vis-à-vis de l'ingénieur de patrons pour lui soumettre de nouveaux besoins (cf. patron « Nouveaux besoins », section 7 du chapitre IV) ou résoudre certaines anomalies détectées au cours de la réutilisation de(s) type(s) d'architectures.

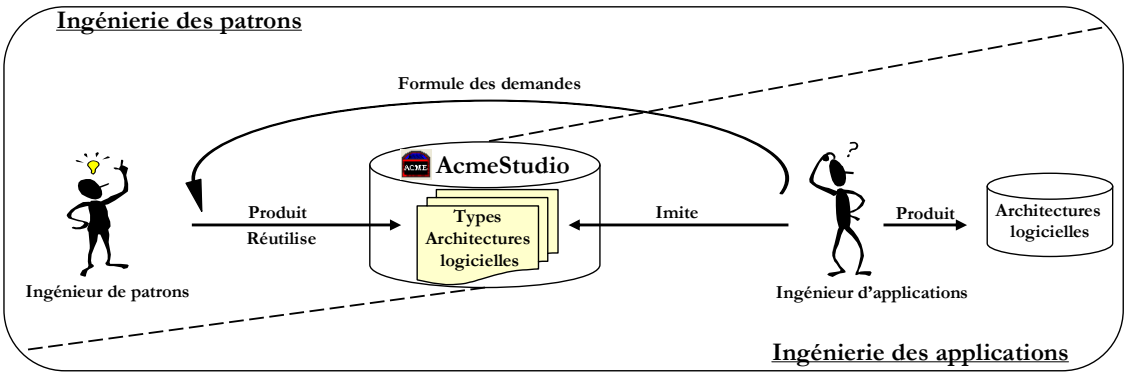


Figure 66 : Cycle de vie de types d'architectures logicielles dans AcmeStudio

Nous avons regroupé les différentes fonctionnalités offertes par AcmeStudio dans le diagramme des cas d'utilisation de la figure 67 selon qu'elles soient destinées à l'ingénieur de patrons ou à l'ingénieur d'applications. Notons que, quelque soit l'acteur, la première étape dans AcmeStudio consiste à créer un nouveau projet pour gérer localement les différents fichiers relatifs à une famille d'architectures ou à une architecture logicielle. Une autre fonctionnalité en commun consiste en la création des représentations. Rappelons que la notion de représentation sous Acme permet de supporter la description hiérarchique des entités conceptuelles. Les autres fonctionnalités sont décrites dans les paragraphes suivants (cf sections 2.2.1 et 2.1.2).

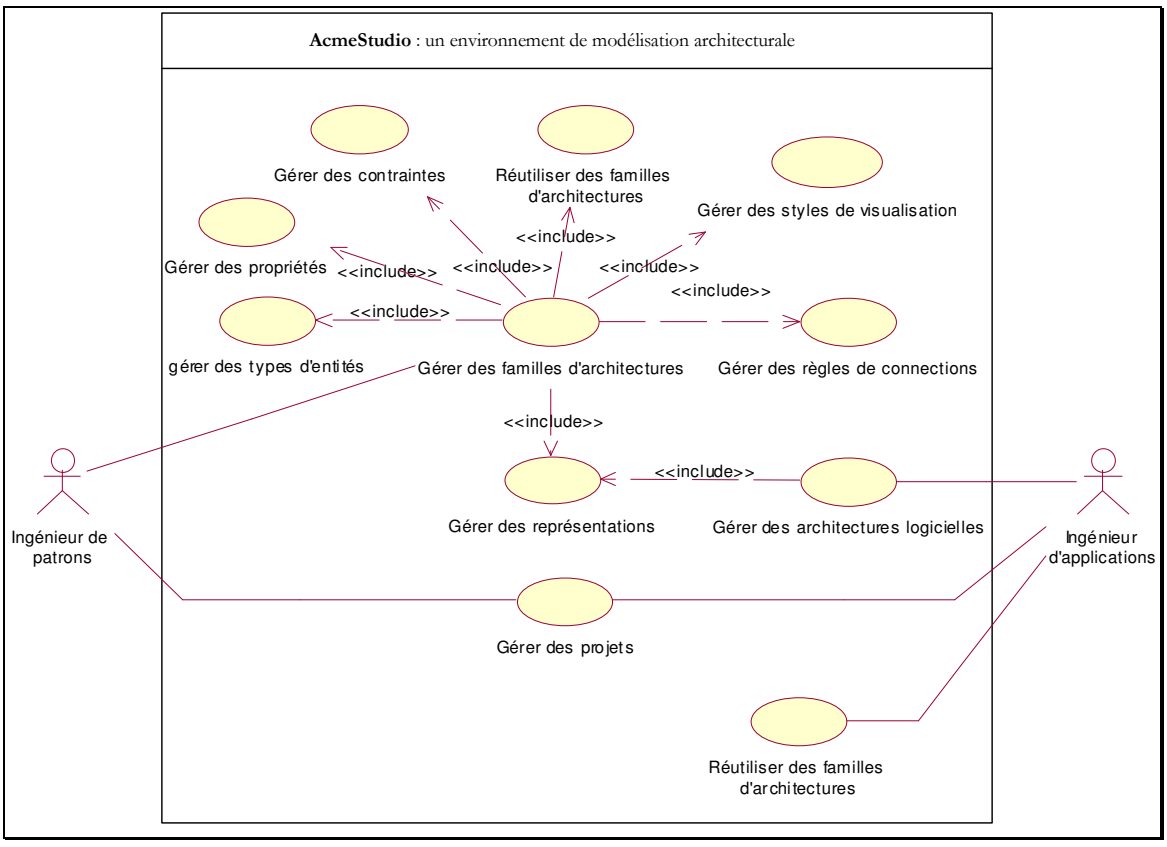


Figure 67 : Diagramme simplifié des cas d'utilisation d'AcmeStudio

2.2.1. L'ingénieur de patrons

AcmeStudio met à la disposition de l'ingénieur de patrons différentes fonctionalités pour gérer de nouvelles familles d'architectures logicielles. La création d'une nouvelle famille d'architectures peut se faire par réutilisation d'entités déjà prédéfinies dans une famille d'architectures existante ou à partir de rien. Pour créer une nouvelle famille d'architectures, l'ingénieur de patrons doit tout d'abord définir les types des différentes entités : ports, rôles, composants et connecteurs. Il peut ensuite définir les types des propriétés des différents éléments conceptuels. L'étape suivante consiste à définir des règles et des contraintes de réutilisation sur les différentes entités ainsi que sur l'architecture globale en utilisant le langage Armani [MONROE, 98]. Pour cette raison, AcmeStudio inclut aussi le contrôleur de contraintes du langage Armani pour vérifier l'ensemble des règles et des contraintes définies dans la description architecturale. La dernière étape consiste à créer des styles de visualisation et des règles de connexion. Les premiers permettent d'attribuer à chaque élément conceptuel une visualisation qui lui sera spécifique au moment de l'instanciation. Les seconds déterminent les modes de connexions possibles entre les différentes entités.

2.2.2. L'ingénieur d'applications

L'ingénieur d'applications a pour tâche de construire des architecturs logicielles comme instances d'un type d'architecture bien déterminé. Cet aspect est particulièrement important puisqu'il permet d'imiter et d'adapter des familles d'architectures.

Notons finalement que AcmeStudio est conçu en tant que "plugin" de l'environnement Eclipse [DES RIVIERES & AL., 04]. Ce dernier peut être employé dans une variété d'applications de modélisation et d'analyse. Eclipse est un environnement de développement ouvert. Il fournit un plugin-environnement permettant des prolongements faciles d'AcmeStudio, de nouvelles analyses, des ajouts de fonctionnalités, et la personnalisation de nouveaux environnements architecturaux conçus en fonction d'une organisation particulière.

2.3. Instrumentation du système PACO

L'instrumentation du système PACO a été réalisée conjointement par les outils AGAP et AcmeStudio (cf. figure 68). Le premier permet de gérer les patrons produit et processus du système PACO, alors que le second est utilisé pour la spécification des familles d'architectures coopératives.

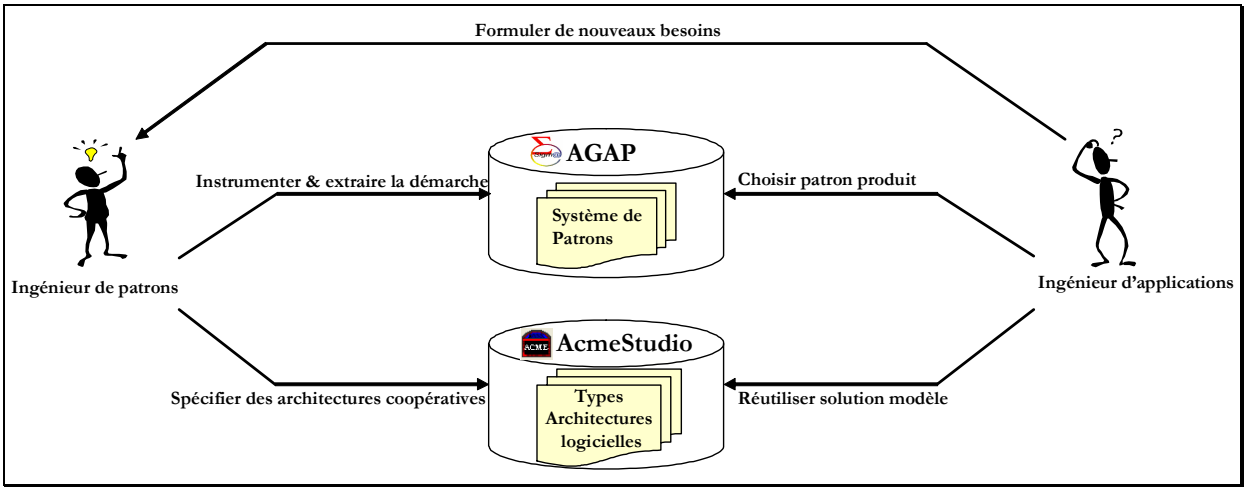


Figure 68 : Instrumentation du système PACO

La fonction d'extraction des systèmes de patrons de l'atelier AGAP a permis de générer un guide méthodologique correspondant à un site web autonome. Nous énumérons quelques fonctionnalités offertes par ce site Web :

- Menus arborescents du système de patrons et du formalisme,
- Présentation des fiches de patrons et de formalismes,
- Cartographie dynamique orientée système de patrons produit compte tenu des relations entres ces derniers,
- Cartographie orientée systèmes de patrons processus à base d'images réactives (préparées par l'ingénieur de patrons),
- Moteur de recherche « full text » autonome.

La copie d'écran de la figure 69 illustre le site web du système PACO réalisé par la fonction d'extraction de l'atelier AGAP.

3. Validation de la démarche

Cette partie illustre l'application du système PACO dans le cadre du projet OSCAR[4] (**O**rganisation des **S**imulations en **C**onception par la **C**apitalisation et la **R**éutilisation). Ce projet a pour objectif de faciliter la capitalisation et la réutilisation des modèles et des démarches de simulation numérique dans le processus de développement des produits [SAIDANE & AL., 03]. Notre objectif au sein du projet est d'étudier les interactions entre deux Systèmes d'Information industriels : le Système d'Information de Calcul (SIC) et le Système d'Information Produit (SIP). Dans la suite, nous présentons brièvement ces deux SI et nous étudions leurs interactions.

