UNIVERSITE SAAD DAHLEB DE BLIDA

Faculté des Sciences
Département d'Informatique

MEMOIRE DE MAGISTER

Option : ingénierie des systèmes et de la connaissance

UN LANGAGE D'ACTION POUR LA SPECIFICATION ET LA VALIDATION DU COMPORTEMENT D'UNE ARCHITECTURE LOGICIELLE

Par

SAADI AbdelFetah

Devant le jury composé de :

Mme H.Abed	Maître de conférence, U. de Blida	Présidente.
Mr M.Benmouhamed	Professeur Université de constantine.	Examinateur.
Mr A.Guessoum	Professeur, U. de Blida	Examinateur.
Mr A.Henni	Maître de conférence INI, Alger.	Promoteur.
Mr D.Bennouar	Chargé de Cours, Université de Blida	Co-Promoteur.
Mme S.Oukid Khouas	Maître de conférence, U. de Blida	Invité.

Blida, 2008

بســـم الله الـــرحمن الـــرحيم

RESUME

L'architecture logicielle est une discipline récente qui s'intéresse aux structures d'un logiciel. Apparue sous ce nom au début des années 90, elle s'adressait plus particulièrement à la conception de logiciels de grande taille ou de familles de produits logiciels. Une architecture logicielle décrit l'ensemble des composants de l'architecture et donne la définition de leur assemblage. Aussi, elle prend en compte les structures d'accueil nécessaires pour le déploiement et l'exploitation du système résultant. Dans ce travail, nous présentons une approche pour la description, la spécification et la validation de comportement d'une architecture logiciel dynamique. Le travail rentre dans le cadre de la proposition de solutions aux problèmes qui se posent actuellement dans la spécification de la dynamique dans les langages de description d'architecture logicielle. L'objectif est de développer un langage de spécification d'architecture logicielle dynamique. Ce langage, appelé SEAL (Simple and Extensible Action Language), est de type langage d'action. Il est inspiré de *Precise Action Semantic* de UML. Il rentre dans le cadre des axes de recherche visant la spécification de modèles exécutables. La mise en œuvre du langage d'action est réalisée dans le contexte d'un environnement graphique qui permet la conception et la validation de l'évolution structurelle d'une architecture. Cette validation se fait par l'exécution d'architecture à un haut niveau d'abstraction.

Mots clés : Architecture logicielle, langage d'action, Description architecturale, Comportement d'architecture, Modèle de composant.

ABSTRACT

Software architecture is a recent discipline which is interested in software structures. Appeared under this name at the beginning of the Nineties, it was addressed, more particularly, to the design of software of big size or families of software products. Software architecture describes the whole of the architecture components and gives the definition of their assembly. Also, it takes into account the reception facilities necessary for the deployment and the exploitation of the resulting system. In this work, we present an approach for the description, the specification and the validation of behaviour of dynamic architecture software. The work enters within the framework of the proposition of solutions to problems currently posed in the dynamics specification in the description languages of software architecture. The aim is to develop a specification language of dynamic software architecture. This language, we called SEAL (Simple and Extensible Action Language), is an Action language. It is inspired from Precise Action Semantic of UML. It enters in the scope of research aiming the specification of executable models. The implementation of the Action language is carried out in the context of a graphical environment which allows the design and the validation of the structural evolution of an architecture. This validation is done by the execution of the architecture on a high-level of abstraction.

Key words: Software architecture, Action language, Architectural description, Architecture behaviour, Component Model.

ملخص

تعتبر هندسة التطبيقات من الانظمة الحديثة التي تهتم بالبنية الداخلية للبرنامج التطبيقي ,ظهرت تحت هذه التسمية في بدايات التسعينيات وتهتم خصوصا بتصميم وتطوير البرمجيات التطبيقية الكبرى والبرمجيات ذات الخصوصيات المشتركة ,تعتمد هندسة التطبيقات على الوصف الدقيق لكل العناصر التي تحتويها كما تعطي التعريف اللازم لكيفية ربط عناصرها الداخلية ,كما تاخذ بعين الاعتبار بنية هياكل الاستقبال الازمة لتشغيل النظام و استغلاله ,في هذا العمل قمنا بتقديم طريقة لوصف و تحديد والتاكد من صحة البرنامج الداخلي للهندسة التطبيقية المتحركة ,هذا العمل يدخل في اطار تقديم الحلول لبعض المشاكل المطروحة حاليافي تحديد الحركية داخل البرامج التي تعتمد على لغات الوصف للهندسة التطبيقية ,الهدف هو تطوير لغة برمجة لتحديد حركية الهندسة التطبيقية المتحركة, هذه اللغة التي قررنا تسميتها SEAL ,تندرج تحت نوع لغات الحركية, تم استنباطها من لغةUML ,استخدام هذه اللغة تم بعد تطوير محيط تصميم يسمح بتحديد حركية الهندسة التطبيقية المتحركة و يسمح كذلك بملاحظة كل المراحل التي تتغير فيهاالهندسة التطبيقية.

كلمات : هندسة التطبيقات, لغات الوصف للهندسة التطبيقية,لغات الحركية, الهندسة التطبيقية المتحركة.

REMERCIEMENTS

J’exprime mes plus grandes reconnaissances et mes vifs remerciements à mon promoteur Monsieur D. BENNOUAR pour le temps et l’attention qu’il a bien voulu consacrer pour bon déroulement de ce travail.

Je remercie très vivement toute l’équipe du département informatique, et tout le personnel, pour les moyens qu’ils ont mis à notre disposition , tous les conseils et les encouragements qu’ils nous ont prodigués tout au long de ce projet.

Mes gratitudes les plus sincères à tous ceux qui ont voulu juger ce travail.

J’exprime mes plus profondes reconnaissances à Madame ABED ainsi qu’aux enseignants du département d'informatique et les membres du laboratoire de recherche et de développement des systèmes informatisés (LRDSI) pour leurs conseils, leurs suggestions et pour l’aide qu’ils m'ont fournis.

Enfin, je remercie tous les enseignants qui ont contribué à ma formation et tous ceux qui ont participé de près ou de loin à la réalisation de ce travail.

Je dédie cet humble travail

à mes très chers parents

à mes frères et sœurs

à mes amis.

TABLE DES MATIERES

LISTE DES ILLUSTRATIONS, GRAPHIQUES ET TABLEAUX.

INTRODUCTION

La notion de composant logiciel occupe aujourd'hui une partie importante du paysage informatique, notamment dans les domaines du génie logiciel, des systèmes distribués ou encore celui des réseaux. Comme tout concept émergent, l'ambiguïté prévaut lorsqu'il s'agit d'en donner une définition claire. Nous pouvons cependant nous reposer sur une définition relativement consensuelle [1] : *Les composants logiciels sont des unités de composition de logiciels*. Ou bien : *Un composant est une entité logicielle, élaborée et vendue par une société et utilisée par un (ou des) client(s) pour composer un logiciel.*

Cette proposition est bien sûr peu satisfaisante mais elle est un point de convergence de nombreux travaux autour du concept de composant. Dès les années 80, le terme de composant sera souvent employé en informatique.

Le domaine des architectures logicielles propose de décrire la structure d'un logiciel comme un assemblage par interconnexion de composants. Une architecture logicielle modélise un système logiciel en termes de composants et d'interactions entre composants. Du point de vue industriel, les besoins en termes de composants logiciels s'expriment plutôt par la nécessité de disposer de technologies permettant de composer des applications à partir d'éléments logiciels réutilisables. Aujourd'hui on trouve de nombreuses solutions industrielles à base de composants tels que Microsoft COM/DCOM, les JavaBeans, les EJB et les composants CCM (Corba Component Model).

Actuellement, un grand intérêt est porté au domaine de l'architecture logicielle et l'approche de conception par assemblage de composant. Cet intérêt est motivé principalement par la réduction des coûts et des délais de développement des applications [4]. En effet, on prend moins de temps à acheter (et donc à réutiliser) un composant que de le concevoir, le coder, le tester, le déboguer et le documenter.

Les études menées dans le domaine du génie logiciel ont montré l'importance de la notion d'architecture logicielle [4]. Cette dernière permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage de composants logiciels. Les axes de travail autour de ce thème sont

multiples, l'idée fondamentale est de profiter des avantages d'une architecture clairement explicitée pour ne laisser en second plan que la programmation d'application.

Les langages de description d'architecture logicielle :

Les architectures logicielles ont pour origine, à la fois les difficultés rencontrées par les concepteurs de gros logiciels, les caractéristiques de la programmation à grande échelle et également les besoins de réutilisation de logiciel. 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 (Figure 1).

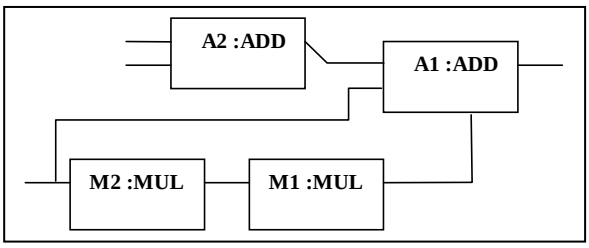

Figure 1: schéma d'une application sous forme diagramme *Box – and – line*

Ce manque de définition rigoureuse vient essentiellement du fait que peu de démarches de conception intègrent cette étape, soit parce que la structure du logiciel est simple, soit parce qu'au travers de l'analyse, les différentes structures du logiciel ont été appréhendées. Cependant, avec l'apparition de systèmes répartis, cette étape de définition d'architecture devient incontournable [1].

Par cette notion de plan de logiciel, la complexité est prise en compte, les possibilités de réutilisations sont augmentées, et il est possible d'utiliser des outils formels sur certaines parties précises de l'architecture et de générer des parties de code automatiquement [6].

Les langages de description d'architecture (ou ADL pour *Architecture Description Language)* répondent en partie à cette problématique en permettant la définition d'un vocabulaire précis et commun pour les acteurs devant travailler autour la spécification liée à l'architecture. Ils spécifient les composants de l'architecture de manière abstraite sans entrer dans des détails d'implantation, définissent de manière explicite les interactions

entre composants d'un système et fournissent un support de modélisation pour aider les concepteurs à structurer et composer les différents éléments [6].

Les langages de description d'architecture sont des langages dits déclaratifs. Ils peuvent être classés en deux grandes familles. *La première* correspond aux langages qui privilégient la description des éléments de l'architecture et leur assemblage structurel, *la seconde* définit les langages qui se centrent sur la description de la configuration d'une architecture et sur la dynamique du système. La première famille de langages correspond à une famille de langages qui est accompagnée d'outils de modélisation, d'analyseur syntaxique et de générateur de code. La deuxième famille de langages regroupe des langages accompagnés d'outils de modélisation et de génération de code mais aussi d'une plate – forme d'exécution ou de simulation d'un système, voire de modification dynamique pendant l'exécution. La particularité de ces langages est de définir un élément d'une architecture (composant ou connecteur) comme une instance. Il devient alors facile et simple de spécifier l'évolution dynamique d'une application au cours de son exécution. Les ADLs les plus connus, sont : UniCon[1], Aesop[4], Darwin[4], C2[1], Wright[6], Rapide[6], OLAN et ACME[1]. Chaque ADL présente des avantages et des inconvénients.

Medvidovic et Taylor [1] ont défini un gabarit de comparaison pour la plupart des langages de description d'architecture existants dans les milieux académique et industriel. Leur étude consiste, d'une part à définir les éléments communs d'un grand nombre d'ADLs (composant, connecteur, configuration) et, d'autre part, à proposer un ensemble de caractéristiques globales pour chaque élément fournissant ainsi un cadre commun favorisant la comparaison entre plusieurs ADLs. Ces caractéristiques sont également un moyen d'évaluer un langage de description d'architecture et sont considérées comme un ensemble de critères que doit prendre en compte un langage pour être considéré comme étant un langage de description d'architecture [1].

Les concepts de base de l'architecture logicielle :

On trouve un ensemble des concepts dans les différents langages de description d'architecture. Ces concepts sont au nombre de trois: *le composant*, *le connecteur*, et *la configuration* (ou *architecture*).

Le composant :

Un composant [1] est une unité de calcul ou de stockage à laquelle sont associées une ou plusieurs unités d'implantation localisées dans un ou plusieurs environnements d'exécution. Il peut ainsi être simple ou composé. Dans ce dernier cas on parle de composite. Sa taille peut être très petite telle qu'une simple fonction mathématique comme elle peut être très importante telle qu'une application complète.

On définit deux parties dans un composant. *Une première partie*, dite extérieure, correspond à son interface. Elle décrit les ressources fournies et les ressources requises par le composant (Figure 2). Les interfaces décrivent les interactions du composant avec son environnement. *La seconde partie* correspond à son implantation et permet la description du fonctionnement interne du composant (Figure 3).

Le type d'un composant est un concept représentant l'implantation des fonctionnalités fournies par le composant. Il s'apparente à la notion de classe que l'on trouve dans le modèle orienté objet. En fournissant un moyen de décrire, de manière explicite, les propriétés communes à un ensemble d'instances d'un même composant, la notion de type de composant introduit un classificateur qui favorise la compréhension d'une architecture et sa conception.

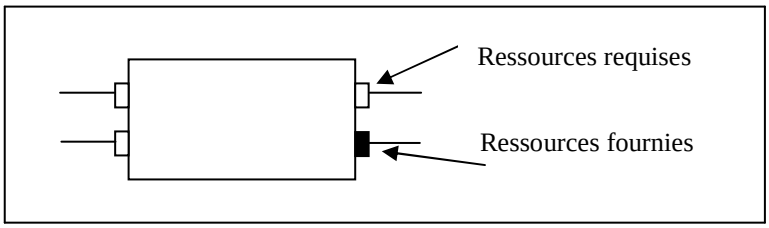

Figure 2 : Schéma d'un composant simple.

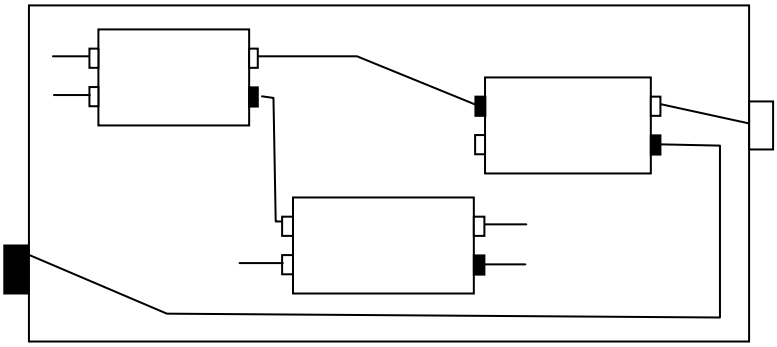

Figure 3 : Schéma d’un composant composite.

La sémantique du composant est exprimée en partie par son interface. Cependant, l’interface ne permet pas de préciser complètement le comportement du composant. La sémantique doit être enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liées à l’architecture.

La sémantique permet de décrire le comportement du composant. Le comportement peut être considéré sous deux formes : un comportement dit statique dans lequel nous nous intéressons aux états discrets du composant à des instants bien précis. Ce type de comportement n’exprime pas comment un tel état a été atteint. Le comportement dynamique quant à lui, explique comment un état est atteint à partir d’un autre.

Les contraintes définissent les limites d’utilisation d’un composant, Une contrainte est une propriété devant être obligatoirement vérifiée lors de l'instanciation d'un composant. Si elle n'est pas respectée dans une architecture, celle ci est alors considéré comme incohérente, donc inacceptable. [6]

Le connecteur :

Le connecteur [1] 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 (Figure 4). 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, c'est – à – dire son interface, qui permet la description des rôles des participants dans une interaction. *La seconde* partie correspond à la description de son implantation. Il s'agit là de la définition du protocole permettant la mise en œuvre de l'interaction. Le connecteur permet également de vérifier l'intégrité de la communication, c'est – à – dire de vérifier que les composants peuvent être connectés correctement.

Le type d'un connecteur correspond à sa définition abstraite qui reprend les mécanismes de communication entre composants ou les mécanismes de décision de coordination et de médiation. Il permet la description d'interactions simples ou complexes de manière générique et offre ainsi des possibilités de réutilisation de protocoles.

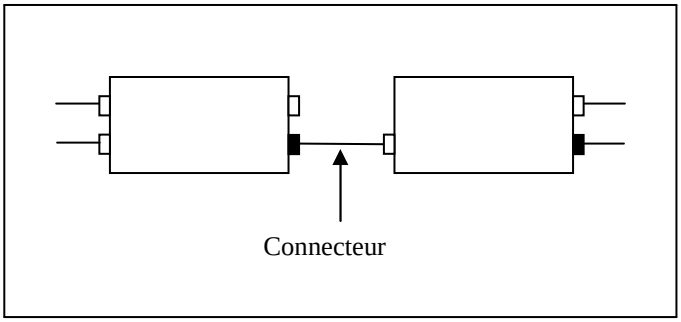

Figure 4 : schéma d'un connecteur.

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

Les contraintes permettent de définir les limites d'utilisation d'un connecteur, c'est – à – dire les limites d'utilisation du protocole de communication associé. Par exemple, le nombre maximum de composants interconnectés à travers le connecteur peut être fixé et correspond alors à une contrainte. [6]

La configuration :

Les configurations d'application, appelées aussi topologies [1] ou architecture, définissent les propriétés architecturales de connectivité et de conformité aux heuristiques de conception, ainsi que des propriétés de concurrence et de répartition dans certains ADLs. Une configuration définit la structure et le comportement d'une application formée de composants et de connecteurs (Figure 5). Une composition de composants, appelée dans certains contextes composites, est une configuration.

La configuration structurelle de l'application correspond à un graphe connexe des composants et des connecteurs formant l'application (figure 5). Elle détermine les composants et les connecteurs appropriés à l'application et vérifie la correspondance entre les interfaces des composants et des connecteurs.

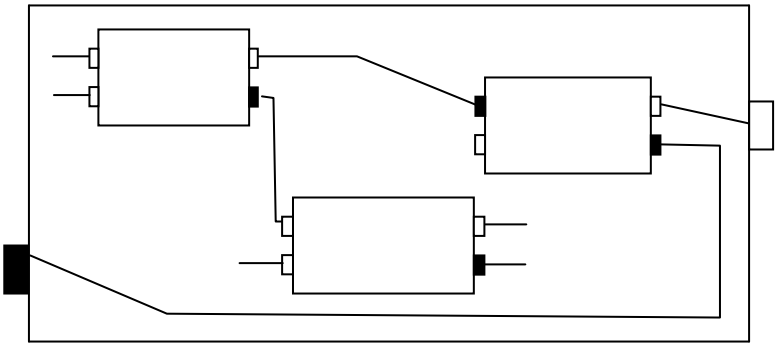

Figure 5 : Schéma d'une configuration.

La configuration comportementale modélise le comportement en décrivant l'évolution des liens entre composants et connecteurs. Elle définit également le schéma d'instanciation des composants au moment de l'initialisation de l'application, ainsi que le placement des composants sur les sites au moment du démarrage du système et leur évolution pendant la vie de l'application.

Une configuration doit permettre de fournir une syntaxe simple et une sémantique permettant de Faciliter la communication entre les différents partenaires d'un projet (concepteurs, développeurs, testeurs, architectes), Elle doit rendre la structure d'une application compréhensible à partir de la configuration sans entrer dans le détail de chaque composant et de chaque connecteur. Elle permet aussi la spécification de la dynamique

d’un système, c’est – à – dire l’évolution de celui – ci au cours de son exécution. La définition de la configuration d’une application doit permettre la modélisation et la représentation de la composition à différents niveaux de détail. La notion de configuration spécifie une application par composition hiérarchique. Ainsi un composant peut être composé de composants, chaque composant étant spécifié lui même de la même manière, jusqu’au composant dit primitif, c’est – à – dire non décomposable. L’intérêt de ce concept est qu’il permet la spécification de l’application par une approche descendante par raffinement, allant du niveau le plus général formé par les composants et les connecteurs principaux, définis eux mêmes par des groupes de composants et de connecteurs, jusqu’aux détails de chaque composant et de chaque connecteur primitifs. [1]

Problématique :

Nous pouvons distinguer deux formes d'architectures: les architectures statiques et les architectures dynamique. Dans une architecture statique, les composants et leurs topologies sont complètement définis à la spécification. La composition et la structure d'une architecture statique ne peut pas changer durant l'exécution. Les architectures statiques ne permettent pas de répondre à un nombre important de besoins, notamment des besoins d'adaptabilité, de résistance aux pannes, de mise à jour à chauds de logiciel, d'optimisation de ressources, d'évolution structurelle ou comportementale. La prise en charge de tels besoins ne peut être accomplie que dans le contexte des architectures dynamiques.

La spécification d'architectures dynamiques permet de planifier à la conception les changements topologiques qui auront lieu à l'exécution. Une architecture dynamique pourra ainsi :

- ✓ Gérer des défaillances d'un système et basculer sur une topologie plus sûre (topologie stable).
- ✓ Gérer un environnement sans cesse en évolution, comme c'est le cas pour les systèmes client – serveur dans lesquels les clients se connectent et se déconnectent continuellement.

L’évolution d’une architecture, n’a pas besoin d’être connue toujours avant la mise en exécution de l’application correspondante. En effet les mécanismes supportant l’évolution permettent de décomposer une architecture en exécution, de remplacer, de créer, de supprimer, d’activer ou désactiver des composants ou connecteurs, et de la recomposer. Ces mécanismes permettent de supporter les évolutions du système, sans arrêter l’exécution de l’application correspondante. [2][5]

Plusieurs problèmes se pose actuellement dans la spécification de la dynamique dans les langages de description d’architecture. Parmi ces problèmes nous pouvons citer ce qui suit :

- ✓ La spécification du comportement est faite par des langages formels. L'utilisation de langage formel dans les ADL, comme WRIGHT, fait face à deux problèmes cruciaux:
  - ❖ La manipulation de langage formel est délicate pour les praticiens.
  - ❖ Les langages formels sont surtout utilisés pour démontrer certaines caractéristiques de l'architecture tels que l'absence d'inter blocage. De plus, le passage à un niveau implémentation par raffinements successifs n'est pas supporté dans les ADL basé sur ce type de spécification [24]
- ✓ Dans une architecture logicielle dynamique, le dynamisme concerne la structure et le comportement. Le dynamise au niveau structurelle concerne principalement les aspects suivants: ajout / suppression / inhibition de composant ou connecteur. Le dynamisme comportemental est la possibilité pour un même composant de se comporter différemment selon l'environnement dans lequel il évolue ou dans le temps. Ces aspects dynamiques sont inexistants dans plusieurs ADL [6]. Dans d'autres ADL, les aspects dynamiques sont soit limités soit spécifique à des situations bien précises.

Notre objectif :

Notre objectif rentre dans le cadre de la proposition de solutions aux problèmes que nous venons de citer. Ces propositions sont faites dans le contexte de l'approche IASA[1] [14] pour l'architecture logicielle. Elles ont pour but de:

- ✓ Mettre au point un langage de spécification d'architecture logicielle dynamique. Ce langage, appelé SEAL (Simple and Extensible Action Language) est de type langage action. Il s'inspire de *Precise Action Semantic* de UML et rentre dans le contexte des actions de recherche dont le but est d'arriver à la spécification de modèles exécutables.
- ✓ Un environnement de conception et validation de l'évolution structurelle d'une architecture. Cette validation se fait par l'exécution d'architecture à un haut niveau d'abstraction.

La grande différence entre *Precise Action Semantic* et le langage SEAL réside dans les deux grands points suivants:

- ✓ SEAL est orienté composant alors que *Precise Action Semantic* est orienté objet. Les éléments fondamentaux que reconnait le langage SEAL sont les éléments du modèle de composant et connecteurs de l'approche IASA.
- ✓ *Precise Action Semantic* est juste une spécification pour laquelle il n'existe aucun langage officiel. SEAL est un langage réel pour lequel a été réalisé un interpréteur permettant d'exécuter des architectures et valider à un haut niveau d'abstraction l'évolution structurelle.

Le langage SEAL possède une syntaxe simple proche des habitudes des praticiens et est mis en œuvre dans le contexte d'un environnement de conception intégré (IDE) appelé IASASTUDIO . Dans cet environnement, l'architecte spécifie les actions en SEAL pour décrire :

- ✓ Les comportements attendus au niveau des ports d'interaction des composants.

- ✓ Les interactions entre ports de composant.
- ✓ Le comportement des composants.
- ✓ L'évolution structurelle d'une architecture.

L'interpréteur SEAL permet, par exécution d'architecture, la validation de l'évolution d'une architecture.

Le plan de la thèse :

Le travail que nous présentons dans cette thèse est organisé comme suit : Le premier chapitre sera consacré à la présentation d'un état de l'art sur la prise en charge des architectures dynamiques dans les différents ADL et dans UML 2.0. Pour chaque ADL, nous donnerons un bref aperçu et nous nous intéresserons aux capacités offertes pour le support de la spécification d'architecture dynamique. Ce chapitre sera conclu par un bilan général Les différents éléments ayant servi à la définition du langage d'action SEAL seront présentés au chapitre 2. Dans ce même chapitre nous étudierons les notions d'action et d'activité dans UML 2.0 ainsi que la spécification d'un système avec le langage SDL. Notre travail s'inscrit dans le contexte du modèle de composant IASA. Les concepts fondamentaux de ce dernier tels les points d'accès, les ports, les connecteurs et les composants. Seront ainsi introduit au chapitre 3. C'est au chapitre quatre que nous décrirons le Langage d'action SEAL qui permet la spécification et la validation du comportement d'une architecture logicielle. La mise en œuvre du langage d'action est réalisée dans le contexte de l'environnement IASASTUDIO. L'environnement de conception IASASTUDIO, qui a été développé sous forme d'un Plug In ECLIPSE[23], sera présenté au chapitre 5. L'évaluation du langage d'action SEAL ainsi que l'environnement de conception réalisé seront discutés au chapitre 6. Cette évaluation s'est faite dans le contexte de la conception et réalisation d'une application de gestion commerciale des produits d'un réseau X25.

---

[1] The Integrated Approcah to Software Architecture

CHAPITRE 1
LA DYNAMIQUE DANS LES ADL ET UML 2.0: UN ETAT DE L'ART

1.1. Introduction:

Les architectures dites dynamiques permettent de planifier à la conception leurs changements structurels et comportementaux durant l'exécution. Ces changements sont nécessaires pour que le système s'adapte à l'environnement sans cesse en évolution. Et pour qu'il puisse répondre aux besoins et exigences qui apparaissent durant la phase exécution (i.e. gérer des défaillances et de basculer sur une architecture plus sûre, gestion de la charge, mise a jour à chaud de composants, arrêt ou redémarrage de composant selon la nécessité)

La spécification d'architecture dynamique est basée sur un ensemble d'action permettant de modifier l'état structurel et comportemental d'une architecture à l'exécution. Nous avons définis dans notre approche plusieurs objectifs concernant d'une part la forme de la spécification et d'autre part les capacités que doit supporter un outil de spécification d'architectures dynamiques.

La spécification doit être à la portée de tout praticien. Elle doit être simple et adaptable aux habitudes des architectes dans leur travail de spécification d'architecture. Ces habitudes sont tirées des premières actions de spécification informelle d'architecture logicielle dans le style *box and line*. La spécification que nous visons doit formaliser ces habitudes sans la rendre complexe. Elle doit être à titre d'exemple capable de déterminer les sous entendus lors de la spécification d'une interaction. Elle doit en outre s'adapter au vocabulaire de l'architecte.

Concernant les capacités d'une spécification d'architecture dynamique, nous avons définis un certains nombres d'actions, que nous avons organisés en 4 catégories :

1. Actions d'évolution structurelle :

   - ✓ Insertion de composant, de connecteurs et de ports dans une architecture.

- ✓ Suppression de composant, de connecteur et de ports.
- ✓ Connexion et déconnexion de port.

2. Evolution comportementale :

   - ✓ Changement de comportement d'un composant, d'un connecteur ou d'un port d'interaction.

3. Contrôle sur les éléments de modélisation :

   - ✓ Autoriser, arrêter un composant, un connecteur ou un port à fonctionner.

4. Validation d'architecture dynamique : La modification de la structure ou du comportement d'une architecture est une action qui doit produire un système correct. La modification dynamique d'une architecture est un processus qui peut être réalisé en plusieurs actions. Durant ce processus le système peut passer par des états instables. Le système d'exécution doit prendre en considération ces états afin qu'ils n'aient pas un impact sur la bonne exécution du système. Le système d'exécution doit vérifier et valider l'architecture après tout changement.

Dans ce qui suit nous présentons un état de l'art des architectures dynamiques dans les différents ADL et dans UML 2.0. Nous avons considérés une quinzaine d'ADL, parmi lesquels nous trouvons : Darwin, Rapide, C2SADEL, ACME, Wright, ArchJava, Fractal, SOFA et UNICON.

Pour chaque ADL, nous donnerons un bref aperçu et nous nous intéresserons aux capacités offertes pour le support des objectifs que nous avons fixé et que nous venons de citer. Ce chapitre sera conclu par un bilan général sur la spécification d''architectures dynamique dans les ADL et dans UML 2.0.

1.2. La dynamique dans les ADLs :

1.2.1 Darwin :

1.2.1.1. Présentation générale :

Le langage Darwin est considéré comme un langage de description d'architecture, bien que celui – ci soit souvent appelé langage de configuration. Un langage de configuration favorise la description de la configuration d'une application, c'est – à – dire la description des interactions entre composants. La particularité de ces langages est qu'un composant est une entité instanciable. La description d'un composant au niveau du langage permet de créer de multiples instances d'un composant lors de l'exécution. Ainsi, ce type de langage se centre sur la description de la configuration et sur l'expression du comportement d'une application plutôt que sur la description structurelle de l'architecture d'un système. [1]

Le concept principal de Darwin est le composant. Un composant est décrit par une interface qui contient les services fournis et requis. Ces services s'apparentent plus aux entrées et sorties de flots de communication qu'à la notion de fonction. Deux types de composants existent. *Le premier* correspond aux composants primitifs qui intègrent du code logiciel, *le second* aux composites qui sont des entités de configuration décrivant les interconnexions entre composants.

Les services requis ou fournis (*require* et *provide*) correspondent à des types d'objets de communication que le composant utilise pour respectivement communiquer avec un autre composant ou recevoir une communication d'un autre composant. La sémantique associée à un composant est celle du processus. Ainsi, une instance de composant correspond à un processus créé, Darwin permet de fixer le nombre d'instances d'un composant lors de son initialisation. [1]

1.2.1.2. Spécification de la dynamique dans DARWIN:

La particularité de Darwin est donc de permettre la spécification d'une partie de la dynamique de l'application en terme de schéma de création de composants logiciels avant, après ou en cours d'exécution. Une caractéristique importante de Darwin est donc sa

capacité à décrire la création dynamique de composants par l'application. Cette caractéristique apporte une amélioration remarquable puisque l'architecture d'une application n'est plus considérée comme un ensemble de composants prévus dès la phase de conception. [2][5]

Le système n'est alors plus considéré comme étant fixe par rapport à sa phase de conception. Il fournit deux principaux mécanismes pour la description de structures dynamiques [5]:

- l'instanciation paresseuse
- l'instanciation dynamique.

Instanciation paresseuse :

L'instanciation paresseuse est une pré – déclaration des instances qui seront effectivement créées non pas lors de la phase d'initialisation, mais dès qu'un premier appel vers l'instance est effectué.

Elle permet à un composant fournissant un service d'être instancié au moment où un autre composant souhaite accéder à ce service. [5]

Exemple :

```
component ClientServer
  {
        inst client : Client ;
        inst serveur : dyn Serveur ;
        bind client.request – serveur.reply
   }
```

L'instance client, qui n'est pas dynamique, est créée normalement lors de l'instanciation du composant composite ClientServer. L'instance serveur ne sera pas créée durant cette instanciation mais seulement lorsque le client sollicitera le service qu'il fournit.

Pour cela le composant serveur est déclaré paresseux par le mot clé dyn. Ainsi, quand le serveur est appelé pour la première fois par le client client.request, l'instanciation est déclenchée. Son service peut être fourni au client par serveur.reply.

L'instanciation paresseuse permet donc, par une spécification de structure dynamique, de décrire la configuration potentielle à l'exécution en déclarant des instances qui seront peut être créées par un appel à leur service. [5]

Ce type d'instanciation permet de décrire des structures dont la taille ne peut être déterminée que dynamiquement. Mais ce mécanisme ne permet d'instancier qu'un seul composant par clause d'interconnexion. [5]

Instanciation dynamique :

L'instanciation dynamique permet de décrire la création d'une instance de composant en lui fournissant des paramètres d'initialisation. D'une manière générale, la création dynamique d'un composant est réalisée au sein d'un composant englobant (composant composite). L'accès à ce composant dynamiquement créé est géré par le composant englobant selon un système de nommage indirect. Toutes ces déclarations sont réalisées à l'aide du langage de configuration de Darwin. [5]

Darwin permet à un composant d'instancier un ou plusieurs composants sans demander de service particulier.

Exemple :

```
component ClientServer
  {     inst serveur : Serveur ;
        bind serveur.creation -- dyn Client
  }
```

Dans cet exemple, seule l'instance *serveur* qui n'est pas dynamique est créée normalement lors de l'instanciation du composant composite *ClientServer*. La déclaration dynamique est réalisée dans la clause d'interconnexion *serveur.creation--dyn Client*.

Pour cela l'opérateur *bind* relie un port de demande de création de composants *creation* en partie gauche avec une instanciation d'un composant *dyn Client*.
Cette interconnexion peut s'effectuer plusieurs fois, ainsi le composant serveur peut instancier plusieurs clients. [5]

Evaluation:

Darwin permet de créer des composants de manière dynamique grâce à la réplication de composants. Par contre il ne propose pas de moyen de les gérer une fois créée. En effet pour les composants d'un autre type, un composant dynamique n'est pas accessible, ce qui limite leur utilisation. De plus Darwin ne permet pas leur suppression.

1.2.2. Rapide :

1.2.2.1. Présentation générale :

Rapide [1] est un langage de description d'architecture dont le but est de vérifier, par simulation, la validité d'une architecture logicielle donnée. Avec le langage Rapide, une application est construite à partir de modules ou de composants communiquant par échange de messages ou d'événements. Rapide fournit également un environnement composé d'un simulateur permettant de vérifier la validité de l'architecture. Les concepts de base du langage Rapide sont les suivants : *la notion d'événement, de composant, et d'architecture.*

L'événement est une information transmise, qui est soit une demande de service, soit une valeur particulière d'un attribut. Le composant ou le module est défini par une interface. Cette dernière est constituée d'un ensemble de services fournis et d'un ensemble de services requis. Les services sont de trois types :

- Les services *Provides.*
- Les services *Requires.*
- Les *Actions* qui correspondent à des appels asynchrones entre composants, deux types d'actions existent : les actions *in* et *out* qui sont des événements acceptés et envoyés par un composant.

L'interface contient également une section de description du comportement (*behavior*) du composant. Cette dernière correspond au fonctionnement observable du composant comme, par exemple, l'ordonnancement des événements ou des appels aux services. Ainsi, l'environnement Rapide peut simuler le fonctionnement de l'application. [1]

De plus, Rapide permet également de spécifier des contraintes (*constraint*) 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. [1]

1.2.2.2. Spécification de la dynamique dans rapide :

Rapide est capable de modéliser l'architecture de systèmes dynamiques dans lesquels le nombre de composants peut varier quand le système est exécuté. Dans rapide toutes les instances sont représentées par des variables. Ainsi pour transcrire cette dynamicité, Rapide introduit deux types de variables particulières : les *placeholder* et les *iterator* auxquelles sont associées des règles de création. [5]

Le nom des variables *placeholder* débute toujours par « ? », elles désignent un objet qui est susceptible d'être présent. Le nom des variables *iterator* débute toujours par « ! », elles désignent une conjonction d'instances d'un certain type. Les règles de création définissent quand, pendant l'exécution, les composants d'un type doivent être créés ou détruits. [5]

Exemple :

```
architecture ClientServer
//Déclaration des instances des composants de l'application susceptible d'exister
?c : Client ; //Référence à une instance de Client
!s : Server ; //Référence à toutes les instances de Server
?service : Service ; //Référence à un bloc de paramètre d'un certain type Service
// Règle d'interconnexion
connect
?c.request(?service) => !s.reply(?service) ;
// Si un client transmet un paramètre de type Service  avec ce type de paramètre, alors
```

// L'événement est transmis à tous les serveurs de l'application avec ces paramètres.
end ClientServer
?c : *Client* est une variable *placeholder* et fait référence à une instance de type *Client.*
!s *: Server* est une variable *iterator* et désigne toutes les instances de *Server* présentes dans L'application.

Ces déclarations de variables définissent un objet devant être présent dans l'architecture ou un ensemble d'objets de tel ou tel type. Il n'est pas nécessaire de définir les instances de composants présents, car cela peut se faire dynamiquement au fur et à mesure de l'exécution.

La partie connexion contient les règles de connexion et les règles de création. Les règles de création définissent les conditions sur les événements qui guident la création de nouveaux composants. Dans l'exemple, la règle de création spécifie l'existence d'un client « *?c »* qui émet avec n'importe quelle donnée de type « *Service »*, alors que la partie droite de la règle est exécutée. Celle ci spécifie que tous les serveurs « *!s »* du système reçoivent la même donnée «*?service »*. Ces règles peuvent être conditionnées par une clause « *where »*. [5]

Evaluation :

Rapide permet de créer des composants dynamiques et possède des mécanismes permettant leur gestion. Par contre, aucun opérateur de création, suppression, migration de composants ou modification d'interconnexions n'est disponible. Rapide donc ne support pas tous les mécanisme offert par les architectures dynamiques.

1.2.3. Olan :

1.2.3.1. Présentation générale :

Olan est un langage de configuration, Son but principal est de fournir un environnement complet pour la construction, la configuration et le déploiement d'applications réparties. Il vise les à atteindre les fonctions suivantes [1]:

- Intégrer du logiciel existant dans une application en l'encapsulant dans un composant Olan,

- Prendre en charge, de façon quasi – automatique, des mécanismes de communication propres aux applications réparties grâce à des objets spécifiques appelés des connecteurs.
- Fournir un moyen de définir une architecture de manière abstraite grâce au langage OCL (Olan Configuration Langage) et en établissant une hiérarchie de composants connectés entre eux.
- Permettre la spécification et le déploiement d'un système réparti en décrivant et en automatisant le placement des composants logiciels sur des nœuds physiques et en gérant, pendant l'exécution, les canaux de communication entre ceux – ci, en fonction de propriétés d'interconnexion définies dans la description de l'architecture logicielle du système.

Le langage OCL fournit plusieurs concepts abstraits qui sont : le composant primitif, le composant composite, et le connecteur.

Le composant primitif est l'unité de base d'une application et l'entité d'intégration de logiciel existant. Il est défini par une interface et une implantation. [1]

1.2.3.2. Spécification de la dynamique dans OLAN :

Concernant l'aspect dynamique, Olan propose un nouveau mécanisme : *les collections*. Une collection de composants est un ensemble nommé d'instances de composants identiques.

A l'exécution, l'évolution d'un composant complexe est associée à la création ou à la suppression d'instances de composants dans une collection, ainsi que l'ajout ou le retrait des connexions entre ses instances. [5]

Une collection de composants *Client* est définie comme spécifié ci – dessous.

Collection = Collection [0..n] of Client ;

Le nombre d'instances de composants à l'exécution est borné. En effet il repose sur un intervalle défini par un nombre minimum et maximum. Cette cardinalité est passée en

paramètres au constructeur de la collection, au moment initial. Deux opérateurs *new* et *delete* permettent respectivement d'ajouter ou de supprimer des instances de composants d'une collection en cours d'exécution. Ils sont appliqués à un identificateur. [5]

- collection.new(idf) ;
- collection.delete(idf);

Les collections permettent de modéliser des composants dynamiques. Par contre, Olan ne présente pas de mécanisme concernant leur gestion.

Evaluation :

OLAN utilise le langage OCL qui permet d'exprimer la description de la dynamique d'un système de manière plus précise que certains ADLs grâce à la notion d'instanciation dynamique et paresseuse et à la notion de collection et de désignation associative. l'originalité de cet ADL est de proposer un moyen de décrire l'administration d'un composant en lui associant des attributs caractérisant, par leurs valeurs. [1]
La création et la suppression d'instances de composants se fait dans une collection.

1.2.4. C2SADEL :

1.2.4.1. Présentation générale :

C2SADEL (Software Architecture Description and Evolution Language) est un ADL dont l'objectif est de décrire des architectures de systèmes hautement distribués, évolutifs, et dynamiques. Les systèmes qu'il souhaite spécifier sont complexes, multi – langages, multi – plates – formes et ont une longue exécution. [5]

1.2.4.2. Spécification de la dynamique dans C2SADEL:

C2SADEL supporte les manipulations dynamiques. Pour cela le langage spécifie un ensemble d'opérations pour l'insertion, la suppression, et la ré – connexion des éléments de l'architecture pendant l'exécution du système : *addComponent*, *removeComponent*, *weld* et *unweld*.

C2SADEL exploite les connecteurs flexibles pouvant supporter le dynamisme. Prenons l'exemple du client – serveur (figure 1.1) auquel nous souhaitons ajouter un client de manière dynamique.

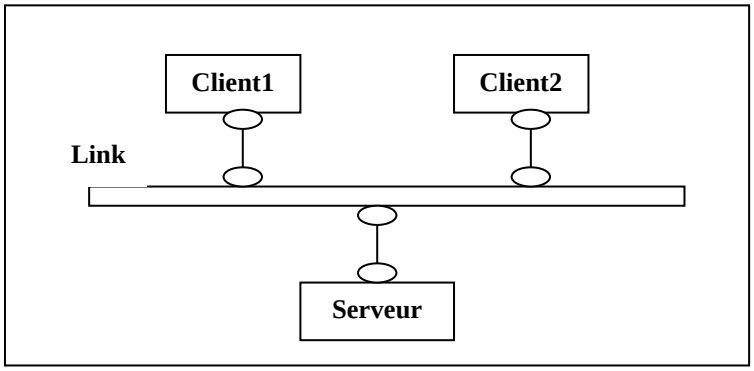

Figure 1.1 : Exemple de client – serveur.

Pour effectuer cet ajout, nous utilisons l'opérateur *addComponent* de la manière suivante :
ClientServer.AddComponent (Client2);
ClientServer.Weld (Link, Client2); // weld = soudure

Le composant *Client2* est ajouté à l'architecture *ClientServer* grâce à l'opérateur *addComponent*. Puis il est connecté au connecteur *Link* grâce à l'opérateur *weld*. Ces deux opérations sont effectuées après que le système ait été construit.
C2SADEL permet aussi de détacher un composant de l'architecture comme spécifié ci – dessous.

Link.SetTopPortFilter (Client1, message_sink);
Le composant *Client1* est momentanément détaché de l'architecture.
De même que C2SADEL permet l'ajout de composants, il propose leur suppression, comme spécifié ci – dessous.
ClientServer.Unweld (Link, Client1);
ClientServer.RemoveComponent (Client1);

Le composant *Client1* est d'abord définitivement détaché du connecteur *Link* grâce à l'opérateur *unweld*, puis il est supprimé de l'architecture grâce à l'opérateur *RemoveComponent.*

Evaluation :

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

1.2.5. Dynamic ACME :

1.2.5.1. Présentation générale :

Dynamic ACME est une extension permettant de décrire la dynamique des architectures. On y différencie deux classes d'éléments architecturaux : les éléments *fermés* (qui sont statiques) et les éléments *ouverts* (qui peuvent évoluer). Ainsi, un élément décrit en ACME est implicitement fermé. [2]

On explicite avec le modificateur *open* qu'un élément peut évoluer. Associé à l'élément, on peut de plus indiquer, via une cardinalité, le minimum et le maximum de ses occurrences. Les occurrences d'un élément *E* sont différenciées par des numéros : *E [1]* est la première occurrence *E* ; *E [*]* représente toutes les occurrences de *E*. Pour une meilleure lisibilité, l'opérateur *Let* permet de donner un nom à une occurrence : par exemple, *Let FirstE=E [1]*, nomme *FirstE* la première occurrence de *E*. [2]

1.2.5.2. Spécification de la dynamique dans Dynamic ACME:

Dynamic ACME permet de spécifier la sémantique de la dynamique. Cette sémantique est décrite à travers des contraintes exprimées par des prédicats. L'idée principale est que les propriétés et les relations entre les éléments ne sont valables pour un élément dynamique que si celui-ci est identifié. [2]

Par exemple, l'attachement dynamique de deux éléments qui s'exprime de la manière suivante:

Attachments {I.Out to F.In}
Ce traduit par: id I and id F => attached – to (I.Out, F.In).

Evaluation :

Dynamic ACME permet de décrire partiellement la dynamique des architectures, il permet de spécifier la sémantique de la dynamique. Cette sémantique est décrite à travers des contraintes exprimées par des prédicats.

1.2.6. Dynamic Wright :

1.2.6.1. Présentation générale :

Dynamic WRIGHT étend WRIGHT et supporte des manipulations pour les changements dynamiques de la topologie d'une architecture.

1.2.6.2. Spécification de la dynamique dans Dynamic Wright :

Récemment Dynamic Wright supporte des manipulations pour les changements dynamiques de la topologie d'une architecture. Pour cela, il définit un *configurateur* décrivant les configurations possibles de l'architecture, une initiale et d'autres sont prises durant l'exécution de l'application.

Dynamic Wright centralise la gestion et le contrôle de la dynamicité dans le configurateur. La solution adoptée met en œuvre des événements de contrôle qui sont émis par les composants dans une vue différente des événements de communication. Ces événements de contrôle sont reçus directement par le configurateur et permettent le déclenchement d'actions de reconfiguration comme : *new, del, attach* et *detach*. [5]

Dans l'exemple du client - serveur, Considérons que le système possède deux serveurs, un serveur primaire qu'il est préférable d'utiliser, et un secondaire qui sert de serveur de secours quand le premier tombe en panne.

Initialement, le connecteur relie le client au serveur primaire. Dès que le serveur primaire tombe en panne la connexion est interrompue. Elle est ensuite rétablie vers le serveur secondaire jusqu'à la restauration du serveur primaire. Pour simuler cette situation, des modifications sont apportées dans la description des composants et connecteurs, des événements de contrôle sont introduits dans leur alphabet. Ainsi le serveur primaire signale

sa panne à l'aide de l'événement de contrôle *control.down* et son rétablissement avec *control.up*. [5]

Dynamic Wright propose aussi des actions *new* et *delete* permettant la création et la suppression de composants et connecteurs de manière dynamique. Or Dynamic Wright est basé sur CSP qui ne peut décrire que des configurations statiques de processus. Pour permettre la création et la suppression de composants, Dynamic Wright restreint les systèmes à ceux qui ont un ensemble fini de configurations possibles. De plus il renomme les événements avec la configuration quand ils apparaissent. Chaque version d'un composant transformé commence par un événement [*Cp.go.p1.Cn1.r1…rn.Cnn.rn]* sélectionnant la version *Cp* où son port *p1* est attaché au rôle *r1* de *Cn1*, … et son port *pn* au rôle *rn* de *Cnn*. [5]
Le configurateur est transformé dans un processus CSP où les actions *new*, *del*, *attach* et *detach* sont compilées dans des événements *Cp.go*… qui sélectionnent la propre sélection du composant à chaque étape de reconfiguration.[5]

Dynamic Wright permet la description de connexions dynamiques et de composants dynamiques, même s'il s'agit de systèmes restreints dus au choix de l'algèbre de processus soutenant le modèle.

Très peu de documentation dans la littérature concerne les composants dynamiques, et Dynamic Wright ne présente pas de mécanisme concernant leur gestion. De plus Dynamic Wright, comme les autres langages, ne supporte que des changements dynamiques devant être connus avant la mise en exécution de l'application correspondant à l'architecture. [5]

Evaluation :

Dynamic Wright permet la description de connexions dynamiques et de composants dynamiques, elle ne supporte que des changements dynamiques devant être connus avant la mise en exécution de l'application correspondant à l'architecture. Donc Dynamic Wright atteint partiellement les objectifs des architectures dynamiques.

1.2.7. ArchJava :

1.2.7.1. Présentation générale :

Archjava est un langage de description d'architecture crée par l'université de Washington [36] Département of Computer Science and Enginering USA dont les concepteurs principaux sont JONATHAN et CRAIG CHAMBERS.

Archjava travaille sur une partie encore peu explorée des langages de description d'architecture (ADL). En effet beaucoup d'ADLs décrivent l'architecture logicielle d'une application dans leur propre langage, *au contraire Archjava étend l'implantation d'un langage,* en l'occurrence JAVA afin d'incorporer les éléments d'architectures à l'intérieur de code. [7]

Un important travail à été réalisé sur le compilateur Archjava afin de respecter l'intégrité des communications entre les composants. L'approche architecture logicielle au travers d'Archjava à pour objectif d'améliorer la compréhension des programmes, de garantir l'architecture de l'application, de permettre une meilleure évolutivité des applications et d'encourager les développeurs à se servir des avantages offerts par l'architecture logicielle. [8]

1.2.7.2. Spécification de la dynamique dans Archjava :

ArchJava offre la possibilité de travailler sur des architectures dynamiques. En effet, il est possible de créer des composants et des liaisons au sein d'un composite. Pour conserver la capacité d'analyse à la compilation des interactions entre composants, toute nouvelle connexion à l'exécution devra avoir été préalablement déclarée.

Quatre fonctions sont offertes par Archjava pour gérer le comportement dynamique d'une application :

- ✓ la création de composants au Runtime,
- ✓ la connexion de composants,
- ✓ la connexion de nouveaux composants sur un même composant.
- ✓ la destruction de composants.

La création dynamique de composant se fait grâce à la syntaxe NEW identique à celle de la programmation orientée objet. Des composants peuvent être connectés, par une instance de port, pendant l'exécution grâce à l'expression CONNECT. La destruction de composants n'est pas générée de façon explicite. Mais comme en java, les composants sont ramassés par le ramasse miettes « *garbage collector »* quant ils ne sont plus référencés ni connectés à d'autres composants. [7]
Dans de nombreuses applications, il peut être nécessaire de connecter plusieurs composants sur un même port sans savoir au départ combien. Archjava permet alors de définir une interface de port. Une nouvelle instance de port sera alors créée à chaque connexion.

Le mot clé *pattern* permet de définir au niveau d'un composite des types de connexions acceptables entre deux types de port.

Cependant, cette prise en charge de la dynamique reste néanmoins très limitée. En effet, il n'y a pas de possibilité de destruction d'instances de composant ni de destruction de liaisons. Il n'est pas possible de déplacer une instance au sein d'un autre composite.

Dans ce sens, les définitions d'architecture logicielle réalisées avec ArchJava sont relativement statiques. Ceci reste cependant cohérent avec la volonté de garantir à la compilation l'intégrité des communications entre composants. La deuxième lacune concerne l'impossibilité de sortir la spécification de cette dynamique à l'extérieur du code métier. En effet, le code lié à la gestion de la dynamique d'une architecture est mêlé au code lié au métier de chaque composant, ce qui complique le développement d'outils d'analyse autour de la dynamique.[7]

Evaluation :

ArchJava offre la possibilité de travailler sur des architectures dynamiques, il est possible de créer des composants et des liaisons au sein d'un composite. Mais toute nouvelle connexion à l'exécution devra avoir été préalablement déclarée, ArchJava atteint partiellement les objectifs des architectures dynamiques.

1.2.8. Fractal :

1.2.8.1. Présentation générale :

Fractal est un ADL qui est ouvert et extensible. C'est un modèle de composant flexible proposé par le consortium ObjectWeb5. Lancé en janvier 2002 par France Telecom R&D et l'INRIA, ce projet a pour but de fournir un modèle de composant assez souple pour pouvoir être utilisé dans n'importe quel langage et sur n'importe quel système, Il définit une syntaxe abstraite indépendante de tout langage de programmation pour la description de l'architecture en termes d'interface, de liaisons, d'attributs . . ., Le principal atout de cet ADL est son extensibilité, tout architecte peut définir un nouveau concept d'architecture, lui associer une syntaxe particulière et étendre l'ADL Fractal pour la prise en compte de ce concept. [10]
Le modèle de composants Fractal cherche à combler ce besoin. Fractal vise à autoriser une définition, une configuration et une reconfiguration dynamique d'une architecture à base de composants, ainsi qu'une séparation claire des préoccupations fonctionnelles et non fonctionnelles. Fractal est construit comme un modèle de haut niveau. [9]

A la base, un composant Fractal est formé d'une membrane et d'un contenu. Les composants peuvent être imbriqués, d'où les notions de sous – composants, de composants composites et de composants partagés. Un composant partagé est un composant appartenant à deux composant – composites qui ne sont pas imbriqués l'un dans l'autre. [9]

1.2.8.2. Spécification de la dynamique dans Fractal :

Dans Fractal, l'architecture logicielle correspond à un graphe d'instances de composant interconnectées. Au cours du cycle de vie cette architecture évolue suite à des opérations de création ou de destruction de liaisons ou d'instances. En Fractal, la description de l'architecture au travers de l'ADL est statique.

Le modèle de composant Fractal offre par l'intermédiaire de son API des mécanismes pour faire évoluer dynamiquement l'architecture d'une application. Il est ainsi possible de reconfigurer dynamiquement une architecture et créer à l'exécution de nouveaux composants. [6]

Fractal offre une API pour naviguer et introspecter la structure des composants : découvrir des interfaces (*component*), le contenu des composites (*content - controller* et *super – controller*) et naviguer le long des liaisons entre interfaces (*binding – controller*). En outre, il est possible de modifier dynamiquement la structure d'une application Fractal c'et à dire modifier le contenu des composites (*content – controller*), créer ou détruire des liaisons (*binding – controller*). [6]
Une deuxième caractéristique intéressante liée à la gestion de la dynamique concerne la création de nouvelles instances de composant. Fractal offre des modèles pour créer dynamiquement de nouveaux composants (l'instanciation simple à l'aide d'une *Factory* ou les *Templates).*

La création à l'aide d'une Factory crée pour un composant primitif la membrane du composant à partir de la définition des interfaces que l'on trouve dans la définition du type de composant. Il place la membrane autour de l'entité réalisant les opérations de l'interface. La création d'un composite à l'aide d'une Factory crée, là aussi, la membrane du composite à partir d'une définition des interfaces du type de ce composite comme un composant normal. Cependant, aucun contenu n'y est ajouté. C'est la principale différence avec la notion de Template qui instancie un composite et une configuration pour ce composite.

La notion de Template enrichit quelque peu la notion de type de composant que l'on trouve dans Fractal en offrant la possibilité de définir un type de composite par l'intermédiaire des interfaces qu'il fournit et qu'il requiert, mais aussi par l'intermédiaire de la configuration mise en place en son sein pour réaliser les opérations liées à ces interfaces. [6]

Tout comme Java, Fractal ne définit pas d'opération explicite de destruction de composants. La seule façon de retirer un composant d'une application est de supprimer toutes ses relations (liaisons, composition) de tous les composants de l'application et de l'arrêter. Rien dans Fractal ne garantit qu'il sera alors détruit, seulement qu'il n'aura plus d'impact sur l'application. [6]

Evaluation :

Fractal offre par l'intermédiaire de son API des mécanismes pour faire évoluer dynamiquement l'architecture d'une application. Il est ainsi possible de reconfigurer dynamiquement une architecture et créer à l'exécution de nouveaux composants. Fractal atteint partiellement les objectifs des architectures dynamiques.

1.2.9. SOFA :

1.2.9.1. Présentation générale :

SOFA (*SOFtware Appliance*) est un modèle de composant développé par l'université Charles (Prague). Son objectif est la construction d'application à partir d'une composition de composants.

SOFA propose les éléments caractéristiques des langages de description d'architecture, à savoir le composant, le connecteur, et la configuration.

Dans SOFA, les composants sont primitifs ou composites. Un composite est défini comme un assemblage de sous – composants alors qu'un primitif ne peut contenir de sous – composants. Un composite ne contient pas de fonctionnalités propres, toutes les fonctionnalités sont définies au sein des primitifs. Les composants de SOFA sont définis par leur *frame* et leur *architecture*.

La notion de *frame* est similaire à la notion de type de composant. C'est une approche boîte noire du composant. Une *frame* définit les interfaces requises et fournies par le composant.

Les connecteurs dans SOFA sont des éléments de premier ordre au même titre que les composants. Ils n'ont pas fondamentalement de différence avec les composants. Ils sont définis à l'aide de *connector – frame* pour la vision boîte noire et *connector – architecture* pour la vision boîte grise. L'objectif des connecteurs de SOFA est de permettre de capturer uniquement la logique métier au sein des composants pendant que les connecteurs prennent en charge la sémantique de l'interaction et les problèmes de déploiement.

SOFA fournit trois types de connecteur prédéfinis :

- CSProcCall avec une sémantique d'appel de procédure,
- EventDelivery avec une sémantique d'envoi de message
- DataStream pour une sémantique d'échange de flux.

Il fournit aussi un type de connecteur utilisateur qui permet de définir sa propre sémantique. Cette dernière sera alors réalisée à l'aide d'un assemblage de composants et de connecteurs. De manière implicite, si aucun connecteur n'est spécifié, un connecteur possédant une sémantique d'appel de procédure est introduit.
Il y a quatre moyens de venir s'accrocher à une interface de composant. La première appelée le *binding* permet de créer une liaison entre une interface requise et une interface fournie de deux composants de même niveau. La deuxième appelée *delegating* lie une interface fournie du composant englobant à une interface fournie d'un de ses sous – composants. La troisième nommée *subsuming* permet de lier un port requis d'un sous – composant à un port requis du composant englobant. Enfin, la dernière nommée *exempting* permet de déclarer qu'une interface d'un composant n'est pas liée. Les trois premiers types d'accroche sont réalisés par des connecteurs.

1.2.9.2. Spécification de la dynamique dans SOFA:

Comme pour Fractal, la description de l'architecture dans SOFA est statique. Une architecture au sens SOFA contient une liste des instances de composant ainsi que leurs liaisons. Si ce schéma sert au moment du déploiement pour l'instanciation et la configuration des composants, ce schéma peut rapidement devenir périmé suite à l'évolution interne de l'application. En outre, suite à une évolution liée à l'action de l'architecte, seul un mécanisme de gestion de version permet de garder une trace de l'évolution de l'architecture.

Une architecture au sens SOFA contient une liste des instances de composant ainsi que leurs liaisons. Si ce schéma sert au moment du déploiement pour l'instanciation et la configuration des composants, ce schéma peut rapidement devenir sans effet suite à l'évolution interne de l'application. En outre, suite à une évolution liée à l'action de

l'architecte, seul un mécanisme de gestion de version permet de garder une trace de l'évolution de l'architecture.

Evaluation :

SOFA utilise uniquement un mécanisme de gestion de version qui permet de garder une trace de l'évolution de l'architecture.

1.2.10. π - SPACE et σπ-SPACE :

1.2.10.1. Présentation générale :

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

Les architectures sont représentées par des configurations de *composites*, de *composants*, et de *connecteurs*. Un composite est un élément composé et définissant une configuration. [2]

Les composants et les connecteurs sont composés [2] :

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

1.2.10.2. Spécification de la dynamique dans π - SPACE et σπ-SPACE :

π-SPACE permet de formaliser la dynamique d'une architecture. Lors d'une description, on peut définir quels éléments sont dynamiques, leurs noms sont alors suffixés du caractère π. Dans l'exemple suivant (Figure 1.2), le client est déclaré comme dynamique. Cela signifie qu'il peut y avoir plusieurs occurrences créées dynamiquement. La clause *whenever* permet

de spécifier des réactions à des évènements. Ici il est spécifié que lorsqu'un nouveau client est créé, celui – ci est attaché au connecteur et que ce dernier évolue (un nouveau port est créé) pour permettre cela.

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

Figure 1.2 : Description d'un composant.

π-SPACE permet de définir des règles de dynamique, afin de préciser les changements topologiques dus à la création dynamique d'un nouvel élément. La dynamique peut être déclenchée par un composant. Pour cela, des ports spécifiques sont définis : *attachementEvolutionPort*, *ComponentEvolutionPort* et *EvolutionPort*. Ces ports permettent d'évoquer des reconfigurations, pour cela ils sont sollicités via le mot-clé *evolvable*. [2]

L'exemple ci – dessous (Figure 1.3) définit le comportement d'un composant. Ce comportement engage deux ports, un (*pC*) qui permet la communication avec un autre élément, et un qui permet de faire évoluer la topologie (*changementAttachement*). Le comportement commence par la réception d'un message via le port *pC*, puis continue par le déclenchement d'un événement pour une évolution via le port *changementAttachement*.

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

Figure 1.3 : Le comportement d'un composant.

Evaluation :

π-SPACE est fondé sur le π - calcul dont il retire des propriétés pour décrire les systèmes dynamiques, il permet de définir des règles de dynamique, afin de préciser les changements topologiques dus à la création dynamique d'un nouvel élément.

1.2.11. ARCHWARE ADL :

1.2.11.1. Présentation générale :

ArchWare ADL fournit la structure de base et les constructions comportementales pour décrire les architectures logicielles évolutives. C'est un langage formel de spécification conçu pour être exécutable et pour supporter l'analyse et le raffinement automatique des architectures. ArchWare ADL est un langage formel défini comme une extension spécifique à un domaine du π-calcul typé d'ordre supérieur.

ArchWare ADL est associé à un langage d'analyse, ArchWare AAL (Langage pour les Analyses des Architectures). Celui – ci est défini comme une extension spécifique à un domaine du μ - calcul. [2]

En ArchWare ADL, les éléments architecturaux sont définis en terme de comportements (*behaviour*). Un comportement est défini par un ensemble d'actions ordonnancées qui spécifie à la fois le traitement interne de l'élément architectural (actions internes) et les interactions avec son environnement (actions de communication). [2]

Un élément architectural communique avec les autres par une interface représentée par un ensemble de *connexions*. Les éléments architecturaux communiquent en transitant des données par ces connexions.

Pour interagir, les éléments architecturaux sont mis en relation par un mécanisme de composition et un mécanisme de liaison, appelé *unification* (c'est une substitution au sens du π-calcul).
Lorsque plusieurs éléments sont composés, ils peuvent interagir lorsque leurs connexions sont liées. [2]

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

1.2.11.2. Spécification de la dynamique dans ARCHWARE ADL :

ArchWare ADL fournit des mécanismes qui encouragent la définition d'architectures dynamiques. D'une part, l'abstraction est un mécanisme permettant la création dynamique. En effet, une abstraction est appliquée à la volée, ainsi plusieurs éléments peuvent être créés dynamiquement à partir d'une même définition. Cela permet l'introduction de nouveaux éléments architecturaux. D'autre part, ArchWare ADL hérite des propriétés du π-calcul d'ordre supérieur et pousse ainsi la mobilité. C'est à dire que des connexions ainsi que des comportements peuvent transiter par d'autres connexions. Cela permet le changement dynamique de la topologie de l'architecture. [2]

Evaluation :

ArchWare ADL fournit des mécanismes qui permettent la définition d'architectures dynamiques. Les connexions ainsi que des comportements peuvent transiter par d'autres connexions. Cela permet le changement dynamique de la topologie de l'architecture.

1.2.12. AML :

1.2.12.1. Présentation générale :

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

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

Dans AML, les architectures logicielles sont représentées comme un ensemble d'éléments (*element*) liés par des relations topologiques (*relationship*) soigneusement décrites et contraintes.
Le mot clé *element* permet de déclarer des instances. La structure est définie par des assertions via le mot clé *assume*.

1.2.12.2. Spécification de la dynamique dans AML :

AML ne permet pas de spécifier la dynamique d'une architecture. Mais il permet de définir des contraintes à travers des assertions temporelles utilisant les quantificateurs *sometimes* et *always* et en vérifiant l'existence d'un élément avec le quantificateur existentiel *id*. AML permet de spécifier comment l'architecture réagit lorsque sa structure évolue. [2]

Evaluation :

AML ne permet pas de spécifier la dynamique d'une architecture. AML permet de spécifier comment l'architecture réagit lorsque sa structure évolue, Il permet de définir des contraintes à travers des assertions temporelles utilisant les quantificateurs.

1.2.13. UNICON :
1.2.13.1. Présentation générale :

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

1.2.13.2. Spécification de la dynamique dans UNICON :

La spécification de la dynamique de l'application, c'est – à – dire le schéma de création ou de suppression des composants n'est pas possible. Il n'existe pas d'abstraction permettant de la décrire. Le cycle de vie d'une application ne peut donc pas être spécifié. [2]

Evaluation :

UNICON ne permet pas la spécification de la dynamique de l'application. Le schéma de création ou de suppression des composants n'est pas possible.

1.3. Bilan sur la Spécification de la dynamique dans les ADLs :

Le tableau 1.1 donne un bilan général sur l'étude de la spécification de la dynamique dans les langages d'architectures logicielle :

	La simplicité de la spécification	**La Validation de la dynamique**	**La création et la suppression des composants et connecteurs.**
Darwin	Difficile à assimiler	La description de la dynamique est limitée.	Permet de créer des composants de manière dynamique grâce à la réplication de composants. pas de moyen de les gérer une fois créée.
Rapide	N'est pas simple	Partiellement il permet d'exprimer la dynamique d'une application	Aucun opérateur de création, suppression, migration de composants ou modification d'interconnexions n'est disponible.
Olan	définir une architecture de manière abstraite grâce au langage OCL	utilise le langage OCL qui permet d'exprimer partiellement la description de la dynamique	La création et la suppression d'instances de composants se fait dans une collection.
C2SADEL	Simple.	supporte les manipulations dynamiques.	Le langage spécifie un ensemble d'opérations pour l'insertion, la suppression, addComponent, removeComponent, weld et unweld.
Dynamic ACME	spécification structurelle d'une architecture logicielle.	permettant de décrire partiellement la dynamique des architectures	Permet de spécifier la sémantique de la dynamique. Cette sémantique est décrite à travers des contraintes exprimées par des prédicats.
Dynamic Wright	Difficile à assimiler	ne supporte que des changements dynamiques devant être connus avant la mise en exécution de	Les actions new et delete permettant la création et la suppression de composants et connecteurs de manière dynamique.

		l'application.	
ArchJava	Basé sur la syntaxe JAVA.	offre la possibilité de travailler sur des architectures dynamiques	Il est possible de créer des composants et des liaisons au sein d'un composite. Mais toute nouvelle connexion à l'exécution devra avoir été préalablement déclarée.
Fractal	Flexible, ouvert et extensible, Il définit une syntaxe abstraite indépendante de tout langage de programmation pour la description de l'architecture.	offre par l'intermédiaire de son API des mécanismes pour faire évoluer dynamiquement l'architecture	Création, suppression à travers une API. Il est possible de reconfigurer dynamiquement une architecture et créer à l'exécution de nouveaux composants.
SOFA		utilise un mécanisme de gestion de version qui permet de garder une trace de l'évolution de l'architecture.	uniquement un mécanisme de gestion de version.
π - SPACE et σπ-SPACE	Difficile à assimiler	fondé sur le π - calcul dont il retire des propriétés pour décrire les systèmes dynamiques.	permet de définir des règles de dynamique, afin de préciser les changements topologiques dus à la création dynamique d'un nouvel élément.
ARCHWARE ADL	C'est un langage formel. délicate pour les praticiens.	fournit des mécanismes qui permettent la définition d'architectures dynamiques	Les connexions ainsi que des comportements peuvent transiter par d'autres connexions. Cela permet le changement dynamique de la topologie de l'architecture.
AML	un métalangage pour la spécification.	ne permet pas d'atteindre tous les objectifs des architectures dynamiques.	Il permet de définir des contraintes à travers des assertions temporelles utilisant les quantificateurs.
UNICON	simple à utiliser car il fournit un modèle de types de composant et de types de connecteur prédéfinis	ne permet pas la spécification de la dynamique de l'application	Le schéma de création ou de suppression des composants n'est pas possible.

Tableau 1.1 : Bilan sur la Spécification de la dynamique dans les ADLs.

1.4. Spécification de la dynamique dans UML 2.0 :

1.4.1. Introduction :

L'objectif de UML 2.0 est de rendre possible la traduction d'un modèle UML en programmes compilables, UML2.0 a introduit les concepts de l'architecture logicielle pour corriger les défaillances de UML1.5. UML 2.0 comporte treize types de diagrammes, UML 1.5 permettait déjà l'utilisation de neuf diagrammes, la version 2.0 en ajoute quatre, Ils se répartissent en deux grands groupes [30] :

1. **Diagrammes structurels ou diagrammes statiques (*UML Structure*) :**
   - ✓ Diagramme de classes (Class diagram)
   - ✓ Diagramme d'objets (Object diagram)
   - ✓ Diagramme de composants (Component diagram)
   - ✓ Diagramme de déploiement (Deployment diagram)
   - ✓ Diagramme de paquetages (Package diagram) → **(Nouveau diagramme de modèle UML2.0).**
   - ✓ Diagramme de structures composites (Composite structure diagram) → **(Nouveau diagramme de modèle UML2.0).**

2. **Diagrammes comportementaux ou diagrammes dynamiques (UML Behavior) :**
   - ✓ Diagramme de cas d'utilisation (Use case diagram)
   - ✓ Diagramme d'activités (Activity diagram)
   - ✓ Diagramme d'états-transitions (State machine diagram)
   - ✓ Diagrammes d'interaction (Interaction diagram) : ils sont représentés par les diagrammes suivants :
     - Diagramme de séquence (Sequence diagram)
     - Diagramme de communication (Communication diagram)
     - Diagramme global d'interaction (Interaction overview diagram) → **(Nouveau diagramme de modèle UML2.0).**
     - Diagramme de temps (Timing diagram) → **(Nouveau diagramme de modèle UML2.0).**

La figure 1.4 montre le schéma général de tous les diagrammes de modèle de UML2.0 :

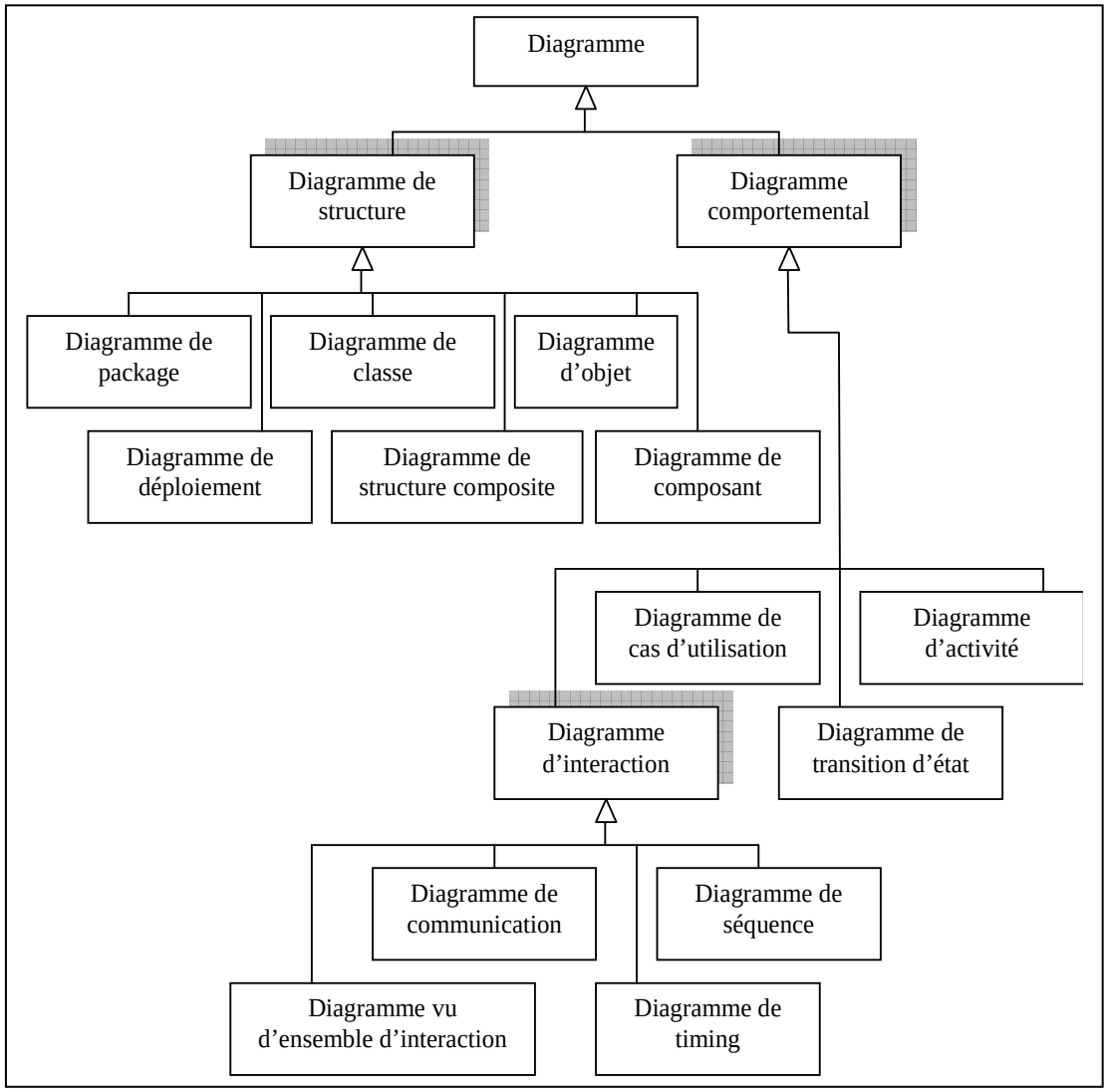

Figure 1.4 : Schéma général de tous les diagrammes de modèle de UML2.0.

Et voici la description des nouveaux diagrammes de modèle UML2.0 [30]:

- **Diagramme de paquetages** : permet de représenter la hiérarchie des modules du projet, leur organisation et leurs interdépendances. Cela simplifie les diagrammes, et les rend donc plus simple à comprendre.

- **Diagramme de structure composite:** permet de décrire la structure interne d'un objet complexe lors de son exécution (au Runtime, décrire l'exécution du programme), dont ses points d'interaction avec le reste du système.

- **Diagramme global d'interaction :** permet d'associer les notations du diagramme de séquence à celle du diagramme d'activité.

- **Diagramme de temps :** permet de modéliser les contraintes d'interaction entre plusieurs objets, comme le changement d'état en réponse à un évènement extérieur.

1.4.2. UML 2.0 et la notion de composant :

La version notée UML 2.0 introduit la notion de diagramme d'architecture, aussi appelée diagramme de structure composite. UML propose une notation unique pour couvrir toutes les phases du développement logiciel. Avec l'ajout de ce diagramme, UML veut combler une des insuffisances de la première version en permettant la spécification de l'architecture d'une application.

1.4.2.1. Présentation du modèle :

Inspiré des ADLs issus de la recherche académique, UML 2.0 propose les trois concepts fondamentaux des langages de description d'architecture à savoir :

- le composant,
- le connecteur,
- et la configuration.

Le composant UML 2.0 :

La notion de composant en UML 2.0 représente un ensemble de constructions qui peuvent être utilisées pour spécifier un système logiciel de taille et de complexité arbitraires.
Un composant UML 2.0 est une entité modulaire et réutilisable fournissant et requérant des interfaces qui peuvent être potentiellement exposées par l'intermédiaire de ports.
Un composant est vu comme une boîte noire. La partie interne d'un composant n'est accessible qu'au travers de ses interfaces. Le composant est une entité d'encapsulation. Ses dépendances sont explicitement définies. Il peut donc être facilement réutilisé au sein de plusieurs développements.

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

Un port est une entité qui émerge d'un composant. Le port se comporte comme un point d'interaction du composant avec l'environnement typé par plusieurs interfaces.

Un port peut être requis, fourni ou complexe. Un port requis ne possède que des interfaces requises et signifie que l'instance du composant doit être connectée à un port de l'environnement fournissant les méthodes requises. Un port fourni ne contient que des interfaces fournies. Un port complexe peut contenir des interfaces requises ou fournies. Ce type de port est très utile pour décrire l'interaction d'un composant avec son environnement dans le cas de service complexe.

Un port peut se voir attacher un comportement pour décrire les interactions qui doivent se produire sur le port. Les ports peuvent être connectés si et seulement si leurs spécifications de structure et de comportement sont *compatibles*.

UML 2.0 définit deux types de composant : le composant basique (*Basic Component*) et le composant empaqueté (*Packaging Component*).

Au niveau des caractéristiques, un composant basique étend la notion de *classe*. C'est une entité instanciable qui interagit avec son environnement par l'intermédiaire de ports ou d'interfaces.
La description du comportement de ce composant est fondée sur des machines à états.
La notion de composant empaqueté étend la notion de composant basique. Si le composant basique définit la manière dont le composant est vu à l'extérieur du composite, le composant empaqueté est une entité composée d'autres éléments liés de façon cohérente tout au long du processus de développement du logiciel.

Le connecteur UML 2.0 :

Comme pour les ADLs, les interactions entre les composants sont décrites par des connecteurs. Un connecteur est une entité qui relie des ports ou des interfaces de

composant. Dans un diagramme d'architecture, une application ou un composant est constitué de l'assemblage d'autres composants par l'intermédiaire de ports interconnectés. Il existe plusieurs types de connecteur :

- Le connecteur d'assemblage qui permet d'assembler deux instances de composant en connectant un port fourni d'un composant au port requis de l'autre composant.
- Le connecteur de délégation qui permet de connecter un port externe au port d'un sous composant interne. L'invocation d'un service sur le port externe sera transmise au port interne auquel il est connecté.

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

La configuration UML 2.0 :

Il y a deux façons de voir le composant en UML, la vision boîte noire présentée précédemment et une vision boîte blanche. Dans la vision boîte noire, le composant n'est défini qu'en fonction des éléments lui permettant d'interagir avec son environnement : ports, interfaces. Au contraire, la vision boîte blanche permet de modéliser les éléments de mise en œuvre des méthodes fournies par le composant.

L'introduction du concept de *partie* (*Part*) dans UML 2.0 rend possible la description de la partie interne d'une classe. Une partie est donc une propriété de la classe, c'est – à – dire elle a le même cycle de vie que l'objet de la classe à laquelle elle appartient. Les parties sont un ensemble d'instances d'autres classes. Comme le composant en UML 2.0 étend la notion de classe, sa structure interne est elle aussi décrite à l'aide de parties qui peuvent être respectivement des instances de classe ou des instances de composant. Si ces parties sont des instances de composant, il est possible de définir l'assemblage entre les ports de ces instances par l'intermédiaire de connecteurs.

Une partie représente une ou plusieurs instances d'une classe ou d'un composant grâce à des contraintes de multiplicité. Sans contrainte de multiplicité, l'objet est construit dès que la partie est construite. La multiplicité permet d'indiquer le nombre minimum et maximum d'instances au sein de la part mais aussi le nombre d'instances créées automatiquement lors de la création de la classe ou du composant contenant le part.

UML 2.0 permet donc, comme dans la plupart des langages de description d'architecture, de définir son architecture de manière hiérarchique. En effet, la notion de part permet de définir la réalisation d'un type de composant à partir d'un assemblage de composants.

1.4.3. Spécification de la dynamique dans UML2.0:

UML 2.0 propose différentes approches qui permettent de caractériser la dynamique de l'application. Entres autres, le diagramme de séquences permet de représenter la création et la destruction d'instances d'objets au cours du cycle de vie de l'application.

Ces constructions permettent de modéliser la reconfiguration d'une architecture en autorisant la création ou la destruction d'instances de composant ou de connecteur. Cependant, aucune opération spécifique n'existe pour cet aspect lié à la reconfiguration. La gestion du redéploiement ou la migration d'un composant au sein d'un autre composite ne sont pas des opérations simples à modéliser sur un diagramme de séquences.

La notion de parties dont on fixe la multiplicité au niveau du diagramme d'architecture est intéressante au niveau de la description de la dynamique. En effet, cela permet sur un diagramme d'architecture de fixer un cadre à l'évolution du contenu d'un composant en termes d'instances contenues. La multiplicité est donc importante à préciser d'abord dans un but descriptif, mais également pour avoir une analyse réaliste du modèle lorsque le modèle UML 2.0 sert de base à la vérification.

1.4.4. L'action Semantics de UML :

Le concept d'Action apparaît à de nombreuses reprises dans la notation UML : Action à exécuter en entrant dans un état ou en tirant une transition (diagramme des états), action à exécuter lors de l'envoi d'un message (diagramme de séquence), etc. La version 1.3 de la notation UML présente très succinctement quelques actions élémentaires telles la création ou la destruction d'un objet, mais fait l'impasse sur leur sémantique et ne propose pas de mécanisme efficace pour composer ces actions. Cette spécification par actions est complémentaire des autre vues UML. En effet, le niveau d'abstraction des diagrammes des états ou d'activité s'avère parfois inapproprié. [19]

La spécification des actions offre les caractéristiques suivantes [19] :

- Des modèles simulables dès la spécification initiale. Ceci afin de pouvoir vérifier le plus tôt possible l'adéquation entre la spécification, les modèles dérivés et leur correspondance avec les besoins de l'utilisateur. Il est possible d'appliquer des techniques de vérification extensive ou de prouver des propriétés de la spécification.
- Un niveau d'abstraction plus élevé qu'un simple langage de programmation. Le but de l'Action Semantics n'est en effet pas d'être le nouveau langage à la mode. Par exemple, l'Action Semantics doit permettre d'exprimer un maximum de parallélisme.
- Une indépendance vis – à – vis des cibles logicielle et matérielle choisies pour l'implantation. Il doit donc être possible de générer du code pour des plates – formes différentes à partir d'une même spécification.
- L'Action Semantics devra favoriser la réutilisation.

1.4.4.1. Présentation de l'Action Semantics :

Cette présentation s'articule autour des trois axes suivants : une extension au méta – modèle UML, un modèle d'exécution et une sémantique des actions. [19]

- ***Le méta – modèle :*** Le but de ce méta – modèle est d'étendre le méta – modèle UML actuel afin de combler ses lacunes concernant la syntaxe de certaines constructions, telles les nombreuses expressions non interprétées : gardes des transitions, actions dans un état, etc. Il définit en particulier une syntaxe abstraite permettant de décrire précisément les actions et leur enchaînement. La granularité est assez fine puisqu'il est possible de décrire le comportement par des actions aussi élémentaires que la création ou la destruction d'un objet, la lecture ou l'écriture d'un attribut, la création ou la suppression d'un lien entre objets, ou encore l'envoi d'un message... des dépendances de contrôle entre ces actions ou des dépendances de données entre les entrées et les sorties de ces actions élémentaires permettent ensuite d'imposer un ordre partiel sur leur exécution. A coté de ces briques de base, des structures plus complexes sont également disponibles : regroupement d'actions

(actions composées, procédures) et structure de contrôle (boucle, choix). Cette syntaxe abstraite est renforcée par des règles de conformité écrites en OCL1, par exemple pour exprimer l'absence de cycle dans les dépendances de données ou de contrôle entre actions. [19]

- ***Un modèle d'exécution :*** Si nous reprenons l'analogie entre l'utilisation d'UML et l'utilisation d'un langage de programmation classique, le méta – modèle est analogue à la grammaire (EBNF) utilisée pour la description de la syntaxe du langage ; un modèle UML peut alors être vu comme un programme, dont la configuration initiale est décrite par les diagrammes de déploiement et d'objets de la notation UML. Cette description n'est finalement rien de plus qu'un arbre abstrait, l'intérêt de la notation graphique d'UML résidant dans la séparation des concepts entre les neuf vues qui apporte une plus grande lisibilité, une plus grande souplesse et une maintenance facilitée, un peu à la manière de la programmation par aspects. Certains modèles sont exécutables par nature (un modèle Java par exemple), selon des conventions bien précises (la fameuse méthode public main). Par contre un modèle décrivant les propriétés de sécurité ou un modèle d'architecture ne sont pas exécutables par construction. Les modèles UML font partie de ces modèles non exécutables de façon standard, mais l'initiative Action Semantics de l'OMG permet de spécifier des propriétés d'exécutabilité que l'on peut rajouter à un modèle UML. [19]

- ***Une sémantique des actions :*** Le modèle d'exécution permet de représenter la mémoire d'une hypothétique machine virtuelle. Il reste à définir comment cette mémoire est modifiée par l'exécution d'une action. Le comportement d'un modèle UML est caractérisé par l'ensemble des historiques de ses objets, chaque historique étant constitué d'une succession de configurations. Ces configurations sont des entités immuables, c'est-à-dire que chaque exécution d'une action à effet de bord entraine la création d'une nouvelle configuration qui est ajoutée à l'historique de l'objet. Les actions complexes sont décomposées en un ensemble d'étapes élémentaires qui donnent chacune naissance à une nouvelle configuration. L'ensemble des étapes élémentaires qui composent l'exécution d'une action est appelé dans l'AS <<cycle de vie>>. Le cycle de vie <<générique>> d'une exécution est composée de quatre étapes : #waiting, #ready, #execute et #complete. Une exécution progresse de l'état

> #waiting à l'état #ready lorsque toutes les données dont elle a besoin sont disponibles (les arguments d'un appel de méthode, par exemple). L'état #execute réalise l'exécution proprement dite en créant une nouvelle configuration qui doit vérifier la post condition de cet état. L'exécution progresse de #execute à #complete lorsque toutes les valeurs de <<retour>> ont été calculées et sont disponibles. La nouvelle configuration dépend des configurations précédentes dans l'historique de l'objet et de l'action en cours d'exécution. L'Action Semantics ne dit pas comment calculer cette nouvelle configuration, mais indique précisément les propriétés qu'elle doit vérifier. En effet, chaque étape élémentaire de l'exécution d'une action est spécifiée par ses pré – conditions et post – conditions. L'exécution ne peut progresser dans une étape que lorsque la pré – condition de celle – ci est vérifiée (sur la configuration courante). Lorsque l'exécution d'une étape élémentaire est terminée, la configuration qui vient d'être créée doit vérifier la post – condition. [19]

Une manière simple de réaliser un interpréteur de modèle UML est donc d'utiliser un moteur d'inférence selon un mécanisme de chaînage avant : une base de faits initiaux est déduite des diagrammes d'objets et de déploiement du système. Lorsqu'une pré – condition (une des règles du moteur d'inférence) est vraie, un nouveau fait correspondant è la post – condition est ajouté è la base des faits, permettant ainsi éventuellement de poursuivre l'exécution. [19]

Evaluation :

Le diagramme de séquences en UML 2.0 permet de représenter la création et la destruction d’instances d’objets. Ces constructions permettent de modéliser la reconfiguration d’une architecture en autorisant la création ou la destruction d’instances de composant (Le connecteur n’est pas instanciable en UML2.0 car il n’est pas considéré comme un type ou un classificateur instanciable et réutilisable). De plus Precise Action Semantics est purement sémantique et ne propose aucune syntaxe concrète permettant son usage.

1.5. Conclusion :

Nous avons, dans cette partie, présenté les différents langages d'architectures logicielles dont certains prennent en charge le caractère dynamique des architectures. La plupart de ces langages ne permettent pas la prise en charge de l'aspect connecteur dynamiques et composant dynamiques. D'autres permettent la création de composants dynamique mais ne spécifié pas les mécanismes de leur gestion une fois créés (C2SADEL, DARWIN), d'autres ne présentent aucun mécanisme de suppression des composants ou des connecteurs.

Globalement, la dynamique de l'architecture n'est prise en compte que par peu d'approches. Par exemple, Wright ne s'y intéresse pas. D'autres modèles de composant comme SOFA ou Fractal fournissent des capacités de reconfiguration dynamique assez avancées. AADL propose par l'intermédiaire des modes de prévoir l'ensemble des configurations possibles pour le système. Ce mécanisme peut se révéler rapidement inefficace si l'architecture du système peut évoluer énormément. ArchJava est un modèle quelque peu à part. N'étant pas au départ un langage de spécification, mais une extension de langage de programmation pour la prise en compte des concepts architecturaux au niveau de l'implantation, la dynamique du système est présente dans un programme ArchJava. Cependant, cette dynamique étant noyée dans le reste du programme, il n'y a pas pour le moment d'outil permettant l'analyse du système. Darwin au contraire propose une description de la dynamique du système. Dans un objectif d'analyse, Darwin propose d'expliciter les mécanismes d'instanciation des composants du système. [6]

Concernant UML 2.0 on a vu qu'il propose par l'intermédiaire de son diagramme de séquences de décrire explicitement la création d'instances de composant. Il permet en outre de décrire un ensemble d'invariants du système au niveau de la description de l'assemblage. Un composite est composé de parties qui peuvent être soit un ensemble d'instances de composant connectées au travers de leur port, soit un ensemble de types de composant liés par des liens d'assemblage. Les types de composant et les ports possèdent alors une cardinalité qui précise le nombre d'instances de composant qui peut appartenir à l'assemblage de composants. Les diagrammes d'architectures dans UML 2.0 présentent une certaine convergence entre les ADLs et le modèle de composant d'UML.

CHAPITRE 2
LES LANGAGES ACTION

2.1. Introduction :

Dans un processus de type MDA (*Model Driven Architecture*), ou d'Ingénierie Dirigée par les Modèles, la description du comportement du système est un des points clé avant de pouvoir poursuivre le processus de modélisation vers la génération de code exécutable dans un langage cible approprié pour le déploiement de l'application comme C, C++ [13].

Pour être capable d'exécuter un modèle, les modeleurs utilisent généralement le langage qui sera utilisé par la suite après la phase de génération de code. Ainsi, il est courant de retrouver dans un diagramme de classes UML des notes écrites en C++ décrivant le comportement d'une opération donnée dans un modèle de conception détaillée par exemple. Cette pratique usuelle dans la phase de modélisation n'est pas du tout en adéquation avec certains principes fondamentaux que sont la maintainabilité des modèles ou la réutilisabilité [13].

Pour permettre la spécification du comportement (donc de la dynamique) dans les modèles de conception de manière indépendante des langages de niveau implémentation, un langage de haut niveau d'abstraction, permettant ce types de spécification de comportement est devenu nécessaires [18]. L'OMG donna le nom de langage d'action à de tels langages. L'utilisation d'un Langage d'action remplaçant ces langages cibles dans la specification des modèles est un besoin qui a été identifié par l'OMG dans le cadre de modélisation UML. C'est ainsi que dans UML1.5 et UML2.0 *Precise Action Semantic* a été introduit [19]. Cependant l'OMG n'a défini aucune syntaxe concrète pour ce langage d'action.