[4] OSCAR est un projet pluridisciplinaire financé par la région Rhône Alpes et qui a fait intervenir plusieurs laboratoires (LSR – équipe SIGMA, GILCO, 3S et LISI) ainsi que des partenaires industriels (Schneider Electric, PCO Technologies et Renault).

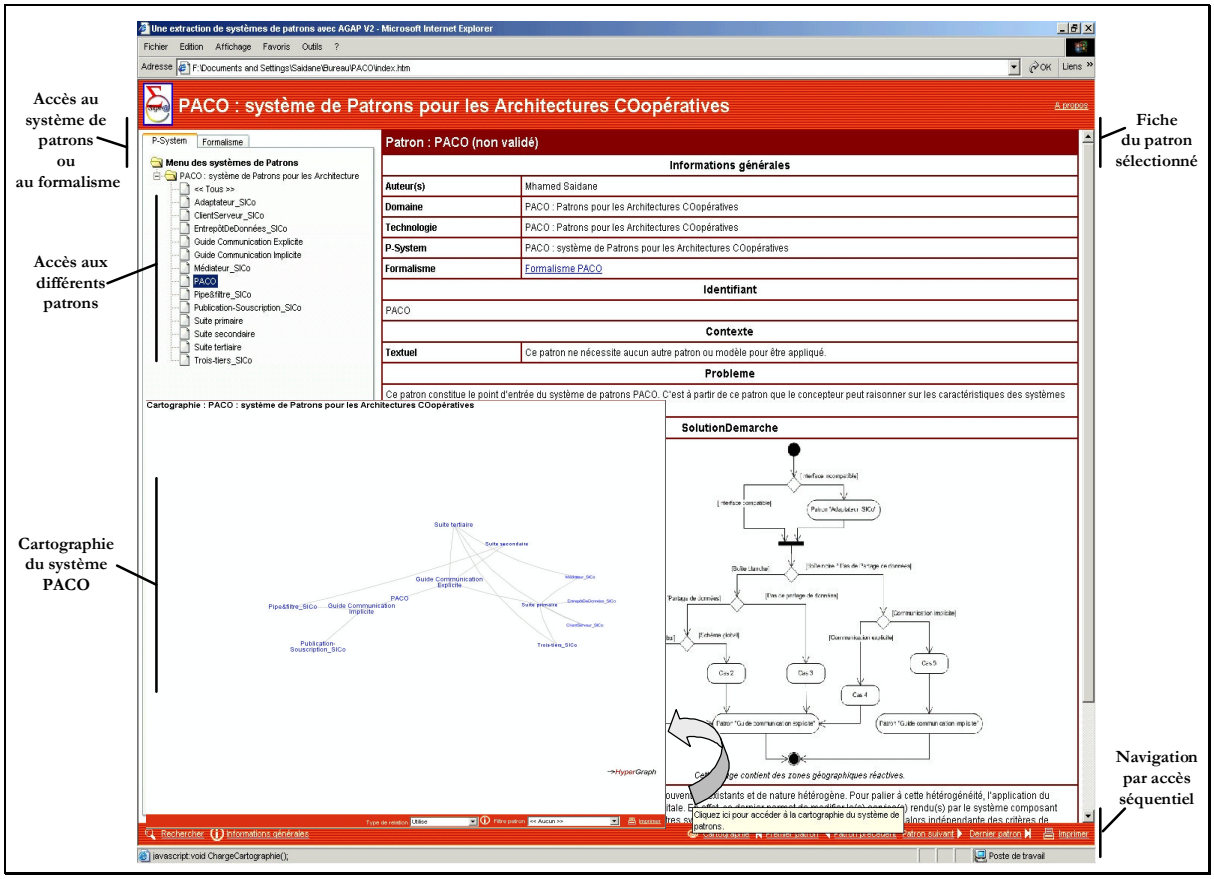


Figure 69 : Site web du système PACO

3.1. Système d'Information de Calcul

Le Système d'Information de Calcul (SIC) supporte l'ensemble des modèles utilisés ou produits par la simulation numérique des produits virtuels. Plusieurs modèles sont nécessaires pour représenter les produits en cours d'élaboration. Cette diversité est due à plusieurs facteurs : les différentes représentations (géométriques, fonctionnelles,…) du même produit à un instant donné de la conception, l'utilisation de plusieurs outils de conception, la diversité des métiers de l'entreprise intervenant sur le produit (mécanique, électrique, thermique, etc.). Pour élaborer, valider et remettre en cause ces modèles, les concepteurs utilisent de nombreuses règles de calcul mécanique afin de mieux estimer les comportements mécaniques leur permettant ainsi de déterminer les meilleurs choix technologiques.

Vue l'ambiguïté des concepts gérés dans le SIC, nous nous sommes basés sur un projet particulier : le projet *Guepar* [TROUSSIER, 99]. La figure 70 représente la chronologie du processus de calcul (cas courant) sur un exemple réel décrit dans un "évènementiel[5]" ou tableau de suivi. Il s'agit d'un exemple concret de démarche de processus de calcul. Cet exemple est emprunté à Schneider Electric. Il décrit une nouvelle conception d'un bras du châssis support de composants pour les tableaux électriques des installations électriques.

[5] Un évènementiel est une représentation permettant au concepteur de tracer le déroulement de ses essais ainsi que les informations à capitaliser.

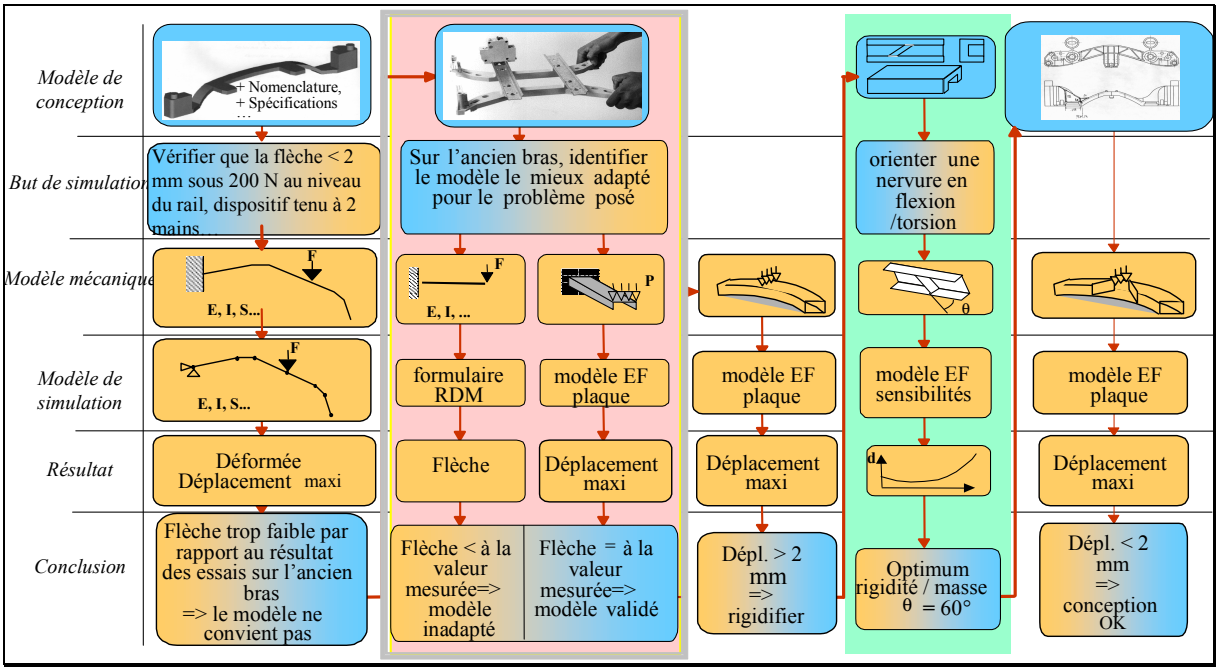


Figure 70 : Un événementiel, démarche de calcul appliquée à l'exemple d'un châssis

Dans la figure 71, nous présentons le diagramme de classes simplifié du SIC déduit à partir de l'exemple de la figure 70 et des informations recueillies auprès des industriels. La résolution d'un projet revient à résoudre plusieurs problèmes de calcul. Pour être résolus, ces derniers font appel à une ou plusieurs études qui à leur tour sont composées d'un ou de plusieurs cycles menés en parallèle. Dans l'événementiel (cf. figure 70) chaque colonne représente une étude ; à titre d'exemple la colonne encadrée en pointillé est une étude composée de deux cycles. Une étude peut être suivie par d'autres, créées au sein d'un même projet ou par une étude appartenant à un autre projet et qui est réutilisée pour débloquer une situation donnée dans le projet en cours. Dans notre exemple, la première étude est suivie par une étude réalisée dans le projet de l'ancien bras (c'est l'étude encadrée dans la figure 70). De même, un cycle possède des suivants dans une étude. L'association récursive « Suivant Chronologique » schématise la relation qui existe entre deux cycles qui se suivent appartenant chacun à une étude différente. L'association « Suivant Ds Etude » schématise le lien qui existe entre deux cycles dont l'un est postérieur à l'autre dans leur étude commune. Les notions de « Modèle de conception », « But de simulation », « Modèle Mécanique », « Modèle de Simulation », « Résultat » et « Conclusion », correspondent aux connaissances métiers à capitaliser.

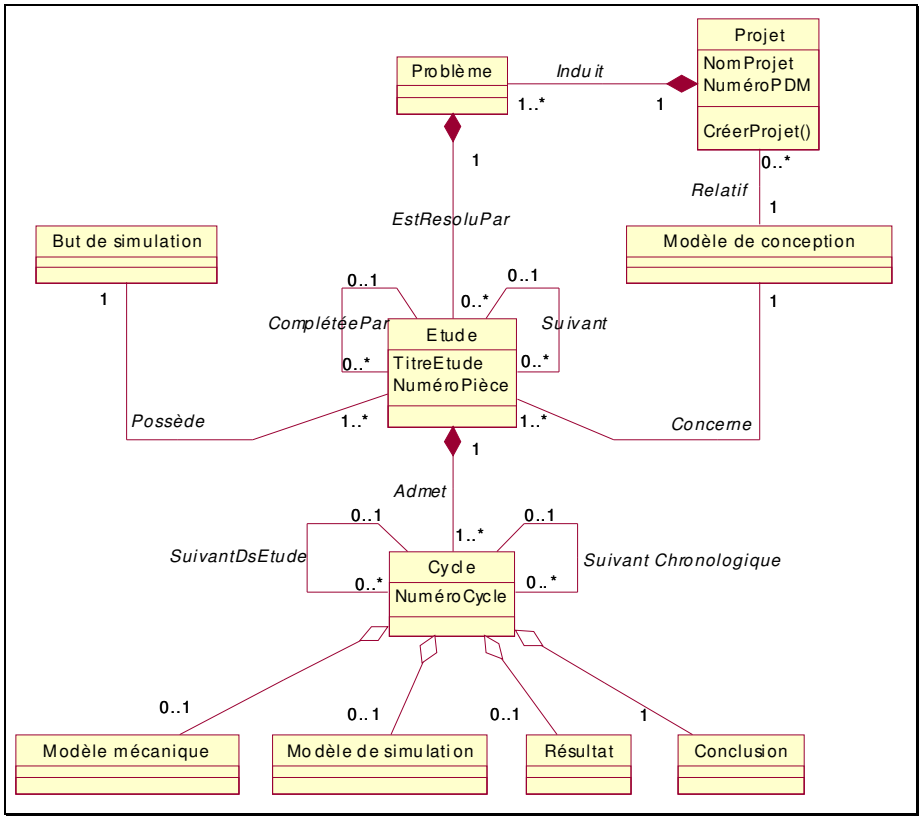


Figure 71 : Diagramme de classes simplifié du SIC

Le SIC co-existe d'une part avec divers systèmes d'information techniques servant d'autres fonctions de l'entreprise liées à la conception des produits, et d'autre part avec un système d'information plus global appelé le "Système d'Information Produit" (SIP).