L'utilisation d'un langage d'action dans la modélisation UML à plusieurs avantages par rapport à l'utilisation d'un langage de programmation classique. Parmi ces avantages nous citons [13]:

- Les modeleurs peuvent réutiliser les modèles même si la cible change.

- Les modèles sont des briques logicielles devant être comprises par tout intervenant qui n'est pas forcément habitué à des langages de type langage de programmation.
- Les langages de programmation sont parfois trop compliqués, trop complexes pour l'usage qu'on en ferait au niveau d'un modèle comportemental.
- Les fonctionnalités de connexions vers d'autres formalismes peuvent être facilitées par cette plus grande simplicité des caractéristiques de modélisation.

Différents Langage action ont déjà été définis mais dans des contextes bien déterminés comme les télécommunications (*ITU – T, languages for telecommunications applications – SDL combined with UML*) ou la définition de « *Precise Action Semantics for the UML»* qui a été adoptée par l'OMG. Cependant cette approche est purement sémantique et ne propose aucune syntaxe concrète permettant son usage [13].

La définition d'une syntaxe concrète à *Action Semantics* a été traitée par plusieurs acteurs comme iUML de Kennedy Carter, l'approche Cassandra de ARTiSAN1, Nucleus BrigePoint de Accelerator Technology ou la solution de Kabira Technology2 avec son langage Kabira Action Semantics.

Nous présentons dans ce qu'il suit les éléments fondamentaux des langages d'action à travers l'étude de quelques langages d'action. C'est à la base des ces éléments que nous définirons par la suite le langage d'action SEAL de l'approche IASA pour l'architecture logicielle.

2.2. Precise Action Semantic de UML 2.0 :

Précise action semantic est basé sur deux concepts: le concept d'action et celui d'activité. Precise Action Semantic se veut général en adressant tous les domaines et se veut indépendant de tout processus de conception.

2.2.1: Le concept d'action :

Le standard UML2.0 [13] définit un package particulier appelé *« Actions »* qui définit en détails comment modéliser toutes les actions d'une application afin d'obtenir un modèle exécutable.

Les actions sont les entités comportementales de base permettant la spécification de modèles UML exécutables. Les actions échangent des flots de contrôle et des flots de données à travers des *InputPins* et des *OutputPins*.

Le standard UML s'attache à définir la sémantique des actions en les regroupant en quatre paquetages : [13]

- Le paquetage *BasicActions* définit les actions d'appel d'opérations, d'envois de signaux et d'invocation de comportements.
- Le paquetage *IntermediateActions* définit des actions d'invocations (diffusion de signaux et envois d'objets qui ne sont pas des signaux), des actions de lecture et d'écriture d'objets, de caractéristiques structurelles et de liens.
- Le paquetage *CompleteActions* définit des actions traitant de la relation entre les objets, les classes et les liens d'objets ainsi que des actions de gestion des événements tels que les acceptations d'appels d'opération.
- Le paquetage *StructuredActions* définit les actions opérant dans le cadre d'activités. Ces actions concernent la manipulation de variables et la gestion des exceptions.

La figure 2.1 illustre le Paquetage *Action* en UML2.0.

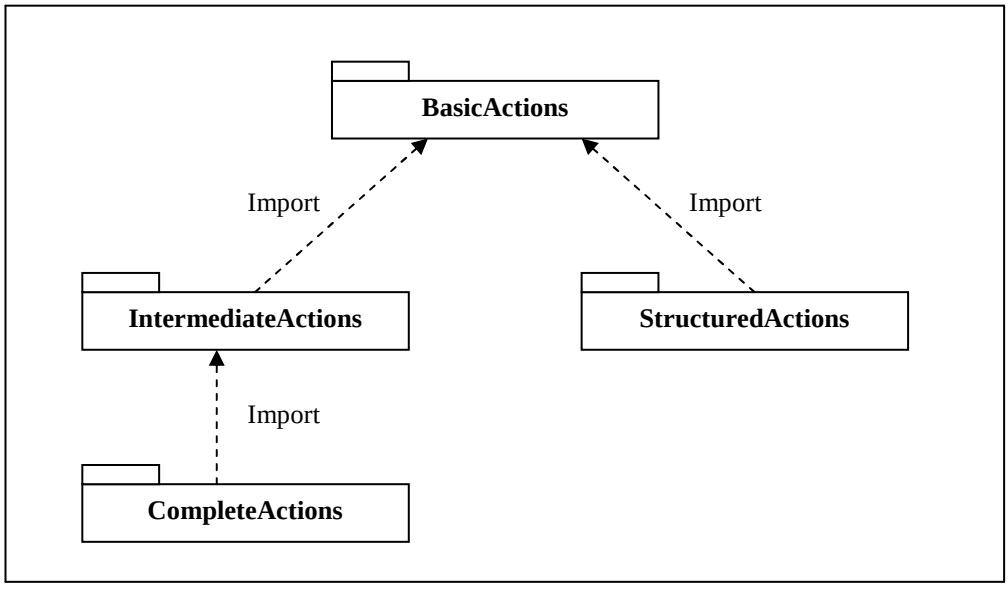

Figure 2.1 : Le Paquetage *Action* en UML2.0.

Le standard UML2.0 s'est attaché à définir une syntaxe abstraite et une sémantique pour le Langage d'action UML. Mais, cette norme n'est pas utilisé car aucun langage de surface ou syntaxe concrète n'est proposée qui satisfasse la sémantique. [13]

2.2.2. : Le concept d'activité :

Une activité est une spécification d'un comportement paramètrè construit à partir de l'enchaînement de sous unités dont les éléments de base sont des actions.

Contrairement aux actions qui sont toutes définies par le standard UML2.0, les activités servent à définir des comportements spécifiés par l'utilisateur. Les activités peuvent être appelées par des actions, Les activités peuvent posséder des paramètres d'entrée ou de sortie.

Le modèle d'activités de UML 2.0 est organisé en différents paquetages reflétant les différents niveaux de sémantique offerts : [13]

- Le paquetage *FundamentalActivities* définit les activités comme noeuds contenant des actions. Ce niveau sémantique est partagé par les activités basiques (*basic activities*) et les activités structurées (*structured activities*).
- Le paquetage *BasicActivities* offre la spécification des activités avec des flots de contrôle et de données entre les actions.
- Le paquetage *IntermediateActivities* définit les activités offrants les flots de contrôle et de données concurrents. Ce niveau de sémantique permet la modélisation de réseaux de Pétri classiques.
- Le paquetage *CompleteActivities* permet la définition d'activités avec des constructions évoluées telles que les transitions pondérées ou encore le « streaming » de données.
- Le paquetage *StructuredActivities* permet la définition d'activités constituées de constructions classiques de programmation structurée comme les boucles ou les branchements conditionnels.
- Le paquetage *CompleteStructuredActivities* permet la définition d'activités avec des flots de données en sortie des boucles ou des branchements conditionnels.

– Le paquetage *ExtraStructuredActivities* permet la définition d'activités contenant des exceptions ou des invocations de comportement sur des ensembles de valeurs.

La figure 2.2 illustre le Paquetage *Activités* en UML2.0.

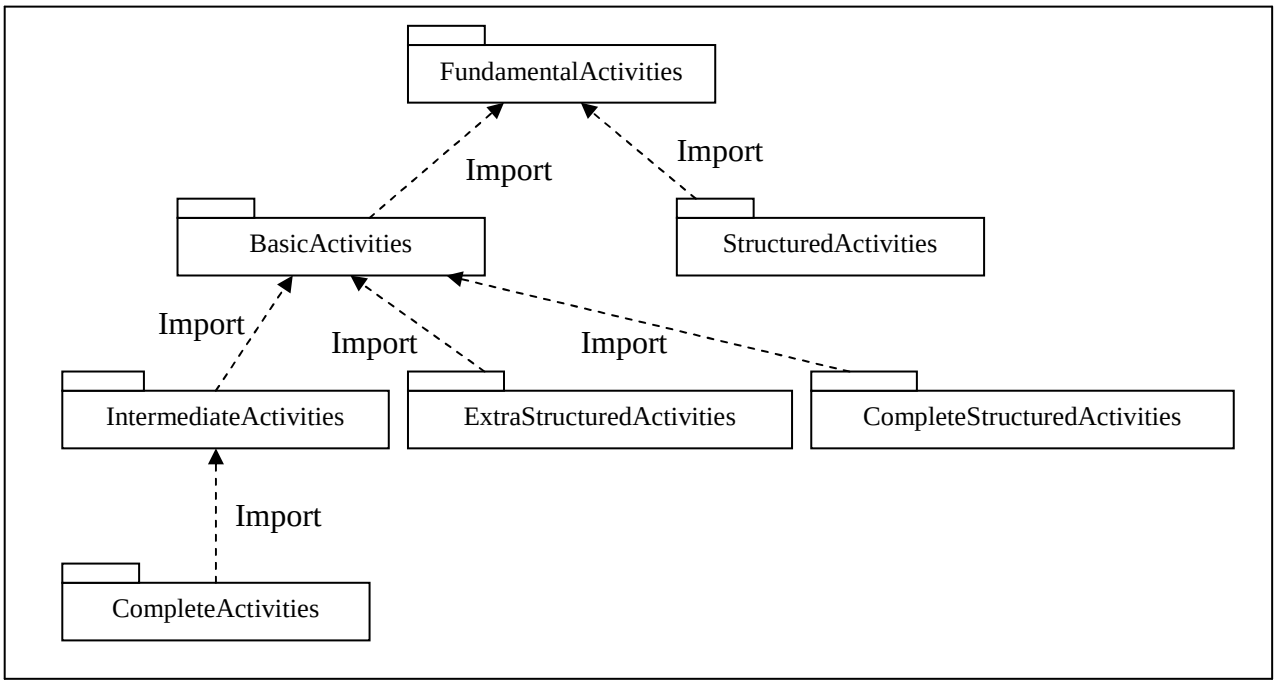

Figure 2.2 : Le Paquetage Activités en UML2.0.

La définition d'un langage respectant la sémantique d'activités de UML 2.0 ne requiert pas obligatoirement l'utilisation de tous ces niveaux de sémantique. Un choix d'un sous ensemble suffisant pour un domaine ou une plate – forme donnée est nécessaire.

Tout utilisateur voulant construire un modèle UML sur les bases présentées ici doit définir son propre langage et doit alors définir une notation textuelle et graphique et définit le lien entre ces constructions et le paquetage *Action* de UML2.0. [13]

2.3. Spécification et description avec SDL :

Le langage SDL (Specification and Description Language) ou Langage de description et de spécification est développé depuis 1972. SDL est une technique de description formelle très utilisée dans le monde des télécommunications. Cependant son cadre d'application

actuel dépasse largement ce domaine, et il est employé avec succès dans des domaines comme les systèmes embarqués ou multimédia il a pour objectifs :

- D'être facile à apprendre, à utiliser et à interpréter,
- De permettre l'élaboration de spécifications,
- Et de permettre des évolutions futures du langage.

Ses domaines d'applications sont très variés mais se concentrent plus spécifiquement sur :

- Les systèmes de télécommunication,
- Les systèmes distribués,
- Les protocoles de communication,
- Et les systèmes temps – réel.

2.3.1. Spécification d'un système :

SDL décrit le comportement d'un système sous la forme de stimulus / réaction, les stimuli et les réactions sont des entités discrètes et contiennent de l'information. En particulier, la spécification d'un système est vue comme étant la séquence de réactions associée à une séquence de stimuli [12].

Un système SDL [12] est composé d'un ensemble de *blocs*. Les blocs sont connectés entre eux et à l'environnement par des *canaux*. A l'intérieur de chacun des blocs, il y a un ou plusieurs *processus*. Un processus est une machine d'états finis étendue fonctionnant de manière concurrente et autonome avec les autres processus. Les processus communiquent de manière *asynchrone* par l'intermédiaire de *signaux*. Aucune synchronisation n'est requise entre l'émetteur d'un signal et son consommateur. Un processus peut envoyer (ou recevoir) des signaux vers (ou de) l'environnement ou les autres processus d'un même système [12].

SDL est fondé sur les machines d'états finis étendues et sur les types abstraits de données. Le concept de types abstraits de données permet d'avoir des spécifications de données rigoureuses et avec plusieurs niveaux d'abstraction [12]

SDL fait appel à des concepts structurels qui facilitent la spécification des grands systèmes et des systèmes complexes. Il est ainsi possible de subdiviser la spécification d'un système en unités faciles à gérer qui peuvent être traitées et comprises de manière indépendante.
La figure 2.3 montre la structure générale d'une spécification de système en SDL :

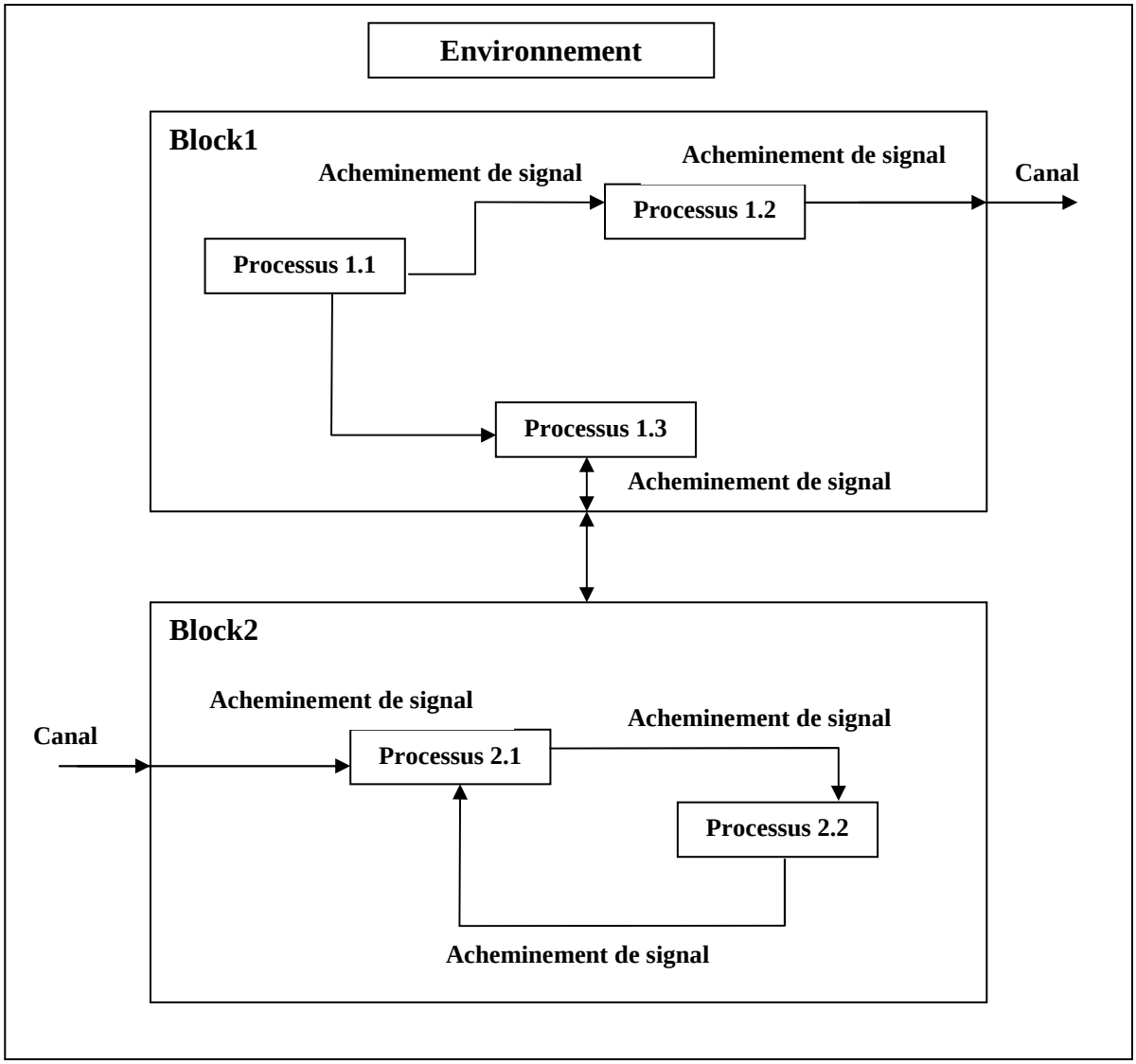

Figure 2.3 : Structure générale d'une spécification de système en SDL [12].

SDL offre le choix entre deux formes pour représenter un système :

- Une représentation graphique (SDL Graphical Representation).
- Une représentation textuelle (SDL Phrase Representation).

Ces deux formes sont des représentations concrètes de la même sémantique de SDL, elles sont équivalentes du point de vue sémantique.

2.3.2. Spécification textuelle de quelques composants d'un système :

Dans ce qui suit nous allons présenté la spécification textuelle de quelques composants d'un système, ces composants sont : Progiciels, système, bloc et Processus.

2.3.2.1. Progiciels et définitions référencées :

Une « spécification SDL » peut être décrite comme une définition de système monolithique ou comme une collection de « package » et de « définition référencée ». Un « package » permet la réutilisation de définitions dans différents contextes. Une « définition référencée » est une définition qui a été supprimée de son contexte de définition pour accroître la visibilité. Elle est insérée à un seul endroit en utilisant une référence. Les définitions qui font partie d'un progiciel définissent des types, des générateurs de données, des listes de signaux, des spécifications distantes et des synonymes. Avant de pouvoir analyser une définition de système concret, chaque référence doit être remplacée par la « définition référencée » correspondante. Dans cette substitution, « l'identificateur » de la « définition référencée » est remplacé par le « nom » dans la référence. [12]

Grammaire textuelle

<spécification SDL> ::=
<liste de progiciels> | [<définition de système> { <définition référencée> }*]
<liste de progiciels> ::= { <définition de progiciel> { <définition référencée> }* }*
<définition de progiciel> ::= { <clause de référence de progiciel> }*

package <nom de progiciel>
[<interface>] <fin>
{
<définition de type de système> | <référence de type de système> | <définition de type de bloc> | <référence de type de bloc> | <définition de type de processus> | <référence de type

de processus> | <définition de procédure> | <définition de procédure distante> | <référence de procédure> | <définition de signal> | <définition de liste de signaux> | <définition de type de service> | <référence de type de service> | <définition de sélection> | <définition de variable
distante> | <définition de donnée> | <définition de macro> }*
endpackage [<nom de progiciel>] <fin>

<clause de référence de progiciel> ::= use <nom de progiciel> [/ <liste de sélection de définitions>] <fin> }*
<liste de sélection de définitions> ::= <sélection de définition> { , <sélection de définition> }*
<sélection de définition> ::= <type d'entité> <nom>
<type d'entité> ::=
system | type block | type process | type service | procedure
| type signal | newtype | signallist | generator | synonym | remote

<interface ::= interface <liste de sélection de définitions>
<définition référencée> ::= <définition> | <diagramme pour la forme graphique>
<définition de système> ::= { <définition de système> | <diagramme de système }
<définition> ::= <définition de type de système> | <définition de type de bloc> | <définition de bloc> | <définition de type de processus> | <définition de processus> | <définition de procédure> | <définition de sous-structure de bloc> | <définition de sous-structure de canal> | <définition de type de service> | <définition de service> | <définition de macro> | <définition d'opérateur>

2.3.2.2. Système :

Décrire un système [12] revient, d'habitude, à décrire :

- Le nom du système,
- Les signaux échangés entre les processus du système et entre les processus du système et l'environnement,
- Les canaux pour la connexion entre les blocs et entre les blocs et l'environnement,
- Les données distantes manipulées par les processus du système,

- Les blocs composant le système.

L'interprétation d'une description de système consiste à créer des instances de processus et à lancer leur exécution. La communication entre le système et l'environnement ou entre les blocs intérieurs au système ne peut se faire qu'au moyen de signaux.

A l'intérieur d'un système, ces signaux sont véhiculés par des canaux. Les canaux relient les blocs entre eux ou à la frontière du système (c'est-à-dire l'environnement). Le système et son environnement interagissent selon un protocole connu des deux parties. Des processus de l'environnement peuvent envoyer ou recevoir des signaux. Un système peut être décrit pour gérer tous les comportements de l'environnement, ou au contraire être focalisé sur certains aspects de l'environnement seulement, les autres sont ignorés. Une « définition de système » est la représentation en SDL d'une spécification ou de la description d'un système. [12]

Grammaire textuelle

<définition de système> ::=
{ <clause de référence de progiciel> }*
system <nom de système> <fin>
{ <entité dans système> }+
endsystem [<nom de système>] <fin>

<entité dans système> ::=
<définition de bloc> | <référence de bloc>
| <définition de canal> | <définition de signal>
| <définition de liste de signaux> | <définition de sélection>
| <définition de macro> | <définition de donnée>
| <définition de variable distante> | <référence de type de bloc>
| <définition de type de bloc> | <définition de type de processus>
| <référence de type de processus> | <définition de procédure>
| <définition de procédure distante> | <référence de procédure>
| <définition de type de service> | <référence de type de service>
<référence de bloc> ::= block <nom de bloc> referenced <fin>

la figure 2.4 montre le schéma général d'un système.

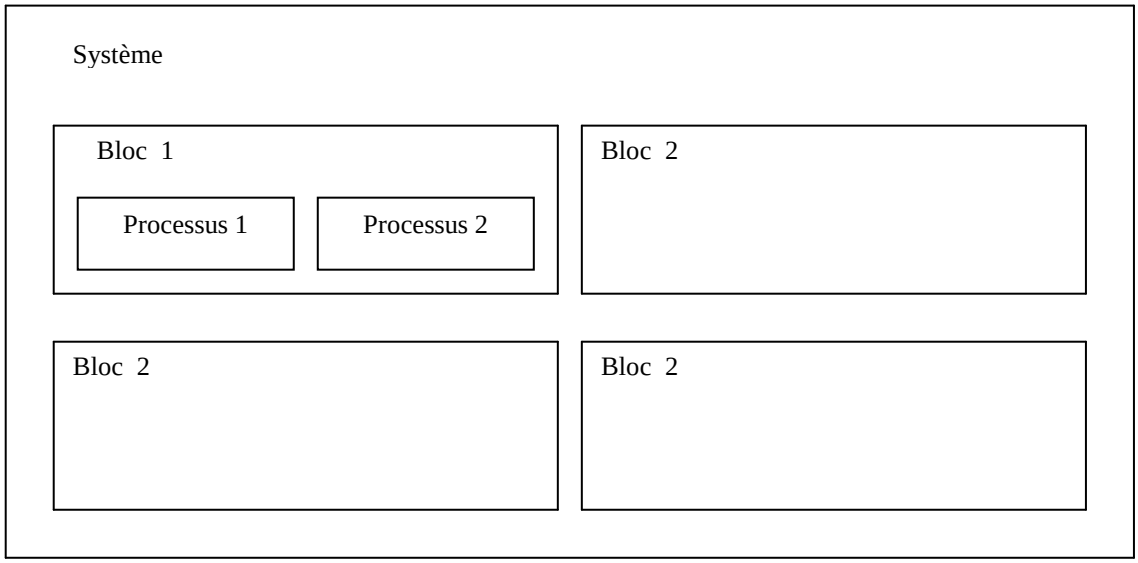

Figure 2.4 : schéma général d'un système.

2.3.2.3. Bloc :

Une « définition de bloc » contient une ou plusieurs définitions de processus d'un système et éventuellement une sous structure de bloc. La définition de bloc permet de regrouper les processus qui tous ensemble assurent une certaine fonction. Une « définition de bloc » assure une interface statique de communication par laquelle les processus peuvent communiquer avec d'autres processus. De plus, elle donne la portée pour les définitions de processus. [12]

Interpréter un bloc consiste à créer les processus initiaux dans le bloc. Décrire un bloc revient, d'habitude, à décrire :

- Le nom du bloc,
- Les signaux visibles à l'intérieur du bloc,
- Les routes pour interconnecter les processus du bloc,
- Les connexions des routes à des canaux pour l'interconnexion de canaux externes au bloc et les routes internes au bloc,
- Les processus composant le bloc.

la figure 2.5 montre le schéma d'un bloc :

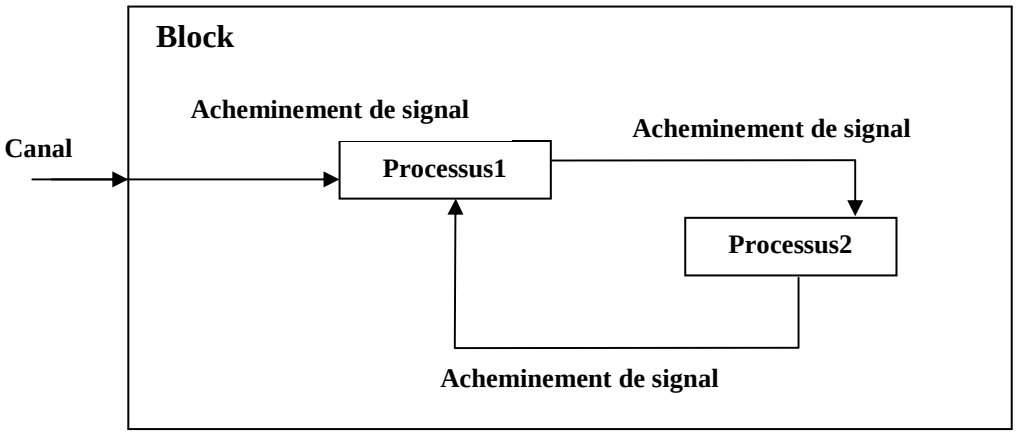

Figure 2.5 : schéma d'un bloc.

2.3.2.4. Processus :

Une « définition de processus » introduit le type d'un processus qui est destiné à représenter un comportement dynamique. Une instance de processus est une machine à états finis étendue communicante qui assure un certain nombre d'actions, appelées transitions, suite à la réception d'un signal donné, chaque fois qu'elle se trouve dans un certain état. La réalisation de la transition aboutit à l'attente du processus dans un autre état, qui n'est pas nécessairement différent de l'état d'origine. [12]

Plusieurs instances du même type de processus peuvent exister au même instant et s'exécuter de manière asynchrone et en parallèle les unes aux autres, et avec d'autres instances de différents types de processus dans le système.

Décrire un processus revient, d'habitude, à décrire :

- Le nom du processus,
- Les paramètres formels du processus,
- Les variables du processus,
- Les réveils manipulés par le processus,
- La description, à l'aide d'états et transitions, du comportement du processus.

2.3.3. Représentation graphique :

La grammaire graphique est décrite dans la norme ITU-T Z.100. Les symboles graphiques pour permettre la compréhension de spécification de systèmes SDL décrits graphiquement. [12] sont indiqués dans la figure 2.6.

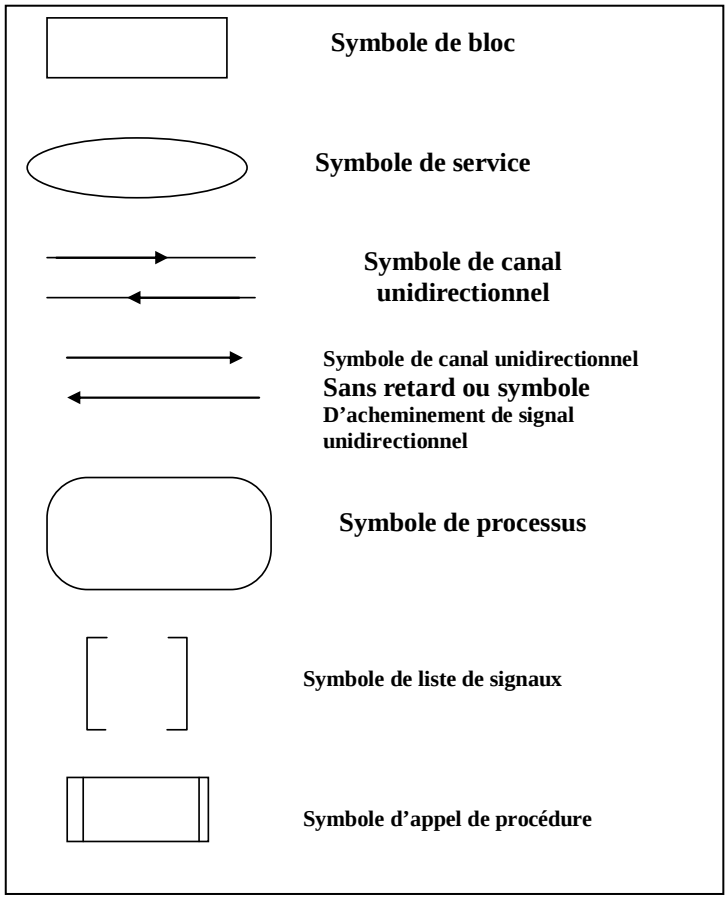

Figure 2.6 : Spécification de systèmes SDL décrits graphiquement.

2.4. Exemples de spécification en SDL :

On présente dans ce qui suit quelques exemples généraux sur la spécification en SDL.

2.4.1. Echange de signaux entre processus :

la figure 2.7 montre un exemple de spécification des échanges de signaux entre processus.

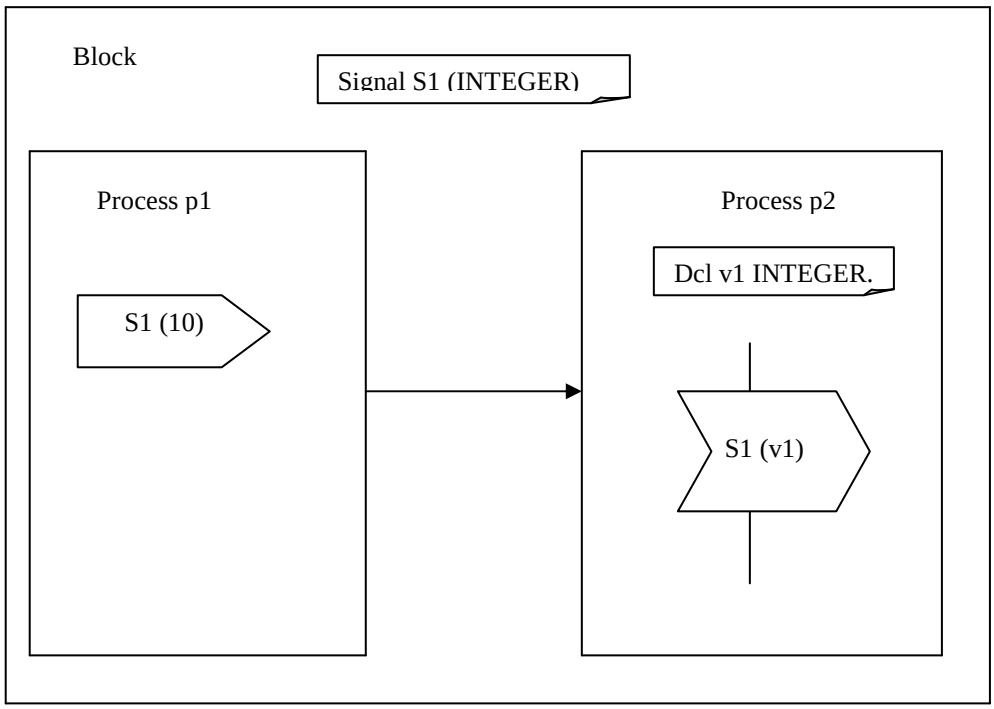

Figure 2.7 : Exemple d'échange de signaux entre processus

```
signal S1(integer) ;
process P1 ;
. . . . . .
output S1(10) to P2

process P2 ;
dcl integer v1 ;
. . .
input S1(v1)
```

Grammaire textuelle

```
<émission de signal> ::=
output <identificateur de signal> [ <paramètres effectifs> ]
{ , <identificateur de signal> [ <paramètres effectifs> ] }*
[ to <destination> ] [ via [ all ] <chemins à emprunter> ]
<destination> ::= <PId de processus> | <identificateur de processus>
<chemins à emprunter> ::= <chemin à emprunter> { , <chemin à emprunter> }*
<chemin à emprunter> ::=
<identificateur de route>
| <identificateur de canal> | <identificateur de porte>
```

2.4.2. Subdivision de bloc:

la figure 2.8 montre un exemple de subdivision de bloc.

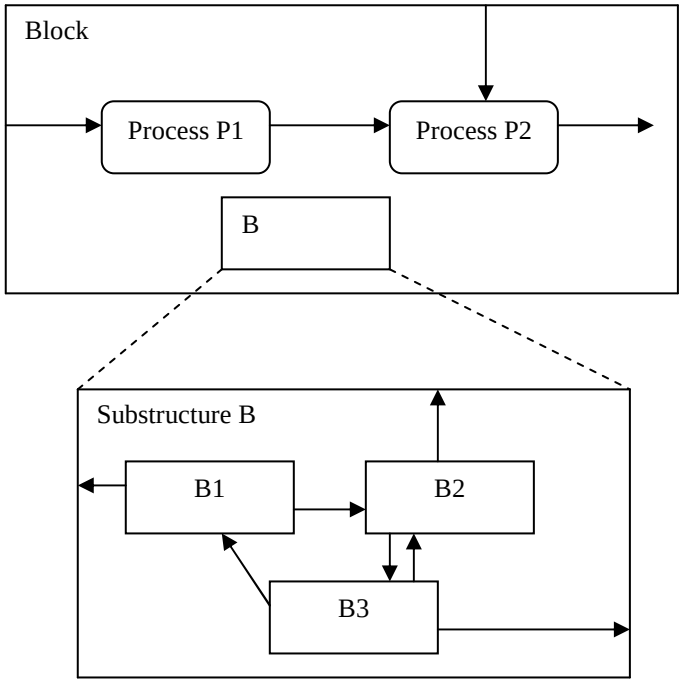

Figure 2.8 : exemple de subdivision de bloc.

2.4.3. Communication :

la figure 2.9 montre la communication entre les blocs.

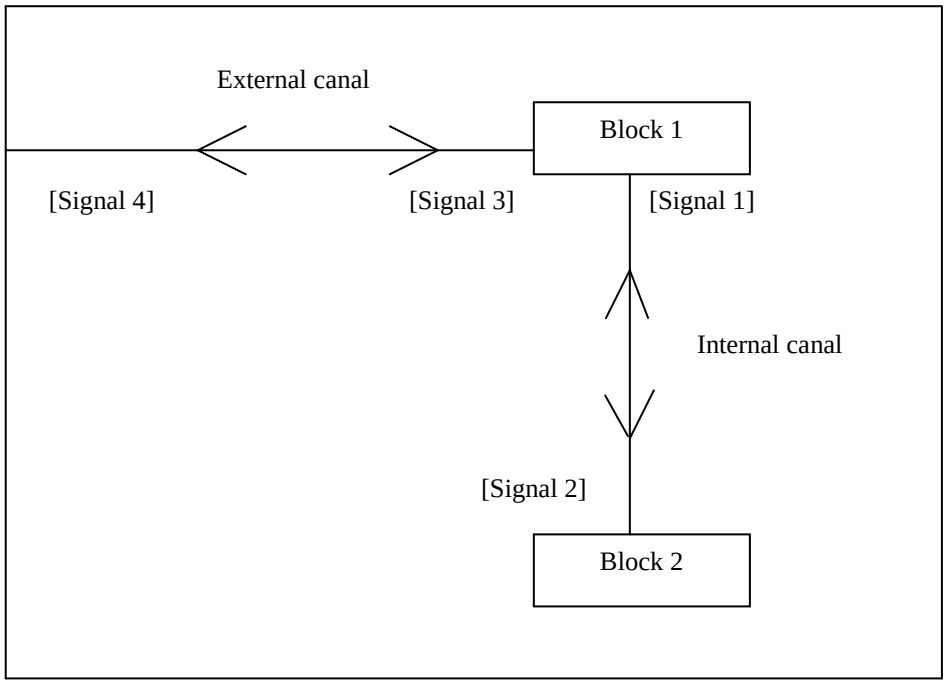

Figure 2.9 : communication entre les blocs [12].

Les blocks de l'architecture sont connectés entre par des channels qui décrivent les différents messages (ou signaux) qui sont échangés entre les blocks.

2.4.4. Comportement :

Le comportement est décrit graphiquement sous forme d'une machine d'état étendue.

Dans cet exemple (figure 2.10) « Variable » de type INTEGER est la seule variable locale au process. La première transition est la transition start qui initialise la variable locale. Un message de demande de connexion est envoyé (conReq), un timer de 5 secondes est démarré (conReqTimer), et l'automate se met dans l'état connecting. Dans l'état connecting si le timer claque -ce qui est l'équivalent de la réception d'un message- on renvoie la demande de connexion jusqu'à 10 fois. Si on reçoit une confirmation de connexion, l'automate passe dans l'état connected. C'est un scénario typique dans les protocoles de télécommunications.

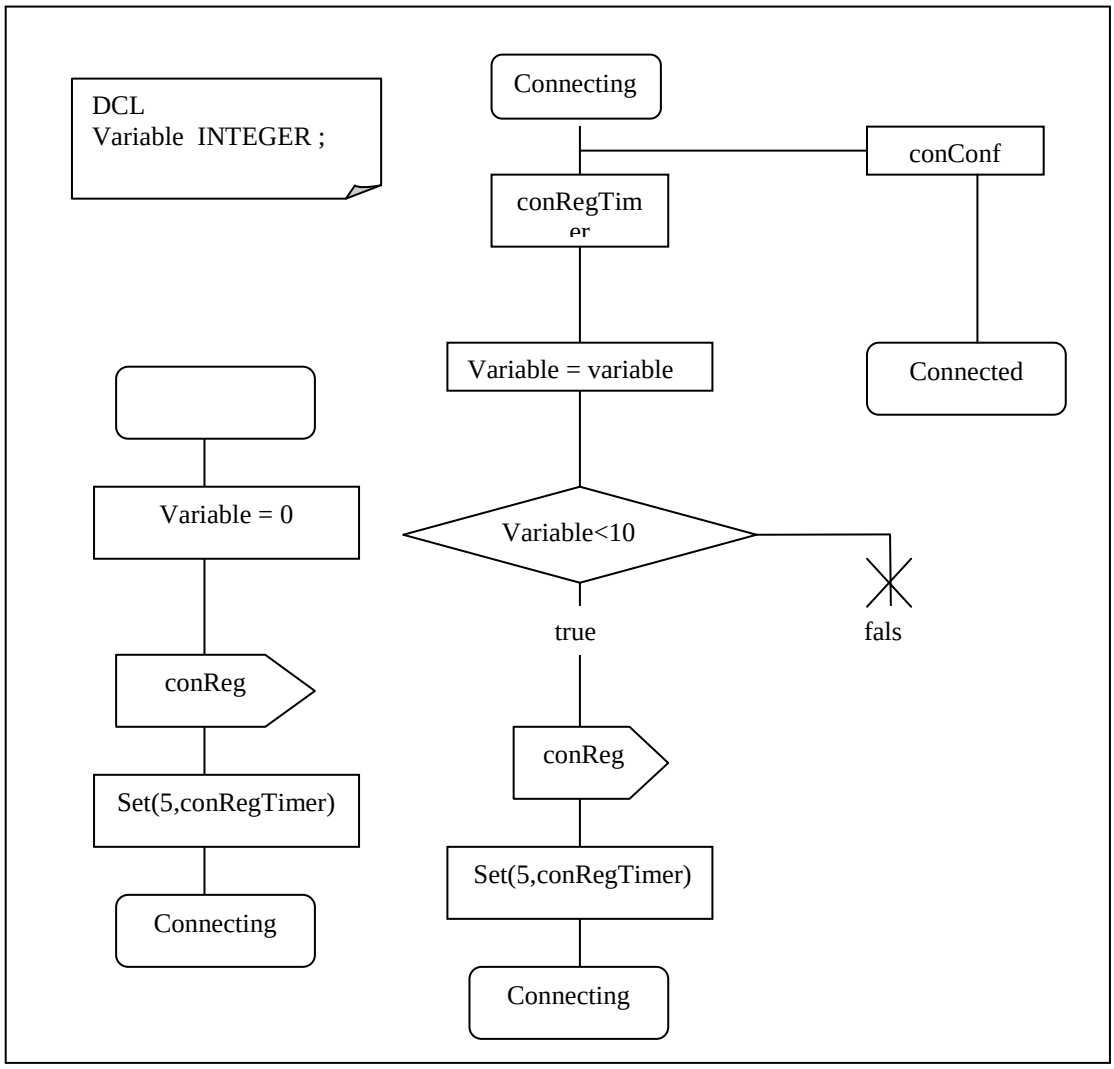

Figure 2.10 : spécification de comportement.

2.5. Conclusion : Bilan SDL / UML

SDL est un langage formel orienté – objet permettant de décrire et de spécifier des systèmes informatiques complexes, Ces propriétés font de SDL un bon langage pour concevoir, décrire, valider et simuler des systèmes temps réel, distribués, communiquant par signaux Ce standard a connu un succès rapide et les premiers outils ont fait l'unanimité dans le secteur des télécommunications. [12]

Par contre le langage SDL est fortement limité à la description des comportements de systèmes réactifs. Encore largement utilisé dans le domaine des télécommunications, SDL permet principalement de spécifier des systèmes complexes, temps réels, répartis, communiquant par signaux. Or ce type de système correspond essentiellement aux composants des réseaux télécoms et ne se prête pas à la modélisation précise d'une majorité des systèmes embarqués ou les systèmes d'information. [12]

D'ailleurs, même dans les télécoms la spécification SDL n'est pas toujours suffisante dans la phase de conception, et les entreprises choisissent l'utilisation d'outils alliant certains concepts SDL pour décrire le système avec des langages de programmation servant à l'implémentation finale. Ceci leur permet également de se défaire du type de donnée abstrait de SDL, trop imprécis, au profit des types complexes des langages utilisés pour l'implémentation sur cible. [12]

UML s'impose par sa flexibilité ainsi que par sa capacité à s'adapter aux cas les plus spécifiques, comme aux cas généraux. Cependant, UML n'a définit aucun langage de surface pour Precise Action Semantic. De plus, *Precise Action Semantic* a été définit pour supporter uniquement les concepts du modèle objet. Et ne reconnait pas les concepts de l'architecture logicielle (composant, port, connecteur etc..) qui ont été introduit dans UML2.0. Ainsi malgré son aspect général et sa puissance, *Precise Action Semantic,* et même si une syntaxe aurait été défini pour ce langage, il ne peut pas être utilisé pour décrire une architecture dynamique qui évoluent au niveau structurel et comportemental

CHPAITRE 3
LES MODELES FONDAMENTAUX DE L'APPROCHE IASA

3.1. Introduction :

L'objectif central du travail présenté dans cette thèse est la définition d'un langage d'action pour l'approche IASA (Integrated Approach to Software Architecture). Les autres objectifs sont :

- ✓ La mise en place d'un interpréteur du langage d'action qui permettra d'exécuter et valider une architecture.
- ✓ La mise en place d'un environnement de conception intégré d'architecture logicielle selon l'approche IASA dans lequel il sera possible de spécifier une architecture logicielle selon le modèle de composant IASA.

Les modèles de base de l'approche IASA représentent l'environnement dans lequel le langage d'action sera mis en ouvre. Ces modèles ont été défini pour permettre la spécification très flexible d'architecture logicielle. Ils permettent en outre la spécification libre de topologies très variées dont certaines ne peuvent être spécifiées par les ADL actuels. Le modèle de composant IASA s'inspire beaucoup des domaines complémentaires à l'architecture logicielle, notamment le domaine de l'architecture matérielle et l'architecture des réseaux. Dans ce qui suit nous présentons les éléments fondamentaux du modèle de composant IASA et nous nous limiterons aux concepts qui seront ciblé par le langage d'action à définir. [14] [15]

Dans ce qui suit nous présentons les éléments de base de modèle IASA. Nous commencerons par la notion de points d'accès puis les ports et les connecteurs et nous termineront par les composants.

3.2. Les Points d'accès :

Le point d'accès [14] [15] est le plus petit élément manipulable dans une architecture et représente l'élément de base pour la définition d'un port. Toute information, quelque soit sa nature arrive ou part d'un composant à travers un point d'accès. Il permet de localiser

les diverses ressources fournies ou requise à travers un port. Il permet de renseigner sur les éléments intervenant dans la réalisation d'un comportement observable sur un port.

Contrairement aux autres modèles de ports, ou le concept de point d'accès correspond à toute une interface, les points d'accès dans notre modèle est orienté vers le support des concepts de base échangés entre deux composants.

Pour l'instant, les concepts supportés par les points d'accès se réduisent à l'échange de données et au transfert du flux de contrôle. Ainsi, un paramètre d'une méthode peut être associé à un point d'accès vu qu'il véhicule une information.

Un point d'accès [14] [15] peut être manipulé indépendamment des autres points d'accès. Il est ainsi possible de tirer une connexion d'un point d'accès d'un port de composant vers un autre point d'accès d'un autre port (figure 3.1), sans que les autres points d'accès des deux ports n'interviennent dans cette connexion. De cette manière il deviendrait possible d'utiliser un paramètre d'une méthode indépendamment de la méthode dans laquelle il est défini. Ce concept n'est pas supporté par les outils et modèle actuels en architecture logicielle.

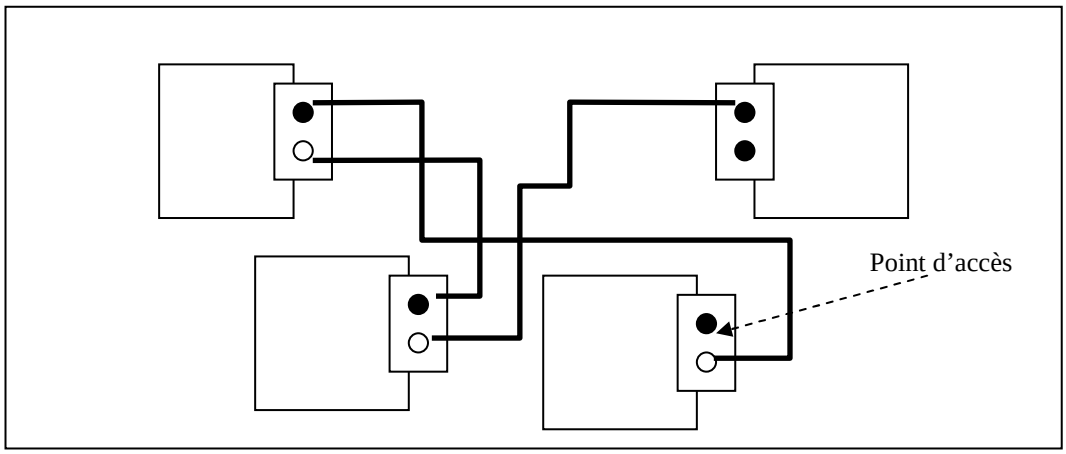

Figure 3.1 : Connexions utilisant les points d'accès.

3.2.1. Les types fondamentaux des points d'accès :

Les points d'accès sont tous représentés par le type de base *IASA_AcessPoint.*. Le type *AnyPoint* possède un mode de communication bien précis, Il est doté de propriété générale

aux points d'accès et d'opérations de base notamment celle destinées à des opérations réflexives. [14][15].
Un point d'accès est destiné soit aux transfert de données *(DataPoint)* ou émettre ou recevoir des actions *(ActionPoint).*

La figure 3.2 représente les types fondamentaux des points d'accès.

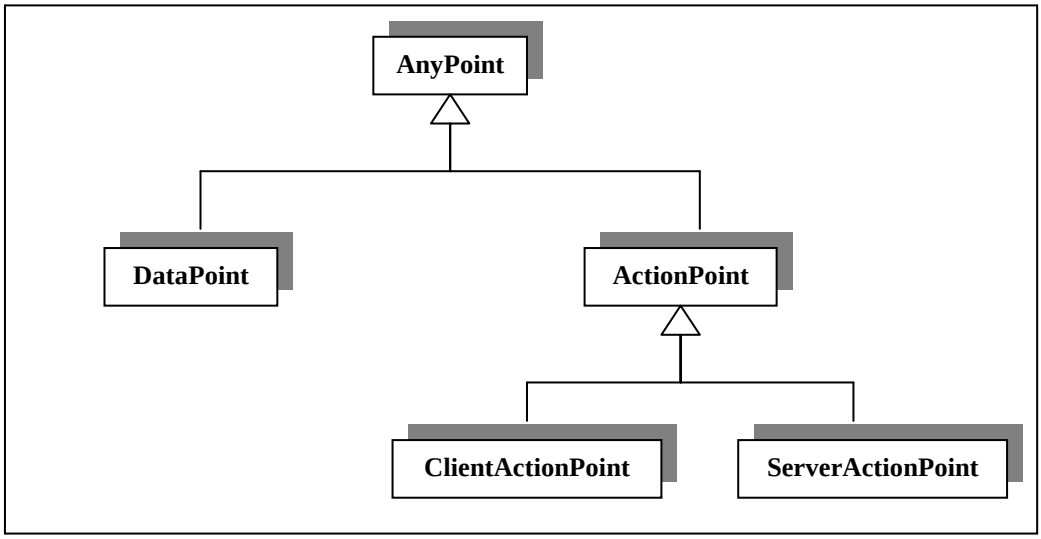

Figure 3.2 : les types fondamentaux des points d'accès.

Un *ActionPoint* indique la présence d'un service qui pourrait être initié à partir de ce point ou réalisé en ce point. Il est dédié aux transferts de flux de contrôle (invocation d'une action, reprise du flux de contrôle par une action).

Un point d'accès peut être marqué ou non marqué (*marked, unmarked*). Un point d'accès marqué est un point d'accès correctement connecté. Cette caractéristique est très utile dans le processus de validation d'une architecture et est exploitée dans un processus de conception du général vers le particulier avec validation progressive de l'architecture durant les différente phase de conception, sans nécessité que les composants utilisés soient effectivement réalisés. [14][15]

Les points d'accès [14] [15] explicite dans notre modèle sont :

3.2.1.1. Les DataPoint :

Les *DataPoint* associé à un type bien précis de donnée : *IntDataPoint*, *DoubleDataPoint*, *StringDataPoint*, ...

Les types associés aux points d'accès peuvent être très complexes. Dans nos objectifs, lors des opérations de gestion dynamique d'une architecture, un composant entier pourrait être transmis et reçus à travers un point d'accès donné supportant de manière explicite le type de composant ou à travers un point d'accès *DataPoint*.

Un *DataPoint* est doté d'un attribut indiquant les sens possible de transferts de donnée. Les valeurs supportées sont : *In*, *out* et *InOut*.

Un *DataPoint* est utilisé pour spécifier un transfert explicite de données. L'architecte peut utiliser un ensemble de *DataPoint* prédéfini ou définir un *DataPoint* qui lui est spécifique.

Les *DataPoint* prédéfini englobent les types de données primitifs que nous retrouvons dans les divers langages de programmation (entier, réel, caractère, booléen) ainsi que des types spécifiques à notre approche, tels que les types de données renseignant sur l'état structurel d'un composant composite.

Dans le contexte actuel où le langage Java est utilisé comme cible dans les diverses opérations de validation de modèles, les types primitifs associé Aux DataPoint correspondent aux types primitifs Java, A titre d'exemple, *IntDataPoint*, *ByteDataPoint*, *CharDataPoint* et *BooleanDataPoint* sont respectivement associé aux types primitifs int, byte, char et boolean.

Par exemple les points d'accès *SubCmpSetDataPoint*, *ConSetDataPoint*, *DConSetDataPoint*, *OpPartPortSetDataPoint*, *CtrlPartPortSetDataPoint*, *ServiceConSetDataPoint* sont associés respectivement à des types de données renseignant sur l'état structurel d'un composite (Liste de composants, liste des connecteurs, liste de port interne).

3.2.1.2. Les ServerActionPoint :

C'est un point associé à un et un seul service. Le composant sur lequel se trouver un *ServerActionPoint* à la charge d'assurer l'activation du service. Le service peut être implémenté partiellement ou totalement par le composant associé au *ServerActionPoint*.

3.2.1.3. Un ClientActionPoint :

Permet au composant associé d'accéder de l'extérieur à un service présent dans un *ServerActionPoint*. Ce dernier peut être associé au même composant que *ClientActionPoint* ou à un autre composant.

3.2.2. Les points d'accès contrôlés :

Un point d'action contrôlée est un *ServerActionPoint* doté d'un contexte d'action dite de contrôle de service, il est représenté par le type *ControledServerActionPoint*. Les actions du contexte de contrôle de service sont *(Enable, Disable, Start, Stop, Restart, Terminate, Delete, Create).*

3.2.3. Les points d'accès d'états :

Ce sont des points d'accès à travers lesquels un service renseigne sur son état. Nous distinguons deux types de point d'état :

- *Un StateDataPoint,*
- *Un StateActionPoint.*

Un *StateDataPoint* est un point de donnée dont l'attribut du sens est fixé à *out*. Il est doté d'un attribut indiquant l'état du service *(Created, Deleted, Terminated, Started, Stopped, Running, Stop, None, Exception).* Selon le mode de fonctionnement du point d'accès seul un sous ensemble de ces états sera valide. Ainsi *Started, Stopped, Killed, Created* sont utilisés en mode pulsionnel et les états tels que *Running, Stop, None* sont utilisé en mode continu.

Un *StateActionPoint* est en fait un *ClientActionPoint* doté des mêmes attributs que le *StateDataPoint*. De plus à chaque état peut être associée une action bien précise.

3.2.4. Les points d'accès d'exception :

Les exceptions sont gérées par trois types de points d'accès.

- *ExceptionActionPoint,*
- *ManagerExceptionActionPoint,*
- *ExceptionDataPoint.*

L'ExceptionActionPoint représente un point à travers lequel un service extériorise un état exceptionnel. Un *ExceptionActionPoint* est un *ClientActionPoint* au niveau duquel est reportée l'action qui devrait permettre de gérer l'état exceptionnel.

Un *ManagerExceptionActionPoint* est le point à travers lequel se fait l'accès au service de gestion de l'exception.

L'ExceptionDataPoint est le point à travers lequel est véhiculée l'information décrivant l'exception.

Dans l'état actuel, toutes les exceptions sont décrites au niveau du type *AnyException* contenant des informations générales sur l'exception (texte décrivant la cause) et le lieu ou l'exception est apparu (nom de service, instance de composant).

A titre d'exemple, la disparition d'un service suite à la suppression dynamique d'une connexion entraînera la provocation de l'exception *UndefinedServiceException* qui pourra être véhiculé par un *ExceptionDataPoint UndefinedServiceExceptionDataPoint*.

Dans l'état actuel, la gestion de l'exception nécessite la combinaison d'un *ExceptionDataPoint* avec soit un *ExceptionActionPoint* soit un *ManagerExceptionActionPoint*.

3.3. Les ports :

Un port [14] [15] quelconque est représenté par le type *AnyPort* (figure 3.4). Un port représente le regroupement logique de plusieurs points d'accès. Le type *AnyPort* est doté de la propriété générale aux ports (nom d'instance, nom de type, instance de composant

associé) et d'un attribut indiquant si tous ses point d'accès sont connecté ou non à d'autres points d'accès. Un certain nombre de fonction de bases sont fournie par ce type, notamment celles permettant des opérations réflexives sur les ports.

Nous distinguons plusieurs sous types de *AnyPort* organisé en quatre familles :

- Ports ordinaires,
- Ports de contrôles,
- Ports d'états,
- Ports d'exception.

3.3.1. Les ports ordinaires :

Les ports ordinaires sont représentés par les types : [14] [15]

- *ClientPort,*
- *ServerPort,*
- *PeerPort,*
- *DataPort.*

- Un *ClientPort* ne peut contenir qu'un point d'accès de type *ClientActionPoint* et zéro ou plusieurs points d'accès de type *DataPoint.*
- Un *ServerPort* ne peut contenir qu'un point d'accès de type *ServerActionPoint,* 0 ou 1 paire de points formés d'un *ExceptionDataPoint* avec *ExceptionActionPoint* et 0 ou plusieurs *DataPoint*.
- Un *PeerPort* qui doit contenir au moins un *ServerActionPoint* et un *ClientActionPoint,* Pour chaque *ServerActionPoint,* peuvent correspondre une paire de points d'accès composé d'un *ExceptionDataPoint* et d'un *ExceptionActionPoint.*
- Un *DataPort* ne peut contient que des *DataPoint*.

Et voici la représentation graphique des ports (figure 3.3) :

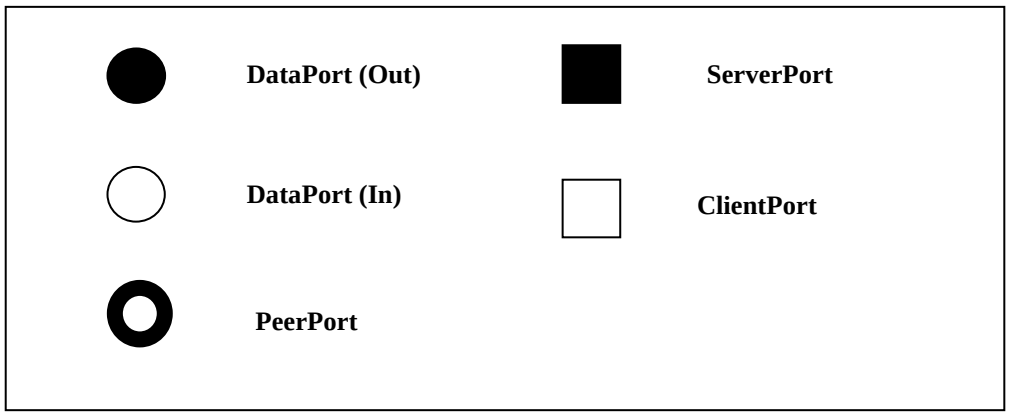

Figure 3.3 : Représentation graphique des types de ports.

3.3.2. Les ports contrôlés :

Les services d'un port peuvent être contrôlés. Le contrôle consiste à les autoriser à fonctionner *(Enable / Disable)*, les lancer ou les arrêter *(Start / Stop)* et les terminer (*Terminate*).

Les ports contrôlables sont du type :

- *ControledServerPort*,
- *ControledPeerPort*.

Un *ControledServerPort* est constitué d'un seul *ControledServerActionPoint*.
Un *ControledPeerPort* contient au moins un *ControledServerActionPoint* et au moins un *ClientActionPoint*.

En plus des *DataPoint*, un port contrôlé peut contenir une paire de points d'exception (*ExceptionDataPoint* et *ExceptionActionPoint*), un point *StateActionPoint* ou un *StateDataPoint* ou les deux.

3.3.3. Les ports d'états :

Ces ports reportent l'état global d'un composant. Pour l'instant, nous avons défini deux catégories d'états significatifs :

- Les états structurels,

- Les états exceptionnels.

- Les états structurels nous renseignent sur l'évolution structurelle d'un composite dans le temps.
- Les états exceptionnels nous renseignent sur des erreurs que les composants n'ont pas été en mesure de résoudre.

Un état structurel nous renseigne sur :

- Les composants formant l'architecture à un moment donné.
- Les connecteurs de la vue interne d'un composite.
- Les connecteurs de délégation.
- Les ports externes d'un composant.
- Les ports internes d'un composant.
- L'état global du composant vis à vis des services fournis et requis.

L'état structurel est renseigné par un *StateDataPort* ne devant regrouper que des *DataPoint* associés aux types représentant les états structurels définis.

3.3.4. Les ports d'états exceptionnels :

Les ports d'états exceptionnels concernent le fonctionnement global d'un composant. Un état exceptionnel de niveau composant peut avoir une influence directe sur tous les services du composant et sur l'existence du composant lui même.
Un certain nombre d'état exceptionnel de niveau composant sont prédéfinis dans l'environnement et d'autres peuvent être définies par l'architecte par sous typage, Les états exceptionnels de base se concentre sur le reports d'erreur suite à des manipulations structurelle au moment de l'exécution de l'architecture :

- Connexion inexistante lors d'une opération d'activation d'une action.
- Port non connecté : lors d'une opération d'activation d'une action ou de transmission de donnée.
- Composant inexistant lors d'une opération d'établissement d'une connexion.
- Port inexistant lors d'une opération d'établissement d'une connexion.

– Conflit de contexte d'action entre ports et connecteurs.

Deux types de ports d'exception sont définis.

- *ExceptionPort,*
- *ManagerExceptionPort,*

– Un *ExceptionPort* consiste en une ou plusieurs paires de points d'accès contenant chacune un *ExceptionDataPoint* et *ExceptionActionPoint*.

– Un *ManagerExceptionPort* consiste en une ou plusieurs paires de points d'accès contenant chacune *ExceptionDataPoint* et *ManagerExceptionActionPoint*.

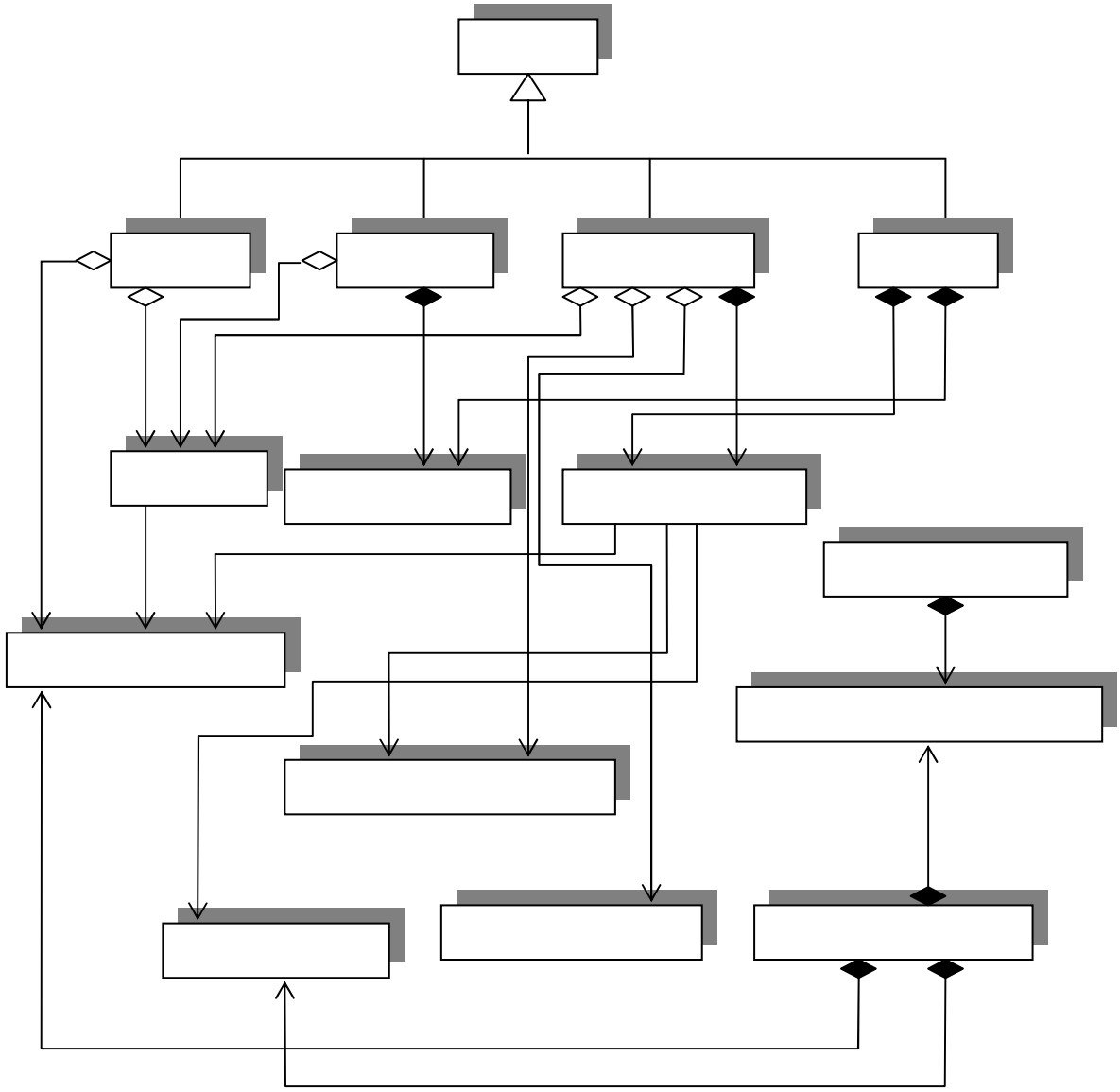

Figure 3.4 : Représentation des différents type de ports.

3.3.5. Connexion de ports :

Lors de l'élaboration de connexion, les points d'accès sont accessibles de manières individuelles au niveau des ports. Ils ne sont connectables que dans une seule topologie de base qui est un point à point. Les autres topologies ne seront possibles que par l'utilisation de composant de connexion comme nous le verrons par la suite.

3.3.6. Les ports prédéfinis :

Ce sont les ports orientés vers les supports de connexion bien définie tels qu'un protocole standard, un style de communication, une interaction avec l'environnement…etc. Ils existent en version ordinaire et en version contrôlée. A titre d'exemple :

- les types *FTPServerPort*, *FTPClientPort*, *CORBAClientPort* et *CORBAServerPort*, sont successivement destinés à des connexions dans le contexte du protocole FTP, ou dans le contexte de l'infrastructure Corba.
- les types *EventIntDataPort*, *EventServerPort*, *EventClientPort* sont destiné à supporter un style de communication par événement.
- le type *EnvServerActionPort* permet l'accès aux services de l'environnement conteneur ou un autre environnement. Le type *ContainerEnvServerPort* ne permet l'accès qu'au service de l'environnement conteneur. Le type *UnixEnvServerPort* permet de communiquer avec un environnement UNIX local ou distant alors que *UnixContainerEnvServerPort* ne permet de communiquer qu'avec le système *UNIX* local. De même le type *EJBEnvServerPort* donne l'accès à des composant évoluant dans un environnement d'exécution EJB alors que le port *EJBContainerEnvServerPort* ne donne l'accès qu'aux services des composant dans le même serveur d'application.

3.4. Les connecteurs :

Le modèle de connecteur que de l'approche IASA se base sue les principes fondamentaux suivant : [14] [15]

- ✓ La connectivité de base est établie entre deux points d'accès par un connecteur dit connecteur élémentaire.
- ✓ Deux point de connexion sont connecté soit dans une topologie point à point soit à travers une infrastructure de communication.

- ✓ Une infrastructure de communication est composée d'un ensemble de connecteurs point à point et de composant de communication.
- ✓ Les topologies autres que point à point sont obtenues par la mise en œuvre des composants de communication et de connecteurs point à point.
- ✓ Les connecteurs élémentaires doivent être manipulable par un ensemble d'opération permettant entre autres de les grouper dans un ou plusieurs connecteurs composé et de les isoler.

L'objectif est donc d'arriver à un modèle capable de prendre en charge les idées de connexion au niveau du modèle mental, de supporter la réalisation de connexion sans grande contrainte.

Les connecteurs et les ports sont mutuellement influençables. Lorsque une instance de connecteur de base relie deux points d'accès, son comportement et les opérations de manipulation sont contraintes par les attributs de communications présents sur les ports.

3.4.1. Les types fondamentaux :

Le modèle de connecteurs repose sur les types de base fondamentaux suivants : [14] [15]

- ➢ *AnyConnectorLine,*
- ➢ *AnyConnector,*
- ➢ *AnyComDevice.*

Le type *AnyConnectorLine* représente un connecteur point à point élémentaire. Contrairement à beaucoup d'ADLs, un connecteur ne possède pas de points terminaux représentant les rôles. Un connecteur représente plutôt un conduit ou canal à travers deux point d'accès. Cette même approche est suivie dans ArchJava, Fractal et Darwin.

Les attributs fondamentaux de ces types sont : son nom et les points qu'ils lient. Ce connecteur est particulier parce qu'une instance de connecteur puisse exister avant même de la lier aux points d'accès.
De plus le fait qu'un connecteur élémentaire relie deux points d'accès et qu'il est manipulable, la réalisation de connexion se fait de manière très flexible et sans contrainte particulière. C'est ainsi qu'il devient possible d'envoyer un point d'accès vers plusieurs autres points d'accès de ports différents.

Le type *AnyConnector* est un connecteur composé de 1 ou plusieurs connecteurs élémentaire. Chaque connecteur élémentaire est identifié (un nom ou un numéro d'ordre) et peut être accessible par des opérateurs faisant partie des composants de communication. Dans une opération de connexion faisant intervenir un connecteur de type *AnyConnector*, tous les connecteurs élémentaires sont utilisés de bout en bout.

Le type *AnyComDevice* représente la technique utilisée pour connecter plus d'un port de composant et pour réaliser certaines opérations spécifiques sur les connecteurs (Filtrage, Multiplexage, distribution ...). Une instance de ce type comprend un certain nombre de port de type A*nyPort*. Le port réel qui serait placé dépendra du type exact de composant de communication, comme nous le verrons par la suite dans la présentation de quelques composant de base.

3.4.2. Les connecteurs spécifiques :

Un certain nombre de connecteurs spécifique ont été défini, parmi ces ports on peut cité : [14] [15]

Le *ClientServerConnectorLine* est un connecteur qui doit lier deux points d'accès de type *ClientActionPoint* et l'autre de type *ServerActionPoint*. Le connecteur ne marquera son sens de fonctionnement que lorsque il est connecté correctement aux deux point d'accès. La connexion de deux composants sans spécification de port par un tel connecteur crée automatiquement un port *ClientPort* sur le premier composant et un *ServerPort* sur le deuxième composant. Chaque port étant muni d'un seul point d'accès.

Le *DataConnectorLine* est un connecteur qui doit lier deux points d'accès de type *ClientActionPoint*. Le sens est défini par le sens des deux points liés. Ces derniers doivent avoir un sens opposé ou ont tous les deux le sens «*InOut*».
Le *ClientServerConnector* lie un port *ClientPort* et *ServerPort*. Il doit disposer d'un et un seul *ClientServerConnectorLine*. Le nombre de *DataConnectorLine* doit être inférieure ou égale au nombre de *DataPoint* du port contenant le moins de *DataPoint.*
Le *PeerToPeerConnetor* contient au moins deux *ClientServerConnectorLine*. Le nombre de *DataConnectorLine* est inférieur ou égale au nombre de *DataConnectorLine* du port contenant le moins de *DataPoint*.

3.4.3. Les composants de connexion spécifique :

Un certain nombre de composants de connexion fondamentaux ont été élaboré. Nous présentons dans ce qui suit les plus usuels. Ces composants sont réactifs aux opérations de connexion et ne dispose au départ d'aucune instance de ports. Les composants de connexions utilisent les opérations réflexives sur les ports connecté pour reconnaître leur type avant de les instancier. Tous les composants de connexion existent sous forme ordinaire ou sous forme contrôlée (dispose d'un port de contrôle).

3.4.3.1. Les répéteurs ou distributeurs (*RepeaterComDevice*) :

Ils permettent de multiplier un même port. De ce fait un port peut être diriger vers plusieurs sorties qui chacun atteindra une cible particulière. Permet donc de réalise une connexion un vers plusieurs. Ce type de composant est utilisé par exemple pour lancer selon une politique définie dans un composant un même service présent sur un de ces ports vers plusieurs d'autres ports, ou pour envoyer des données à plusieurs destinations. Le port d'entrée du composant et les ports de sortie sont totalement compatibles. Le port d'entrée est soit un *DataPort* ou un *ServerPort* et les ports de sorties sont tous soit des *DataPort* si le port d'entrée est un *DataPort,* soit tous des *ClientPort* si le port d'entrée est un *ServerPort*. Tous les ports sont dotés du même nombre de point d'accès.
Lorsque il est utilisé avec une entrée *ServerPort*, le répéteur possède une logique proche du composant de raffinage d'actions, cependant son rôle est plus léger car la logique d'enclenchement des actions se trouve au niveau composant et non au niveau répéteur.

3.4.3.2. Les concentrateurs (*ConcentratorComDevice*) :

Permet à plusieurs clients un accès à un même service. Le composant est doté de plusieurs ports d'entrée de type *ServerPort* et d'un seul port de sortie *ClientPort*.

3.4.3.3. Les multiplexeurs (*MuxComDevice, DemuxComDevice*):

Un multiplexeur est composé de plusieurs entrées, une sortie et un sélecteur d'entrée. Les entrées et les sorties sont soit toutes des *DataPort* soit toutes des *ActionPort*. Le port de sélection peut être un *DataPort* ou un *ActionPort*. C'est à travers le port de sélection (par donnée ou par action) qu'un port d'entrée bien précis sera connecté à la sortie. Les données ou les actions des ports non sélectionnés seront bloquées ou temporisé ou ignoré selon la politique de fonctionnement associé au multiplexeur.

Quand au démultiplexeur il réalise l'opération inverse d'un multiplexeur en reportant l'entrée sélectionnée vers la sortie.

3.4.3.4. Les diffuseurs :

Les ports d'un diffuseur sont tous de type *PeerPort*. Une information une action ou donnée mise sur un port atteindra simultanément tous les ports. Tous les point d'accès *ServerActionPoint* et *DataPoint* en *In* et *InOut* forme une ressource critique dont l'accès est synchronisé. Si un port de composant dispose d'un des ports critiques, il dispose alors de tout le diffuseur jusqu'à ce qu'il le libère.

3.4.3.5. Les commutateurs :

Les ports d'un commutateur sont tous de type *PeerPort*, composé chacun de deux *ServerActionPoint*. Le deuxième *ServerActionPoint* offre le service de création / suppression dynamique de connexion. Les ports de composants connecté au commutateurs doivent être des *PeerPort* doté d'au moins deux *ClientActionPort*. Dans un commutateurs, les connecteurs sont crée et éliminée dynamiquement. A l'état repos, aucune connexion n'existe.

Lorsque un composant veut se connecter à un autre composant à travers le commutateur, le *PeerPort* du composant exprime à travers son deuxième *ClientActionPoint*, le souhait d'établir un lien avec un autre *PeerPort* connecté au commutateur. Si le *PeerPort* de destination est libre, une connexion est alors établie. A la fin de l'interaction entre les deux *PeerPort*, ces derniers libèrent le connecteur qui serra alors éliminé par le commutateur. Si le port de destination est engagé dans une connexion, la demande sera refusée et le composant devra prendre la décision adéquate (réessayer par exemple).

3.4.4. Les connecteurs prédéfinis :

Dans le processus de définition d'une architecture logicielle, l'architecte réalise les connexions point à point selon l'une des deux approches suivantes :

- Il exploite des connecteurs prédéfinis, ce qui est souvent le cas.
- Il crée des instances des divers types de connecteurs de base. Ces instances doivent être complété de manière explicite ou à partir des point de connexion qu'ils relient.

A travers les connecteurs prédéfinis, le système prend en charges les diverses techniques de connexion de base, les diverses protocoles de communication, les infrastructures de communication objets, et les connexions avec l'environnement d'évolution des composants.
Les connecteurs prédéfinis connectent les ports prédéfinis correspondants.

Exemple :

Le type *FtpConnerctor* représente le protocole *FTP* et doit relier un port de type *FtpClientPort* à un port *FtpServerPort*.

Le type *EJBEnvConnector* permet de se connecter à un environnement, EJB conteneur ou distant. Il doit utiliser un *EJBEnvClientPort* et un *EJBEnvServerPort*.

3.5. Les composants :

Le modèle de composant distingue entre deux grandes catégories de composants : [14] [15]

- *Les composants primitifs,*
- *Les composants composites.*

Les opérations d'élaboration d'architecture logicielle se concrétise par la réalisation d'un composant composite.
Le modèle de composant définit une organisation de la vue externe à laquelle doit adhérer n'importe quel composant (primitifs, composites, anciennes applications) et une organisation de la vue interne, applicable exclusivement aux composites.

La vue externe est représentée par le concept *d'enveloppe*. La vue interne est organisée en deux grandes partie : *la partie opérative et la partie contrôle*.

3.5.1. Les types fondamentaux :

Un composant [14] [15] est un type. Tous les types de composant sont des sous types de *AnyComponent* (figure 3.5). Ce type renseigne sur les éléments fondamentaux d'un composant à savoir le nom de type, le nom d'instance, une liste de ports et une liste d'enveloppe applicable.

Deux sous type ont été défini pour distinguer clairement entre composant primitif et composant composite. Ce sont les types *PrimitiveComponent* et *CompositeComponent*.

Le type *CompositeComponent* renseigne de plus sur l'organisation interne du composant en partie opérative et contrôle et maintient une liste de composant et une liste des connecteurs.

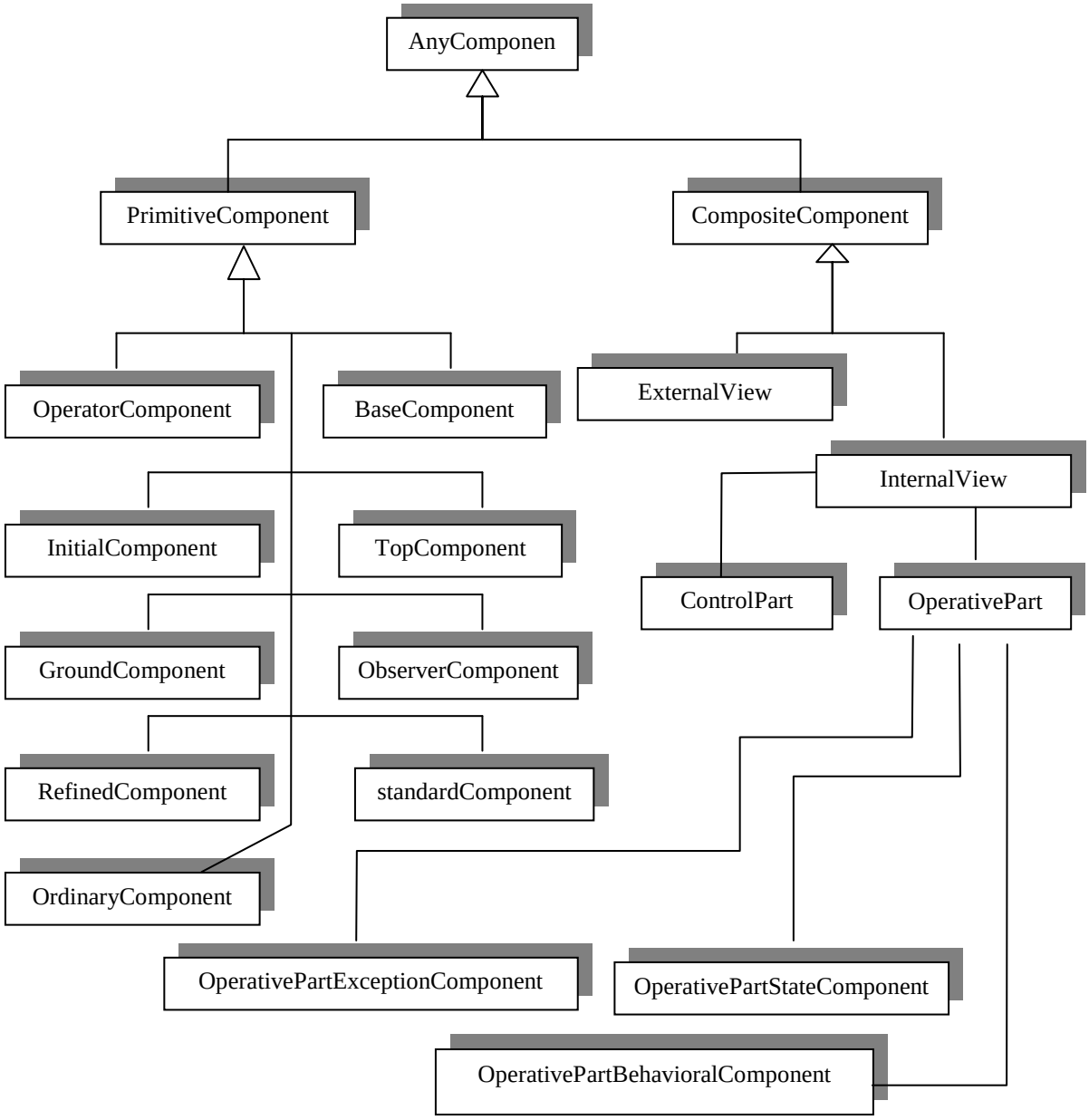

Figure 3.5 : représentation des différents type de composant.

Comme on a déjà précisé La vue externe est représentée par le concept *d'enveloppe*. La vue interne est organisée en deux grandes partie : *la partie opérative et la partie contrôle*.

3.5.2. Organisation de la vue externe : la notion d'enveloppe :

La vue externe de tout composant est formée d'une enveloppe munie d'un certain nombre d'adaptateurs (figure 3.6). L'objectif primordial de l'enveloppe est de permettre une isolation totale de la vue interne d'un composant du monde externe. [14][15]

Les ports considérés dans une opération d'assemblages sont les ports de l'enveloppe. Les ports de la vue interne seront rattachés aux ports de l'enveloppe par les adaptateurs de celle ci et qui seront entre les ports de la vue externe et ceux de la vue interne.

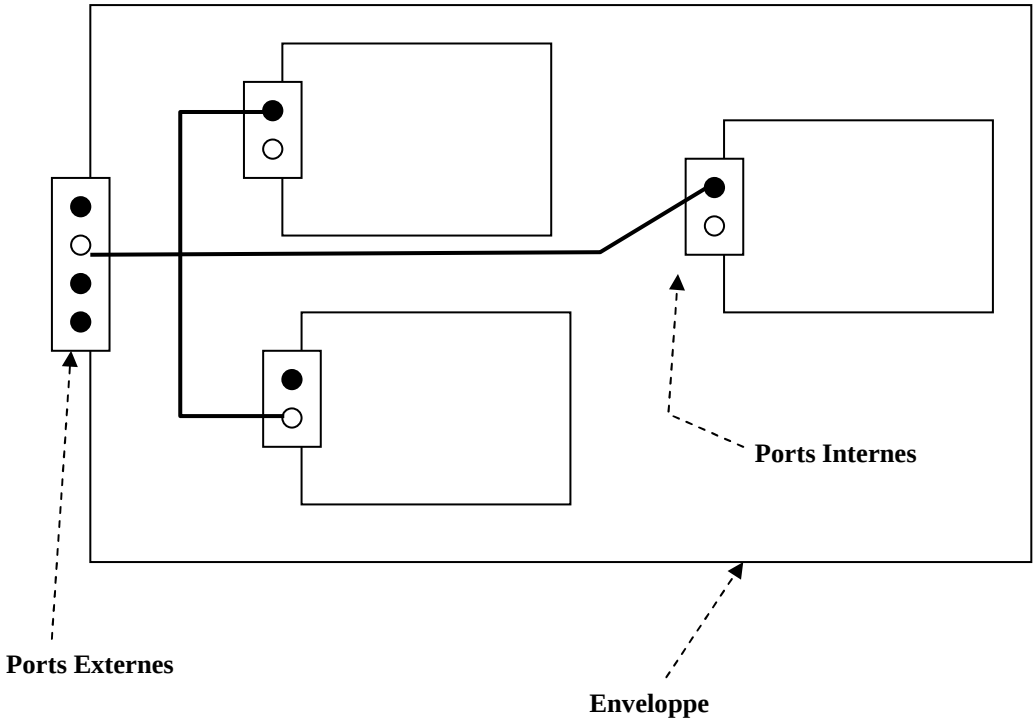

Figure 3.6 : La notion d'enveloppe.

Les enveloppes sont représentées par le type *AnyWrapper*. Ce dernier est doté d'un ensemble d'attributs, notamment l'instance de composant auquel une instance de *AnyWrapper* ou un de ses sous types est associé.

3.5.3. La Notion d'enveloppes Spécifiques:

La notion d'enveloppe a été introduite principalement pour prendre en charges les divers besoins lors de l'élaboration d'une architecture. Cette prise en charge est réalisée par un ensemble d'enveloppes particulières : [14] [15]

- *Les enveloppes à ports transparents* : Ces enveloppes, n'utilisent pas d'adaptateurs entre les ports internes et externes. Les ports externes sont une copie des ports internes. L'usage de ce type d'enveloppe par l'architecte représente un guide efficace dans l'opération de génération du code de

l'application, dans lequel l'opération de réduction de la redondance d'adaptateurs sera faite de manière efficace.

- *Les enveloppes marquées :* Dans une enveloppes marquées, qu'elle soit ordinaire ou transparente, tous les points d'accès peuvent être marqué. Dans une enveloppe totalement marquée tous les points d'accès sont marqués. Dans une enveloppe marqué partiellement vers l'extérieur seules les services requis *ClientActionPoint* et les *DataPoint* dans le sens est *In* ou *InOut* sont marqués. Dans une enveloppe marqué partiellement vers l'intérieur seules les services fournis *ServerActionPoint* et les *DataPoint* dans le sens est O*ut* ou *InOut* sont marqués. Ce type d'enveloppe est utilisé dans le processus de vérification de l'état global c'est à dire pour la vérification de la stabilité d'un type.
- *Les enveloppes de déploiement :* Ces enveloppes sont dotées de ce qu'on appelle un cas de déploiement qui maintient un plan de déploiement opérationnel. Elles prennent en charge la résolution des divers aspects liés au cas de déploiement correspondant à un composant, par la mise en œuvre d'un ensemble d'adaptateurs qui y seront attachés.

3.5.4. Notion d'application d'une enveloppe :

L'application d'une enveloppe à un composant permet d'atteindre l'objectif pour lequel l'enveloppe a été définie. Ainsi pour la génération effective d'une application c'est une enveloppe de déploiement doté d'un cas de déploiement qui sera appliquée. Pour la vérification de la stabilité d'une architecture, c'est l'enveloppe marquée qui sera appliquée.

3.5.5. Conséquence de la technique d'enveloppe :

En plus de ces objectifs premiers cités précédemment, la notion d'enveloppe permet d'atteindre d'autres objectifs aussi importants :

- Au niveau interne il devient possible de raisonner de manière indépendante et sans aucune contrainte, du monde externe. Grâce à la présence d'un ensemble d'adaptateurs au niveau enveloppe, le concepteur concentre plutôt son travail sur la détermination des ports les plus adéquat pour sa vue interne, sans se soucier de l'environnement de déploiement. La structure d'un port interne d'un composant n'est pas influencée par la structure de

ports des composants externes. A titre d'exemple, lors de la conception d'un port à travers lequel les actions correspondent aux actions d'un protocole standard disons HTTP, le concepteur pour des raison de facilité de conception ou efficacité, choisi pour faire correspondre à ce port une interface ordinaire composé de plusieurs méthodes. Cependant le port au niveau de l'enveloppe correspond réellement à un client du protocole et l'adaptateur doit transformer les appels de procédure en actions réelles du protocole (Utilisation d'un socket IP, Datagramme ...).

- La transformation de n'importe quel composant provenant de n'importe quelle source en un composant prêt à être exploiter dans les opérations d'assemblage selon notre approche.
- Il est possible d'associer à des instances d'un même type de composant des enveloppes différentes dans un même composite Ainsi selon la situation de composant, un composant portera l'habit le plus convenable.

3.5.6. Organisation de la vue interne :

La vue interne est organisée en deux parties.

- *Une partie opérative,*
- *Une partie contrôle*.

3.5.6.1. La partie opérative :

Elle contient les composants représentés par leur interconnexion, et la logique générale du composite à un moment bien précis. Elle est représentée par un sous type du type *OperativePart*.

Parmi les éléments important de cette partie opérative nous retrouvons une liste de ports qui seront par la suite lié à l'enveloppe par des connecteurs de délégation. Les instances de ports de cette liste sont les mêmes instances de ports présents sur les composants de la partie opérative.

Les instances de composants et connecteurs sont soit statiques soit dynamiques :

- Une instance statique est définie à la spécification. Elle est toujours présente dans la partie opérative et ne peut être supprimée.
- Une instance dynamique peut disparaître et apparaître durant tout le cycle de vie de l'instance de son composite. Elle peut ainsi être créée, ou être supprimée.

la figure 3.7 montre un exemple de schéma générale d'un composant composite :

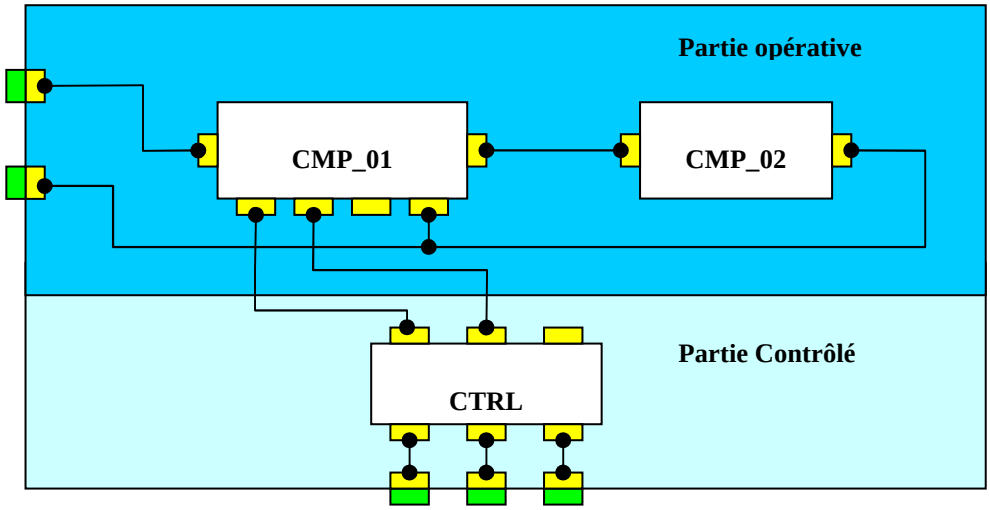

Figure 3.7 : Schéma générale d'un composant composite.

3.5.6.2. La partie contrôle :

La partie contrôle réalise les diverses opérations sur la partie opérative. Parmi ces opérations nous citons la gestion du fonctionnement des divers services (arrêt, lancement en séquence ou en parallèle) et le contrôle de l'évolution structurelle. La partie contrôle est représentée par le type *ControlPart*.

Les opérations de mise en œuvre dans le temps des composants de la partie opérative ainsi que la structuration dynamique de la partie opérative représente le comportement de cette dernière. La mise en œuvre de ce comportement est assuré par un composant comportemental de type *BehavioralComponent*.