3.2. Système d'Information Produit

Le Système d'Information Produit (SIP) gère l'ensemble des informations techniques nécessaires aux activités de développement des produits. L'objectif d'un Système d'Information Produit (SIP) est de gérer l'ensemble du patrimoine informationnel du produit tout au long de son cycle de vie, depuis sa première évocation en stratégie d'entreprise jusqu'à sa destruction [GZARA, 00]. Autrement dit un SIP gère l'ensemble des informations relatives aux produits.

L. Gzara dans [GZARA, 00] propose un référentiel produit fixant la terminologie des divers concepts gérés dans les SIP. Dans cette section, nous présentons uniquement un fragment du diagramme de classes représentant les concepts qui nous intéressent dans le cadre de la coopération entre SIP et SIC. Le diagramme de classes de la figure 72 présente particulièrement le concept de produit et d'article. Un article est un composant du produit, depuis la pièce élémentaire (ne présentant aucun intérêt à être décomposée) jusqu'au produit lui-même, en passant par toutes les strates intermédiaires de composition. Un article peut donc être composé (un ensemble composé en interne) ou catalogue (matières premières, pièces élémentaires, ensembles inséparables, etc.). Il peut être aussi un article à variante permettant de décliner divers types de produits à partir d'une conception de base. Par contre un article constant ne possède pas de variantes. C'est donc un article "concret", obligatoirement retenu dans la composition d'un type de produit donné.

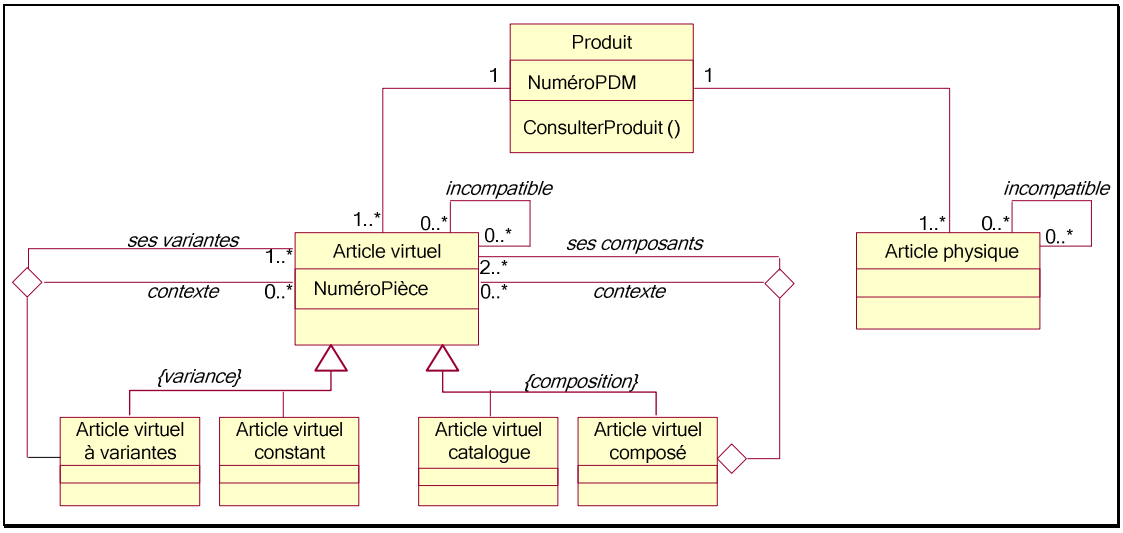


Figure 72 : Diagramme de classes partiel du SIP

3.3. Modélisation de la coopération entre SIC et SIP

L'un des objectifs du projet OSCAR est d'étudier les interactions entre Système d'Information Produit (SIP) et Système d'Information de Calcul (SIC). En effet, ces deux SI échangent des informations et partagent des données et des processus.

1. **Flux d'information :** Le SIP gère des données et des processus sur les produits. Ces processus sont chacun composés de diverses activités de calcul. Ces dernières utilisent des données du modèle du produit gérées dans le SIP et produisent d'autres données, résultats du calcul, supportées dans des modèles mécaniques et de simulation. Ces données du modèle mécanique ou de simulation sont ensuite à réintégrer dans le modèle de produit du SIP afin de capitaliser les calculs et les simulations effectuées. La figure 73 illustre cet échange de flux d'information entre les deux systèmes.

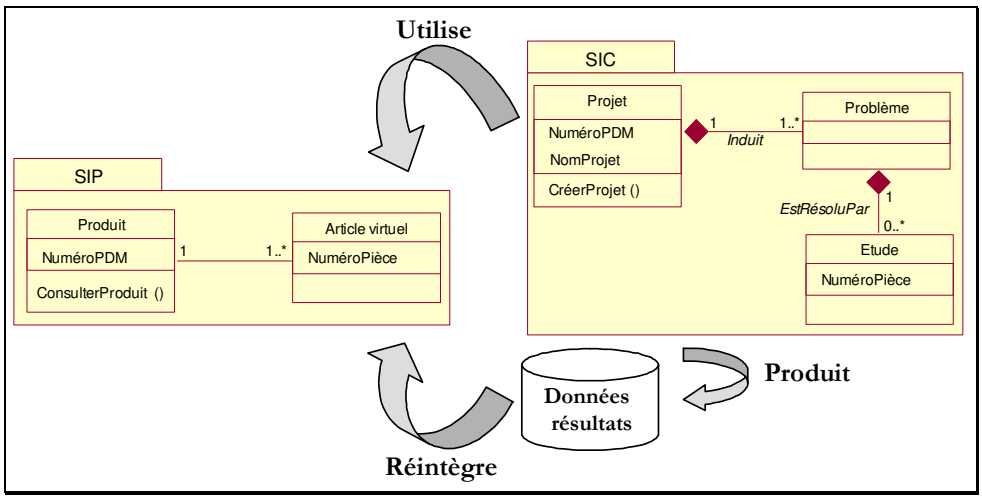


Figure 73 : Flux d'information échangé entre SIC et SIP

2. **Partage de données et de processus :** Dans les diagrammes de classes présentés dans la figure 71 et la figure 72, nous remarquons aussi l'existence des attributs communs aux

SIC et SIP. Le premier est l'attribut "NuméroPDM" dans la classe « Projet » puisqu'un projet dans le SIC projet garde le même numéro qui lui a été attribué dans le SIP. Le second est l'attribut "NuméroPièce" dans la classe « Etude » qui correspond à la notion d'article dans le SIP. Les deux SI échange aussi des services. Dans la suite, nous choisissons comme scénario de coopération entre SIC et SIP, la gestion d'un nouveau projet dans le SIC (cas d'utilisation "Gérer_Projet") (cf. Figure 74).

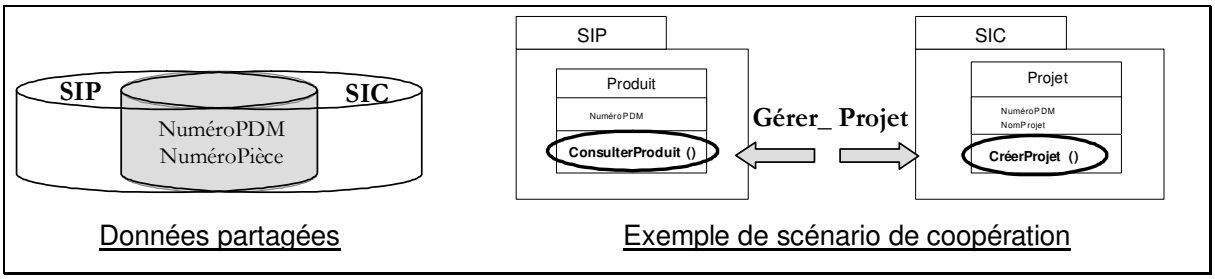


Figure 74 : Echange de données et de processus entre SIC et SIP

Celui-ci fait appel au cas d'utilisation "ConsulterProduit" du SIP et "CréerProjet" du SIC. En effet, la création d'un nouveau projet dans le SIC nécessite la vérification de l'existence du "NuméroPDM" dans le SIP ("ConsulterProduit"). Si le "NuméroPDM" existe, une requête de création d'un nouveau projet est envoyée au SIC ("Créer_Projet") ; sinon un message d'erreur est retourné à l'utilisateur. Le diagramme de séquence de la figure 75 décrit le scénario du cas d'utilisation "Gérer_Projet".