Malgré qu'un *ControlPart* est conçu pour contenir n'importe quel type de composant, et de ce fait permettrait de supporter des comportements très complexes, au niveau pratique nous avons imposé la contrainte suivante :

Un *ControlPart* ne peut être composé que d'une seule instances des types de composant suivant :

- *BehavioralComponent,*
- *OperativePartStateComponent,*
- *OPerativePartExceptionComponent.*

L'instance du *BehavioralComponent* est obligatoire alors que les instances des deux autres types sont optionnelles.

Un *BehavioralComponent* est un type primitif dont la réalisation est faite complètement dans le langage d'action à réaliser. (Dans notre cas c'est le langage SEAL qui sera définit par la suite)

Le *BehavioralComponent* peut avoir un ou plusieurs comportements. Ces derniers peuvent être définis à l'intérieur ou importé de l'extérieur.

Le *BehavioralComponent* dispose de plusieurs ports par défaut permettant de sélectionner un comportement interne ou de charger un comportement externe.

Par cette approche il devient de spécifier pour une même architecture de base divers comportements, chacun fixant à sa manière l'évolution de la partie opérative. Cette tendance pourrait ouvrir la voie vers des architectures basé sur une catégorie bien précise de composant et un nombre d'instance bien limité (du à un nombre de licences par exemple) et dont un fonctionnement bien précis sera déterminé à partir du plan d'interconnexion qui sera réalisé dynamiquement.

A Chaque instanciation ou suppression de composant composite dans une partie opérative le *BehavioralComponent* de l'instance crée ou supprimé reporte cette information (nom

d'instance, nom de type) au *BehavioralComponent* du composite dans lequel il a été instancié ou supprimé.

3.5.7. Les composants spécifique :

Ils sont au nombre de trois et sont utilisé dans la partie contrôlée :

- *OperativePartBehavioralComponent,*
- *OperativePartStateComponent,*
- *OPerativePartExceptionComponent.*

Nous avons présenté précédemment les éléments essentiels du premier, et dans cette section nous présenterons les deux autres :

3.5.7.1. Le composant d'états (OperativePartStateComponent) :

Le composant d'état est un composant prédéfinis, paramétrable, pouvant être utilisé dans la partie contrôle. Il sert à reporter les divers états de la partie opérative. Il maintient diverses listes et des attributs renseignant sur les divers états qu'il met à dispositions à travers un ensemble de DataPoint organisé en un seul port. A chaque opération d'ajout / suppression de composant ou connexion, le composant d'état est informé par le BehavioralComponent de la partie contrôle

En plus des états structurels, le composant reporte les états globaux d'un composant. Nous avons définis deux grands états globaux: indiquant si ce dernier est stable ou non. Un composant stable est un composant pour lequel tous les points d'accès sont marqués. L'état global est en fait un et logique de tous les états globaux des composants de la partie opérative.

Une instance de composant OperativePartStateComponent peut être active dans l'environnent ou passive. Une instance active crée pour chaque nouveau composant incorporé dans la partie opérative un point DataPoint qu'elle relie au port d'état global du nouveau composant instancié. Si le composant est éliminé, la connexion et le DataPoint

associé sont éliminés. En mode passif, les opérations citées sont prises en charge par l'architecte.

3.5.7.2. Le composant OPerativePartExceptionComponent :

Ce composant purement comportemental a pour rôle essentiel de rediriger une exception apparue sur un port vers un seul port (une sorte de multiplexeur). Ce dernier représente le port Exception du composant qui devrait par la suite être relié à l'enveloppe par un connecteur de délégation.

3.5.8. Les états globaux des types de composants :

Un type de composant peut être dans un état stable ou instable. Un Composant stable peut par la suite être opérationnel ou non. Le type *AnyComponent*, *PrimitiveComponent* et *CompositeComponent* sont par définition des composants stable.

Un composant est dit stable (Sinon il est instable) si lorsque nous lui appliquons une enveloppe marquée (les points d'accès externes sont définis), alors tous les points d'accès seront marqués. La vérification de stabilité ne prend pas en considération les *DataPoint* dont le sens de transfert de données est *Out*. Ce type de *DataPoint* n'est considéré que pour la vérifier si un type est *totalement stable*.
Un *composant opérationnel* est un composant stable doté d'un plan de déploiement opérationnel.

CHAPITRE 4
SEAL : UN LANGAGAE D'ACTION POUR LA SPECIFICATION DE LA DYNAMIQUE DANS UNE ARCHITECTURE LOGICIELLE

4.1. Introduction :

Dans les architectures dynamique ou évolutive la topologie peut changer durant l'exécution, suite à des opérations sur les éléments d'une architecture tels que l'ajout, changement, suppression de composants de ports et de connecteurs.

Les architectures dynamiques permettent de planifier à la conception leurs changements topologiques à l'exécution. Cet aspect important n'a cependant pas attiré une grande attention dans les divers travaux de recherche. Les architectures évolutives, sont nécessaires pour que le système soit capable de gérer des défaillances et de basculer sur une architecture plus sûre, ils sont nécessaires pour permettre à une application de s'adapter avec un environnement sans cesse en évolution et d'être dynamiquement mise à jour et administrable.

L'évolution d'une architecture, n'a pas besoin d'être connue toujours avant la mise en exécution de l'application correspondante. En effet les mécanismes supportant l'évolution permettent de décomposer une architecture en exécution, de remplacer, de créer, de supprimer, d'activer ou désactiver des composants ou connecteurs, et de la recomposer. Ces mécanismes permettent de supporter les évolutions du système, sans arrêter l'exécution de l'application correspondante.

Le but de ce chapitre est de décrire un Langage d'action permettant notamment de modéliser la spécification et la validation du comportement d'une architecture logicielle.

4.2. Présentation de langage « SEAL » :

La spécification des comportements c'est à dire les interactions et le comportement de la partie opérative dans un composant composite se fait dans un langage d'action que nous avons appelé SEAL (Simple and Extensible Action Langage). SEAL est doté d'une grande puissance d'expression due principalement à la notion de contexte d'actions

(*ActionContext*). La version actuelle de l'interpréteur de SEAL ne s'intéresse qu'à l'exécution d'un composant composite, plus exactement du composant *BehavioralComponent* dans le but de montrer la validité des opérations de transformation architecturale comme l'ajout ou la suppression des composants et de connexions. [14][15]

Ce langage à été définit dans le but de montrer la validité des opérations de transformations architecturales (évolution structurelle d'une architecture logicielle). Il repose sur des contextes d'actions fondamentaux et sur tous les types prédéfinis de l'approche IASA (composants, ports, point d'accès, connecteurs, exception etc..)

Ce langage a été défini pour répondre à plusieurs attentes. Il permet ainsi : [14][15]

- La spécification des comportements observables sur les ports.
- La spécification des interactions (comportement observable sur un connecteur).
- La spécification du comportement global de la partie opérative d'un composant. .
- La définition de composants primitifs (composant comportementaux) indépendant des langages d'implémentation.

La spécification des actions décrivant les aspect comportementaux d'une architecture apparaissent de manière claire dans les clauses behavior) et actioncontext) d'une description SEAL Celle ci est structurée en une hiérarchie de clauses, dont le premier niveau est illustré dans la figure suivante : [14][15]

```
package    nomDePackage;
import     listeDePackage // package de composant et de connecteurs
component nomDuTypêDeComposant {
        // Définition de types internes. Pour être instancié dans d'autres composants,
        // Un type doit être définis à l'extérieur du composant, soit dans un même fichier
        // Ou dans un fichier différent.
port {   // Définition de types internes de port     }
connector
     {     // Définition de types internes de connecteurs.     }
component
    {     // Définition de types internes de composant.     }
// Fin de la définition des types internes
//   Définition des instances
ports
   {    // Définition des ports du composant   }
operativepart
  {  // Description de la partie opérative:
     //  Instance de composant, type internes de composant, et connexions}
controlpart
  { // Description de la partie contrôle: instance de composant et connexion inter –
    //  Composant De la partie contrôle et avec les composants de la partie opérative.
  }
behavior
{   // Comportement du composant; Ce comportement représente celui du composant
   //  OpPartController de la partie control
   // Le comportement est décrit soit en SEAL,  transformable directement en
  // Langage de programmation ou en java ou c'est une référence à un fichier Java
}
properties
 { // Spécification des propriétés non fonctionnelles. Pour l'instant,
   // Seule la Description des informations de déploiements est supportée
}
} // Fin description type de composant
```

Figure 4.1 : Description textuelle de l'architecture. [14][15]

Les autres clauses où apparaissent les descriptions SEAL sont les clauses internes de la clause port et les clauses décrivant les connecteurs au niveau des clauses ***operativepart*** et ***controlpart***. Ces clauses seront discutées plus en détail par la suite.

4.2.1. Notion d'action :

Les actions [14][15] sont les entités comportementales de base qui échangent des flots de contrôle et des flots de données à travers des points d'entrée de données et sortie de données. Une action est identifiée par un nom qui est toujours associé à un contexte de

validité de l'action. Elle peut correspondre en pratique à une activité très simple ou une activité très complexe.

4.2.2. Les types d'actions :

Les actions définies dans un contexte peuvent être de l'un des types d'action suivants : [14][15]

- Les actions primitives
- Les actions abstraites
- Les actions composées

4.2.2.1. Les actions primitives :

Une action primitive doit obligatoirement correspondre à une implémentation. Elle n'est pas décomposable même si en réalité elle correspond à une activité très importante. Nous retrouvons ce type d'actions au niveau de contexte attaché aux ports de composants primitifs, de composants finaux et au niveau de certains connecteurs et ports prédéfinis.

4.2.2.2. Les actions abstraites :

Une action abstraite est une action composée d'actions appartenant à une autre instance de contexte. Une action abstraite peut être vide. Cette dernière peut être inconnu lors de la création de l'action abstraite.

4.2.2.3. Les actions composées :

Une action composée est définie à partir des actions de la même instance de contexte. Elle représente une interaction ou une partie d'interaction. Elle peut être composée d'actions primitives, d'autres actions composée ou d'actions abstraites.

Le langage repose sur un ensemble de contextes dit fondamentaux et sur tous les types prédéfinis de l'approche intégrée relatifs aux composants, ports, point d'accès et connecteurs.

4.2.3. Notion de contextes d'actions :

Un contexte regroupe [14][15] un ensemble d'actions valables pour un domaine d'action particulier. Cette notion a été introduite pour fixer le domaine de raisonnement d'un architecte et lui fournir juste les outils suffisant pour exprimer ses concepts dans des situations bien précise.

Une action est toujours définie dans un contexte d'action. Un contexte d'action est un espace de nom. Le nom d'une action est relatif à son contexte. A titre d'exemple, la spécification d'une interaction ne nécessite pas tous le jargon du langage d'action (opérations d'accès et manipulation d'instances de composants et ports etc..), ni les actions de contrôle dynamique de la structure d'un composant (ajout, suppression de port, de composant, de connecteurs etc..). De plus dans une interaction, le vocabulaire sera contraint par les ports interconnectés.

Un contexte d'action peut contenir des actions primitives, des actions abstraites, des actions composées. Il peut être manipulé par des opérations d'ajout, suppression ou modification d'action.

4.2.4. Les contextes d'action dans le langage « SEAL » :

On trouve deux types de contextes : [14][15]

- Les contextes fondamentaux,
- Les contextes de supports.

4.2.4.1. Les contextes fondamentaux :

Les actions propres du langage sont organisées dans trois contextes de base prédéfinis (figure 4.2) :

- Le contexte racine *RootActionContext*,
- le contexte de contrôle *ControlActionContext*,
- Le contexte des interactions *InteractionActionContext*.

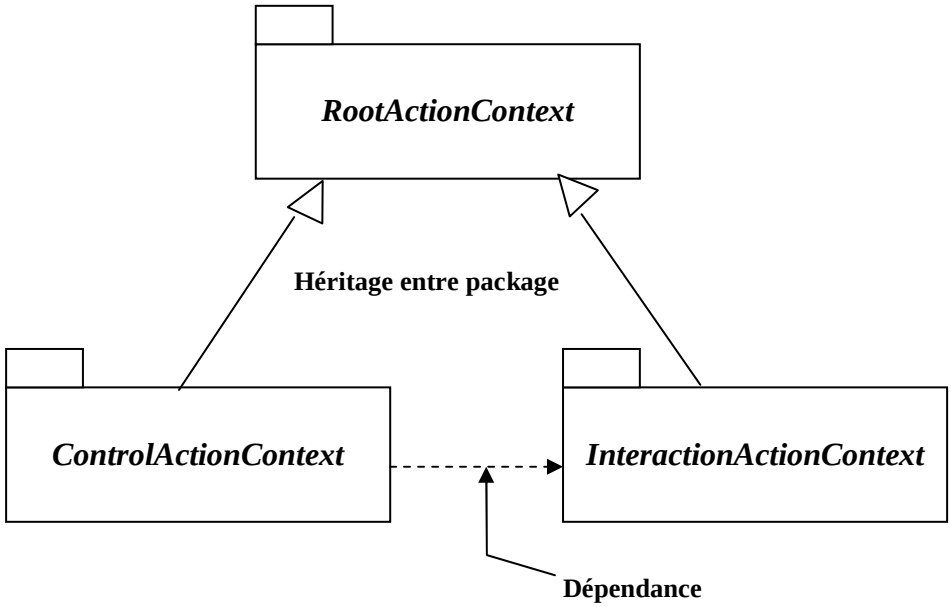

Figure 4.2 : les contextes fondamentaux.

Le RootActionContext :

Le *RootActionContext* contient les action fondamentales représentée par :

- Les opérateurs usuel permettant de manipuler les différentes instances des divers types de bases (accès / modification des états et action dans un port),
- Les structure de contrôle conditionnelle et répétitive (*if then Else*, *while*),
- Les opérations de manipulation de contexte,
- La gestion des exceptions.

A titre d'exemple, les opérations d'attachement de contexte aux connecteurs et ports, la manipulation des instances (ajout d'action à un port, changement d'attributs d'une instance etc..) sont défini au niveau du *RootActionContext*. Tout contexte défini est soit une extension ou une restriction directe ou indirecte du contexte de base *RootActionContext*.

L'InteractionActionContext :

C'est un sous ensemble du *RootActionContext*, et dans lequel nous ne pouvons trouver que les actions permettant de spécifier des interactions de bases. Dans l'état actuel,

InteractionActionContext ne supporte que deux opérateur : l'opérateur de spécification séquentielle d'action et l'opérateur de spécification parallèle d'action.

Les actions d'envoi et réception de donnée (*send, receive*), les actions de demande et lancement d'actions (*request, invoke*) et les opérations de modification d'attributs de ports et connecteurs.

Le ControlActionContext :

Il est destiné à décrire les comportements de la partie opérative, et plus particulièrement le contrôle de sa structure durant l'exécution. Le *ControlActionContext* utilise toutes les capacités de *RootActionContext* et y ajoute des actions de contrôle des service (*EnableService, DisableService, StartService, StopService, KillService, CreateService*), les contrôle globaux de composant *(EnableComponent, DisableComponent)* et le contrôle de la structure dynamique des composant.

Parmi les actions de manipulation dynamique de la structure d'un composant, nous trouvons notamment les opérations de création d'instances (*CreateComponent, CreatePort, CreateClientActionPoint, CreateServerActionPoint, CreateDataActionPoint, CreateConnector, etc.*.) et de suppression d'instances (*DeleteComponent, DeleteConnector, DeleteAccesPoint*).

La figure 4.3 donne un exemple d'actions décrivant le comportement du composant *OpPartController*.

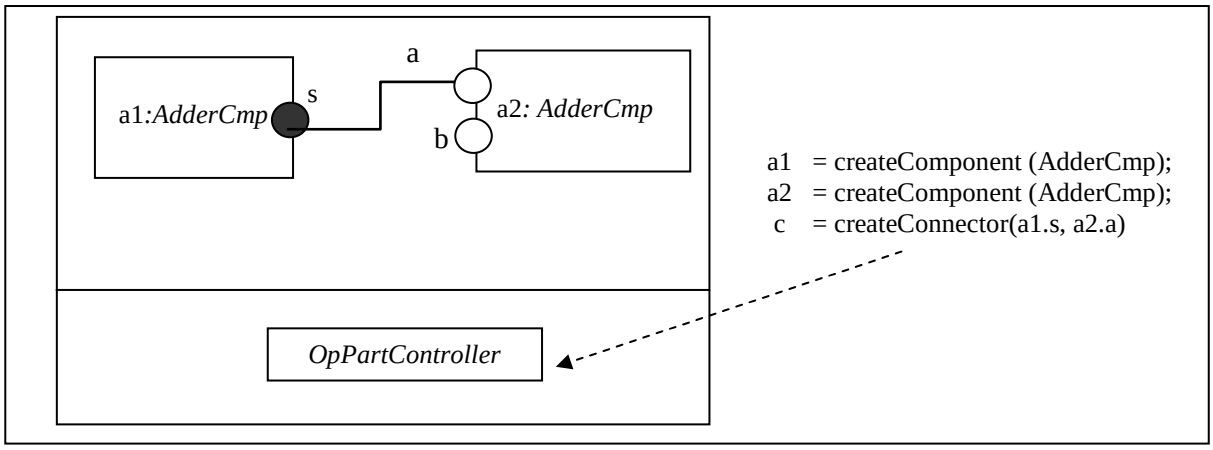

Figure 4.3 : Exemple d'actions décrivant le comportement du composant OpPartController.

4.2.4.2. Les contextes de supports :

4.2.4.2.1. Les contextes systèmes :

La notion de contexte permet d'adapter le langage à divers environnements d'exploitation et aux besoins propres des architectes. Le langage peut supporter un nombre très important de contexte et est de base muni de plusieurs contextes standard de spécification d'interaction.

Ceux ci font partie du *SystemIntercationActionContext*. Nous y retrouvons des contextes qui regroupent les actions des protocoles standards de communication comme le FTP ou HTTP, des environnements de programmation comme UNIX ...etc. Donc une commande du protocole FTP, et une commande UNIX Shell sont des actions présentes successivement dans les contextes *FtpIntercationActionContext* et *UnixIntercationAxtionContext*.

Un contexte système est ouvert pour un enrichissement par d'autres contextes ou un réajustement par suppression de contextes jugés obsolètes. Le contexte de support *SystemInteractionContext* représente l'endroit ou les *plug – in* pour SEAL seraient installés.

Dans l'approche actuelle, toutes les actions de ces contextes de supports doivent avoir une définition complète. Le contexte de support représente un premier mécanisme d'extension du langage d'action SEAL. [14][15]

4.2.4.2.2. Les contexte de projet :

Les *ProjectIntercationActionContext* sont des contextes élaborés par les architectes durant le processus de conception pour leurs besoins propres. [14][15]

La définition d'un nouveau type de contexte se fait en deux grandes étapes :

- L'extension ou la restriction d'un contexte de base. Ce dernier doit obligatoirement être un *InteractionActionContext*. Si aucun contexte de base n'est spécifié, le nouveau contexte étend le contexte *InteractionActionContext*. Dans l'état actuel, il n'est possible d'utiliser qu'un seul contexte de base. Le contexte nouvellement crée disposera de tout ou partie du vocabulaire de son contexte de base.
- L'enrichissement éventuel du nouveau contexte par de nouvelles actions primitives ; abstraites ou composée.

Dans un projet, la définition d'un contexte d'interaction se fait à partir d'un autre contexte d'interaction et un contexte de contrôle à partir d'un autre contexte de contrôle. La création de nouveaux contextes se fait soit de manière explicite à partir d'un autre contexte local ou appartenant à un autre projet soit de manière implicite à partir du contexte d'action correspondant dans le *RootActionContext*. Cette approche garantira que dans tout contexte il y'aura les mécanismes de base nécessaires à la spécification d'interactions ou de comportements de composants. Un contexte nouvellement crée dispose de tout ou partie du vocabulaire de son contexte de base.

Les contextes d'interactions de projet sont associés aux connecteurs et aux ports. Ils peuvent être manipulés par l'ajout, la suppression et la modification des actions primitives, abstraites ou composées. La modification concerne souvent l'aspect marquage, le nombre de points d'entrée et sortie de l'action et le type de ces points. L'ajout d'une nouvelle action à un contexte nécessite obligatoirement son marquage. La possibilité d'enrichissement de

contextes par de nouvelles actions, ouvre une nouvelle voie à travers laquelle, nous assurons l'extensibilité du langage d'action.

La suppression d'une action d'un contexte, alors que celle – ci est instanciée dans un port ou dans un composant comportemental, provoque une exception dans le processus de conception nécessitant un nouveau départ plus sain de ce dernier.

4.2.5. Instanciation de contextes au niveau connecteur :

Une instance de contexte est créée lors de l'établissement d'une connexion (création d'un connecteur) entre deux port ou l'instanciation d'un composant de communication. Les connecteurs ou composant de connexion et les ports qu'ils connectent partagent la même instance de contexte d'action. Des connecteurs différents ne peuvent pas partager une même instance de contexte même s'ils utilisent le même type de contexte. L'instanciation de contexte pour une connexion peut se faire par influence des ports de composants connectés. [14][15]

Lors de la construction d'une connexion, si le type de connecteur est associé à un contexte, et les ports ne sont associé à aucun contexte, alors le contexte du connecteur sera associé aux deux ports. Si un port dispose d'un contexte, et que l'autre port et le connecteur ne sont associés à aucun contexte, le contexte du port sera alors associé à l'autre port et au connecteur. [14][15]

Dans le cas ou l'une des composantes d'une connexion est associée à un type de contexte différent des autres composantes, la connexion ne pourra pas être établie. Au niveau dynamique, ce dernier cas se traduira par la génération de l'exception *ActionContextConflictException*. Les port de composant primitifs et finaux sont associés à des contextes d'actions constants, qu'il n'est pas possible de modifier.

4.2.6. Attachement de contexte d'action aux connecteurs :

Lors de l'établissement d'une nouvelle connexion, correspondant à la création d'un nouveau type de connecteur, un contexte d'action est alors crée pour ce connecteur. Ce contexte d'action est alors enrichi (ou rétrécie) et constituera une référence pour la définition des

contextes d'action associée aux ports (figure 4.4), la construction des ensembles d'actions au niveau des ports et la spécification des interactions. [14][15]

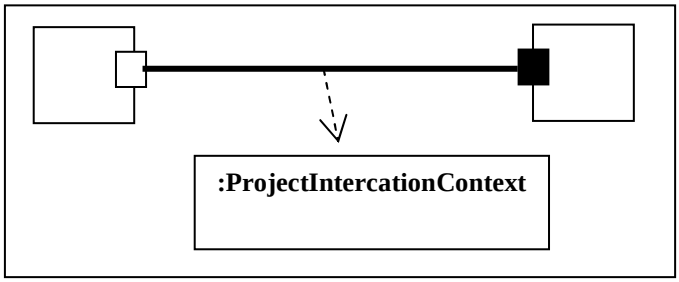

Figure 4.4 : Attachement de contexte d'action aux connecteurs.

Les ports connectés doivent posséder des ensembles d'actions compatibles avec le contexte associé au connecteur. La détermination de la compatibilité se base principalement sur le nom des actions, le nombre de point d'entrée sortie, le type des points d'entrée sortie et l'ordre des points d'entrée sortie. Dans le cas ou la connexion est établie entre instances de type de composants déjà définis (i.e. primitifs, opérationnel), les ports sont déjà associés à un contexte d'actions.

Dans une connexion, Il est fort probable que les contextes d'actions des ports et celui du connecteur utilisent pour une même action des noms différents. La mise en compatibilité des divers contextes d'actions est le premier travail après création du type de connecteur. Cette mise en compatibilité est assurée par le mécanisme d'alias (*ActionAlias*) supporté par les contextes d'actions. Le mécanisme d'alias indique de manière explicite la correspondance entre les noms et les points d'entrée sortie de l'action. La définition des alias peut se faire au niveau port ou au niveau connecteur. Au niveau port, la nouvelle action alias d'une ancienne action est introduite pour correspondre à une action du connecteur. Au niveau connecteur, pour réaliser des correspondances entre actions de ports et de connecteurs, il est nécessaire de définir pour une action, autant d'alias que de ports connectés. Les ports de composants primitifs et finaux sont associés à des contextes d'actions constants, qu'il n'est pas possible de modifier. [14][15]

4.2.7. Etat initiaux des actions et des ports :

Lors de l'association d'un type de contexte à un port les actions d'un *ServerActionPoint* sont marquées par les mêmes marques spécifiées dans le type de contexte. Si un

ClientActionPoint est associé à un contexte, toutes les actions au niveau de ce *ClientActionPoint* ne seront marquées que lorsque ce point d'accès sera relié directement à un *ServerActionPoint*. Dans ce cas le marquage est pris du *ServerActionPoint*. Si une connexion est supprimée le marquage des actions au niveau d'un *ClientActionPoint* revient à son état initial (non marqué). Ce marquage sera répercuté sur tous les *ServerActionPoint* qui dépendent du point d'accès client déconnecté. [14][15]

4.2.8. Etat des actions dans les instances de contexte :

Une action peut être marquée (donc complètement définie) ou non marquée. Dans un type de contexte les actions primitives et les actions composées formées uniquement d'actions primitives sont par défaut marquées. Les actions abstraites et les actions composée d'au moins une action abstraite sont par défaut non marquées.

Le marquage d'action est une opération menée sur les instances de contexte durant les opérations de création de connexion. Elles ne concernent que les actions non marquées (action abstraites et action composée non marquées).

Une action abstraite ne passera à l'état marqué que suite aux diverses opération de connexion et raffinement. Le développement en arbre d'une action abstraite peut montrer si celle ci peut être marquée ou non. Une action abstraite éligible à un marquage correspond à un arbre dont les feuilles sont des actions primitives dans leurs contextes de définition.
Une action composée non marquée est éligible à un marquage si toutes ses actions sont marquées. [14][15]

4.2.9. La grammaire de langage :

Il s'agit ici la grammaire utilisée pour le langage SEAL au format EBNF, les caractères sont entre « ' » et les mots clé entre « ″ », un « # » représente une option, un « * » représente une répétition, un « + » représente une répétition d'au moins une occurrence, un | sépare différentes possibilités.

Voici donc la grammaire de ce langage (figure 4.5) :

```
Programme : := (statement)*
statement : := CompoundStetement
              |déclaration
              | ″if ″  ‘(‘ expression  ‘)’ statement (″Else″ statement) #
              |″For″  ‘(‘ forint ‘ ;’ forcond ‘ ;’ foriter ‘)’ statement
              |″while″  ‘(‘ expression  ‘)’ statement
              |″Switch″  ‘(‘ variableNom  ‘)’ (caseExpression)+
                                  ‘(‘DefaultExpression’)’ endSwitch

CompoundStetement : := ‘{‘   (statement)*   ’}’

Expression : := conditionExpression
ConditionExpression : := RelationExp (( = | ! =) RelationExp)
RelationExp : := IDENT (< IDENT ) #
                 | IDENT (>  IDENT ) #
                 | IDENT (<=  IDENT ) #
                 | IDENT (>=  IDENT ) #

declaration : := Type variableNom

Type : :=  identifieur
          |BaseType

baseType : :=  void
              |boolean
              |int
              |float
              |string
              |double
              |long

identifieur : := IDENT
```

IDENT : := Alpha(Alpha | Digit)*

CaseExxpression : := ″Case″ ‘(‘ value ‘)’ ‘ : ’ (GroupeAction)

DefaultExpression : := ″default″ : (GroupeAction)

Digit : :=’0’….’9’

Alpha: :=’a’….’z’ | ’A’…..’Z’

Num_int : :=(Digit)+

Num_float : :=(Digit)+(‘.’(Digit)+)

String = : :=’’’ Alpha(Alpha | Digit)* ’’’

VariableNom : := variableDeclar (‘,’ varaibleDeclar)*

VariableDeclar : := IDENT varInit

varInit : := (‘=’ Num_int | Num_float | String | true | false)

Figure 4.5 : La grammaire de langage.

4.2.10. Quelques constructeur du langage d’action :

4.2.10.1. Action de création des composants :

En UML2.0, une action de création d’un objet instancie un *classifier* qui ne peut pas être abstrait. L’instance alors est mise en sortie de l’action (dans un *OutputPin*). Il n’y a pas de point de variation sémantique.

Le sens donné dans notre langage à cette action est presque le même, La notation utilisée est définie par les règles suivantes :

CreateComponent ::= Instance = CreateComponent (NomComposan) ;

NomComposant ::= Identifier

Instance ::= Identifier

L’exemple suivant (figure 4.6) donne la représentation graphique et textuelle équivalente de cette action.

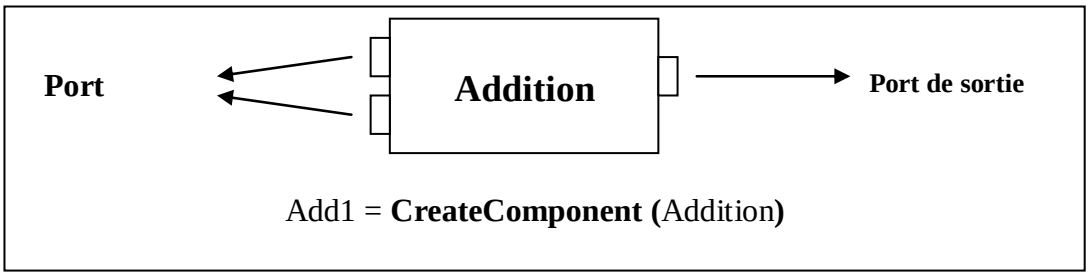

Figure 4.6 : Action de création des composants.

5.2.10.2. Action de suppression des composants :

DeleteComponent(NomComposant) ; NomComposant ::= Identifier

Exemple

La figure 4.7 montre l'état de l'architecture avant l'application de l'action de suppression des composants.

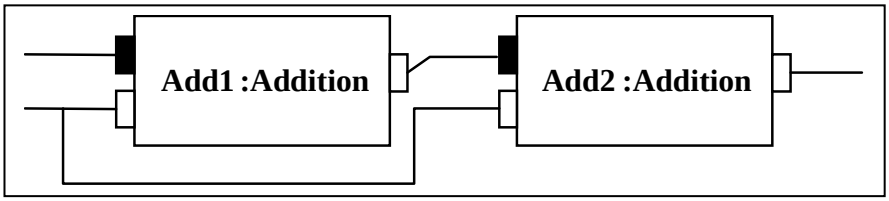

Figure 4.7 : Architecture avant l'application de l'action de suppression des composants.

Après l'excusions de *DeleteComponent (Add2) ;*

On aura le schéma suivant (figure 4.8) :

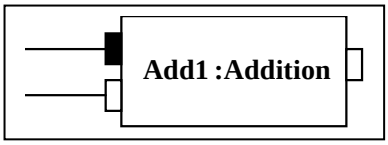

Figure 4.8 : Architecture après l'application de l'action de suppression des composants.

4.2.10.3. Action de création d'un connecteur entre deux port :

Syntaxe :

NomConnecteur = *CreateConnecteur*(NomComposant.NomPort,NomComposant.NomPort)

Exemple :

La figure 4.9 montre l'architecture avant l'application de l'action de création d'un connecteur entre deux ports.

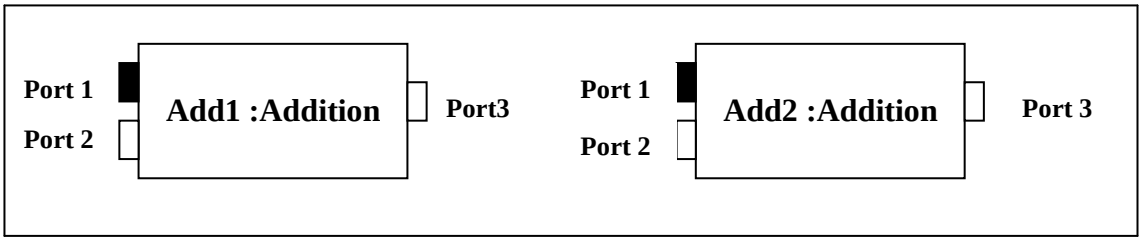

Figure 4.9 : Architecture avant l'application de l'action de création d'un connecteur entre deux ports.

Après exécution de l'action :

Connecteur1 = *CreateConnecteur*(Add1.port3,Add2.port1) ;

On aura le résultat suivant (figure 4.10) :

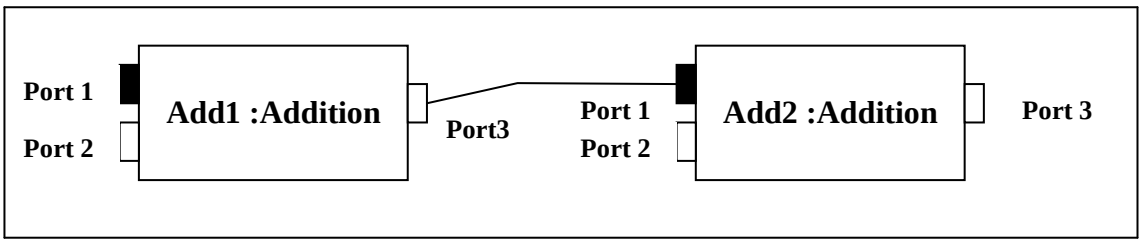

Figure 4.10 : Architecture après l'application de l'action de création d'un connecteur entre deux ports.

4.2.10.4. Action de suppression d'un connecteur entre deux ports :

De même de la création on fait la suppression de la manière suivante :

DeleteConnecteur(NomConnecteur) ;

4.2.10.5. Actions d'activation / désactivation des composants:

Pour l'activation d'un composant il suffit de spécifié son nom :

EnableComponent ::= EnableComponent NomComposant ; NomComposant ::= Identifier

De même pour la désactivation :

DisableComponent ::= DisableComponen NomComposant ; NomComposant ::= Identifier

4.2.10.6. Actions appliquées sur les services :

Ce sont des actions permettant le contrôle des différentes services, parmi ces action on peut cité :

- EnableService() : Activation d'un service.
- DisableService() : Désactivation d'un service.
- StartService() : Démarrage d'un service.
- StopService() : Arrêt d'un service.
- KillService() : Tué un service.

4.2.10.7. Actions appliquées aux Ports :

Ce sont des actions permettant le contrôle des ports, parmi ces action on peut cité :

- EnablePort() : Activation d'un port.
- DisablePort() : Désactivation d'un port.
- StartPort () : Démarrage d'un port.

- StopPort() : Arrêt d'un port.

4.2.10.8. Actions appliquées sur les actions au niveau des points d'accès :

NomComposant.NomPort.NomPoint.Action = NomAction;

Exemple:

Add1.Port1.Point1.Action = 'Action1' ;

4.2.10.9. Actions de chargement de comportement :

loadBehavior ::= loadBehavior (Num_Port);

Où Num_Port est le numéro de port concerné par le chargement.

4.2.10.9. Actions d'activation de comportement au niveau de port :

activateBehavior ::= activateBehavior(behaviorId) ;

Où behaviorId est l'identifiant de béhavior (comportement).

4.2.11. La spécification de la dynamique par le langage d'action :

Comme on a déjà indiqué, La description des actions SEAL, apparaissent de manière claire dans les clauses décrivant les comportements et les contextes d'actions d'une description textuelle qui sera bien détaillé dans la section qui suit, celle ci est structurée en une hiérarchie de clauses.

4.2.12. Présentation de la spécification textuelle de l'architecture logicielle :

La spécification textuelle de l'architecture logicielle est structurée en une hiérarchie de clauses représentées comme suite :

4.2.12.1. La clause package :

Elle représente la structure générale de notre architecture et voici la spécification textuelle associé (figure 4.11) :

```
package    nomDePackage;
import      listeDePackage // package de composant et de connecteurs
component nomDuTypêDeComposant {
          // Définition de types internes. Pour être instancié dans d'autres  composants,
          // Un type doit être définis à l'extérieur du composant, soit dans un même fichier
          // Ou dans un fichier différent.
port { // Définition de types internes de port  }
connector
      {  // Définition de types internes de connecteurs.   }
component
    {  // Définition de types internes de composant.  }
// Fin de la définition des types internes
//   Définition des instances
ports
   { // Définition des ports du composant }
operativepart
  { // Description de la partie opérative:
    //  Instance de composant, type internes de composant, et connexions }
controlpart
 {
   // Description de la partie contrôle: instance de composant et connexion inter –  Composant
   // De la partie contrôle et avec les composants de la partie opérative. }
behavior
{
  // Comportement du composant; Ce comportement représente celui du composant
  //  OpPartController de la partie control
  // Le comportement est décrit soit en SEAL,  transformable directement en
 // Langage de programmation ou en java ou c'est une référence à un fichier Java
}
properties
 {// Spécification des propriétés non fonctionnelles. Pour l'instant,
```

```
  // Description des informations de déploiements
 }
} // Fin description type de composant
```

Figure 4.11 :description *de la* clause package.

Exemple :

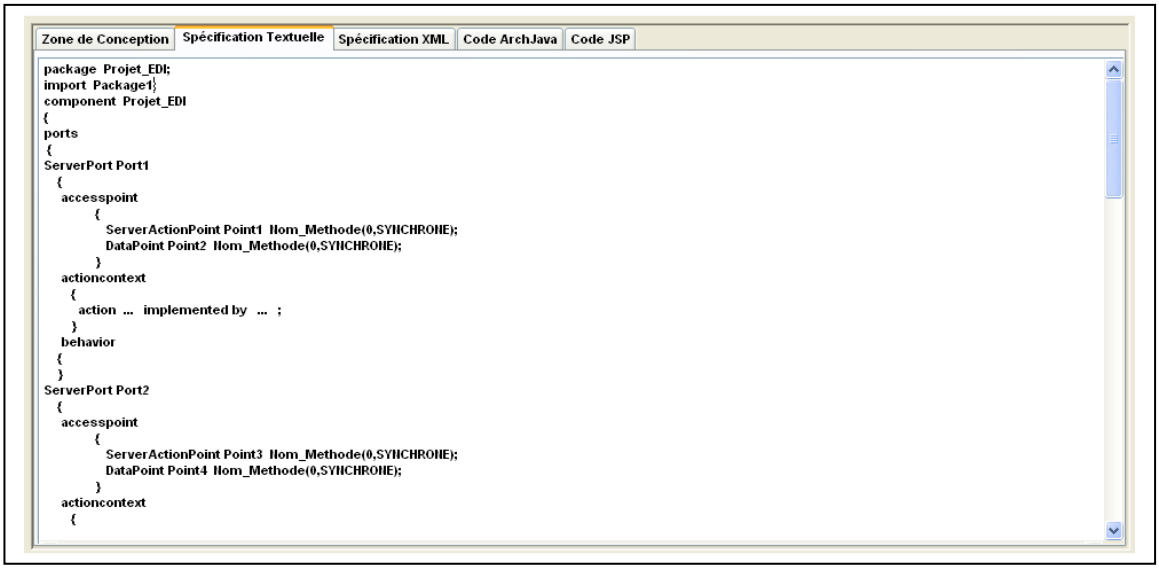

Figure 4.12 : La spécification textuelle de la clause package.

Et voici comment importer des packages (figure 4.13) :

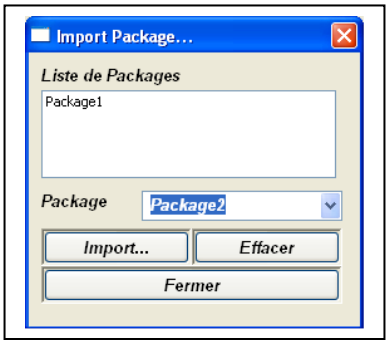

Figure 4.13 : Importation des packages.

4.2.12.2. La clause port :

Cette clause est spécifiable indépendamment dans un fichier ou à l'intérieur de la définition d'un composant et dans ce cas le type est considéré comme interne (figure 4.14). Dans les deux définitions il devrait être possible de spécifier un certain nombre de type de port dans

une clause port englobante ou directement sans nécessité de le mettre dans la clause port englobante. La clause englobante est introduite uniquement pour l'organisation du code. Et voici la spécification textuelle associé.

```
port { // Clause englobante:
    port nomDuType1 {
        accesspoint{}
        actioncontext{}
        behavior {}
        }
    port nomDuType2 {
        accesspoint{}
        actioncontext{}
        behavior {}
        }
} // Fin de la description des types de ports
```

Figure 4.14 : Spécification de la clause port.

4.2.12.3. La clause accesspoints :

Permet la spécification des points d'accès.

TypeDuPointD'accès nonInstance liste de paramètres

Exemples :

ServerActionPoint pMainAp (Temp, SYNCHRONE/ASYNCHRONE); // Max 1 par port
ClientActionPoint pMainAp (0, SYNCHRONE); // Max 1 par port
IntDataPoint pMainStatus (OUT, 0, SYNCHRONE) ; // 0n ou plusieur
StringDataPoint pin1 (in, 0, synchrone);
ComplexDataPoint pin2 (in, 0, synchrone);

4.2.12.4. La clause actioncontext :

la figure 4.15 montre la spécification de la clause actioncontext.

action *listeD'actions* implemented by *Signature d'une méthode;*
action *listeD'actions* ; // Aucune voie pour l'implementation n'est encore définie. Ce sont des actions abstraites

Signature d'une méthode = Instance_Datapoint nomMethodName (liste instanceDataPoint) | Instance_Datapoint nomMethodName

Figure 4.15 : Spécification de la clause actioncontext .

Exemple :

action fire implemented by pMainStatus fireMethod();
action *xAction* implemented by *pin1 xMethod(pin1, pin2);*

4.2.12.5. La clause behavior (pour les ports) :

Déclaration de variables locales primitives servant à créer des expressions logiques et déclaration du nom de toutes les règles utilisés, voici dans ce qui suit la déscription de la spécification textuelle associe (figure 4.16).

```
behavior
 {
        rules liste des règles;
        // Définition une par une de toutes règles déclarées
          rule nomDeLaRegle
            {
              precondition:  Expression logique
              pattern   suitesactionNormale; actioinencasdepanne;
              postcondition:  action de positionnement de variables ou de
                              Déclenchement d'autres actions souvent des exceptions
               failure: action a exécuter en cas de panne si l'action fail est citée dans la
                        Zone fail d'une règle.
            }
  }
```

Figure 4.16 : Spécification de la clause behavior.

Suite d'action normale : une suite d'action avec possibilité d'utiliser des paramètres ou non. Chaque action est séparée d'une autre par une virgule. La suite se termine par un point virgule. Souvent, la liste se termine par l'action success qui déclenchera en fait les postconditions. Si aucune action success n'est spécifiée en fin de liste d'action, aucune postcondition ne sera réalisée

Action en cas de panne : Ces actions seront réalisées si un problème apparaît dans le cours normal d'exécution d'une règle.

Lorsque l'action fail est utilisée, les actions au niveau de la section failure seront exécutées avant l'action qui suit fail.

Exemple1:

```
behavior{
    rules fireRule;
    rule  fireRule {
    precondition: ; // Aucune précondition nécessaire
    pattern: fire,success;
    postcondition:;
  }
```

Exemple 2 :

```
FTPClientPort  pFtp
{
accesspoint
        {   ClientActionPoint cFtpAp (0, SYNCHRONE);
            StringDataPoint    pFtpReplies (IN, 0, SYNCHRONE);
         }
actioncontext
        {use system.FTPIntercationContext }
behaviour
{
    boolean ftpConnexionSet = false;
    rules    successOpen, errorOpen, closeConnexion, getTicketFile,
             ftpOpenException, ftpReset;
     rule   successOpen {
              precondition: !ftpConnexionSet;
              pattern: open(hostname, username, password),
              receive(FTP_OPEN_OK),success; fail ftpOpenException;
              postcondition: ftpConnexionSet = true;       }
     rule  errorOpen
          {
           precondition: !ftpConnexionSet;
```

```
        pattern: open(hostname, username, password),
        receive(FTP_OPEN_ERROR),success; fail ftpOpenException;
         postcondition:;  }
 rule  closeConnexion
     {
        precondition: ftpConnexionSet;
        pattern: close, receive(FTP_CLOSE_OK),success; fail ftpError;
        postcondition: ftpConnexionSet = false; }
rule  getTicketFile
    {
        precondition: ftpConnexionSet;
         pattern: rename(OLD_NAME, NEW_NAME), get(NEW_NAME),
        delete(NEW_NAME), success; fail ftpError;
         postcondition: ;  }
 rule  ftpReset
   {
        precondition:;
         pattern: close, success;
         postcondition: ftpConnexionSet = false;}
```

4.2.12.6. La clause ports (instanciation de type de port) :

Cette clause permet d'instancier les divers ports et leur ajouter si nécessaires des aspects spécifiques. Cependant ces instances de port ne constituent pas de nouveau types.

ports {instanciation de ports}

Instanciation de port :

TypeDePort nomDuPort {description supplémentaire du port}

Description supplémentaire de port:

En général les actions supplémentaires ne sont nécessaires que lorsque le port est défini à ce niveau.

accesspoint{ point d'accès supplémentaires} actioncontext{action supplémentaire} behavior {comportement supplémentaire}

Exemple :

```
ports {
        MainCmpPort   pMain {
       // Clauses accesspoint, actionContext et behavior supplémentaire.
       accesspoint
        { ServerActionPoint pMainAp (0, SYNCHRONE);
          IntDataPoint  pMainStatus    (OUT, 0, SYNCHRONE); }
       actioncontext
       {action fire implemented by  pMainStatus fireMethod();}
       behaviour
        {
              rules fireRule;
                            rule  fireRule {
                            precondition: ; // Aucune précondition nécessaire
                            pattern: fire,reply,success;
                            postcondition:;
                            } // Fin fireRule
       }
       } // Fin description de pmain
…      …..  ……
}// Fin description des ports
```

Exemple :

La figure 4.17 représente un exemple de la clause ports, et la figure 4.18 montre un exemple sur les règles.

```
ports
{
ServerPort Port1
  {
   accesspoint
         {
           ServerActionPoint Point1  Nom_Methode(0,SYNCHRONE);
           DataPoint Point2  Nom_Methode(0,SYNCHRONE);
         }
   actioncontext
    {
     action ...  implemented by  ... ;
    }
   behavior
  {
  }
ServerPort Port2
  {
   accesspoint
         {
           ServerActionPoint Point3  Nom_Methode(0,SYNCHRONE);
           DataPoint Point4  Nom_Methode(0,SYNCHRONE);
         }
   actioncontext
    {
```

Figure 4.17 : Exemple de la clause ports.

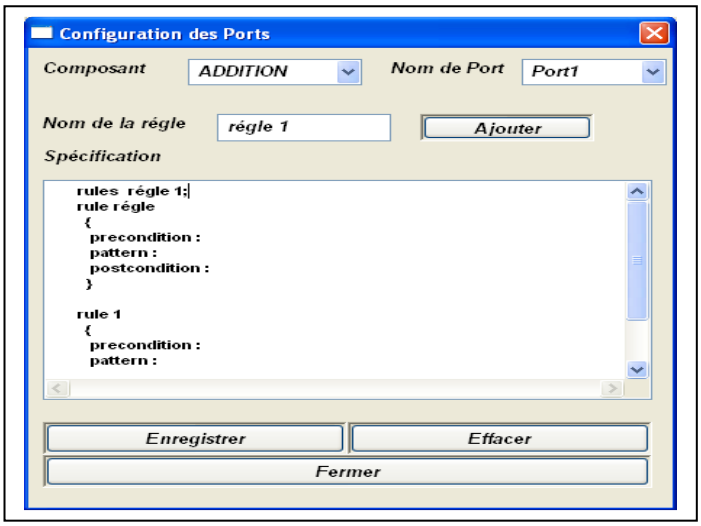

Figure 4.18 : Exemple sur les règles.

4.2.12.7. La clause connector :

Comme pour les ports, les types de connecteurs peuvent être défini dans une seule clause ce qui donne plus de clarté et d'organisation ou dans des clauses connector indépendantes. Et voici la spécification textuelle associé (Figure 4.19) :

```
Connector
{   Définition de type de connecteurs}

Définition d'un type de connector :
Connector  nomDuNouveauType
{
   Déclaration des rôles;
   Définition de l'architecture du connecteur en utilisant d'autres types de connecteurs ;
   Définition du contexte d'action ;
   Définition du comportement ;
}
```

Figure 4.19 : Spécification de la clause connector .

Déclaration des rôles :

Possibilité de plusieurs lignes de déclaration de rôles, Une ligne est comme suit;

roles *liste de rôles séparé par des virgules et se terminant par un point virgules;*

Définition de l'architecture :

Commence par le mot clé architecture.

Architecture
{Déclaration des connecteurs utilisés dans l'architecture
Attachement des rôles des divers connecteurs}

Déclaration des types de connecteurs utilisés (instanciation) :

Plusieurs lignes sont possibles :

TypeDeConnecteurExistant liste de Nom d'instances;

Connections des rôles :

map nonmDeRole to nomDeRole1, ….. nomDeRole1;

Définition du contexte d'action:

Liste de nom d'action spécifié par le type et le nom.

Définition des interactions:

Une interaction montre comment les rôles interagissent. Ces rôles seraient par la suite reliés au port de composant. Les rôles utilisent les actions de bases qui sont *invoke, request, send* et *receive*.

Une interaction peut être source d'Exception si certaines conditions sont violées. La description d'une interaction est similaire à la description d'un pattern d'une règle.

```
connector {
        connector FireCon {
                roles sRole, cRole;
                        architecture {
```

```
                    CSconnector csc;
                    map sRole to  csc.sRole;
                    map cRole to csc.cRole:
                    }
                    actioncontext {fire (BooleanOutputPin status);}
                    behavior{
                    interaction fire  raises FireException {
                            cRole.request(fire); sRole.invoke (fire);
                            sRole.send(status);cRole.receive(status);
                            success;
                            fail raiseFireException;
            }
      }     // Fin description du type FireCon
} // Fin de la clause définition de nouveau types de connecteurs
```

4.2.12.8. La clause operativepart:

Voici la spécification textuelle de cette clause (Figure 4.20):

```
operativepart{
        components {  Typecomposan1  Instancecomposant1;
                      ……..
                      TypecomposanN  InstancecomposantN;
                    }

        connexions
                  {
                   DelegateConnector  NomConnecteur1  { map connexion des
ports; }
                    ……
                   DelegateConnector  NomConnecteurN  { map connexion des
ports; }
                  }

} // Fin description de la partie opérative
```

Figure 4.20 : Spécification de la clause operativepart.

4.2.12.9. La clause controlpart:

Exemple général :

```
controlpart {
        components
                {
```

```
                    OpPartController opPartController;
                    ActiveOpPartException  opPartException;
                }
        connexions {
                FireCon  fireX25gcCon {
                        binding   {  bind cRole to opPartController.x25gc;
                                     bind  sRole to x25.pmain;
                        }
                        // Éventuels Contextes et interaction additionnelles
                        actioncontext { } behavior{ }
                }
                FireCon  firelcCon {
                        binding   { bind cRole to opPartController.lc;
                                    bind  sRole to lc.pmain;
                        }
                } // Fin clause connexions
}// Fin de la clause controlpart
```

4.2.12.10. La clause behavior (pour l'opPartController):

behavior
{ // Définition du comportement du composant *opPartController* }

4.2.12.11. La clause properties :

Exemple général :

properties

```
{   // Description du déploiement des composants
architecture
  {  // Définition des éléments de l'architecture.
        environment local
                {
                machine localhost;
                os System_Exploitation;
                }
        environment NomEnvironement
                {
                        machine  NomMachine;
                        os   System_Exploitation;
                }
}
```

La figure 4.21 montre un exemple sur la clause propriétés.

Figure 4.21 : Exemple sur la clause propriétés.

```
deployment
{ // Définition des éléments de déploiement
 deploymentcase {THREAD,  PROCESS; COMPOSITE}
 deploymentmap map1
    {
     deploy ComposantComposite as PROCESS in local; // deployment de composite
     deploy InstanceComposant as MAIN_THEAD in  COMPOSITE; // déploiement des
     deploy InstanceComposant    as PROCESS in  COMPOSITE;      // instances
    }
}

} // Fin description
```

La figure 4.22 montre un exemple sur la clause déploiement.

Figure 4.22 : Exemple sur la clause déploiement.

4.2.13. La grammaire associe à cette spécification :

Il s'agit ici la grammaire utilisée pour la spécification textuelle d'une architecture logicielle au format EBNF, les caractères sont entre « ' » et les mots clé entre « " », un « ? » représente une option, un « * » représente une répétition, un « + » représente une répétition d'au moins une occurrence, un | sépare différentes possibilités.

Voici donc la grammaire EBNF :

Voir la grammaire dans annex_2.

4.2.14. Validation par exécution de composant :

Le comportement d'un composant est celui observable sur la partie opérative. Nous distinguons deux types de comportements: le déroulement des interactions mettant en oeuvre les divers services des composants et les opérations de contrôle sur la structure effectuées par la partie contrôle. [14][15]

L'un des objectifs premiers de SEAL est de permettre la spécification de l'évolution structurelle des composants composites et la validation de cette évolution. La validation se fait par l'exécution de composant par l'interpréteur SEAL.

L'exécution d'un composant est restreinte actuellement à l'interprétation du comportement de la partie opérative décrite au niveau du composant OpPartController de la partie contrôle.

4.2.15. Élément de base pour l'exécution d'un composant :

L'exécution d'un composant nécessite la présence dans sa partie contrôle et dans toutes les parties contrôles de toutes les instances constituant la partie opérative des trois composants optionnels OpPartStateCmp, OpPartExceptionCmp et OpPartLogCmp. Si dans les parties contrôles un de ces composants n'est pas instancié, il y est alors instancié temporairement par l'interpréteur. L'interpréteur SEAL utilise toutes les composantes de la partie contrôle

de toutes les instances de composant et instancie les trois composants optionnels de la partie contrôle en mode actif. [14][15]

4.2.16. Technique de validation :

La technique de validation actuelle d'une architecture se base sur les deux principes suivants : [14][15]

- Déterminer les moment significatifs dans le processus d'évolution d'une architecture.
- Valider l'architecture après chaque moment significatif par la vérification du concept de stabilité (montrer qu'un composant est stable).

La validation de composant est réalisée, juste après chaque moment significatif. Elle consiste à appliquer au composant une enveloppe marquée vers l'extérieur et à vérifier la validation de tous les niveaux de la hiérarchie de composition du composant.

Les moments significatifs actuellement considérés concernent les événements principaux suivants.

- L'initialisation du système.
- Le passage d'un état autorisé à un état non autorisé d'un composant et vis versa.
- Le passage d'un état autorisé à un état non autorisé d'un service et vis versa
- La création et la suppression de points d'accès, de ports, de composant et de connecteurs.

4.2.17. Processus de validation :

Nous avons défini deux techniques à travers lesquels se fait la validation de l'évolution structurelle: [14][15]

- Dans la première technique, l'architecte écrit le code SEAL d'évolution de l'architecture.

- Dans la deuxième technique, l'architecte crée des *snapshost* réalise la validation des *snapshot* et génère le code SEAL correspondant qui sera intégré au composant de contrôle (*OpPartController*).

Dans une première réalisation de la validation, et dans un souci de réduire la complexité de l'interpréteur, les aspect déploiement de composant et analyse des interactions n'ont pas été considérés. De plus nous nous sommes limités dans la version actuelle à quatre actions : ajout et suppression de connecteurs et de composants.

Dans cette approche, l'interpréteur commence l'analyse de validité d'un composite à partir du premier niveau de sa hiérarchie de composition d'un composite. Il réalise ainsi l'exécution et l'analyse d'un composant selon la démarche suivante : [14][15]

- Vérification de la stabilité du premier niveau de la hiérarchie de composition par application d'une enveloppe fermée vers l'intérieur à toutes les instances du composite et d'une enveloppe fermée vers l'extérieur pour le composite lui – même. Dans le cas ou le composite parait instable, l'architecte doit corriger la source d'instabilité. Dans ce type d'analyse, souvent c'est un port client qui n'est pas connecté.
- Application d'une enveloppe fermée vers l'extérieure au composite et exécution des actions contenues dans le composant de contrôle de la partie opérative. Après chaque moment significatif, l'interpréteur réalise une opération de vérification de la stabilité (application d'enveloppe fermée vers l'extérieur). Durant cette opération il y'a création des diverses vues instantanée de l'architecture dans lesquelles pourraient apparaître les sources d'une éventuelle instabilité.

A la fin de l'exécution nous obtenons une liste de *snapshot*. L'analyse de ces *snapshot* permettra de déterminer l'origine des erreurs.

La figure 4.23 représente le schéma à la fin de l'exécution.

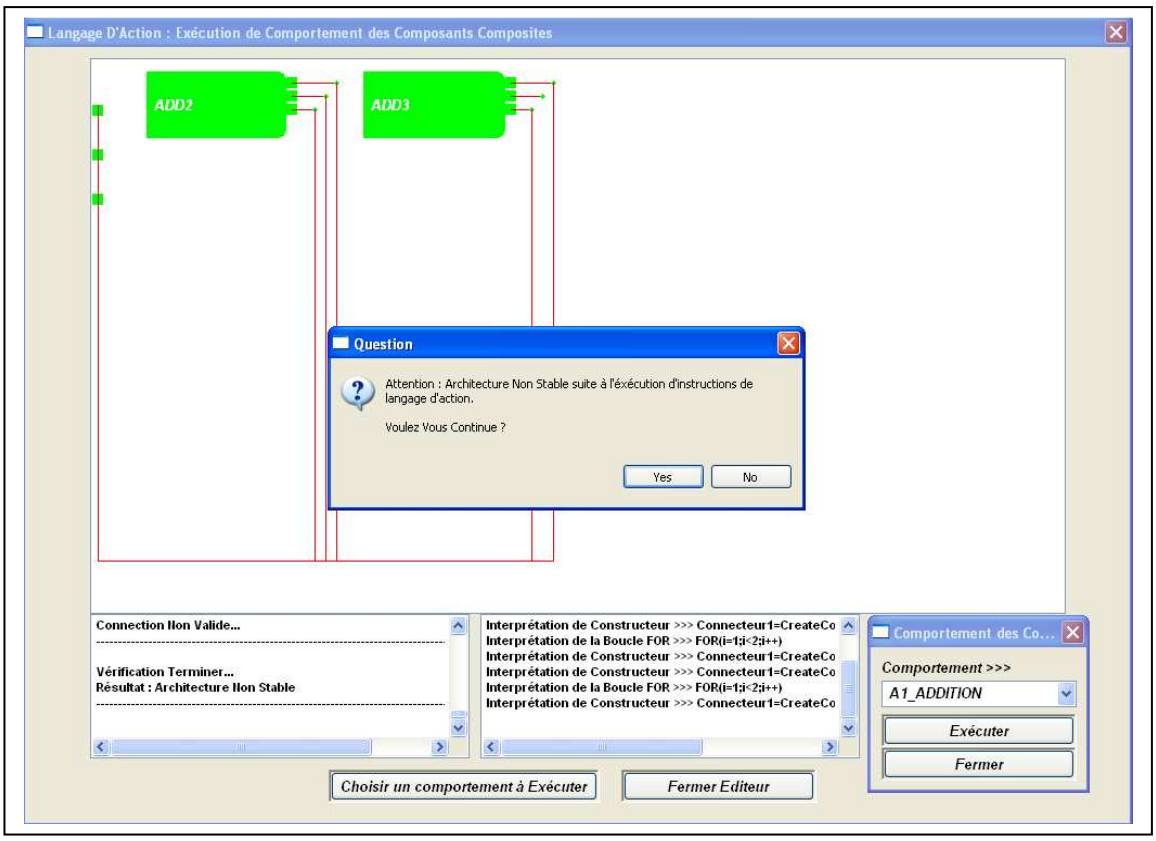

Figure 4.23 : schéma de fin de l'exécution.

4.3. Conclusion :

La spécification des comportements c'est à dire les interactions et le comportement de la partie opérative dans un composant composite se fait dans le langage d'action SEAL (Simple and Extensible Action Langage), Celui ci est doté d'une grande puissance d'expression due principalement à la notion de contexte (*ActionContext*). La version actuelle de l'interpréteur de SEAL ne s'intéresse qu'à l'exécution d'un composant composite, plus exactement du composant *BehavioralComponent* dans le but de montrer la validité des opérations de transformation architecturale comme l'ajout ou la suppression des composants et de connexions. Ce langage à été définit donc dans le but de montrer la validité des opérations de transformation architecturale et l'évolution structurelle de l'architecture. Il repose sur des contextes fondamentaux et sur tous les types prédéfinis relatifs aux composants, ports, point d'accès et connecteurs.

CHAPITRE 5

IASASTUDIO : UN ENVIRONNEMENT GRAPHIQUE POUR LA SPECIFCATION ET LA VALIDATION D'ARCHITECTURE LOGICIELLE

5.1. Introduction :

Pour la mise en œuvre de tous les concepts décrits précédemment nous avons développé un environnement de développement graphique, appelé IASASTUDIO, permettant de spécifier et valider le comportement d'une architecture logicielle.

La technique de spécification dans IASASTUDIO est d'une grande simplicité. Elle permet de prendre en charge de la manière la plus directe possible le modèle mental de l'architecte sous ses deux aspects, structurel et comportemental. Cette technique de spécification est en fait un renforcement du modèle informel à base de boite noire et d'interconnexions. Il permet en outre la prise en compte des aspects dynamiques d'une architecture.

A travers les modèles de base de l'approche IASA, l'environnement IASASTUDIO permet de faire des spécifications d'architecture logicielle à un haut degré d'abstraction, sans contraintes des concepts lié directement aux techniques d'implémentation, notamment le concepts d'interface sur lequel se base la quasi – totalité des modèles de composant et connecteurs actuels.

5.2. Présentation générale :

La fenêtre principale de IASASTUDIO (Figure 5.1) est composée d'un ensemble de fenêtres dont les principales sont :

- ✓ La fenêtre : Création du nouveau projet.
- ✓ La fenêtre : Importation des packages.
- ✓ La fenêtre : Importation des bibliothèques ArchJava.
- ✓ La fenêtre : Configuration des Ports.

- ✓ La fenêtre : Configuration des Propriétés et de Déploiement.
- ✓ La fenêtre : Validation de l'architecture.
- ✓ La fenêtre : Importation des composants déjà crées (Composant prédéfinit).
- ✓ La fenêtre : Implémentation des composants primitifs.
- ✓ La fenêtre : Saisie des instructions du langage d'action SEAL.
- ✓ La fenêtre : Affichage de l'architecture interne d'un composant.
- ✓ La fenêtre : Génération de code de la spécification textuelle de l'architecture.
- ✓ La fenêtre : Génération du code ARCHJAVA de l'architecture.
- ✓ La fenêtre : Spécification XML de l'architecture.
- ✓ La fenêtre : Choix du comportement d'un composant à exécuter.
- ✓ La fenêtre : Résultat de l'exécution du comportement de langage d'action.

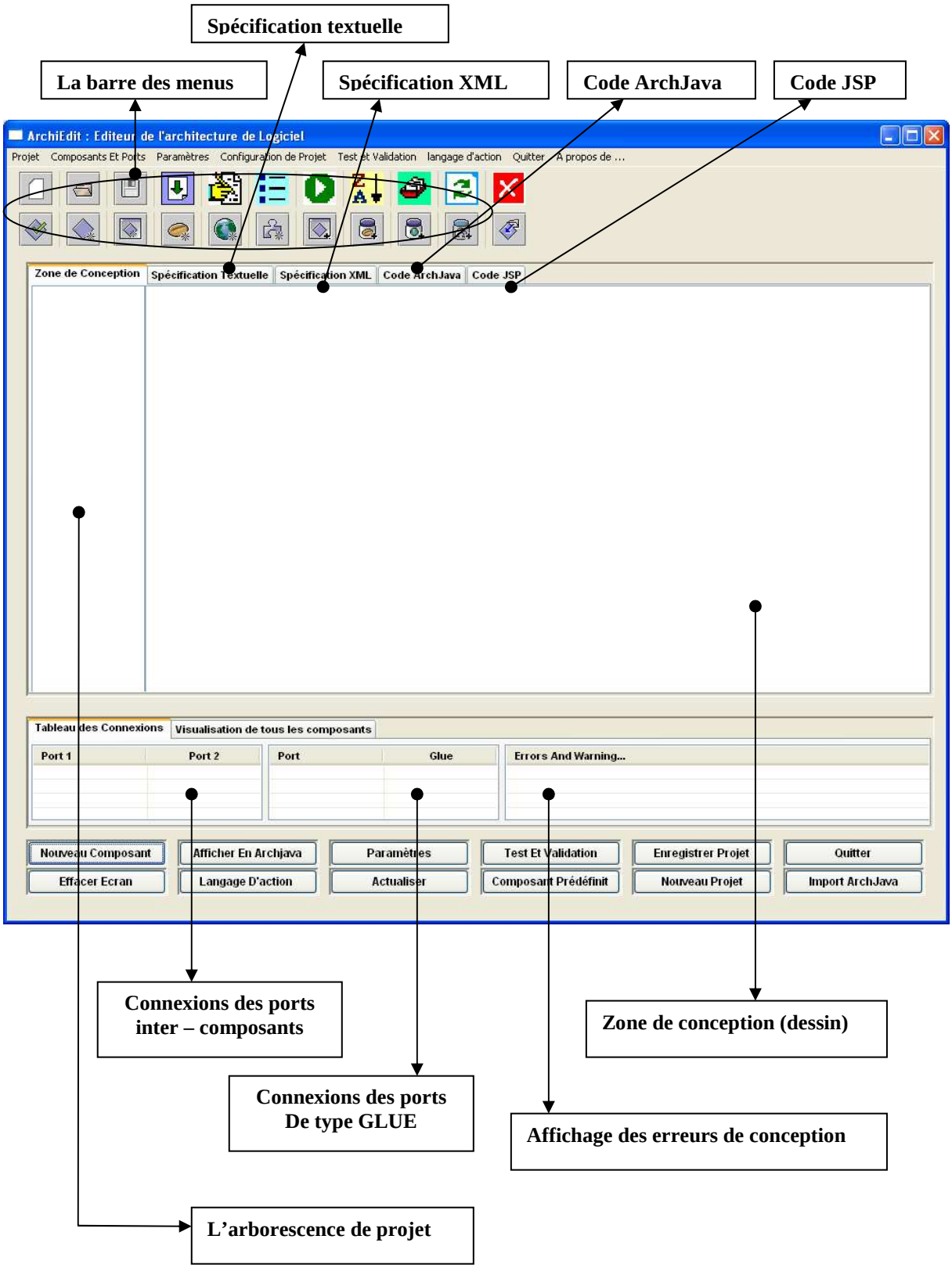

Figure 5.1 : Interface générale de « IASASTUDIO ».

Dans ce qui suit nous nous limitons à la présentations de quelques fenêtres de IASASTUDIO. Ensuite, nous montrerons leur mise en oeuvre à travers un exemple.

5.2.1. La fenêtre : Création du nouveau projet

Pour créer un nouveau projet il faut activer le choix « Projet → Nouveau Projet » dans la barre de menu et la fenêtre de la figure 5.2 apparaît :

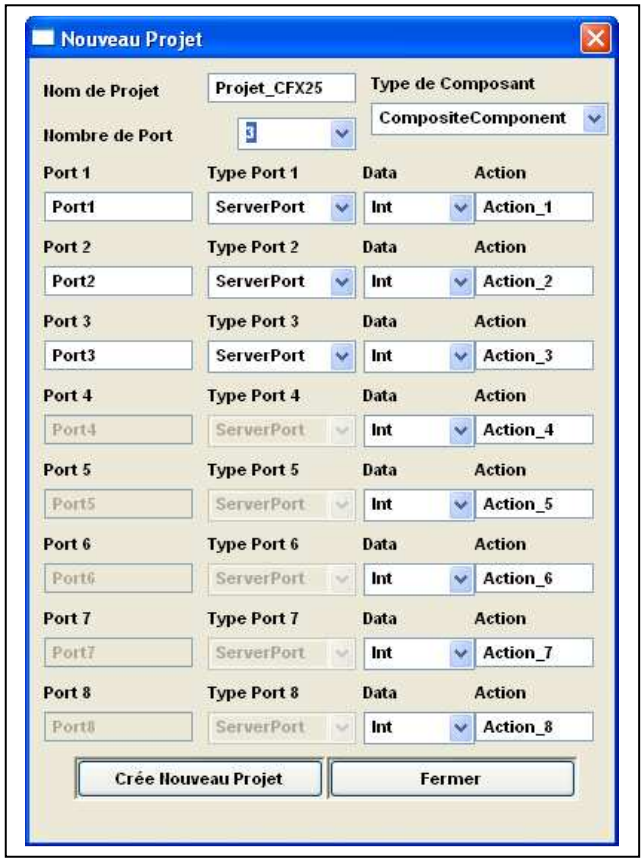

Figure 5.2 : Création du nouveau projet.

Dans la zone « Nom de Projet » il faut spécifier pour le projet un nom qui sera exploité pour nommer aussi le package global de l'architecture. Il est nécessaire de spécifier le type de projet (Composant Global). Le nombre de ports doit être aussi spécifié. Pour chaque port nous devons préciser le type (figure 5.3) ainsi que l'action à exécuter au niveau de ce port.

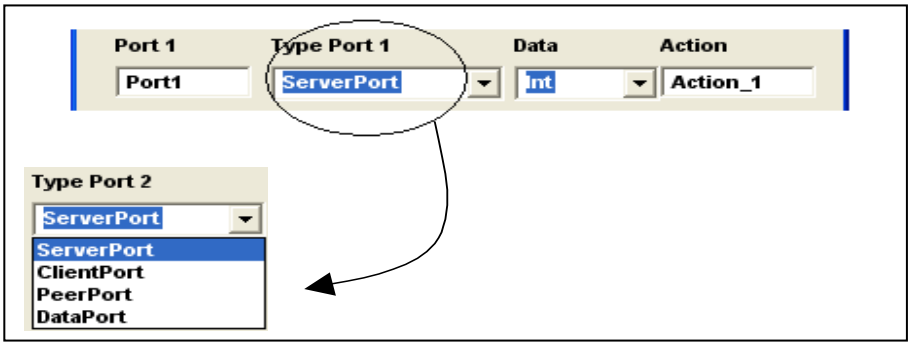

Figure 5.3 : Choix de type de port.

Lorsque la création du projet est confirmée par l'activation du bouton « Crée Nouveau Projet » dans la fenêtre de définition de projet (Figure 5.2), alors une autre fenêtre apparaît. Cette dernière permettra la définition de la structure interne de l'application. Nous rappelons qu'une application entière est considérée dans l'approche IASA comme étant un composant.

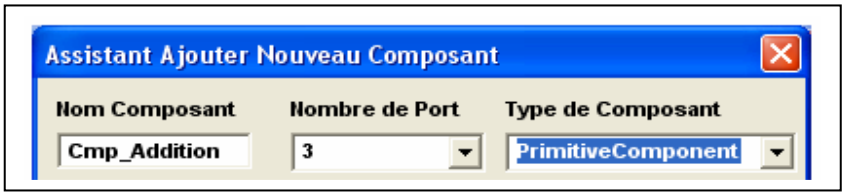

Figure 5.4 : Création du nouveau composant de projet.

Cette nouvelle fenêtre permet de créer un composant composite ou primitif (selon le choix de l'architecte), elle contient les mêmes informations que le composant Projet global (type de composant, nombre de ports, type de port, nom action).

Une fois le projet crée, nous devons importer les divers packages nécessaire à la génération de code SEAL et aussi à la génération de code dans le langage cible choisi. Actuellement IASASTUDIO reconnaît deux langages cibles: Le langage ArchJava et le langage JSP.

A titre d'illustration, la figure 5.5 montre la conception du composant composite EDI correspondant au « Projet_EDI ». Ce composant (ou application) est composé de :

- Trois instances de composants primitifs (c1, c2, c3) de type de composant cmp ce type est un composant primitifs.

- une instance A1 du type de composant ADDITION. Ce type est un composant composite.

- Une instance P1 du type de composant composite PROJET.

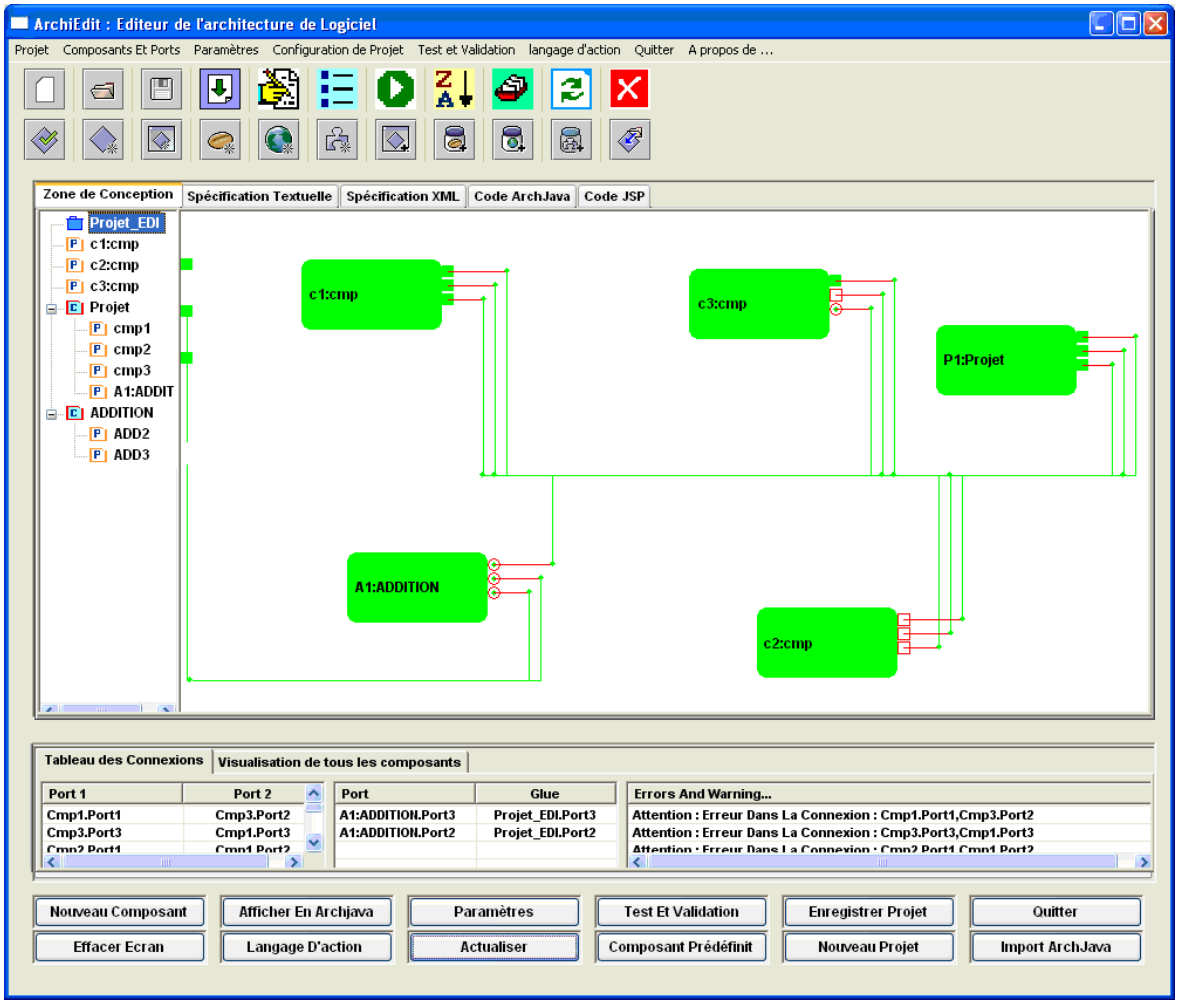

Figure 5.5 : Exemple général du composant composite « Projet_EDI ».

A gauche de la zone de conception (Figure 5.5), une fenêtre d'inspection de la hiérarchie de composition du composant à réaliser est affichée. L'arborescence montre clairement les types utilisés et les instances de ces types ainsi que les différents composants composites et primitifs du projet, le "P" indique que le composant est primitif et le "C" pour indiquer que le composant est composite (Figure 5.6).

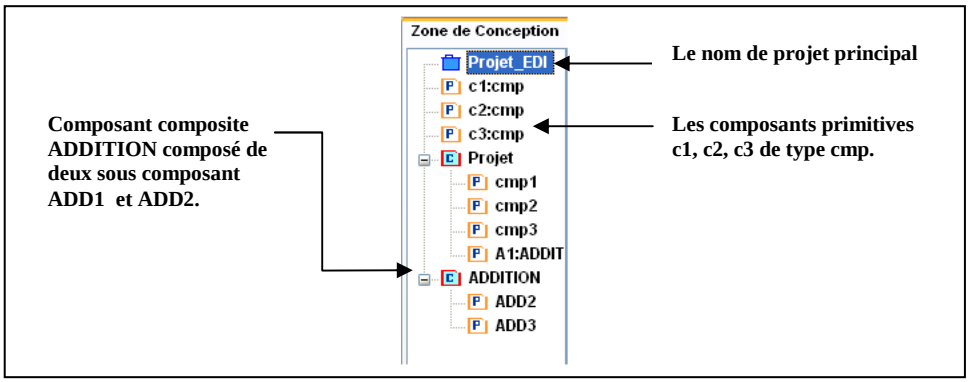

Figure 5.6 : Arborescence des composants.

Dans la figure 5.7 on remarque le composant c1 qui est de type composant Cmp, il est doté de trois ports de trois type différents : un ServerPort, ClientPort et PeerPort.

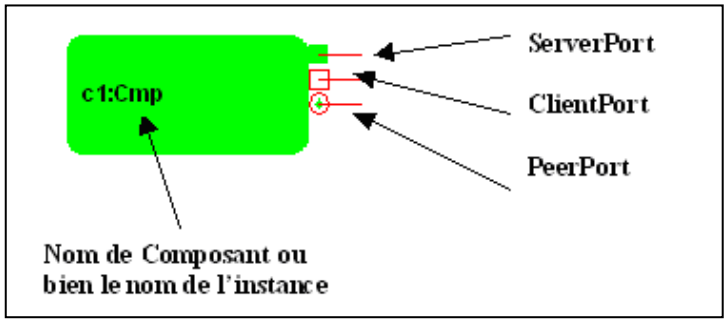

Figure 5.7 : Exemple de la structure générale d'un composant et des ports.

La figure 5.8 donne des informations sur les connexions et les Glue des ports, nous rappelons que deux type de validation son supportés : La validation des connexions inter – composant et la validation des connecteurs de délégation (Glue).

La validation est indiquée par des messages qui renseignent sur la validité des connexions.

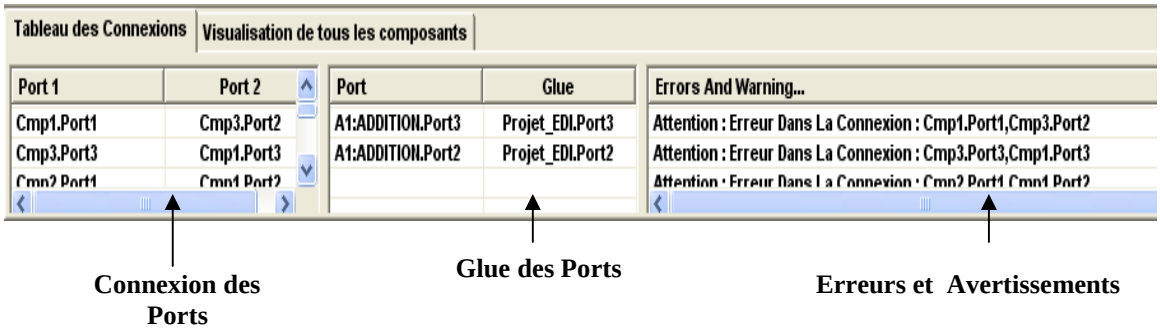

Figure 5.8 : Informations sur les connexions et les Glue des ports.

5.2.2. La fenêtre : Importation des packages

L'importation des packages se fait par l'activation du menu « Configuration de Projet → Importation des Packages ». Qui fera apparaître la fenêtre de la figure 5.9, Cette fenêtre permet de spécifier les divers package à importer pour le projet.

Figure 5.9 : Importation des packages.

5.2.3. La fenêtre : Importation des bibliothèques ArchJava

Pour la génération de code dans le langage cible ArchJava, il est nécessaire d'importer les bibliothèques nécessaires par l'activation du menu « Configuration de Projet → Importation des librairie ArchJava ». Suite à cette activation, la fenêtre de la figure 5.10 s'ouvre :

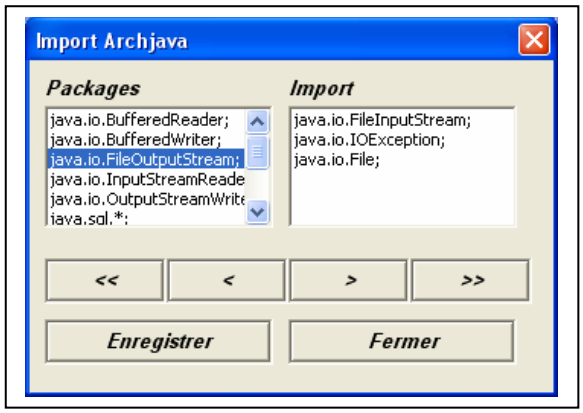

Figure 5.10 : Importation des bibliothèques ArchJava.

5.2.4. La fenêtre : La configuration des Ports

La configuration des ports est réalisée dans la fenêtre configuration de port (Figure 5.11) qui est accessible à travers le menu « Configuration de Projet → Configuration des Ports ».

La configuration de port permet de spécifier le comportement observable sur le port. Le comportement est spécifié en langage SEAL. Il consiste à attacher au port un certain nombre de règles comme indiqué sur la figure 5.11.

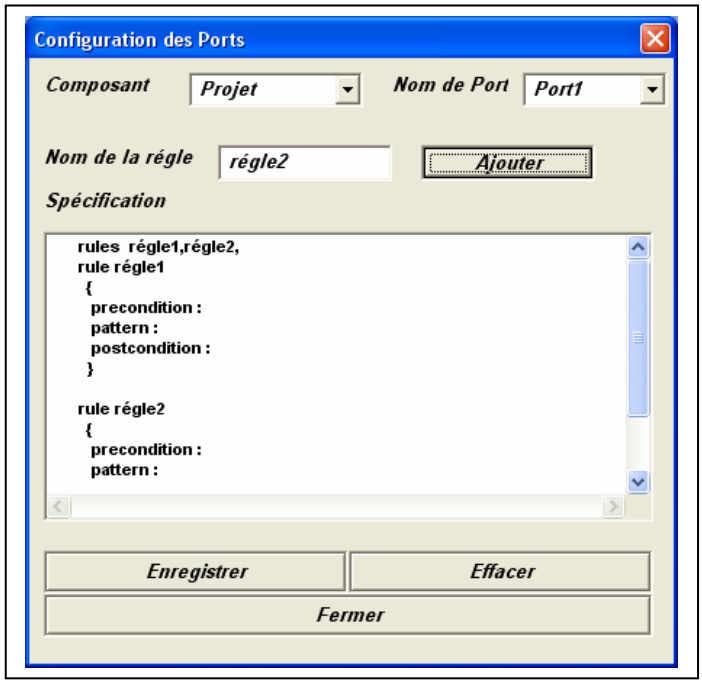

Figure 5.11 : Configuration des ports.

5.2.5. La fenêtre : Configuration des Propriétés et de Déploiement

IASASTUDIO permet de spécifier l'architecture de déploiement et le déploiement des composant dans l'architecture à travers la fenêtre « Configuration des Propriétés et de Déploiement » (Figure 5.12) qui est accessible suite à l'activation du menu « Configuration de Projet → Configuration des Propriétés et de Déploiement ».

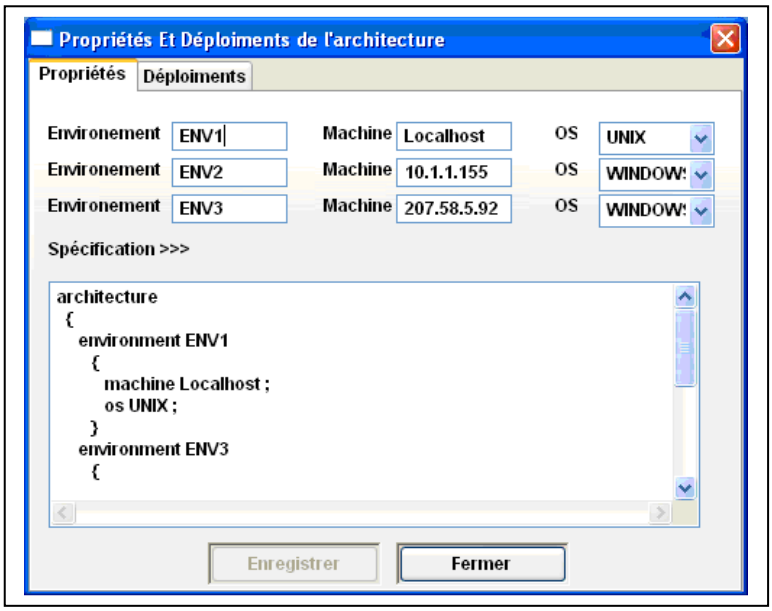

Figure 5.12 : Configuration des Propriétés.

Sur la figure 5.12 l'environnement de déploiement ENV1 est représenté par la machine locale dotée du système d'exploitation UNIX.

L'environnement de déploiement ENV2 est une machine dont l'adresse IP est 10.1.1.155 et est dotée d'un système d'exploitation WINDOWS.

L'environnement de déploiement ENV3 est une machine dont l'adresse IP est 207.58.5.92 et est dotée d'un système d'exploitation WINDOWS.

La fenêtre propriétés de déploiement (Figure 5.13) spécifie de manière précise l'environnement ou devra être déployé chaque composant ainsi que la qualité du composant dans l'environnement de déploiement

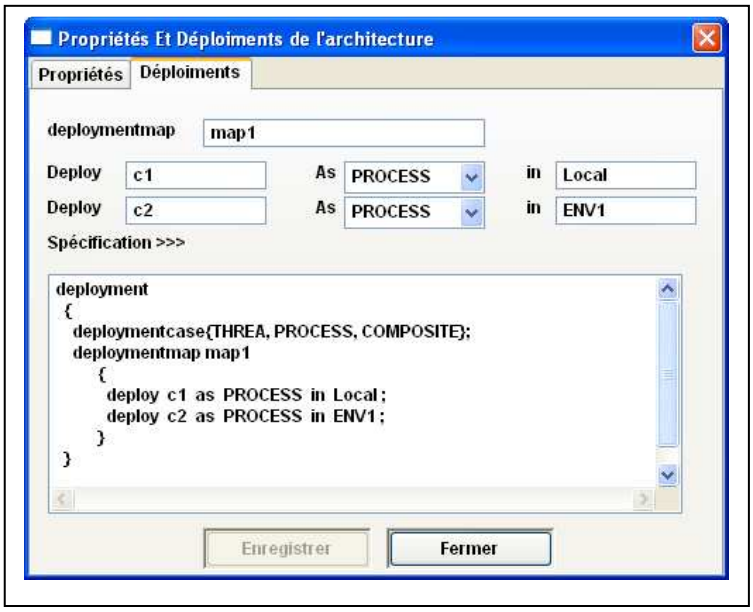

Figure 5.13 : Configuration de Déploiement.

Sur la figure 5.13, nous remarquons que le composant c1 sera déployé comme un PROCESS dans l'environnement Local.

Le Composant c2 sera déployé comme un PROCESS dans l'environnement ENV1.

5.2.6. La fenêtre : Test et Validation de l'architecture

La fenêtre « Test et Validation de l'architecture » (Figue 5.14) offre les diverses facilités pour la validation d'une architecture. Elle est accessible par le menu : Test et Validation → Test et Validation de l'architecture.

Deux type de validation son supportés : La validation des connexions inter – composant et la validation des connecteurs de délégation, souvent appelée Glue.

La validation est indiquée par des messages qui renseignent sur la stabilité d'une architecture.

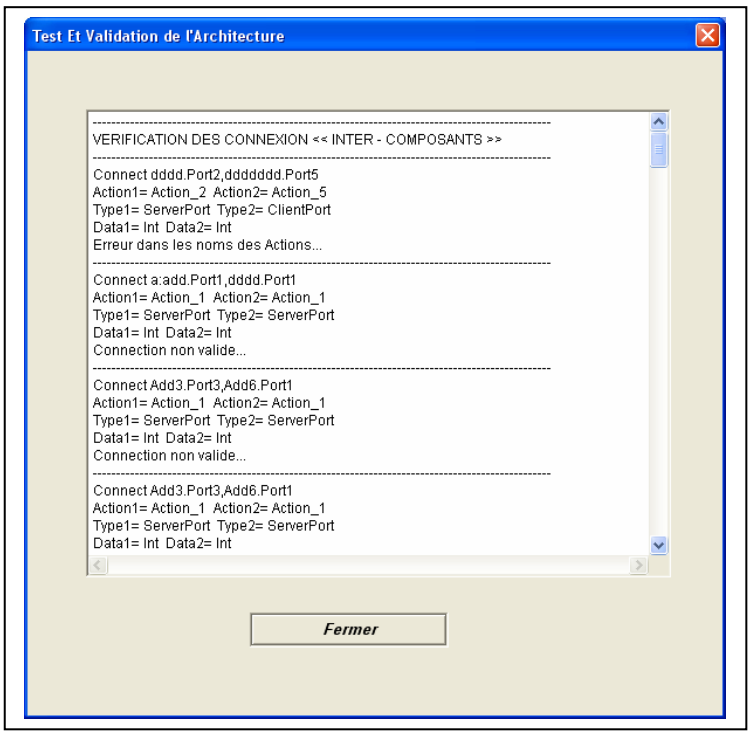

Figure 5.14 : Test et Validation de l'architecture.

Nous rappelons ici que les enveloppes marquées sont utilisées dans le processus de vérification de l'état global (stabilité) d'un composant. Il est possible de marquer totalement ou partiellement les points d'accès. Le marquage d'un point d'accès signifie que ce dernier est correctement connecté. Une enveloppe totalement marquée est une enveloppe dont tous les points d'accès sont marqués. Dans une enveloppe marquée partiellement vers l'extérieur, seuls les services requis *ClientDataPoint* et les *DataPoint* dont le sens est *in* ou *inout* sont marqués. Dans une enveloppe marquée partiellement vers l'intérieur, seuls les services fournis *ServerDataPoint* et les *DataPoint* dans le sens est *out* ou *inout* sont marqués.