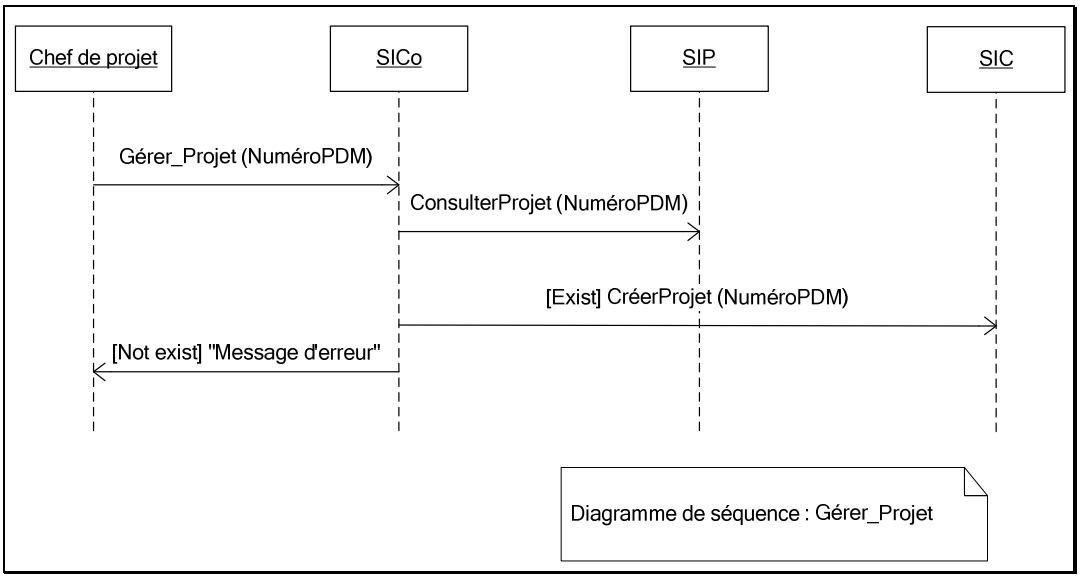


Figure 75 : Diagramme de séquence "Gérer_Projet"

3.4. Application du Système PACO

Afin de choisir le patron produit à appliquer dans le cadre de la coopération du SIC et du SIP, nous commençons, tout d'abord, par étudier le mode de coopération des deux SI :

- **Boîte blanche :** le concepteur connaît les flux de données gérés et échangés entre le SIC et le SIP. Leur structure est ainsi accessible par le concepteur et peut être modifiée et adaptée pour les besoins de la coopération.
- **Partage de données et absence d'un schéma global :** les deux SI partage des données puisque les activités de calcul utilisent des données du modèle de produit géré dans le SIP et réciproquement.
- **Explicite et synchrone :** la communication entre les deux SI est explicite et synchrone. Comme nous venons de voir dans la section précédente, la création d'un nouveau projet dans le SIC dépend de l'existence du "NuméroPDM" dans le SIP.
- **Point à point et bidirectionnelle :** la communication entre les deux SI est bidirectionnelle. D'un côté, les données du modèle du produit du SIP sont utilisées par les calculs et les simulations pour construire le modèle mécanique et / ou de simulation. De l'autre côté, Les données du modèle mécanique ou de simulation sont à réintégrer dans le modèle produit du SIP afin de capitaliser les calculs et les simulations effectuées.

Pour naviguer dans les patrons processus du système PACO et choisir le patron produit qui répond à ces critères de coopération, nous utilisons la cartographie orientée systèmes de patrons processus (images réactives) réalisée grâce à la fonction d'extraction de l'atelier AGAP (cf figure 76).

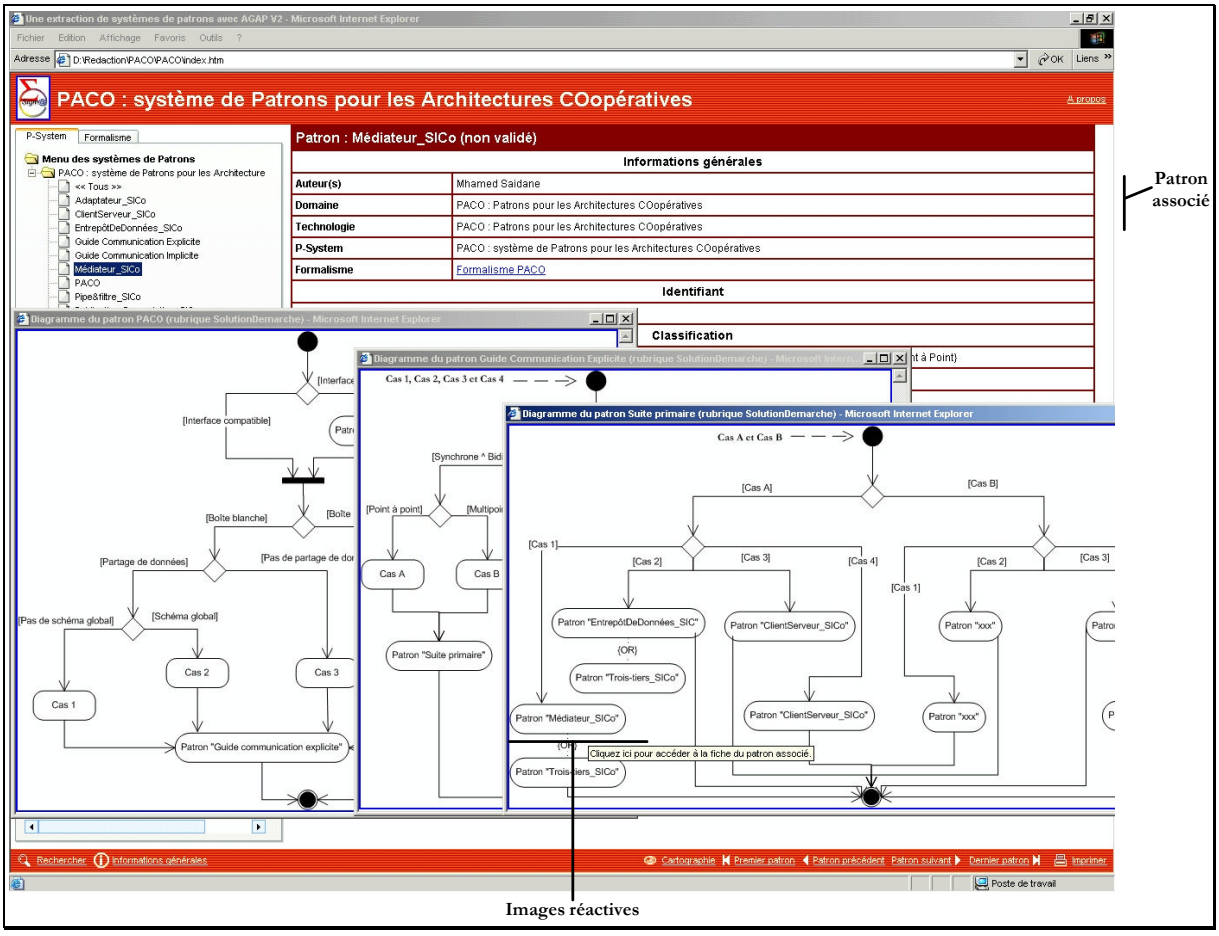


Figure 76 : Copie d'écran de la cartographie

Le mode de coopération sélectionné nous conduit vers deux patrons produit : le patron « Médiateur_SICo » et le patron « Trois-tiers_SICo ». Dans le cadre de ce projet, la mise en place d'une architecture client / serveur à trois niveaux serait très coûteuse et ne correspond pas aux attentes de nos partenaires industriels. Le choix du patron « Médiateur_SICo » est plus adéquat pour la modélisation de la coopération entre le SIC et le SIP.

Dans la figure suivante (cf. figure 77), nous présentons la première étape pour la réutilisation du patron « Médiateur_SICo » et qui consiste en l'instanciation de la solution semi-formelle du patron. Comme nous l'avons déjà évoqué, l'utilisation des diagrammes de composants d'UML 2.0 nous garantit uniquement une meilleure compréhension de l'architecture logicielle.

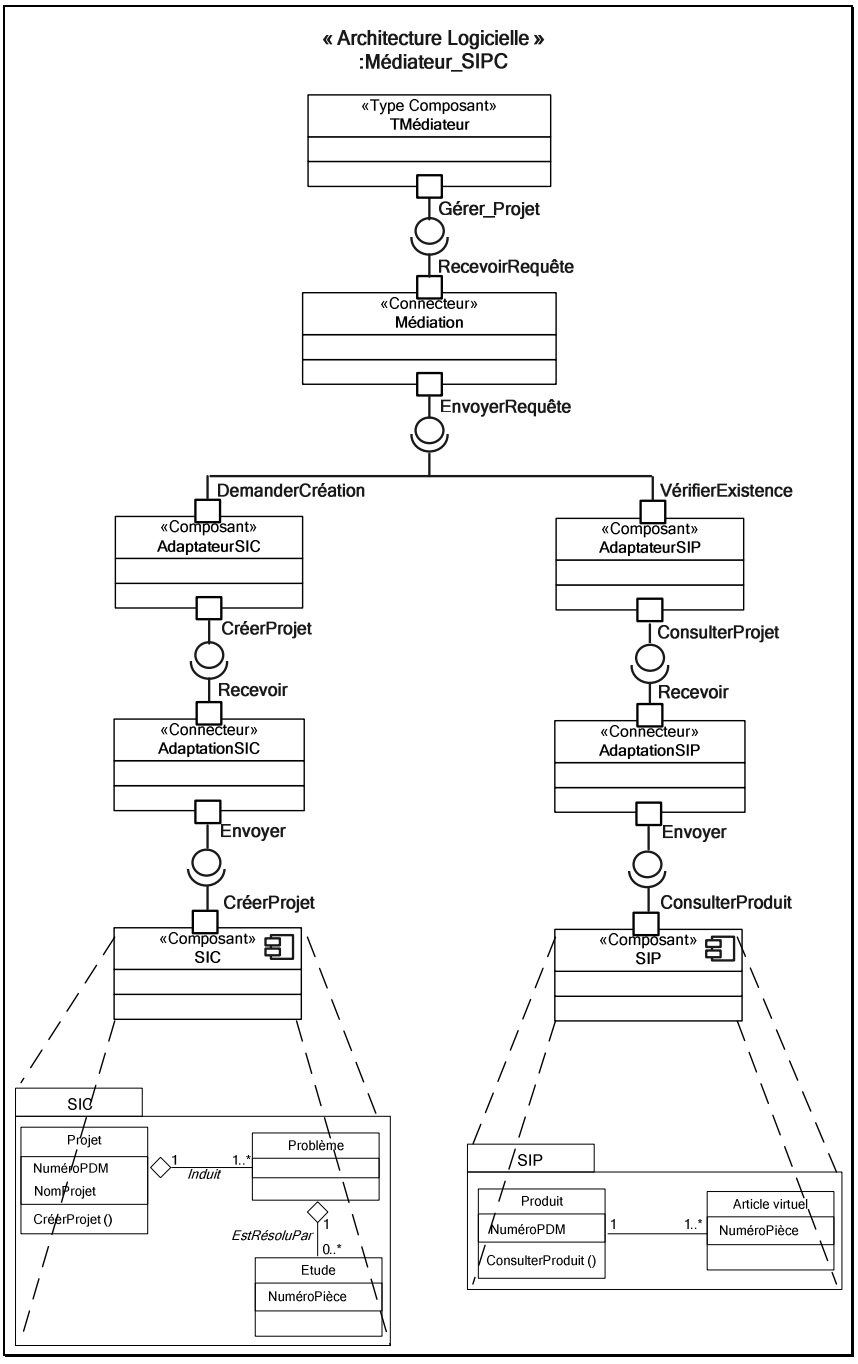


Figure 77 : Instanciation du diagramme de composants du « Médiateur_SICo »

La réutilisation effective de la famille d'architectures « Médiateur_SICo » est réalisée "naturellement" par l'outil AcmeStudio. La copie d'écran de la figure 78 illustre l'imitation de la spécification de la solution formelle du patron « Médiateur_SICo ».

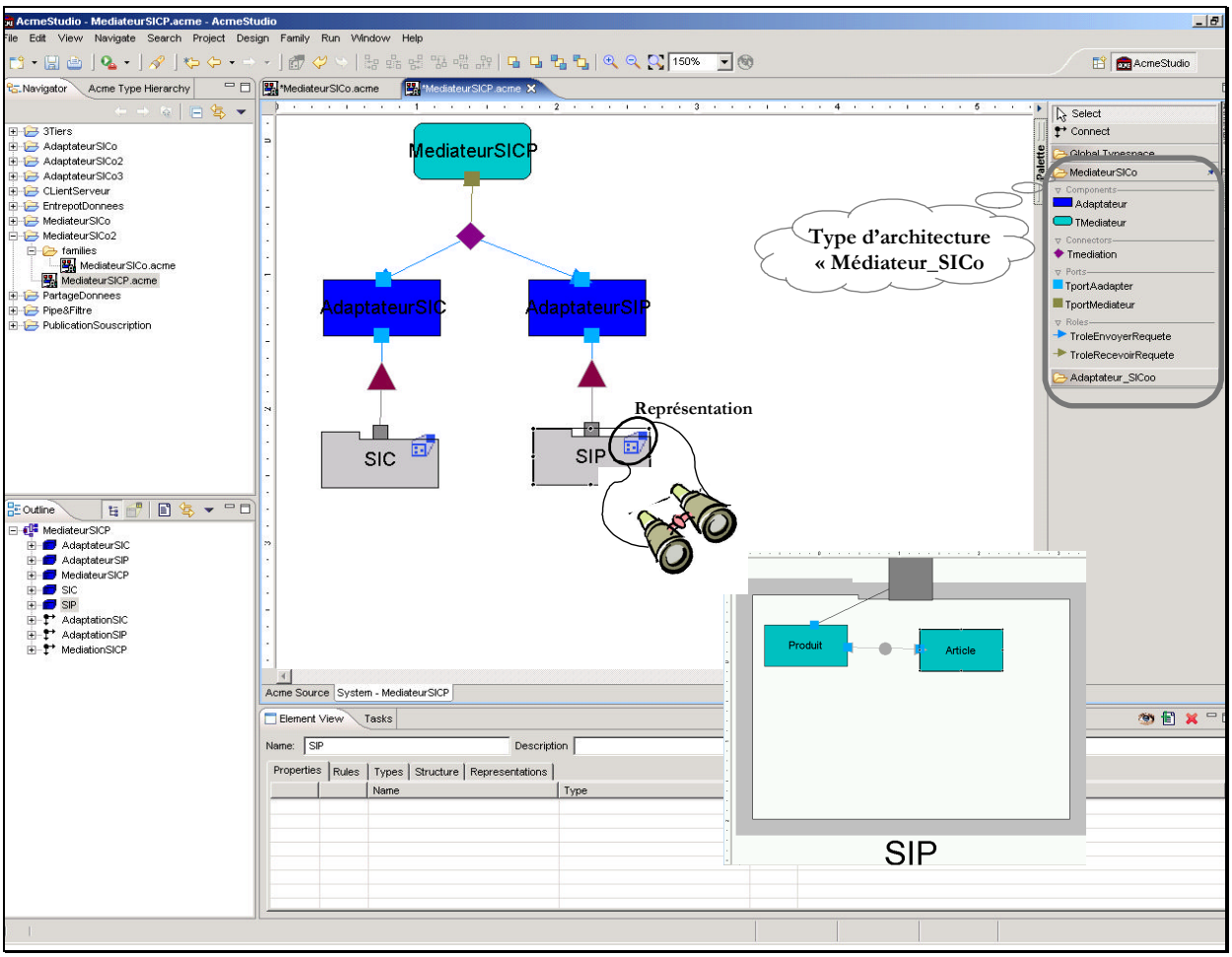


Figure 78 : Copie d'écran de l'outil AcmeStudio

Le code source suivant décrit la spécification formelle de l'architecture logicielle du « Médiateur_SICP ».

```
import $AS_PROJECT_PATH/families/MediateurSICo.acme;
System MediateurSICP : MediateurSICo = new MediateurSICo extended with {

  Component AdaptateurSIC : Adaptateur = new Adaptateur extended with {
    Port CreerProjet : TportAdaptee = new TportAdaptee;
    Port DemandeCreation : TportAadapter = new TportAadapter;
  };

  Component MediateurSICP : TMediateur = new TMediateur extended with {
    Port GererProjet : TportMediateur = new TportMediateur;
  };

  Connector Mediation : Tmediation = new Tmediation extended with {
    Role TroleEnvoyerRequete0 : TroleEnvoyerRequete = new TroleEnvoyerRequete;
  };

  Attachment MediateurSICP.gérerProjet to Mediation.RecevoirRequete;
```

```
Attachment AdaptateurSIC.DemandeCreation to Mediation.EnvoyerRequete;
Component SIC : TsysComp =  new TsysComp extended with {
   Representation RepresentationSIC = {
      System RpresentationSIC = {
         Component Projet = {
            Port p0;
            Port CreerProjet;
         };
         Component Probleme = {
            Port p0;
            Port p1;
         };
         Connector conn0 = {
            Role r1;
            Role r2;
         };
         Attachment Projet.p0 to conn0.r2;
         Attachment Probleme.p0 to conn0.r1;
         Component Etude = {
            Port p0;
         };
         Connector conn1 = {
            Role r1;
            Role r2;
         };
         Attachment Probleme.p1 to conn1.r2;
         Attachment Etude.p0 to conn1.r1;
      };
      Bindings {
         Projet.CreerProjet to SIC.CreerProjet;
      }
   };
   Port CreerProjet : TportSysComp =  new TportSysComp;
};

Connector AdaptationSIC : Tconnecteur =  new Tconnecteur;

Attachment AdaptateurSIC.CreerProjet to AdaptationSIC.Recevoir;
Attachment SIC.CreerProjet to AdaptationSIC.Envoyer;

Component AdaptateurSIP : Adaptateur =  new Adaptateur extended with {
   Port ConsulterProduit : TportAdaptee =  new TportAdaptee;
   Port VerifierExistence : TportAadapter =  new TportAadapter;
};

Component SIP : TsysComp =  new TsysComp extended with {
   Representation RepresentationSIP = {
      System RepresentationSIP = {
         Component Produit = {
            Port p0;
            Port ConsulterProduit;
         };
         Component Article = {
            Port p0;
         };
         Connector conn0 = {
            Role r1;
            Role r2;
         };
         Attachment Article.p0 to conn0.r1;
         Attachment Produit.p0 to conn0.r2;
```

```
        };
        Bindings {
          Produit.ConsulterProduit to SIP.ConsulterProduit;
        }
      };
      Port ConsulterProduit : TportSysComp =  new TportSysComp;
  };

  Connector AdaptationSIP : Tconnecteur =  new Tconnecteur;

  Attachment SIP.ConsulterProduit to AdaptationSIP.Envoyer;
  Attachment AdaptateurSIP.VerifierExistence to Mediation.TroleEnvoyerRequete0;
  Attachment AdaptateurSIP.ConsulterProduit to AdaptationSIP.Recevoir;
};
  Attachment AdaptateurSIP.RequeteAadapter to MediationSICP.TroleEnvoyerRequete0;
  Attachment AdaptateurSIP.RequeteAdaptee to AdaptationSIP.Recevoir;
};
```

4. Conclusion

Ce chapitre a illustré, d'une part, les outils pour supporter de la solution proposée pour l'ingénierie des Systèmes d'Information Coopératifs (SICo), et d'autre part, la validation de la démarche sur un cas concret. Ce dernier point a été mené en appliquant le système PACO dans le cadre du projet régional OSCAR. Certes, ce projet industriel ne suffit pas pour valider l'ensemble de l'approche proposée. Néanmoins, la modélisation de la coopération entre les deux systèmes d'information industriels (le Système d'Information de Calcul et le Système d'Information Produit) nous a permis de confronter cette approche à un cas réel. Ceci avait pour objectif de vérifier les points suivants :

1. La modélisation de la coopération par réutilisation de patrons est facilement applicable ;
2. Les critères de coopération proposés sont adéquats à des cas de coopération concrets ;
3. La navigation dans les patrons processus par la cartographie générée avec l'atelier AGAP présente une aide quant au choix du patron produit à appliquer ;
4. Les patrons produit proposés répondent à des problèmes récurrents rencontrés dans des projets industriels ;
5. La solution modèle des patrons produit est facilement réutilisable sous AcmeStudio.

Toutefois, le système de patrons proposé reste à enrichir par d'autres patrons spécifiant d'autres des familles d'architectures dédiées à la coopération. Les critères de coopération énumérés sont aussi à raffiner pour que chaque mode de coopération conduit à un seul patron produit.

Quant à l'aspect outillage, nous avons extrait le système PACO sous forme d'un site web autonome facilitant la navigation entre les différents patrons. Les spécifications formelles des différents patrons produit ont été intégrées dans AcmeStudio en tant que familles d'architectures prédéfinies. Nous joignons à ce manuscrit un cédérom contenant le site web du système PACO ainsi que l'outil AcmeStudio avec les familles d'architectures coopératives prêtes pour la réutilisation.

CHAPITRE VII

CONCLUSION & PERSPECTIVES

1. Bilan

L'objectif de cette thèse est de proposer une méthode d'ingénierie pour les Systèmes d'Information Coopératifs (SICo).

L'hypothèse principale sur laquelle nous nous sommes basés au début de ce travail est le manque de formalismes consensuels pour la modélisation des SICo. Etant un domaine relativement récent, les approches actuelles n'offrent pas de solutions suffisamment intégrées pour la modélisation de la coopération entre Systèmes d'Information (SI). La complexité du domaine présente aussi un frein à la réutilisation des modèles conceptuels qui peuvent décrire de tels systèmes. Ceci est d'autant plus vrai que ces modèles sont souvent dispersés et mal documentés. La tâche du concepteur des SICo devient ainsi difficile en l'absence de guide méthodologique spécifique au domaine.

La définition d'architectures logicielles pour les SICo paraît une solution prometteuse pour maîtriser leur complexité. Une description architecturale permet de raisonner sur l'architecture globale du système à un niveau d'abstraction élevé en faisant abstraction des détails techniques. Dans le cadre des SICo, elle permet aussi d'exprimer l'ensemble des systèmes composants ainsi que leurs interactions.

Partant de ce constat, les objectifs fixés s'articulent au tour de quatre points principaux :

❶ Proposer des modèles de spécification réutilisables de SICo à un niveau d'abstraction élevé pour gérer la complexité des architectures logicielles coopératives,

❷ Distinguer un ou des langages de modélisation permettant de fournir un vocabulaire sémantiquement riche pour exprimer explicitement les interactions entre les systèmes composants de la coopération,

❸ Fournir une démarche pour orienter le concepteur dans son choix vers l'architecture la plus adéquate à son problème,

❹ Utiliser la technique des patrons pour capitaliser l'ensemble des modèles et la démarche puis documenter la méthode proposée, promouvant ainsi, une ingénierie de SICo par réutilisation.

L'objectif de l'état de l'art que nous avons mené en première partie de cette thèse, était d'étudier les différentes techniques de coopération ainsi que des architectures logicielles pouvant être capitalisées dans le cadre des SICo. Ceci nous a permis, dans une étape suivante, d'étudier les langages de modélisation qui peuvent offrir une description riche et cohérente pour de telles architectures.