L'étude de stabilité d'un composant consiste donc à appliquer une enveloppe marquée vers l'extérieur pour le composite et des enveloppes marquées vers l'intérieur pour chaque instance de la partie opérative et contrôle.

Après avoir appliqué ces enveloppes, il suffit de trouver au moins un port requérant un service ou une donnée (port client ou un port de données dont le sens est *in*) non connecté à un port marqué pour décider que le composite est instable.

5.2.7. Saisie de code dans le langage SEAL et dans le langage cible :

La saisie du langage SEAL est réalisée pour définir le code du composant de contrôle de la partie contrôle d'un composant, la définition des interaction et la spécification des comportement observables sur les port d'interaction. La saisie du code en langage cible est réalisée uniquement pour les composants primitifs.

Dans les deux cas, la fenêtre de saisie du code apparaît suite à un double click sur une instance de composant ou sur le type de composant au niveau de l'inspecteur de la hiérarchie de composition.

La figure (5.15) montre la fenêtre de saisie du code SEAL pour le composant de contrôle de la partie contrôle.

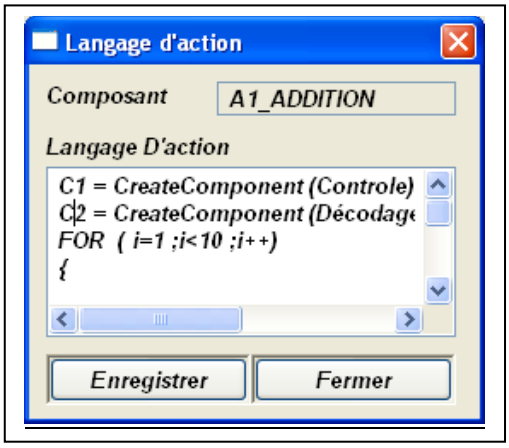

Figure 5.15 : Saisie des instructions du langage d'action SEAL.

Les composants primitifs correspondent toujours à une implémentation. Un composant primitif doit être soit importée soit défini dans IASASTUDIO en utilisant le langage cible adéquat. A titre d'exemple, si le langage cible est ArchJava, le composant primitif doit être écrit soit en ArchJava (mise en œuvre de concept e l'architecture logicielle sans passer par le modèle IASA) soit directement en JAVA. Dans IASASTUDIO, la fenêtre de saisie du code en langage cible relatif à un composant primitif est accessible par un double click sur une des instances du composant primitif ou sur le type du composant primitif dans la fenêtre d'inspection de la hiérarchie de composition. Le code saisi correspond aux divers services fournis par le composant primitif.

En se basant sur la description SEAL, IASASTUDIO génère toutes les entêtes méthodes correspondant aux actions supporté par le composant primitifs (action déclarées aux niveau de tous les point d'accès de type ServerActionPoint). IASASTUDIO inspecte rapidement le code saisi pour vérifier les présences d'appel de méthodes qui correspondent aux actions déclarées au niveau des points d'accès de type ClientActionPoint. Pour la vérification du code source, IASASTUDIO utilise le compilateur du langage cible. La validation du code source est complètement à la charge de son concepteur.

Figure 5.16 : Implémentation des composants primitifs.

5.2.8. La génération de code décrivant l'architecture

Dans IASASTUDIO, la conception est réalisée presque totalement de manière graphique. Le code SEAL est écrit dans des situations particulières. Cependant cette tache d'écriture du code SEAL est très légère et est réduite à la définition du comportement du composant de contrôle de la partie contrôle, la spécification des interactions associées aux connecteurs et le comportement observable sur les ports. Le code écrit pour chaque cas est souvent très léger. La totalité du code SEAL représentant l'architecture d'une Application est généré automatiquement dans IASASTUDIO au fur et a mesure de l'avancement de la définition d'une architecture. IASASTUDIO permet en plus de générer le code complet de l'architecture dans le langage cible choisi et en XML. A titre d'illustration, la figure 5.17 montre le code SEAL associé au projet EDI.

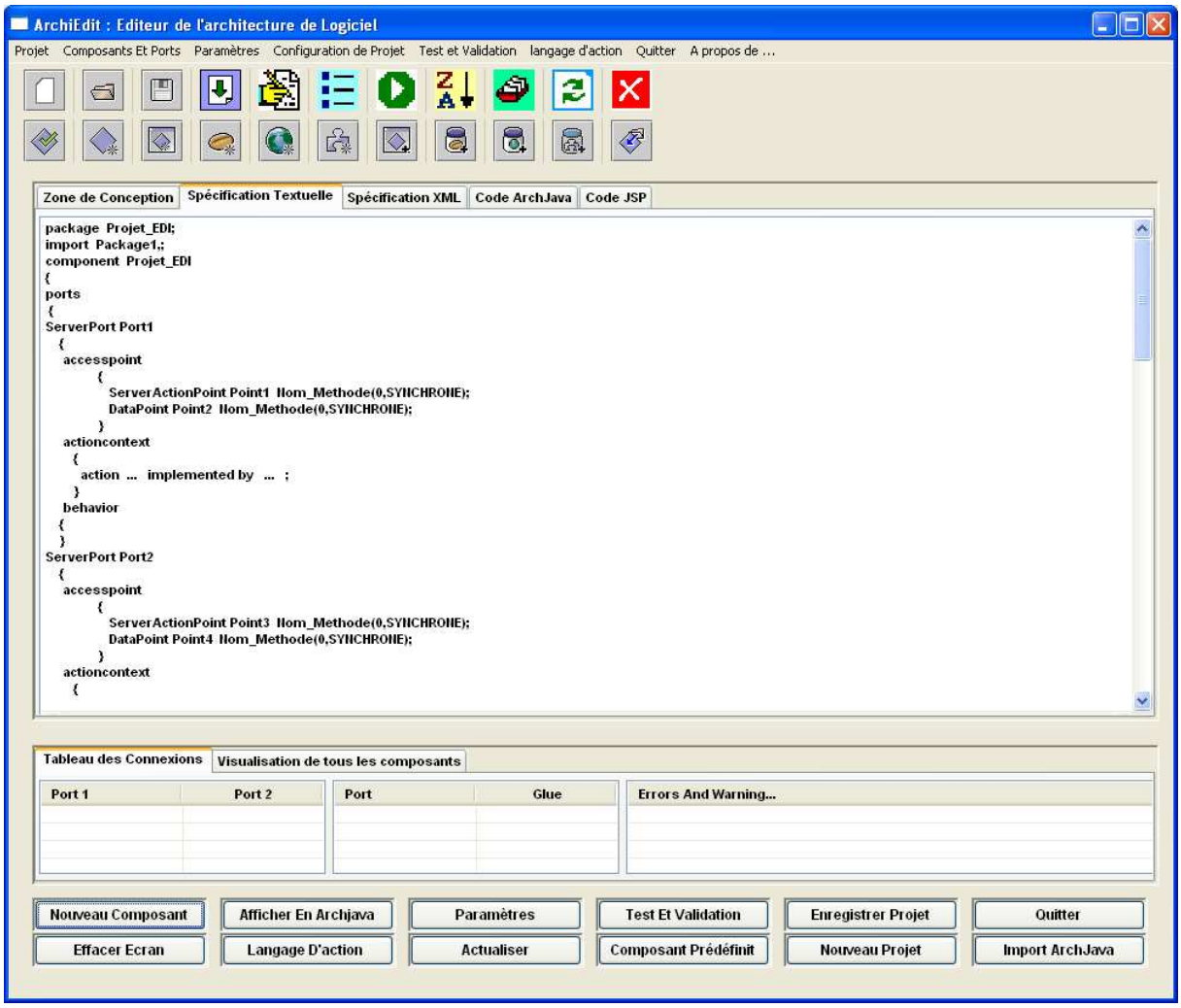

Figure 5.17 : La génération de code de la spécification textuelle de l'architecture.

La génération du code SEAL est accompagnée d'une vérification structurelle de l'architecture. A titre d'exemple la figure 5.18 avertit le concepteur d'un manque d'un fichier de configuration d'un port.

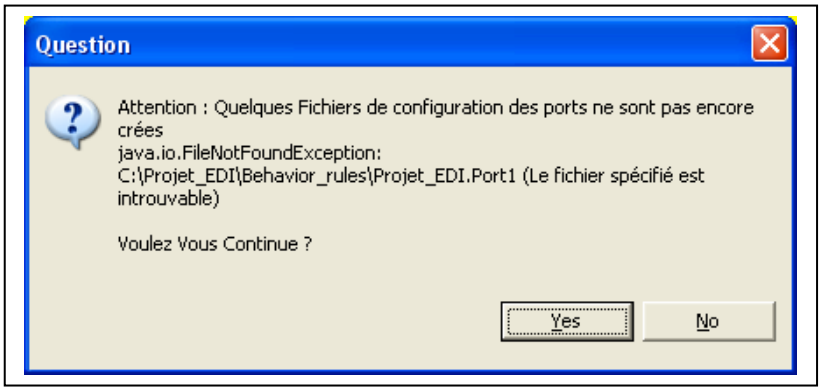

Figure 5.18 : Message d'erreur indique qu'un port ou une propriété n'est pas bien configurée.

La figure 5.19 montre la génération de l'architecture dans le langage cible. Dans le cas de la figure 5.19 c'est le langage ArchJava qui est considéré.

Des messages d'erreurs ou d'avertissement sont générés si le code ArchJava est incomplet. En général les erreurs concernent l'oubli de la définition d'un composant primitif (Figure 2.20)

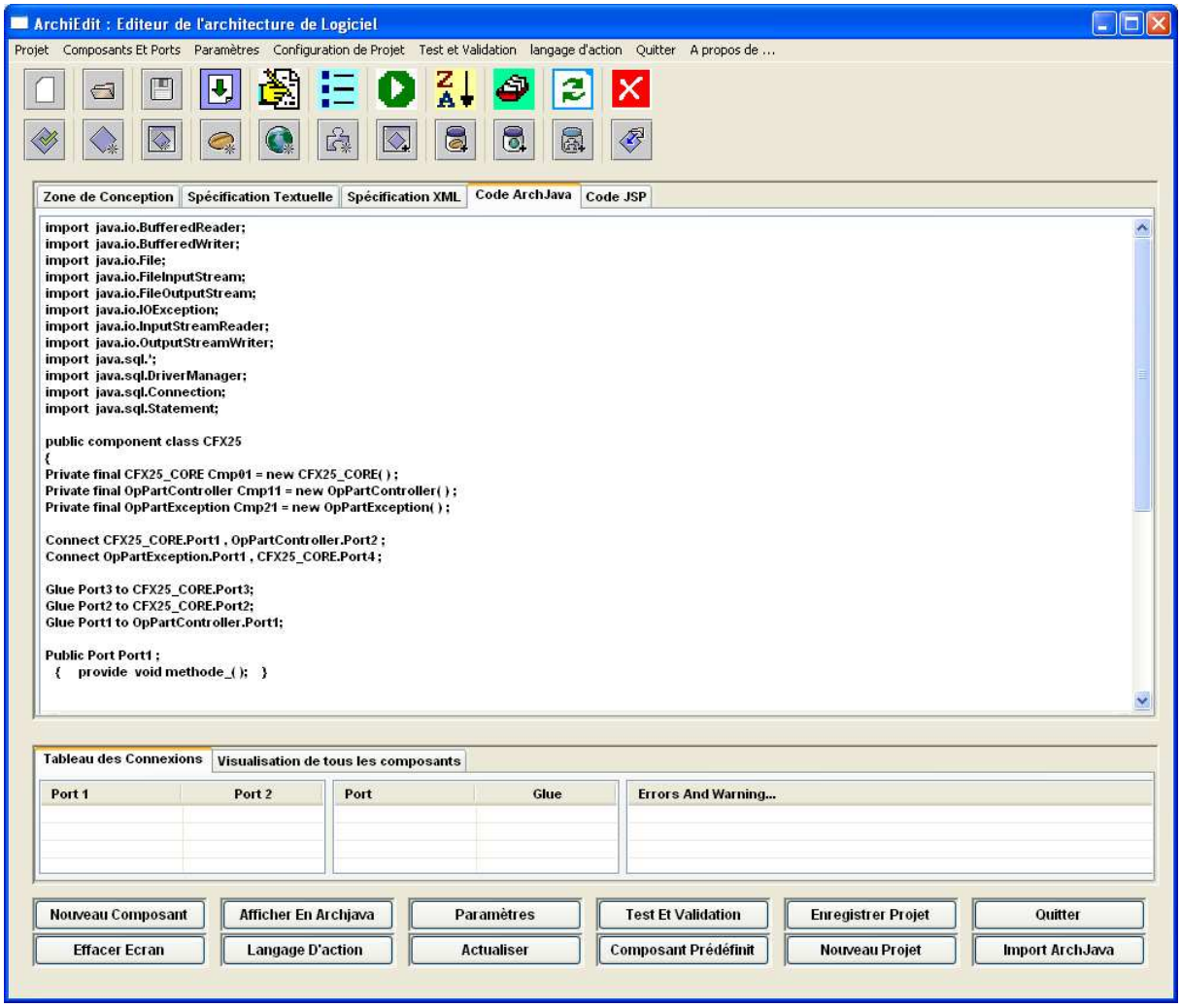

Figure 5.19 : La génération du code ARCHJAVA de l'architecture.

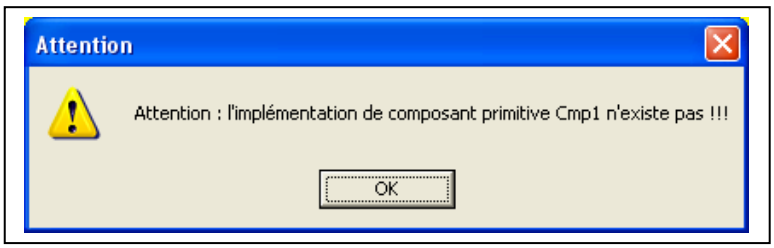

Figure 5.20 : Message d'erreur indique qu'il existe des composants

Primitifs qui ne sont pas implémentés.

Pour la spécification XML on a utilisé un ensemble de balise de code xml pour définir notre architecture, ces balises sont définit dans le chapitre suivant dans le tableau 6.1.

La figure 5.21 monte la génération de la spécification XML de l'architecture.

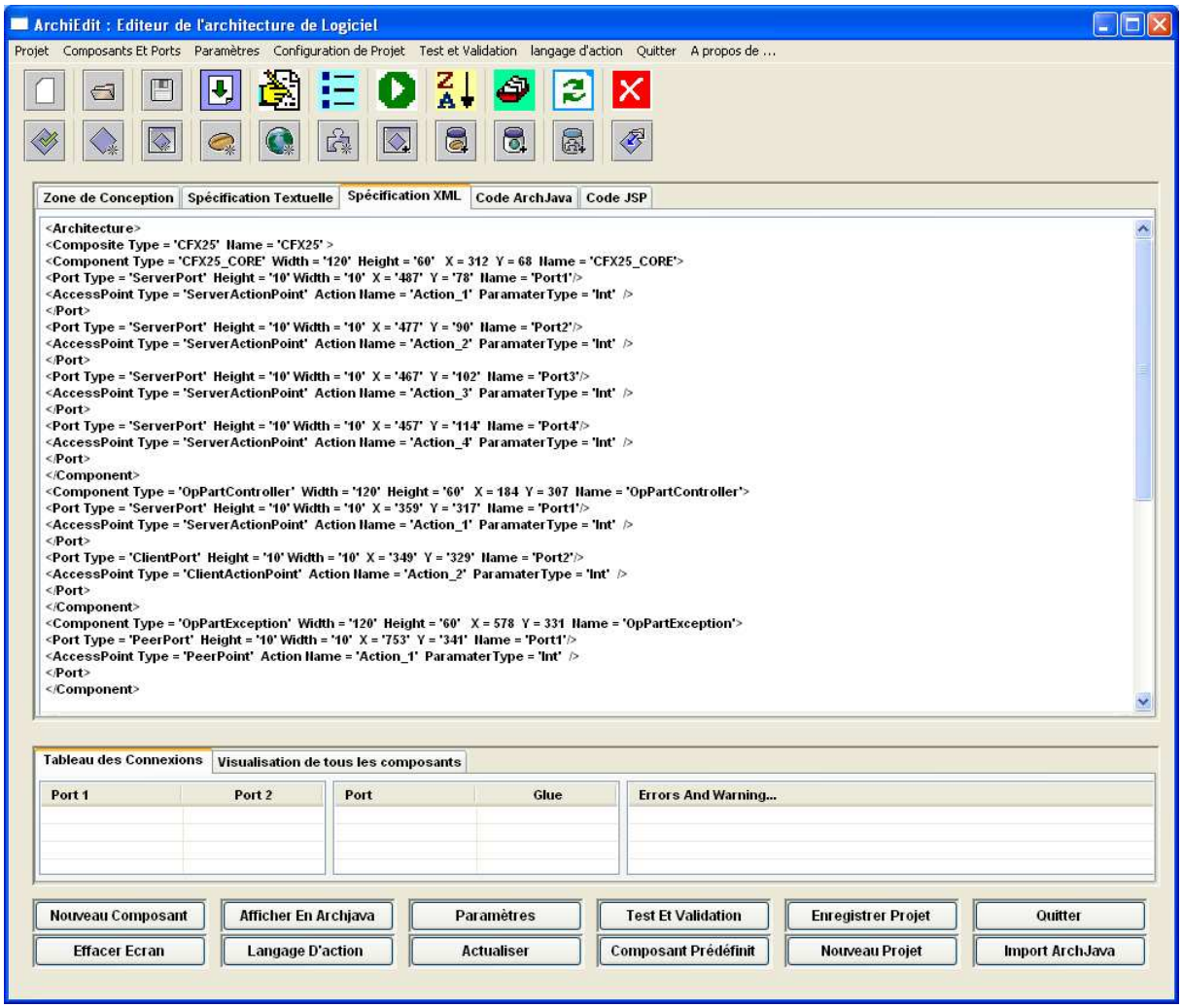

Figure 5.21 : La spécification XML de l'architecture.

5.2.9. La fenêtre : Choisir du comportement d'un composant à exécuter

Pour exécuter un langage d'action sur un composant composite il faut cliquer sur le bouton « Langage d'action → Choisir un comportement à exécuter ».

Figure 5.22 : Choisir du comportement d'un composant à exécuter.

Le comportement de l'instance A1 de composant composite ADDITION est alors choisi. Donc il suffit de cliquer sur exécuter.

5.2.10. La fenêtre : Résultat de l'exécution du comportement de langage d'action

Le résultat de l'exécution après l'application du langage d'action sur le composant sélectionné est apparaît a la figure 5.23.

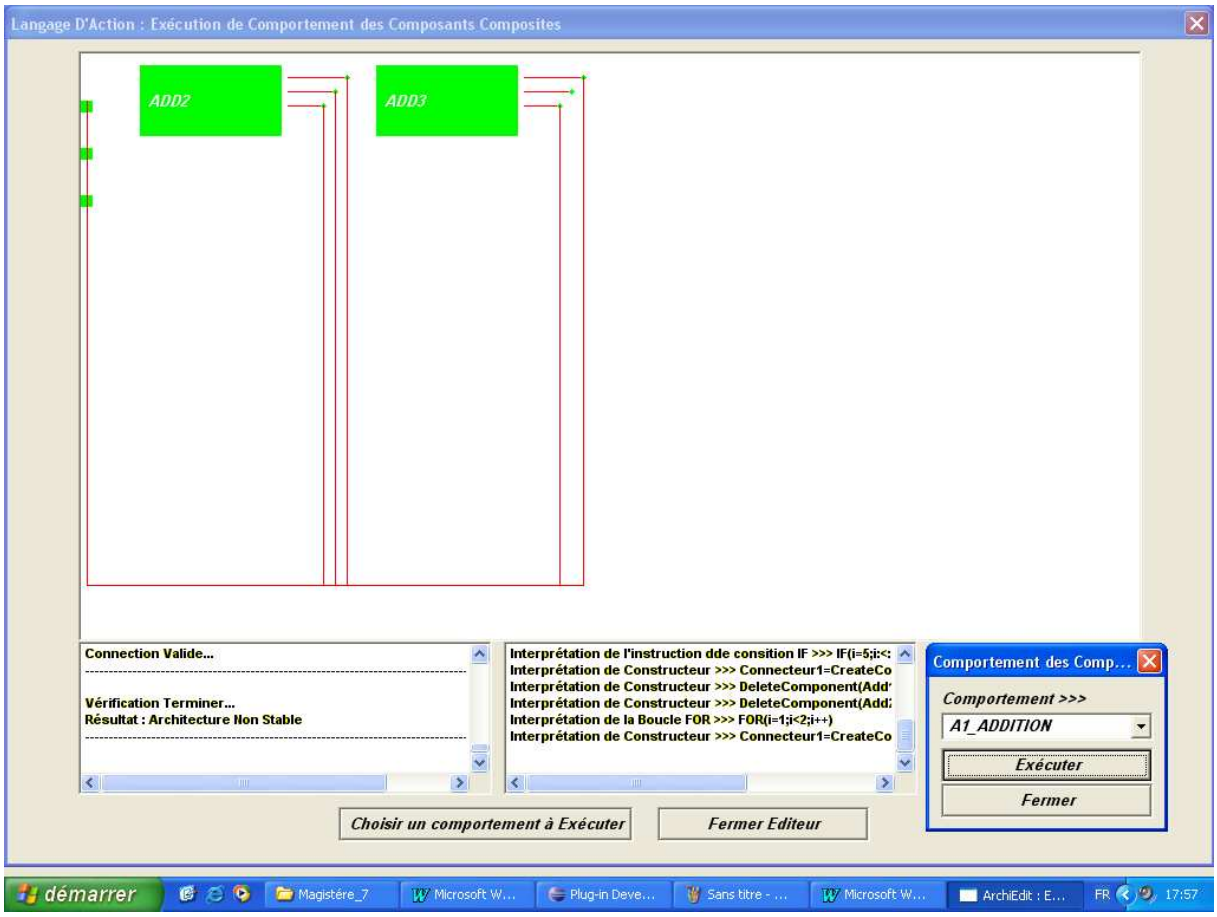

Figure 5.23 : Résultat de l'exécution du comportement de langage d'action.

Si après l'exécution d'une instruction du langage d'action on trouve qu'il y a une erreur, un message nous indique que l'architecture n'est pas stable (Figure 5.24).

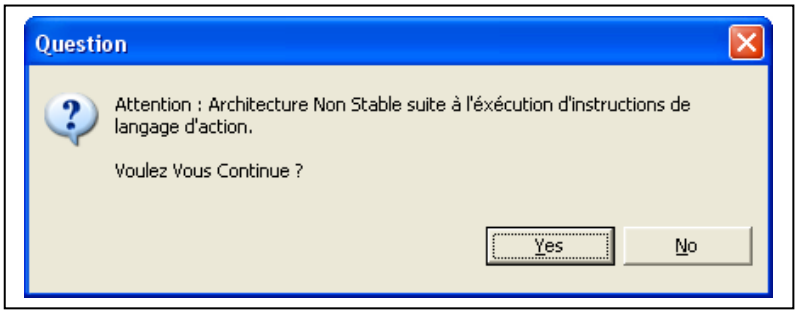

Figure 5.24 : Message d'erreur indique que l'architecture n'est pas stable.

Le résultat de l'exécution des instructions de langage SEAL et les différentes étapes d'interprétation de ces instructions est montré dans la figure 5.25.

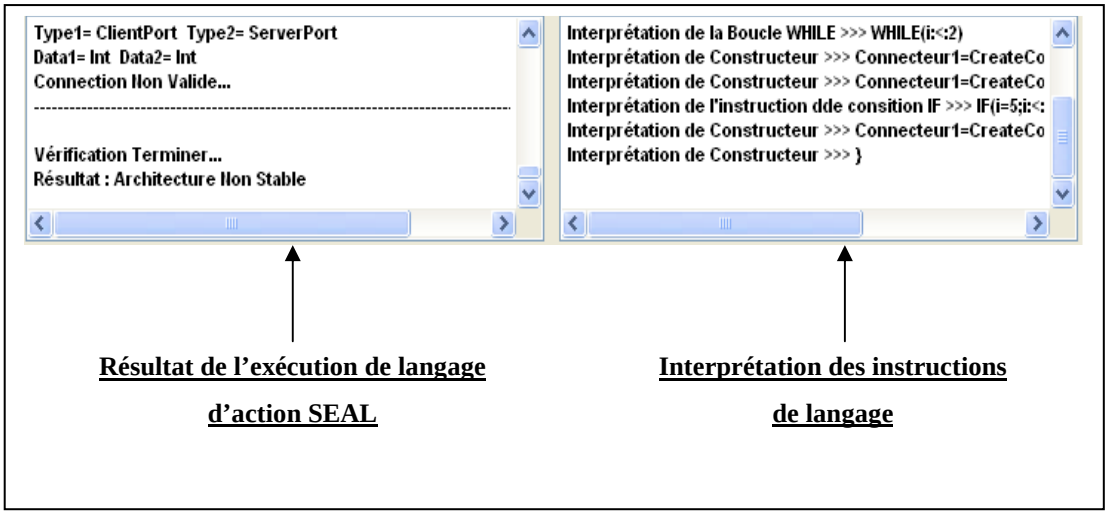

Figure 5.25 : Résultat de l'exécution des instructions de langage SEAL et les différentes étapes d'interprétation de ces instructions.

5.3. Conception d'application centrée sur le Web :

IASASTUDIO permet de concevoir des applications selon une architecture à niveau basée sur un serveur Web, en utilisant l'approche IASA. Nous allons illustrer cette capacité à travers une partie d'un projet de E – gouvernement dont l'objectif est de réaliser une E-APC.

Les JSP ou Java Server Pages sont une technologie Java qui permettent la génération de pages web dynamiques, les JSP permettent d'introduire du code Java dans des tags prédéfinis à l'intérieur d'une page HTML. La technologie JSP mélange la puissance de Java côté serveur et la facilité de mise en page d'HTML côté client. Donc la technologie JSP permet de séparer la présentation sous forme de code HTML et les traitements sous formes de classes Java définissant un bean ou une servlet.

Une JSP est habituellement constituée :

- De données et de tags HTML.
- De tags JSP.
- De scriptlets (code Java intégré à la JSP).

Les fichiers JSP possèdent par convention l'extension ".jsp".

Pour la génération de code JSP deux cas sont possibles :

Cas 1 :

L'architecture ne contient que des composants de type Page JSP, c'est – à – dire chaque composant primitif représente une page JSP, et donc l'architecture globale qui est un composant composite représente donc tout ou une partie d'un site.

Dans ce cas il faut crée un projet de type « Projet JSP » (Figure 5.26), et concevoir l'architecture de site correspondante, donc le site (le composant composite) sera constitue d'un ensemble de page JSP interconnecter entre eux à travers des ports et des connecteurs qui respect le modèle IASA. (Exactement comme il est indique pour les architectures logicielles).

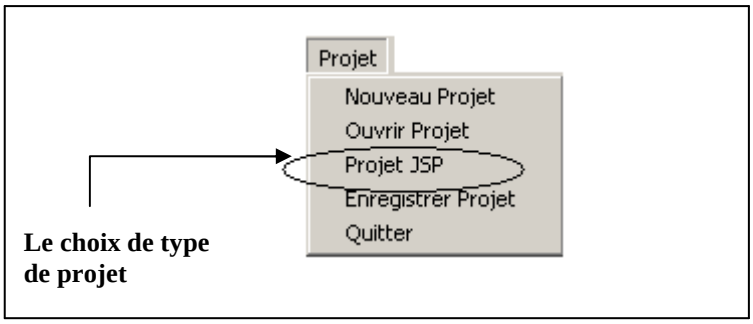

Figure 5.26 : Schéma présente l'icône de création d'un projet JSP.

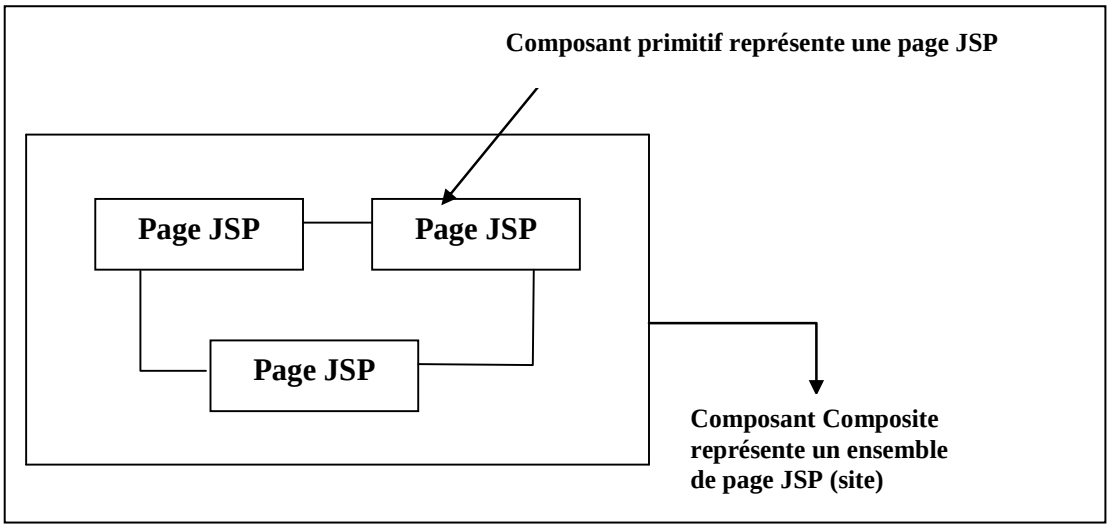

Figure 5.27 : Composant Composite représente un ensemble de page JSP.

Les inter – connexion et la structure des composants et des ports et des connecteurs sont crées selon le modèle IASA qui est déjà définit.

Dans ce cas nous avons trouvé qu'une page JSP classique peut regrouper des codes qui peuvent être généré automatiquement, parmi ces codes on peut citer :

- Le code qui permet la connexion vers une base de donnée (chargement de driver, Spécification de SGBD,….).

- Le code pour l'ouverture et la fermeture d'un flux d'entrer sortie de donnée (fichiers).

- Les importations des packages et des librairies.

Prenons l'exemple suivant qui représente un petit site web d'apc (composant composite Site_APC) composer de deux pages de type JSP (Figure 5.28):

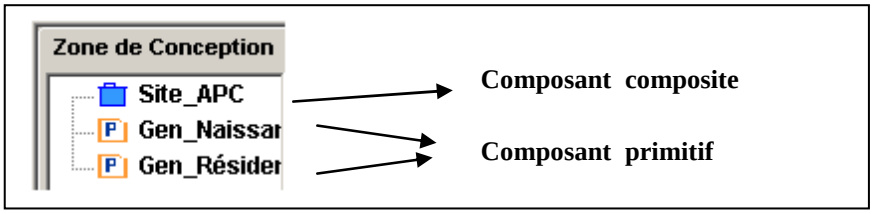

Figure 5.28 : Site web « Site_APC » composer de deux pages JSP.

- Gen_Naissance : Page JSP pour la génération des actes de naissances, composant primitif doté de deux port, un ServerPort pour la fourniture des actes et un ClientPort qui demande le numéro de l'acte.

- Gen_Résidence : Page JSP pour la génération des fiches de résidence, composant primitif doté de deux port, un ServerPort pour la fourniture des actes et un ClientPort qui demande le numéro de l'acte.

La figure 5.29 montre le schéma général de l'architecture selon le modèle IASA :

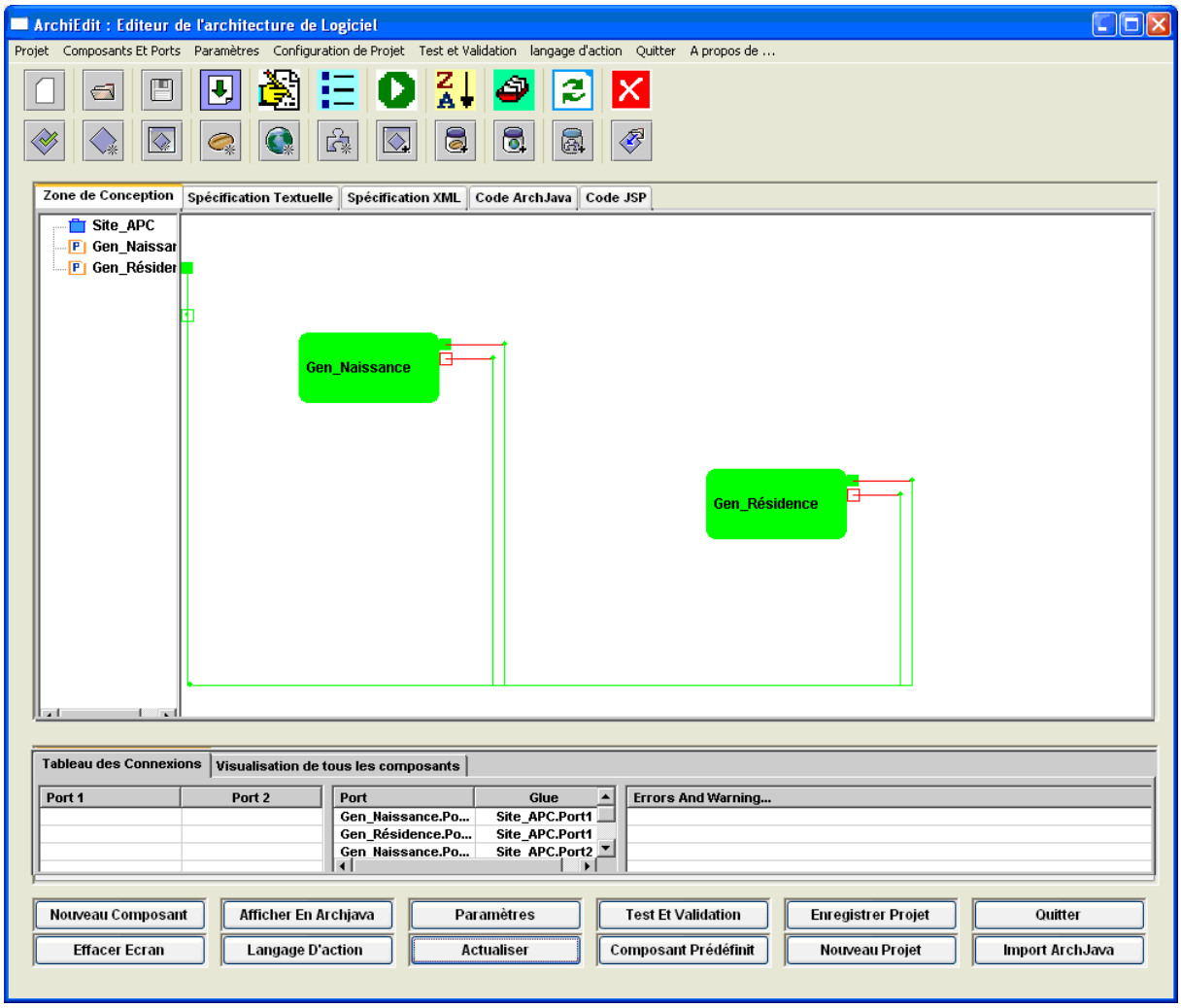

Figure 5.29 : Schéma général de l'architecture.

Maintenant on doit spécifié quelques informations sur chaque composant primitif de l'architecture, pour cela un double clique sur un composant primitif ouvre la fenêtre suivante qui nous demande d'indiquer quelques informations sur la page JSP, on commence par des informations sur la connexion vers une base de donnée, puis l'utilisation des entrer sortie, et enfin l'importations des packages et des libraires. Si une étape n'est pas utilisé dans la page JSP il suffit de passé a l'étape suivante en cliquant sur le bouton « Suivant ».

La première fenêtre qui correspond aux propriétés de la connexion et qui concerne les informations sur les bases des données est représentée par la figure 5.30.

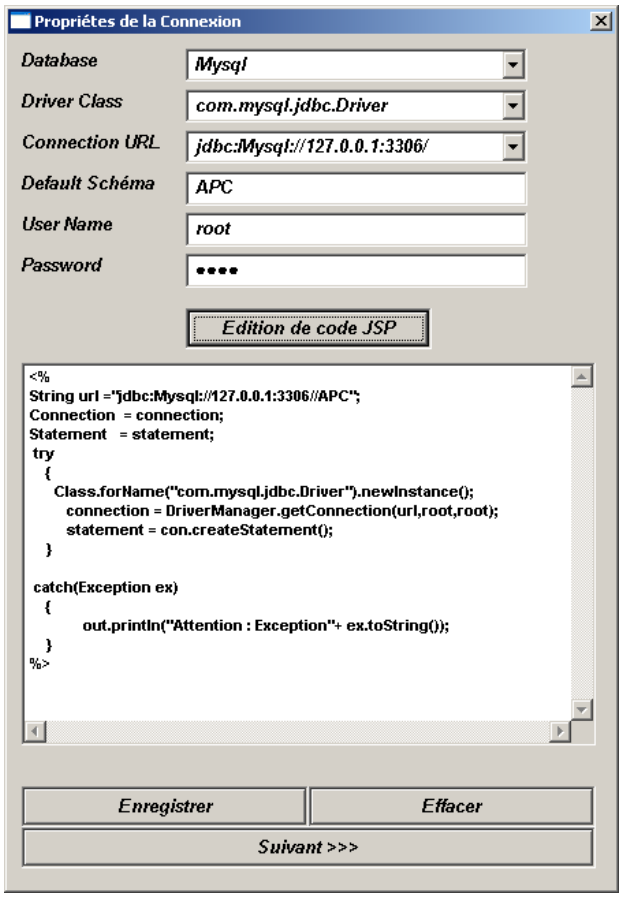

Figure 5.30 : Propriétés de la connexion.

Il faut spécifié dans cette page le SGBD utilisé par la page JSP et le driver correspondant, il faut aussi indiquer l'url et le nom de la base des donnée (ici APC) ainsi que l'utilisateur et le mot de passe, un code est généré automatiquement sur la base des informations déjà indiquer par l'architecte, ce dernier est inséré directement dans le composant qui représente la page JSP correspondante.

En cliquant sur « Suivant » la fenêtre de la figure 5.31 qui indique les propriétés des fichiers est apparaître :

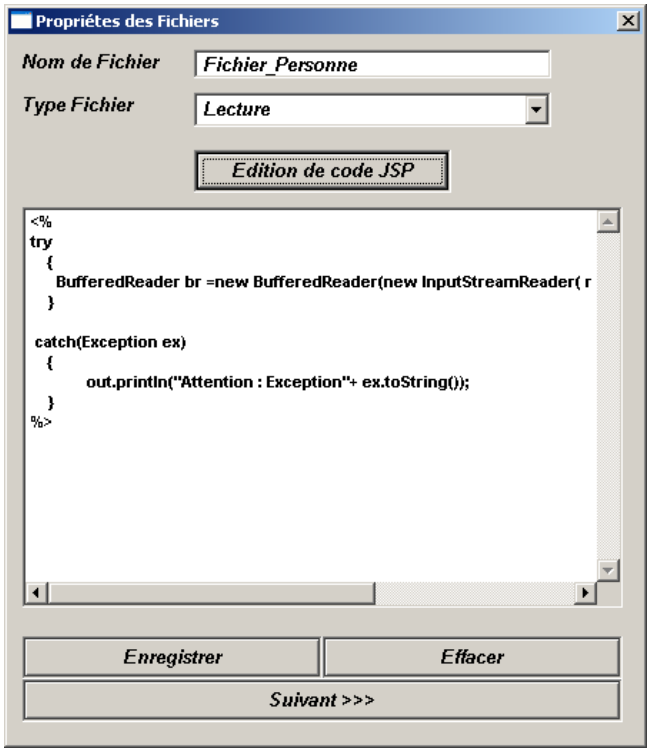

Figure 5.31 : Propriétés des fichiers.

Il suffit simplement d'indiquer le nom de fichier et le type de flux à ouvrir :

- Lecture,
- Ecriture.

En cliquant sur « Suivant » la fenêtre suivante est apparaître :

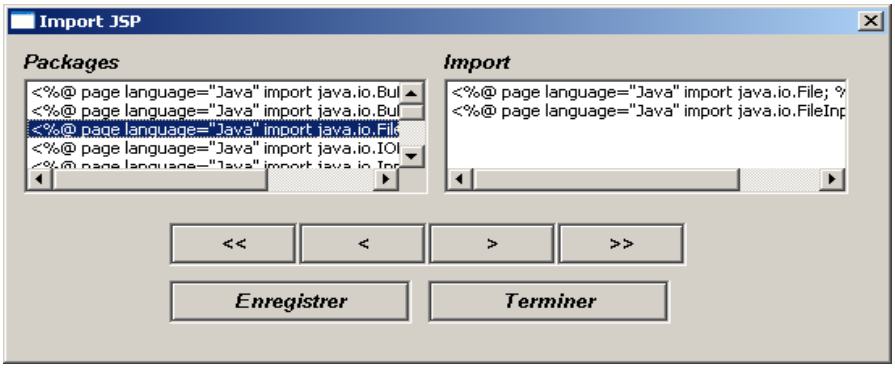

Figure 5.32 : Importation des librairie JAVA pour le code JSP

Cette étape permet à l'architecte de spécifier les packages et les librairies à importer.

On clique en fin sur « Terminer » et on refait le même travail avec les autre composant de type Page JSP.

Et enfin le code JSP générer de composant Gen_Résidence.jsp est présenté dans la figure 5.33.

Le code archjava et le code de la spécification textuelle sont aussi générer automatiquement et respecte toujours la structure de l'architecture et de modèle IASA.

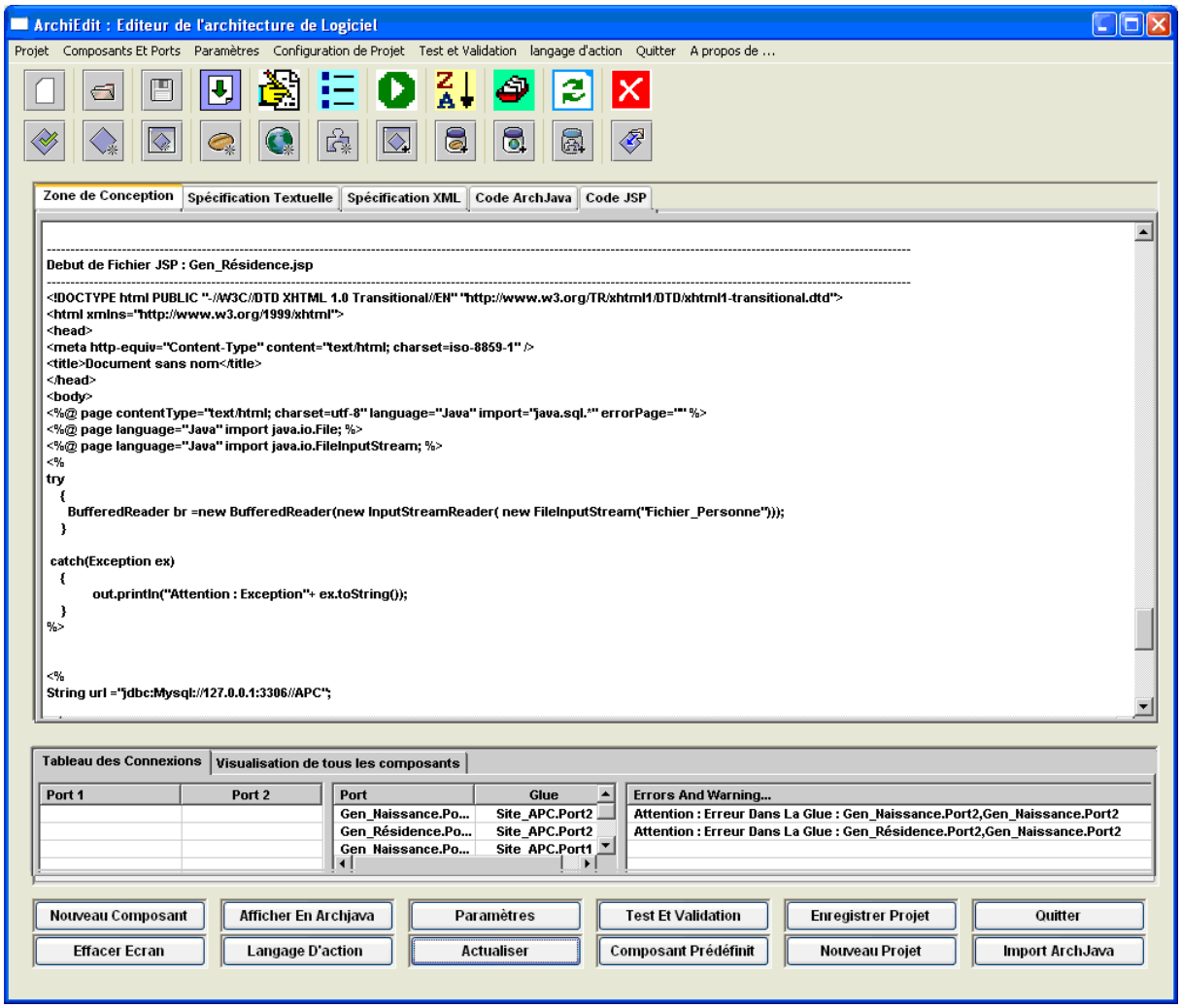

Figure 5.33 : Le code JSP générer de composant Gen_Résidence.jsp

Cas 2 :

L'architecture est composer de plusieurs composant dont certains représente des pages JSP et d'autre non, dans ce cas avant de passer à l'implémentation des composants primitifs il faut choisir entre deux types d'implémentation de composant :

- Soit Implémentation Archjava.
- Soit Implémentation JSP.