L'investigation dans la littérature des techniques de coopération et des langages de modélisation susceptibles de répondre aux objectifs fixés au départ, nous a conduit à fixer les éléments de base qui présentent le fondement de la méthodologie proposée. Il s'agit de développer une méthode d'ingénierie pour les SICo basée sur les patrons.

Le système PACO (Patrons pour les Architectures COopératives) proposé est composé de patrons produit et de patrons processus. Les premiers formalisent des modèles d'architectures logicielles coopératives. Les seconds formalisent des fragments de démarche pour aider le concepteur de SICo dans son processus de modélisation. Cependant, un ensemble de choix a été fait pour organiser les différentes architectures et faciliter leur réutilisation.

Les patrons produit du système PACO capitalisent les familles d'architectures coopératives les plus connues et utilisées dans la littérature. Pour les spécifier, nous avons utilisé conjointement le langage orienté objet UML 2.0 et le langage de description d'architectures Acme. L'utilisation d'un langage semi-formel permet de décrire graphiquement la solution modèle. Les diagrammes de composants d'UML 2.0, offrent une description claire et compréhensible favorisant ainsi la communication entre les différents acteurs du développement. Le langage Acme offre une description textuelle formelle de la structure des familles. Il permet de profiter du cadre sémantique qu'offrent les ADL pour fournir des descriptions d'architectures cohérentes pouvant être réutilisées de manière sûre et rigoureuse.

Nous avons ensuite défini un ensemble de critères pour caractériser les systèmes composants et les connecteurs dans des architectures coopératives. L'ensemble des critères énumérés constitue le fruit de l'étude de l'existant menée dans la première partie du document. Nous avons ensuite combiné les différents critères en excluant les combinaisons illogiques entre certains critères de nature antagonique. Les patrons processus proposés dans le système PACO présentent dans des diagrammes d'activités les différents modes de coopération définis. Chaque patron processus présente un fragment de démarche et est basé sur un ensemble de critères de coopération. L'enchaînement des différents patrons processus conduit au mode de coopération souhaité et mène vers le patron produit qui répond à ce mode de coopération.

L'utilisation des techniques de patrons d'une manière générale et particulièrement dans le cadre de l'ingénierie de SICo, nécessite la mise en place de deux processus complémentaires :

- Un processus pour la réutilisation proposé à l'ingénieur de patrons. Ce processus consiste en l'identification et la spécification des familles d'architectures logicielles ainsi que leur organisation dans les patrons processus en se basant sur les différents modes de coopération.

- Un processus par la réutilisation dédié à l'ingénieur d'applications, le concepteur de SICo dans notre cas. Ce processus est basé sur la recherche, la sélection et l'imitation d'un patron pour spécifier des architectures logicielles coopératives. Le processus de recherche et de sélection est assisté par les différents patrons processus constituant la démarche. Le processus d'imitation et d'adaptation est supporté par l'outil du langage Acme, AcmeStudio qui permet la réutilisation des familles d'architectures déjà prédéfinies par l'ingénieur de patrons.

Enfin, pour valider le système PACO, nous nous sommes appuyés sur un cas réel issu d'un projet industriel.

En résumé les principales contributions de notre travail sont :

❶ Des patrons produit pour documenter et capitaliser des familles d'architectures coopératives. Ils regroupent des architectures de coopération très utilisées et peuvent représenter une sorte de référentiel du domaine.

❷ Un méta-modèle pour définir les concepts gérés lors de la description structurelle des architectures coopératives. Celui-ci limite le vocabulaire d'UML et d'Acme et identifie les éléments de base permettant la spécifcation de telles architectures.

❸ Une contribution à une meilleure description des architectures coopératives par la combinaison d'un langage semi-formel et d'un langage formel pour spécifier graphiquement et textuellement la solution modèle des patrons produit.

❹ Des critères de coopération pour classifier les architectures coopératives en se basant sur les différents modes de coopération entre les systèmes d'information.

❺ Un guide méthodologique basé sur des patrons processus définissant une démarche propre à l'ingénierie des SICo.

2. Perspectives

Le travail de thèse a permis la mise en place d'un cadre conceptuel pour l'ingénierie des Systèmes d'Information Coopératives (SICo). Il s'agit d'un système de patrons, formulant des modèles et une démarche pour la spécification de la coopération entre Systèmes d'Information. Cette thèse peut se prolonger vers plusieurs perspectives de recherche à moyen et long terme qui peuvent faire l'objet de quatre points.

① Compléter les spécifications des solutions produits

Ce premier point concerne le raffinement des différents patrons produit proposés. En effet, le travail mené concerne le choix des langages, de l'organisation du système de patrons et des outils à utiliser. Il présente une première version destinée à être améliorée. Les améliorations souhaitées concerne la description statique et dynamique des solutions formelles. La solution modèle décrite en Acme présente un squelette de spécification qui peut être enrichi, notamment en collaborant avec des spécialistes côté métier, comme par exemple des spécialistes des Bases de Données Fédérées ou des Systèmes Multi-Agent. Il est alors intéressant de rendre ces spécifications plus complètes pour tenir compte des différentes propriétés des familles d'architectures proposées ainsi que la sémantique qui leur est associée. Ce premier point concerne la description statique des familles d'architectures. Le deuxième

point est relatif à leur description dynamique. La mise en œuvre de cette perspective débutera par une étude plus approfondie des moyens offerts par le langage Acme pour la description du comportement dynamique des architectures logicielles.

La dernière étape du travail visée concerne l'enrichissement du système de patrons par l'ajout d'autres patrons produit couvrant les différents modes de coopérations définis. En effet, au stade actuel, les patrons produit ne résolvent pas l'ensemble des modes de coopération énumérés.

② Réduire la multiplicité entre mode de coopération et patron produit

Actuellement, un mode de coopération peut mener vers plusieurs solutions modèles et inversement, une solution modèle peut répondre à plusieurs modes de coopération. Nous sommes donc face à deux situations. En premier lieu, un patron produit peut répondre à plusieurs modes de coopération. C'est le cas par exemple des familles d'architectures qui peuvent répondre à une communication synchrone ou asynchrone. Dans ce cas, c'est l'ingénieur d'applications qui décide de la nature de la communication au moment de l'instanciation. La deuxième situation est relative au mode de coopération qui peut mener vers plusieurs patrons produit. Pour l'instant, la rubrique « Problème » distingue les architectures voisines. Dans ce dernier cas de figure, le raffinement des critères de coopération paraît nécessaire. L'authentification d'autres critères peut en effet élargir la panoplie des modes de coopération proposés et différencier les familles d'architectures proches. Nous pouvons compléter la démarche proposée en introduisant, à titre d'exemple, un critère qui différencie une communication directe d'une communication indirecte ; ou bien ajouter une famille de critères qui caractérisent l'architecture globale, etc. Nous pouvons espérer, dans une première étape, d'arriver à une relation un à plusieurs (1-n) entre patron produit et mode de coopération (c'est-à-dire, un mode de coopération mène vers une seule famille d'architecture logicielle). Toutefois, nous estimons qu'il faut analyser davantage le cas inverse. En effet, une famille d'architecture pourra toujours répondre, par exemple, à une communication synchrone ou asynchrone.

③ Validation

Ce point concerne la validation du système PACO. Certes, le projet industriel OSCAR nous a apporté des éléments de réponse quant à l'application de la méthode sur des cas réels. Toutefois, cette expérimentation reste insuffisante pour valider l'ensemble des patrons proposés. La pratique de la méthode proposée sur d'autres projets concrets nous permettra d'avoir de plus amples retours d'expérience. De ce fait, il serait possible d'étudier de manière plus concrète les limites de la solution. Nous pourrions aussi déceler l'adéquation de la solution modèle offerte par rapport aux besoins exprimés par le concepteur. Cela permettrait d'approfondir l'examen des avantages et des inconvénients de l'application des patrons produit : par exemple, évaluer l'adéquation de la solution modèle dans une coopération à grande échelle.

④ Intégration avec d'autres travaux

Cette perspective concerne l'extension de ce travail par d'autres travaux en cours. Dans le cadre des travaux de l'équipe ABLE (Architecture Based Languages and Environments) à l'University Carnegie Mellon, des travaux sont en cours pour proposer un outil permettant de traduire les spécifications Acme en UML [GARLAN & AL., 02]. L'objectif est d'automatiser le passage des conceptions architecturales aux modèles orientés objet. L'intégration de cet outil

dans notre démarche assistera le passage des spécifications Acme aux diagrammes de composants UML. Ceci permettra d'encourager l'utilisation des spécifications formelles et de tirer profit de la richesse de leur vocabulaire. Notons qu'un premier prototype a déjà été réalisé dans le cadre du projet "OC Cyber Rift" [CHENG, 03]. Ce prototype permet de traduire des spécifications Acme en UML-RT (sous Rational Rose Real-Time). De même, dans le cadre des travaux de notre équipe [ARNAUD & AL., 04], une thèse en cours porte sur l'imitation des diagrammes UML. Le but visé est de proposer un outil permettant la réutilisation effective de la solution semi-formelle (diagramme de classes UML) des patrons produit sous l'atelier AGAP. Cette réutilisation (imitation) doit être complétée afin de ne pas perdre "l'essence" du patron. L'intégration de ces deux outils dans la méthode proposée donnera plus de flexibilité au concepteur SICo. La réutilisation de la solution modèle peut être effectuée à partir de la spécification Acme ou des diagrammes de composants UML. Ces derniers ne seront plus alors traités uniquement comme des "images" favorisant la compréhension, mais comme des spécifications réutilisables.

Bibliographie

A

[ALEXANDER, 79] Alexander C., *The Timeless Way of Building*, OXFORD UNIVERSITY PRESS, 1979.

[ALLEN & AL., 94] Allen R., Garlan D., *Formalizing Architectural Connection*, IEEE Conference on Software Engineering, California, 1994.

[ALLEN, 97] Allen R., *A formal Approach to Software Architecture*, Thèse de Doctorat, School of Computer Science Carnegie Mellon University, mai 1997.

[ALLEN & AL., 97] Allen R. and Garlan D., *A Formal Basis for Architectural Connection*, ACM Transactions on Software Engineering and Methodology, vol. 6, n°. 3, p. 213-249, juillet 1997.

[ALLEN & AL., 98] Allen R., Douence R., Garlan D., *Specifying and analyzing dynamic software architectures*, Conference on Fundamental Approaches to Software Engineering, Portugal, Mars 98.

[ANGLEROT & AL., 00] Anglerot S., Bonnet G., Regnault G., *Les Agents Intelligents Sur Internet*, Ecole polytechnique de l'université de Nantes, 2000.

[ARNAUD & AL., 04] Arnaud N., Front A., Rieu D., *une approche par méta-modélisation pour l'imitation des patrons*, Congrès Inforsid, Biarritz, mai 2004.

B

[BATINI & AL., 86] Batini C., Lenzerini M., Navathe S. B., *Conceptual Database Design*, Edited by the Benjamin, Cummings Publishing, 1992.

[BENSLIMANE, 99] Benslimane D., *Systèmes d'information coopératifs : une approche à base de médiation et d'agents,* Mémoire d'Habilitation à Diriger des Recherches, Université de Bourgogne, janvier 1999.

[BOND, 90] Bond H. S., *Distributed Decision Making in Organisation*, IEEE Transactions on Systems, Man and Cybernitics Conference, novembre 1990.

[BOOCH & AL., 99] Booch G., Rumbaugh J., Jacobson I., *Le guide de l'utilisateur UML,* Eyrolles, 1999.

[BOUGHZALA, 01] Boughzala I., *Démarche méthodologique de conception de systèmes d'information coopératifs interagents pour la gestion des connaissances*, Thèse de Doctorat, Spécialité Informatique, Université Paris VI, décembre 2001.

[BOULANGER & AL., 97] Boulanger D., Dubois G., *Objets et coopération de systèmes d'information,* dans Ingénierie objet : Concepts et techniques, Edition C. Oussallah, InterEdition, p. 339-376, 1997.

[BOURON, 92] Bouron T., *Structure de communication et d'organisation pour la coopération dans un univers multi-agents*, Thèse de Doctorat, Spécialité Informatique, Université Paris VI, novembre 1992

[BRENNER & AL., 98] Brenner W., Zarnekow R., Wittig H., *Intelligent Software Agents : Foundations and Applications*, Springer-Verlag, Berlin, 1998

[BRODIE & AL., 92] Brodie M., Ceri S., *On Intelligent and Cooperative Information Systems : a workshop summary*, Journal of Intelligent and Cooperative Information Systems, vol. 1, n°. 2, p. 249-289, 1995.

[BUSCHMANN & AL., 96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M., *Pattern-oriented Software Architecture – A System of Patterns*, John Wiley and Sons, 1996.

[BUSSE & AL., 99] Busse S., Kutsche R. D., Leser U., Weber H., *Federated Information Systems: Concepts, Terminology and Architectures,* Technical Report Nr. 99-9, Berlin University, 1999.

C

[CAUVET & AL., 01] Cauvet C., *Ingénierie des Systèmes d'Information*, Dans la série du traité IC2 – Information Commande Communication-, Editions Hermès, 2001.

[CHENG, 03] Site Internet de Cheng O., http://gs01.isri.cs.cmu.edu/~zensoul/projects/acme-uml/index.cgi dernière mise à jour 2003, consultation octobre 2004.

[COAD, 92] Coad P., *Object-Oriented Patterns*, Communication of ACM, vol. 35, n°. 9, septembre 1992.

[COAD, 95] Coad P., *Object Models : Strategies, Patterns and Applications,* Yourdon Press Computing Series, 1995.

[CONTE, 97] Conte A., *Développement de système d'information à l'aide de patrons – Applications aux bases de données actives*, Thèse de Doctorat, spécialité Informatique, UJF - Grenoble 1, France, décembre 1997.

[CONTE & AL., 01] Conte A., Fredj M., Giraudin J. P., Rieu D., *P-Sigma : un formalisme pour une représentation unifiée de patrons*, Inforsid'01, Martigny, juin 2001.

[COPLIEN, 96] Coplien J. O., *Advanced C++ Programming Styles and idioms*, Addison-Wesley, 1992.

[COUTURIER, 05] Couturier V., *L'ingénierie des systèmes d'information coopératifs par réutilisation : une approche à base de patterns*, Thèse de Doctorat, spécialité Informatique, Université Jean Moulin, Lyon 3, décembre 2004.

D

[DADAM, 96] Dadam P., *Distributed Databases and Client/Server Systems*, Springer-Verlag, 1996

[DAVIS, 80] Davis R., *Report on the workshop on distributed A. I. independent assertions for integration of heterogeneous schemas*, VLDB Journal, vol. 1, n°. 1, p. 81-127, juillet 1992.

[DEREMER & AL., 76] DeRemer F., Kron H., *Programming-in-the-Large Versus Programming-in-the-small*, IEEE Transactions on Software Engineering, p. 321-327, juin 1976.

[DUPONT, 95] Dupont Y., *Problématique et résolution contextuelle des conflits de fragmentation dans l'intégration de schémas,* Revue ISI, vol. 3, n°. 1, p. 29-58, 1995.

F

[FEHLING & AL., 83] Fehling M., Erman L., *Report on the Third Annual Workshop on DAI*, SIGART Newsletter 84, p. 3-12, avril 1983.

[FERBER, 95] Ferber J., *Les systèmes multi-agents : Vers une intelligence collective*, Inter Editions, 1995.

[FERBER, 97] Ferber J., *Les systèmes multi-agents : un aperçu général*, Technique et Sciences Informatiques, vol.16, n°8, 1997.

[FOWLER, 97] Fowler M., *Analysis Patterns – Reusable Object Models,* Addison-Wesley, 1997.

G

[GARLAN & AL., 94] Garlan D., Allen R., Ockerbloom J., Exploiting Style in Architectural Design Environments, Software Engineering, The second ACM Sigsoft, p. 175-188, décembre 1994.

[GAMMA & AL., 95] Gamma E., Helm R., Johnson R. E., Vlissides J., *Design patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995.

[GARLAN & AL., 97A] Garlan D., Monroe, Wile D., *ACME : An Architecture Description Interchange Language*, Computer Science Departement, Carnegie Mellon University, novembre 1997.

[GARLAN & AL., 97B] Garlan D., Monroe, Wile D., ACME : *Architectural Description of Component-Based Systems*, Computer Science Departement, Carnegie Mellon University, 1997.

[GARLAN & AL., 02] Garlan D., Cheng S. W., Kompanek A. J., *Reconciling the Needs of Architectural Description with Object-Modeling Notations*, Science of Computer Programming, Special issue on Unified Modeling Language, vol. 44, P. 23-49, 2002.

[GZARA, 00] Gzara L., *Les patterns pour l'ingénierie des Systèmes d'Information Produit*, Thèse de Doctorat, spécialité Génie Industriel, INPG, décembre 2000.

H

[HAREL, 87] Harel D., Statecharts : *A Visual Formulation For Complex Systems*, Science of Computer Programming, vol. 8, n°.3, p.231-274, juin 1987.

[HEIMBIGNER & AL., 85] Heimbigner D., McLeod D., *A federated Architecture for Information Management,* ACM Transactions on Office Information Systems, vol. 3, n°. 3, p. 253-278, 1985.

[HEISEL & AL., 97] Heisel M., Lévy N., Using LOTOS Patterns to Characterize Architectural Styles, Theory and Practice of Software Development, Springer-Verlag, vol. 1214, p. 818-832, 1997.

[HERIN & AL., 01] Hérin D., Espinasse B., Andonoff E., Hannachi C., *Des systèmes d'information coopératifs aux agents informationnels*, ISI, Hermès, chapitre 8, 2001.

[HOARE, 87] Hoare C. A. R., *Communicating Sequential Process*, Communications of the ACM, vol. 21, p. 666-677, août 1978.

[HOFMEISTER AL., 99] Hofmeister C., Nord R. L., Soni D., *Describing Software Architecture with UM*L, TC2 1st Working Conference on Software Architecture, 1999.

I

[INVERADI & AL., 95] Inveradi P., Wolf. A. L., *Formal Specification and Analysis of Software Architectures Using the Chemical Machine Model*, IEEE transactions on Software Engineering, vol. 21, n°. 4, avril 1995.

[ISSARNY & AL., 98] Issarny V., Saidakis T., Zarras A., *A survey of Architecture Description Languages*, Technical Report, C3DS Project, janvier 1998.

J

[JAUSSERAN, 05] Jausseran E., *Démarche Symphony étendue, formalisation et expérimentation sur un Système d'Information Hospitalier*, Mémoire d'Ingénieur C.N.A.M, Spécialité Informatique, Grenoble, juillet 2005.

[JENNINGS & AL., 98] Jennings N. R., Sycara K., Wooldridge M., *A Roadmap of Agent Research and Development*, Autonomous Agents and Multi-Agent Systems, n°1, p. 7-38, 1998.

K

[KORICHE , 95] Koriche F., *Conception d'une méthode de modélisation des systèmes d'information coopératifs*, Actes du congrès INFORSID'95, p. 433-451, Grenoble, juin 1995.

[KRISHNAMURTHY & AL., 91] Krishnamurthy R.,Litwin W., Kent W., *Language Features for Interoperability of Databases with Schematic Discripancies*, ACM Sigmod, Colorado, p. 40-49, 1991.

L

[LESSER & AL., 87] Lesser V. R., Corkill D. D., *Distributed Problem Solving Networks*, John Wiley and Sons, New York, p. 245-255, 1987.

[LITWIN & AL., 82] Litwin W., Boudenant J., Esculier C., Ferrier A., Glorieux A. M., La Chimia J., Kabbaj K., Moulinoux C., Rolin P., Stangret C., *SIRIUS System for Distributed Data Management,* In Distributed Database, p. 311-366, 1982.

[LITWIN & AL., 86] Litwin W., Abdellatif A., *Multidatabase Interoperability,* IEEE Computer, décembre 1986.

[LIU & AL., 95] Liu L., Lu C., *The distributed interoperable object model and its application to large-scale interoperable database systems,* International Conference on Information and Knowledge Management, 1995.

[LUCKHAM & AL., 95] Luckham D. C., Kenney J. L., Augustin L. M., Vera J., Bryan D., Mann W., *Specification and Analysis of system Architecture Using Rapide*, IEEE Transactions on Software Engineering, vol. 21, n°. 4, p. 336-355, Avril 1995.

M

[MEDVIDOVIC & AL., 97] Medvidovic N., Taylor R., *A Framework for Classifying and Comparing Architecture Description Languages,* ACM Sigsoft Software Engineering, vol. 22, n°. 6, novembre 1997.

[MEDVIDOVIC & AL., 99] Medvidovic N., Rosenblum S., *Assessing the Suitability of a Standard Design Method for Modeling Software Architectures*, TC2 1st Working IFIP Conference on Software Architecture, 1999.

[MEDVIDOVIC & AL., 00] Medvidovic N., Taylor R. N., *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, vol. 26, n°. 1, p. 70-93, janvier 2000.

[MICHELIS & AL., 99] De Michelis G., Dubois E., Jarke M., Matthes F., Mylopoulos J., Papazoglou M., Pohl K., Schmidt J., Woo C., Yu E., *Cooperative Information Systems : A Manifesto*, Cooperative Information System : Trends and Directions, Academic Press, 1997

[MONROE & AL., 97] Monroe R. T., Kompanek D., Melton R., Garlan D., *Stylized Architecture, Design Patterns, and Objects,* IEEE Software, p. 43-52, janvier 1997.

[MONROE, 98] Monroe R. T., *Capturing Software Architecture Design Expertise With Armani,* Technical Report, Carnegie Mellon University School of Computer Science, octobre 1998.

[MULLER, 96] Müller J. P., *The design of intelligent agents : A layer approach*, Lecture Notes of Computer Science, Springer-Verlag, vol.1177, 1996.

N

[NG & AL., 96] Ng K., Kramer J., Magee J., *A CASE Tool for Software Architecture Design*, Journal of Automated Software Engineering, vol. 3, n°. 3/4, p. 261-284, août 1996.

[NODINE & AL., 00] Nodine M., Fowler J., Ksiezyk T., Perry B., Taylor M., Unruh A., *Active Information Gathering in InfoSleuth,* In International Journal of Cooperative Information Systems, p. 3-28, 2000.

O

[OMG, 91] Group (Object Management), *The Common Object Request Broker : Architecture and Specifications*, Rapport technique, 1991.

[OMG & AL., 03] OMG, *UML 2.0 superstructure final adopted specification*, Object Management Group, document ptc/03-08-02, août 2003.

[OREIZY & AL., 98] Oreizy P., Medvidovic N., Taylor N. R., *Architecture-based runtime software evolution*, International Conference on Software Engineering, Kyoto, Japan, avril 1998.

[OUSSALAH & AL., 97] Oussalah M., *Ingénierie Objet – Concepts et Techniques*, InterEditions, mai 1997.

[OUSSALAH & AL., 99] Oussalah M., *Génie Objet, Analyse & Conception de l'évolution d'Objets*, Editions Hermès, 1999.

[OUSSALAH & AL., 05] Oussalah M., *Ingénierie des Composants Logiciels : principes et fondements*, Editions Vuibert, juin 2005.

P

[PARENT & AL., 96] Parent C., Spaccapietra S., *Intégration de bases de données : panorama des problèmes et des approches,* Revue ISI, vol. 4, n°. 3, p. 333-358, 1996.

[PIECHOKI] Piechocki L., *UML en français*, http://uml.developpez.com/lp/index-cours.html

[PORTLAND, 94] About the Portland Form, http://c2.com/ppr/about/portland.html, 1994.

[PRATT, 86] Pratt V., *Modelling Concurrency With Partial Orders*, International Journal of Parallel Programming, vol. 15, n°. 1, p. 33-71, 1986.

R

[RIEU, 99] Rieu D., *Ingénierie des systèmes d'information*, Mémoire d'Habilitation à Diriger les Recherches, Institut National Polytechnique de Grenoble, 1999.

[DES RIVIÈRES & AL., 04] Des Rivières J., Wiegand J., *Eclipse : A platform for integrating development tools*, IBM Systems Journal, vol. 43, n°. 2, 2004.

[ROBBINS & AL., 98] Robbins J.E., Medvidovic N., Redmiles D. F., Rosenblum D. S., *Integrating Architecture Description Languages with Standard Design Method,* 20th International Conference On Software Engineering, p. 209-218, 1998.

[ROLLAND & AL., 88] Rolland C., Foucault O., Benci G., *Conception des Systèmes d'Information – la méthode REMORA*, Editions Eyrolles, 1988.

S

[SAIDANE, 01] Saidane M., *Vers un système de patron pour la modélisation des Systèmes d'Information Coopératives*, GDR-PRC I3, Lyon, 2001.

[SAIDANE & AL., 02] Saidane M., Giraudin J.P., *Des patrons pour l'ingénierie de la coopération des systèmes d'information*, Numéro spécial Connaissances métier dans l'ingénierie des Systèmes d'Information, vol. 7, n°. 4, Hermès, 2002.

[SAIDANE & AL., 03] Saidane M., Baizet Y., Blanco E., Pourroy, Rieu D., *Vers un méta-outil de capitalisation et d'organisation de simulations*, Inforsid'03, Nancy, juin 2003.

[SAIDANE & AL., 04] Saidane M., Rieu D., *Towards a patterns system for modeling cooperative information system*, ISCA, CSITeA'04, Le Caire, Egypte, décembre 2004.

[SCHEUERMANN & AL., 90] Scheuermann P., Elmagarmid A., Garcia-Molina H., Manola F., McLeod D., Rosenthal A., Templeton M., *Report on the workshop on heterogeneous database systems held at northweston university,* ACM Sigmod Record, vol. 19, p. 23-31, décembre 1990.

[SCHMIDT, 91] Schmidt K., *Cooperative Work : a Conceptual Framework*, Distributed Decision Making : Cognitive Models for Cooperative Work, John Willey and Sons, Chichester, p. 75-110, 1991.

[SCHMIDT, 99] Schmidt D. C., *Wrapper facade : a structural pattern for encapsulating functions within classes,* Report magazine, 1999.

[SCIORE & AL, 94] Sciore E., Siegel M., Rosenthal A., *Using semantic values to facilitate interoperability among heterogeneous information systems,* ACM Transactions on Database Systems, vol. 19, p. 254-290, juin 1994.

[SELIC & AL., 98] Selic B., Rumbaugh J., *Using UML for Modeling Complex Real-Time systems*, Rational Software Corporation, mars 1999.

[SHAW & AL., 96A] Shaw M., Garlan D., *Software Architecture : Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, 1996.

[SHAW & AL., 96B] Shaw M., DeLine R., Zelesnik G., *Abstractions and Implementations for Architectural Connections*, International Conference on Configurable Distributed Systems, mai 1996.

[SHETH & AL., 90] Sheth A., Larson J., *Federated Databases Systems for Managing Distributed Heterogeneous and Autonomous Databases*, ACM Computing Survey, vol. 22, n°. 3, septembre 1990.

[SHETH & AL., 92] Sheth A., Kashyap V., *So Far (Schematically) Yet So Near (Semantically)*, In proceedings of the IFIP DS-5, Conference on semantics of Interoperable Database Systems, Australia, novembre, 1992.

[SPACCAPIETRA & AL., 92] Spaccapietra S., Parent C., Dupont Y., *Model independent assertions for integration of heterogeneous schemas,* VLDB Journal, vol. 1, n°. 1, p. 81-127, juillet 1992.

[SPIVEY, 89] Spivey J., *The Z Notation : A Reference Manual*, Prentice Hall, 1989.

T

[TASTET, 04] Tastet L., *AGAP : un Atelier de Gestion et d'Applications de Patrons*, Mémoire d'Ingénieur C.N.A.M, Spécialité Informatique, Grenoble, mars 2004.

[TROUSSIER, 99] Troussier N., *Contribution à l'intégration du calcul mécanique dans la conception de produits techniques : proposition méthodologique pour l'utilisation et la réutilisation*, Thèse de Doctorat, spécialité mécanique, UJF Grenoble, octobre 1999.

V

[VESTAL, 94] Vestal S., *Mode changes in real-time architecture description language*, Proceedings of the second International Workshop on Configurable Distributed Systems, Mars 1994.

W

[WARMER & AL., 98] Warmer J., Klippe A., *The Object Constraint Language: Precise Modeling with UML*, Addision-Wesly, 1998.

[WOOLDRIDGE & AL., 95] Wooldridge M., Jennings N. R., *Intelligent Agents : Theory and Practice*, The Knowledge Engineering Review, vol.10, n°2, p. 115-152, 1995.

Annexe 1 : Le formalisme P-Sigma

Le formalisme P-Sigma a été défini dans le cadre des activités de l'équipe Sigma du laboratoire LSR. Ce travail a pour objectifs :

- d'uniformiser la représentation des patrons produit et processus ;
- d'améliorer la formalisation de l'interface de sélection des patrons ;
- de proposer une organisation de systèmes de patrons.

Le formalisme est constitué de trois parties : *Interface*, *Réalisation* et *Relations*. Chaque partie regroupe un certain nombre de rubriques. Une rubrique est composée d'un ou de plusieurs champs de nature différente (texte, diagramme UML). Une rubrique peut être représentée de différentes manières.

1. Rubrique «Interface »

L'interface d'un patron est composée de cinq rubriques servant à la sélection d'un patron.

Rubrique	**Définition et Champs**
Identifiant	Définit le couple problème / solution à partir duquel le patron pourra être référencé. Constitue la clé principale de communication entre le concepteur de patrons et les utilisateurs. Champ : ▪ Un champ textuel.
Classification	Définit la fonction du patron par un ensemble de mots-clés du domaine (termes du domaine). Donne intuitivement la classification du domaine. Champs : ▪ Un champ textuel ▪ Eventuellement un champ formel sous la forme d'une expression logique utilisant les mots clés du domaine.
Contexte	Décrit la pré-condition pour l'application du patron. Peut être obtenu en appliquant la solution modèle d'un ou de plusieurs patrons ; les noms des patrons correspondants constituent alors le champ formel. Champs :

	▪ Un champ textuel ▪ Eventuellement un champ formel : {patron}
Problème	Définit le problème résolu par le patron. Champ : ▪ Un champ textuel
Force	Définit les apports induits par l'application du patron. Champs : ▪ Un champ textuel ▪ Eventuellement champ formel : expression logique des critères de qualité associés à une technologie

2. Rubrique «Réalisation»

La partie réalisation d'un patron comprend quatre rubriques présentées dans le tableau suivant :

Rubrique	**Définition et Champs**
Solution **Démarche**	Indique la solution du problème en terme de processus à suivre. Un diagramme d'activités permet éventuellement de représenter la démarche sous la forme : [garde] patron Champs : ▪ Un champ textuel ▪ Eventuellement un champ formel de type diagramme d'activités.
Solution **Modèle**	Décrit la solution en terme des produits attendus après l'application du patron. Champs : ▪ Un champ textuel ▪ Un champ de type diagramme de classes ▪ Eventuellement un champ formel de type {diagramme de séquence}
Cas **d'application**	Décrit des exemples d'imitation de la Solution Modèle. Rubrique optionnelle mais fortement conseillée pour faciliter la compréhension de la solution du patron. ▪ Un champ textuel ▪ Un champ de type diagramme de classes ▪ Eventuellement un champ de type {diagramme de séquence}
Conséquence d'application	Présente les limites et les bénéfices de l'application de la solution. Peut inclure un nouvel ensemble de problèmes faisant apparaître la nécessité d'appliquer de nouveaux patrons.

	Champ : ▪ Un champ textuel

3. Rubrique «Relation»

La rubrique relation est composée de quatre rubriques correspondantes aux quatre types de relations possibles entre les pattrons.

Rubrique	Définition
Utilise	Si un patron P1 utilise un patron P2, alors : ▪ La solution démarche de P1 doit être exprimée en utilisant le patron P2. ▪ La classification de P2 peut être enrichie par rapport à celle de P1 : de nouveaux mot clés sont éventuellement ajoutés dans la classification de P2. ▪ Le contexte de P2 peut être enrichi par rapport au celui de P1.
Raffine	Si un patron P1 raffine un patron P2, alors : ▪ Le problème de P1 doit être une spécialisation de celui de P2. ▪ La classification de P1 peut être enrichie par rapport à celle de P2. ▪ La force de P1 peut être enrichie par rapport à celle de P2. ▪ Le contexte de P1 peut être enrichi par rapport à celui de P2.
Requiert	Si un patron P1 requiert un patron P2, alors : ▪ L'application de P2 doit être un pré-requis à l'application de P1. ▪ P2 doit apparaître dans le contexte de P1.
Alternative	Un patron P1 est une alternative d'un patron P2 si les deux patrons se différencient par leur force qui justifie deux solutions différentes au même problème : ▪ P1 et P2 ont la même classification, le même contexte et le même problème. ▪ Seule la rubrique « Force » des deux patrons est différente.