La fenêtre suivante permet de faire ce choix :

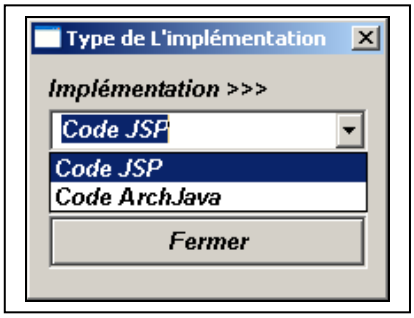

Figure 5.34 : Le choix entre les deux types d'implémentation.

Le traitement et la génération de code JSP reste le même comme il est déjà présenté, et pour le code archjava le traitement ne change pas.

5.3.1. Le code ArchJava générer pour l'architecture (Site_APC) :

voici ci – dissous Le code ArchJava générer pour l'architecture « Site_APC »

```
public component class Site_APC
{
Private final Gen_Naissance Cmp01 = new Gen_Naissance( ) ;
Private final Gen_Résidence Cmp11 = new Gen_Résidence( ) ;

Glue Port1 to Gen_Naissance.Port1;
Glue Port1 to Gen_Résidence.Port1;
Glue Port2 to Gen_Naissance.Port2;
Glue Port2 to Gen_Résidence.Port2;

Public Port Port1 ;
   {     provide  void methode_( );    }

Public Port Port2 ;
   {     requiers void methode_( );    }
```

```
public void Execute()
  { // Fonction permet de lancer les traitements des sous composants;   }
}

public component class Gen_Naissance
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
 }
public component class Gen_Résidence
 {
    public port Port1
          {provide Int  Action_6 (); }
    public port Port2
          { requiers Int  Action_7 (); }
 }
```

Figure 5.35 : Le code ArchJava générer pour l'architecture « Site_APC ».

5.3.2. Le code JSP générer pour l'architecture (Site_APC) :

La figure 5.36 montre le code JSP générer pour l'architecture « Site_APC ».

```
--------------------------------------------------------------------------
Debut de Fichier JSP : Gen_Naissance.jsp
--------------------------------------------------------------------------
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Document sans nom</title>
</head>
<body>
<%@ page contentType="text/html; charset=utf-8" language="Java" import="java.sql.*"
errorPage="" %>
<%@ page language="Java" import java.io.BufferedReader; %>
<%@ page language="Java" import java.io.BufferedWriter; %>
<%@ page language="Java" import java.io.File; %>
<%@ page language="Java" import java.io.FileInputStream; %>
<%@ page language="Java" import java.io.FileOutputStream; %>
<%@ page language="Java" import java.io.IOException; %>
<%@ page language="Java" import java.io.InputStreamReader; %>
<%@ page language="Java" import java.io.OutputStreamWriter; %>
<%@ page language="Java" import java.sql.*; %>
<%@ page language="Java" import java.sql.DriverManager; %>
<%@ page language="Java" import java.sql.Connection; %>
<%@ page language="Java" import java.sql.Statement; %>
<%
try
    {
      BufferedReader br =new BufferedReader(new InputStreamReader( new
FileInputStream("Fichier_Personne")));
    }

 catch(Exception ex)
    {
              out.println("Attention : Exception"+ ex.toString());
    }
%>
<%
String url ="jdbc:Mysql://localhost:3306//APC";
Connection  = connection;
```

```
Statement   = statement;
 try
    {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
          connection = DriverManager.getConnection(url,Root,root);
          statement = con.createStatement();
    }

 catch(Exception ex)
    {  out.println("Attention : Exception"+ ex.toString());   }
%>
</body>
</html>
------------------------------------------------------------------------
Debut de Fichier JSP : Gen_Résidence.jsp
------------------------------------------------------------------------
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Document sans nom</title>
</head>
<body>
<%@ page contentType="text/html; charset=utf-8" language="Java" import="java.sql.*"
errorPage="" %>
<%@ page language="Java" import java.io.BufferedReader; %>
<%@ page language="Java" import java.io.BufferedWriter; %>
<%@ page language="Java" import java.io.File; %>
<%@ page language="Java" import java.io.FileInputStream; %>
<%@ page language="Java" import java.io.FileOutputStream; %>
<%@ page language="Java" import java.io.IOException; %>
<%@ page language="Java" import java.io.InputStreamReader; %>
<%@ page language="Java" import java.io.OutputStreamWriter; %>
<%@ page language="Java" import java.sql.*; %>
<%@ page language="Java" import java.sql.DriverManager; %>
<%@ page language="Java" import java.sql.Connection; %>
<%@ page language="Java" import java.sql.Statement; %>
<%
try
    {
      BufferedReader br =new BufferedReader(new InputStreamReader( new
FileInputStream("Fichier_Personne")));
    }

 catch(Exception ex)
    {  out.println("Attention : Exception"+ ex.toString());  }
%>
<%
String url ="jdbc:Mysql://localhost:3306//APC";
Connection  = connection;
Statement   = statement;
 try
    {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
          connection = DriverManager.getConnection(url,Root,root);
          statement = con.createStatement();
    }

 catch(Exception ex)
    {   out.println("Attention : Exception"+ ex.toString());  }
%>
</body>
</html>
```

Figure 5.36 : Le code JSP générer pour l’architecture « Site_APC ».

CHAPITRE 6
EVALUATION DE LANGAGE SEAL

6.1. Introduction :

Le langage SEAL a été évalué dans le cadre de la réalisation d'une application de gestion commerciale des produits du réseau X25. Dans le processus de réalisation de cette application nous avons mis en en œuvre les concepts de l'approche IASA (les modèles de composant, port et connecteur, et le processus de conception) ainsi que les divers concepts du langage SEAL.

Une première cible du processus de transformation de la spécification architecturale en une description dans un langage de programmation a aussi été évaluée dans ce travail. Cette première cible est représentée par langage ArchJava. Le choix de ArchJava est dû principalement à sa capacité de représenter les concepts de composants et de connecteurs.

6.2. Présentation de l'application de gestion commerciale des produits de réseau X25 :

Le système de gestion commerciale de réseau X25 a pour objectif de générer les factures des clients de l'opérateur utilisant le réseau à partir des tickets de taxations émis et des données clients. De plus, il met à disposition un fichier contenant les informations nécessaires au service comptable.

Le but est de mettre en place un site capable de facturer les clients de l'opérateur quelque soit le réseau utilisé.

Les données en entrée du système sont :

- Les tickets de taxations de divers constructeurs.
- Les données clients saisies par l'interface de l'application.
- Le plan tarifaire (tarifs, plages horaires….).
- Le paramétrage du système.

Les autres données en entrée correspondent à la saisie des services ou des diverses données de configuration.

Les types de données intermédiaires présentes dans le système :

- Tickets de taxations réceptionnés,
- Tickets de taxations valorisés et réconciliés,
- Les Tickets de taxations valorisés sont conservés puis stockés pendant une période donnée.

Les données en sortie du système sont :

- Impressions des factures des clients facturables.
- La facture telle qu'elle a été envoyée au client (visualisable par un opérateur).
- La facture détaillée de chaque client facturé (visualisable par un opérateur).
- Mise à la disposition de la comptabilité, d'informations sur le montant des factures et les clients.
- Le journal du système et les alarmes enregistrées.

6.3. Architecture de réseau X25 :

Le protocole X.25 est un protocole relatif à la commutation de paquets. Il a été adopté en 1976 par le CCITT (consultative committee on international télégraphy and téléphony). Parmi les réseaux internationaux utilisant X25 nous citons Transpac en France et géré par France Télécom, EPSS pour TCTS (Trans Canada Telephone System) au Canada. Datapac et Telenet pour Telenet Communication Corps aux USA. En Algérie le premier réseau X25 porte le nom de DZPAC et qui était géré par le ministère des PTT.
Ce réseau est actuellement géré par *Algérie Télécom*. Cette dernière a depuis quelque année installée un deuxième réseau X25, et appelé Mégapac permettant d'offrir des débits plus important que le DZPAC.

Le protocole X.25 contient les trois premières couches du protocole, à savoir [URL04]:

- La norme X.21 pour le niveau physique.
- Le sous-ensemble LAP-B de la norme HDLC pour la couche liaison.
- La norme X.25 pour la couche réseau.

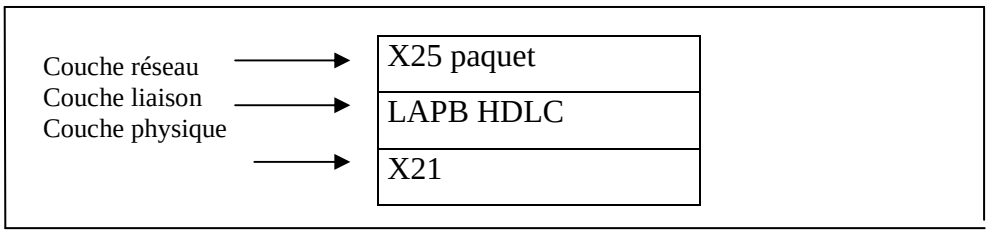

Figure 6.1 : Couche réseau.

La recommandation X.25 défini un protocole entre l'interface ETTD (Equipement Terminal de Traitement de Donnée) et un ETCD (Equipement de terminaison de circuit de données) pour la transmission de paquets [URL04].

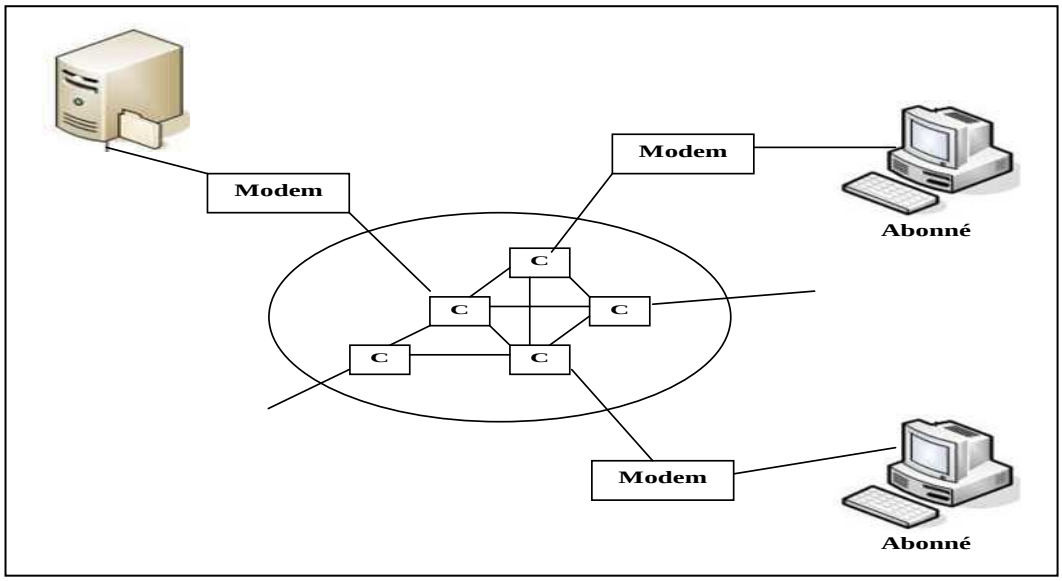

Figure 6.2 : La transmission de paquets.

Comme tout réseau numérique, le réseau X25 est doté d'une station spéciale dite station de supervision. Une station de supervision est un ordinateur ordinaire doté généralement d'une grande capacité de stockage. Le logiciel de supervision gère le réseau et réalise l'opération de collecte de ticket de taxation à partir de tous les commutateurs du réseau. Souvent dans un même réseau nous trouvons 2 stations de supervision. Une station principale et une station de secours.

Dans le cas de *Algérie Télécom*, les deux réseaux appartiennent à deux compagnies différentes et utilisent des technologies différentes. Ceci se répercute sur les tickets de taxation. Ainsi *Algérie Télécom* est confronté à deux formats différents de ticket. Un format *SAGEM* et un format *ALCATEL*

6.4. Objectif du système de gestion commerciale des produits de réseau X25:

6.4.1. Objectifs principaux :

Les objectifs principaux du système de gestion commerciale des produits du réseau X25 sont :

- De générer les factures des clients du réseaux à partir des tickets de taxations émis et des données clients.
- De mettre à disposition un fichier contenant les informations nécessaires au service comptable.
- De réaliser le suivi des paiements.

Les autres objectifs sont :

- Le suivi des opérations anormales du système par la mise en place de politique de déclenchement d'alarme et de niveau de gravité de ces alarmes.
- L'établissement de diverses statistiques concernant l'exploitation du réseau.

6.4.2. Objectifs complémentaires :

Pour arriver à ces objectifs il faudrait avoir les éléments complémentaires suivants :

- La gestion des clients.
- La gestion des abonnements. Un client peut avoir 0, 1 ou plusieurs abonnements.
- La définition des divers produits à commercialiser : ce sont notamment des services et la location d'équipements.
- La définition des tarifs des produits à commercialiser.

- La définitions de la manière avec laquelle un produit est commercialisable : Volume d'informations, temps de connexion, location à des prix forfaitaires etc.
- La détermination des produits consommés. Ceci se fait essentiellement à partir de deux éléments fondamentaux :

➢ Les tarifs des tickets de taxations. Ces derniers indiquent les volumes, durée et qualité des services consommés.

➢ Les tarifs fixes liés aux droits d'abonnement, à la location des équipements et aux services complémentaires.

Le schéma suivant récapitule les fonctionnalités générales du système visé :

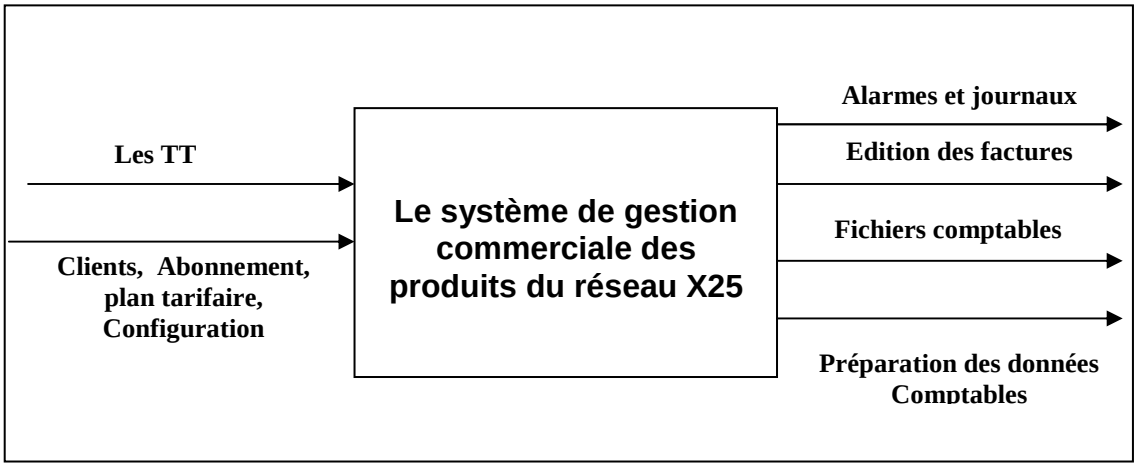

Figure 6.3 : Vue générale et informelle du système

6.5. Architecture informelle du Système :

Voici l'architecture informelle du Système de gestion des produits de réseau X25.

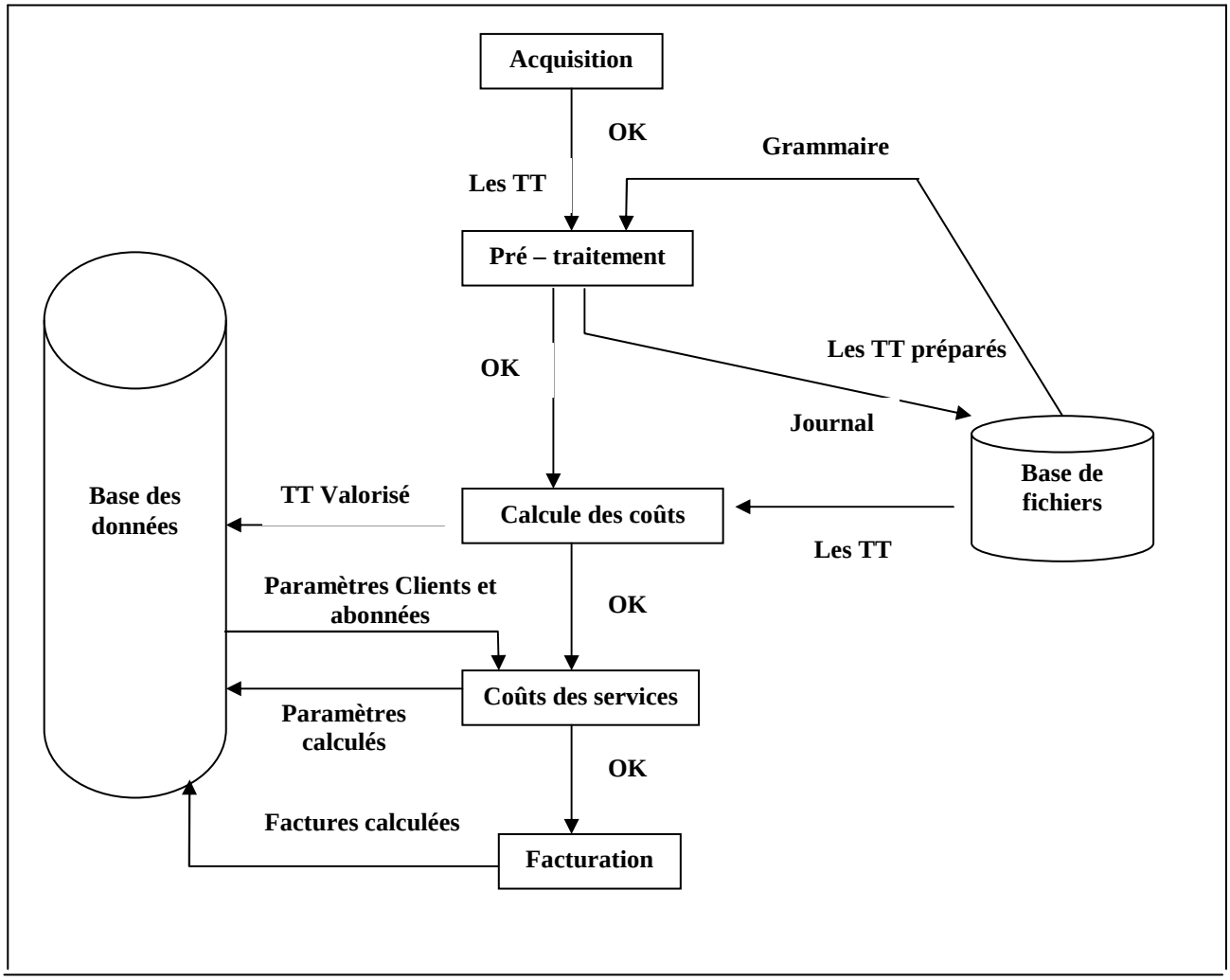

Figure 6.4 : Architecture générale du système

6.6. Processus de développement :

Pour l'évaluation de notre approche nous avons suivi le processus de conception de l'approche IASA [15]. Ce processus comprend les grandes phases suivantes :

Etape 1 : Initialisation de projet:

Projet AP, AD

AP est le nom du composant à réaliser (Une application dans notre approche est un type composant)

AD est l'architecture de déploiement que le composant peut exploiter

1. Définition de la vue externe (Prise en charge des besoins)
   1.1. Définition des ports : détermination des services fournis et requis et des contextes d'action
2. Soit LCMP une liste de couple (composant, architecture de déploiement) à réaliser.
   LCMP = {(AP, AD)}

Etape 2 :

Pour tout couple (APX, ADX) de LCMP (choisir le couple. Au départ c'est le composant correspondant au projet et qui est la racine de l'arbre de conception)

1 Elaboration de la vue Interne de APX

Le contrôleur est un élément fondamental de la vue interne, définir la vue interne en prenant en considération le rôle du contrôleur

TANT QUE la vue n'est pas stable

(les actions suivantes ne sont pas obligatoirement ordonnés)

- Identifier les types de composants nécessaires pour APX : Créer de nouveau type par généralisation plutôt que de modifier des types instancié (modification de port)
- Définir les vues externes des nouveaux types de composant (Port, contexte d'action) (Ceci est le cas dans une approche top down)
- Etablir/Modifier/Supprimer les connexions
- Etablir les contextes de connecteurs et de composite
- Définition et ajustement du comportement du composant *OpPartController* de la partie contrôle.
- Vérifier la stabilité du composant (application aux instances d'enveloppes marquées)
- Réajuster les vues externes des nouveaux types de composants
- Définir le cas de déploiement des instances selon AD
- Ajouter les nouveaux types à une liste intermédiaire ILCMP de couple Composant, Architecture de déploiement

2 Fin d'une itération: le composant est stable

3 Ajouter les nouveaux types de composant à LCMP: LCMP = LCMP + ILCMP.

4 Refaire à partir de 1

Et voici l'application de ce processus sur le système de gestion commerciale des produit de réseau X25.

Etape1 : initialisation de projet.

Le nom CFX25 est le nom du composant global qui représente l'application de gestion commerciale des produits de réseau X25.

Cette application doit être distribuée et déployée dans les environnements *Unix* et *Windows*. Les données importantes produites par ce système seront mises dans une station *Linux*. Celle ci dispose du *SGBD Oracle* (Oracle 8i).

L'application X25GC doit être distribuée et déployée dans les environnements ***Unix*** (*Solaris, Linux*) et ***Windows*** (Windows XP). Les composants lourds, nécessitant un temps d'exécution assez long et produisant un nombre important de données, seront mis dans une station *SUN* sous *Solaris*. Celle – ci dispose du *SGBD Oracle* (Oracle 8i). Les machines *Windows* et *Unix* et les stations de supervision, sont interconnectés dans le contexte d'un réseau IP privé (Figure 6.2).

Etape 2 – 1ère ITERATION:

1. Elaboration de la vue externe de projet CFX25 :

La vue externe de projet CFX25 selon l'approche IASA est présentée dans la figure suivante : (figure 6.5).

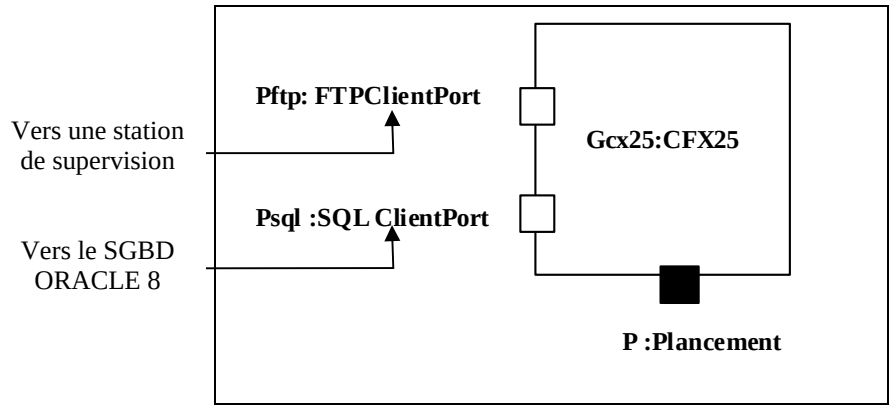

Figure 6.5 : La vue externe de projet CFX25 selon l'approche IASA

La vue externe de l'application est représentée par le composant global CFX25 qui contient les ports suivant :

- ✓ *FTPClientPort* : est de type ClientPort pour l'acquisition des données a partir des deux stations de supervision, ce port utilise le protocole de transfert FTP.
- ✓ *SQLClientPort* : est de type ClientPort pour toutes les opérations d'accès à la base des données (insertion, suppression, modification, consultation).
- ✓ *Plancement* : est de type ServerPort pour le lancement de l'application.

2. Elaboration de la vue interne du composant CFX25 :

La vue interne du composant CFX25 est montrée dans la figure 6.6.

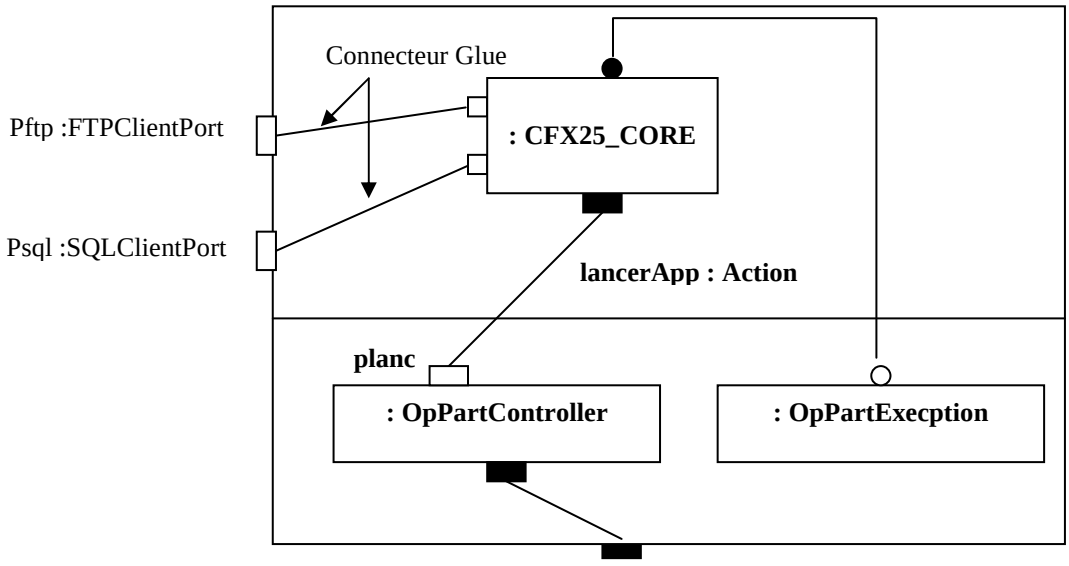

Figure 6.6 : La vue interne de composant CFX25.

2.1. Détermination des composants nécessaires à la réalisation du composant CFX25 :

Nous avons identifié un seul composant, pour cela nous définissons un type de composant :

CFX25_CORE : représente le corps de l'application de la gestion commerciale.

2.2. Définition de la vue externe des sous composants :

Le composant CFX25_CORE représente l'application complète de gestion commerciale de réseau X25, elle est dotée d'un ensemble de ports pour la communication avec le composant englobant, le port de lancement permet de lancer l'application et celui

d'exception permet d'envoyer toutes les exceptions vers la partie contrôle (composant OpPartException).

2.3. Définition des contextes d'actions :

Nous définissons les contextes particuliers aux interactions, ce sont des contextes qui seront affectés aux ports aux connecteurs.

Les actions d'accès au système de fichier sont primitives et correspondent à des opérations d'entrée sortie directement supportées par le langage cible.
CFX25Context c'est le contexte général du composant composite *CFX25* il est composé de trois contexte d'actions de base de SEAL:

- FtpIntercationContext,
- SQLIntercationContext,
- FSContext.

2.4. Comportement du composant de contrôle :

En SEAL le composant comportemental (*OpPartController*) est un composant primitif possédant des ports spécifiques pour le chargement de comportement à partir d'une source externe, la figure 6.7 montre un exemple de chargement de comportement pour le composant comportemental *OpPartController* :

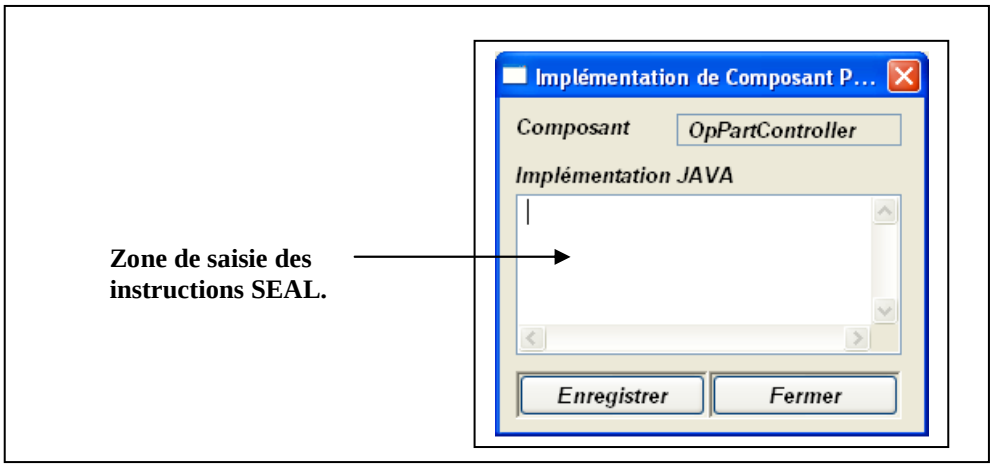

Figure 6.7 : Exemple d'implémentation d'un composant primitif.

Le composant de contrôle (*OpPartController*) aura les instructions de contrôle de ce composant, ces instructions sont présentées par la figure 6.8.

```
Planc.request LancerApp  // Instruction de contrôle qui permet de lancer
                         //  L'application.
```

Figure 6.8 : les instructions de contrôle de composant *OpPartController*.

La figure 6.9 montre le comportement de composant *OpPartController* avec l'outil IASASTUDIO.

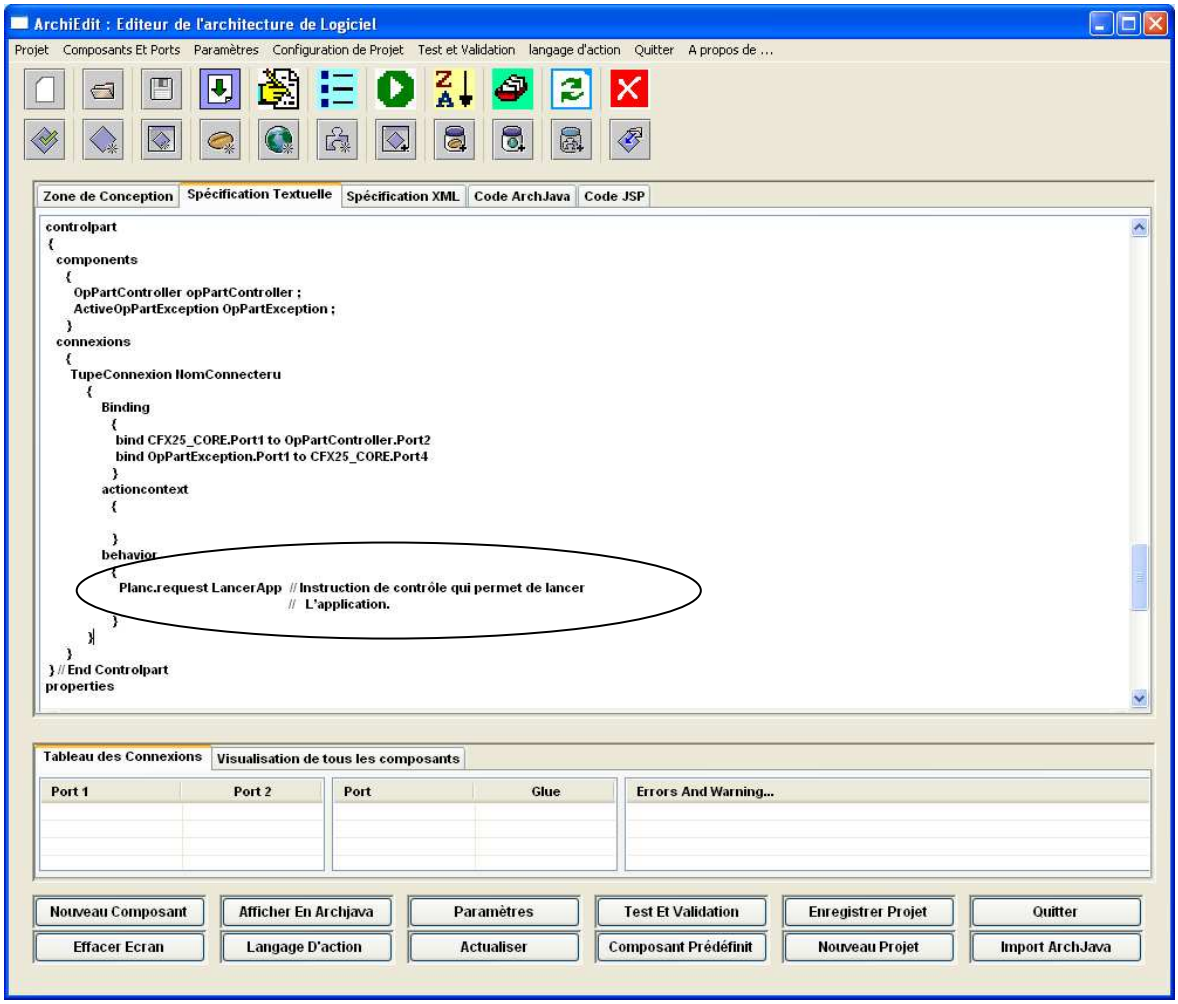

Figure 6.9 : Le comportement de composant *OpPartController*
Avec l'outil IASASTUDIO.

Le code ci – dissous montre une partie de code qui concerne les règles appliquées au niveau des ports, ici nous avons pris le port Port2 de type ClientPort.

```
ClientPort Port2
   {
    accesspoint
            {
              ClientActionPoint Point3  Nom_Methode(0,SYNCHRONE);
              DataPoint Point4  Nom_Methode(0,SYNCHRONE);
            }
    actioncontext
      {
            // context d'action.
      }
    behavior
   {
     rules  régle1;          // liste des règles
     rule régle1             // spécification de la premier règle.
      {
       precondition :
       pattern :
       postcondition :
      }
  }
 }
```

Figure 6.10 : Les règles appliquées au niveau des ports.

La figure 6.11 montre les règles appliquées au niveau des ports avec l'outil IASASTUDIO.

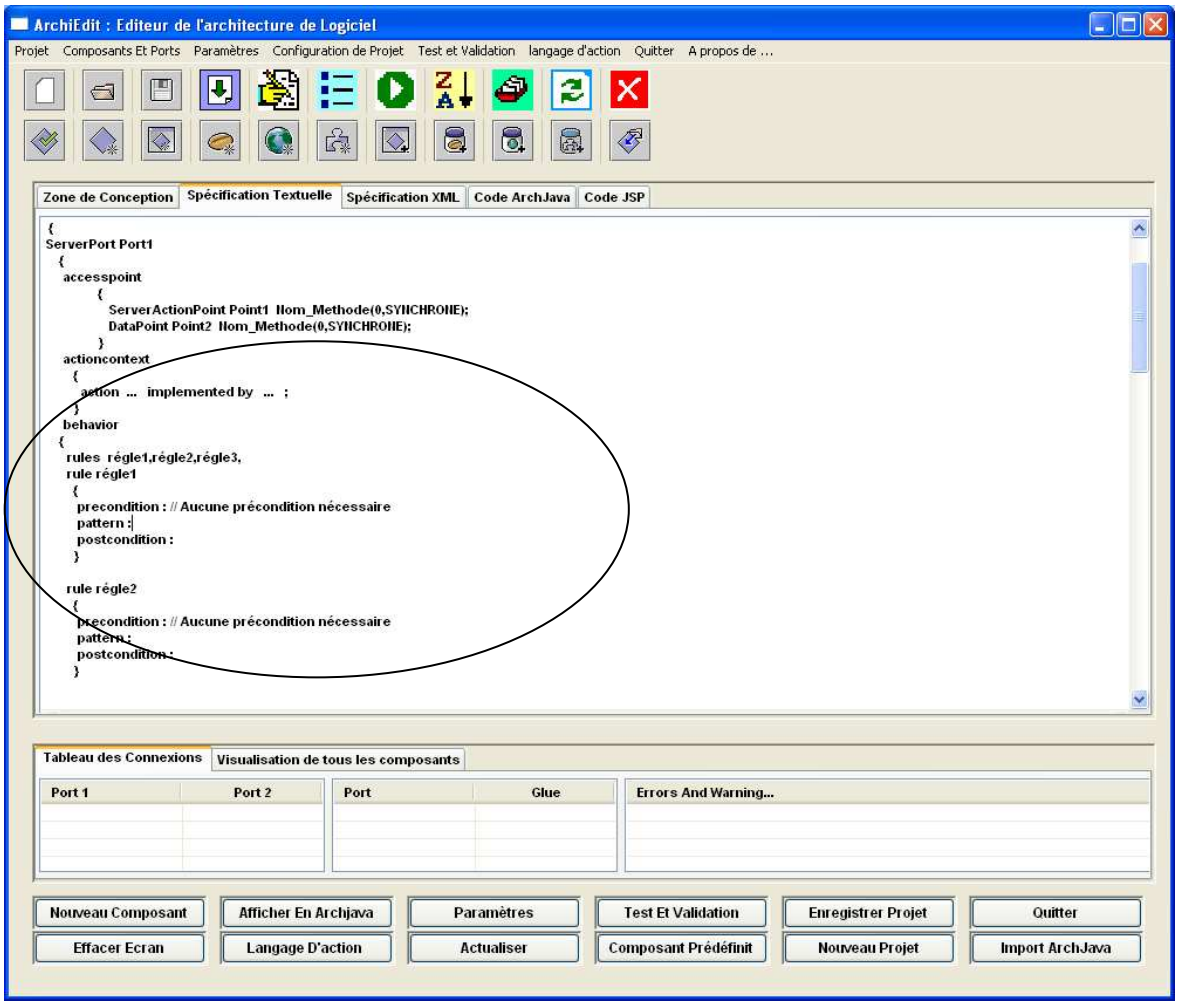

Figure 6.11 : Les règles appliquées au niveau des ports (IASASTUDIO).

2.5. Etude de la stabilité :

Nous rappelons que les enveloppes marquées sont utilisées dans le processus de vérification de l'état global (stabilité) d'un composant. Il est possible de marquer totalement ou partiellement les points d'accès. Le marquage d'un point d'accès signifie que ce dernier est correctement connecté. Une enveloppe totalement marquée est une enveloppe dont tous les points d'accès sont marqués. Dans une enveloppe marquée partiellement vers l'extérieur, seuls les services requis *ClientDataPoint* et les *DataPoint* dont le sens est *in* ou *inout* sont marqués. Dans une enveloppe marquée partiellement vers l'intérieur, seuls les services fournis *ServerDataPoint* et les *DataPoint* dans le sens est *out* ou *inout* sont marqués.

L'étude de stabilité du composant *X25GC_CORE* consiste à appliquer une enveloppe marquée vers l'extérieur pour le composite et des enveloppes marquées vers l'intérieur pour chaque instance de la partie opérative et contrôle (*X25GC_ CORE* et *OpPartController*).

Après avoir appliqué ces enveloppes, il suffit de trouver au moins un port requérant un service ou une donnée (port client ou un port de données dont le sens est *in*) non connecté à un port marqué pour décider que le composite est instable.

3. Réalisation avec l'outil IASASTUDIO :

3.1. Schéma générale de l'architecture :

Le schéma général de l'architecture est présenté dans la figure 6.12.

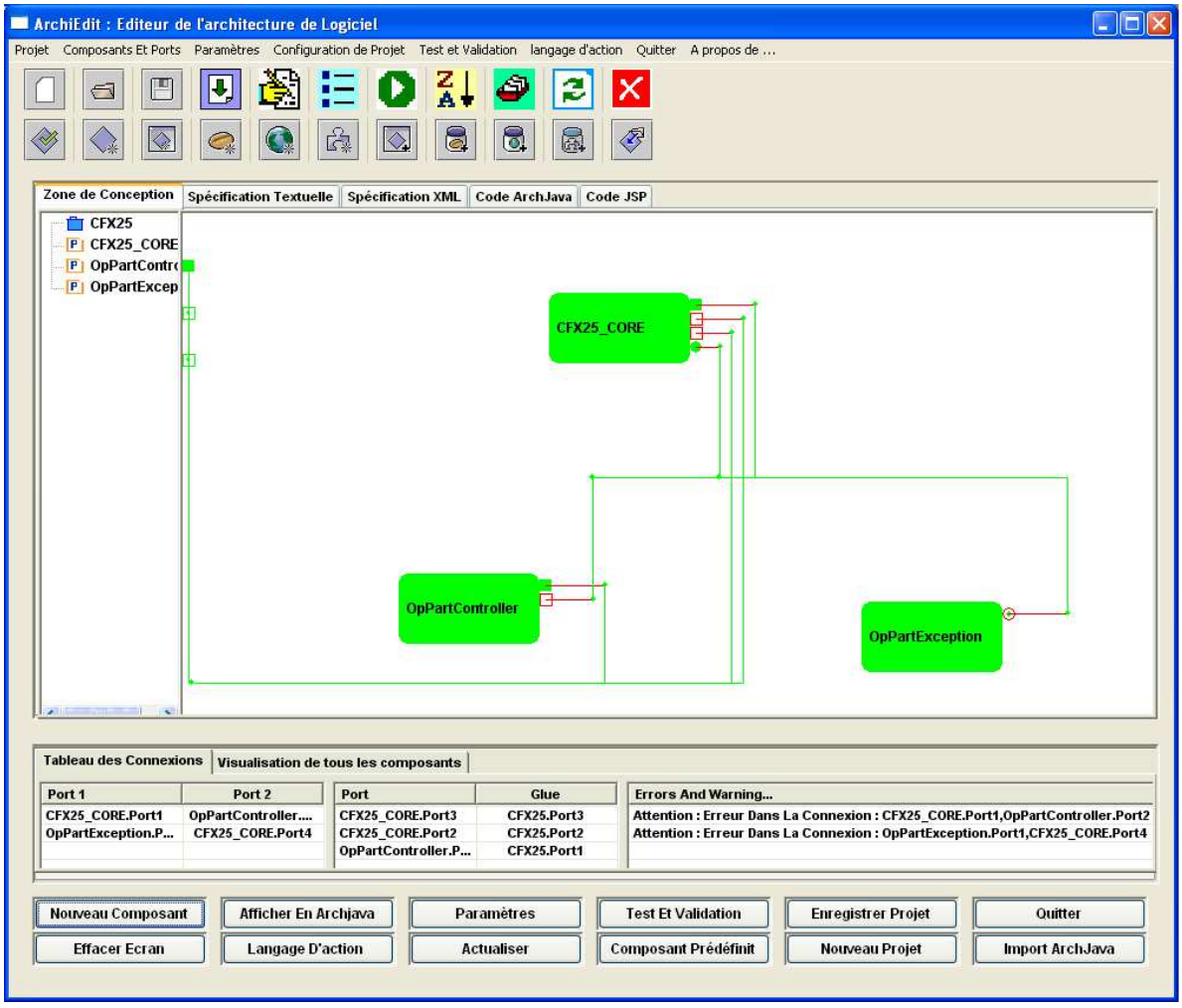

Figure 6.12 : Schéma générale de l'architecture.

Le détail et la spécification complète sont dans l'annexe_1 codes 7.1.

3.2. Génération de code ArchJava :

Pour transformer une conception selon l'approche intégrée dans un langage cible, il est nécessaire de définir un certain nombre de règles de transformation. Ces règles concernent les points suivants :

- Les ports
- Les connecteurs simple et de délégation,
- Les composants.

En ArchJava il existe deux type de ports : les ports provides et les ports requires, donc nous avons converti les ports de notre approche de type « ServerPort » au type provides dans ArchJava et les ports de notre approche de type « ClientPort » au type requires dans ArchJava.

Le code ArchJava associé à l'architecture est présenté dans la figure ci – dissous (figure 6.13).

```
public component class CFX25
{
Private final CFX25_CORE Cmp01 = new CFX25_CORE( ) ;
Private final OpPartController Cmp11 = new OpPartController( ) ;
Private final OpPartException Cmp21 = new OpPartException( ) ;

Connect CFX25_CORE.Port1, OpPartController.Port2;
Connect OpPartException.Port1, CFX25_CORE.Port4;

Glue Port3 to CFX25_CORE.Port3;
Glue Port2 to CFX25_CORE.Port2;
Glue Port1 to OpPartController.Port1;

Public Port Port1 ;
  {    provide  void methode_( );   }
Public Port Port2 ;
  {    requiers void methode_( );   }
Public Port Port3 ;
  {    requiers void methode_( );   }
public void Execute()
 { // Fonction permet de lancer les traitements des sous composants;   }

}

public component class CFX25_CORE
 {
   public port Port1
        {provide Int  Action_1 (); }
   public port Port2
```

```
        {provide Int  Action_2 (); }
    public port Port3
        {provide Int  Action_3 (); }
    public port Port4
        {provide Int  Action_4 (); }
   public void CFX25_CORE()
   {

   }
 }
public component class OpPartController
 {
    public port Port1
        {provide Int  Action_1 (); }
    public port Port2
        {requiers Int  Action_2 (); }
   public void OpPartController()
   {

   }
 }
public component class OpPartException
 {
    public port Port1
        {provide Int  Action_1 (); }
   public void OpPartException()
   {
   }
 }
```

Figure 6.13 : Le code ArchJava de composant CFX25.

La figure suivante montre la génération de code ArchJava avec IASASTUDIO.

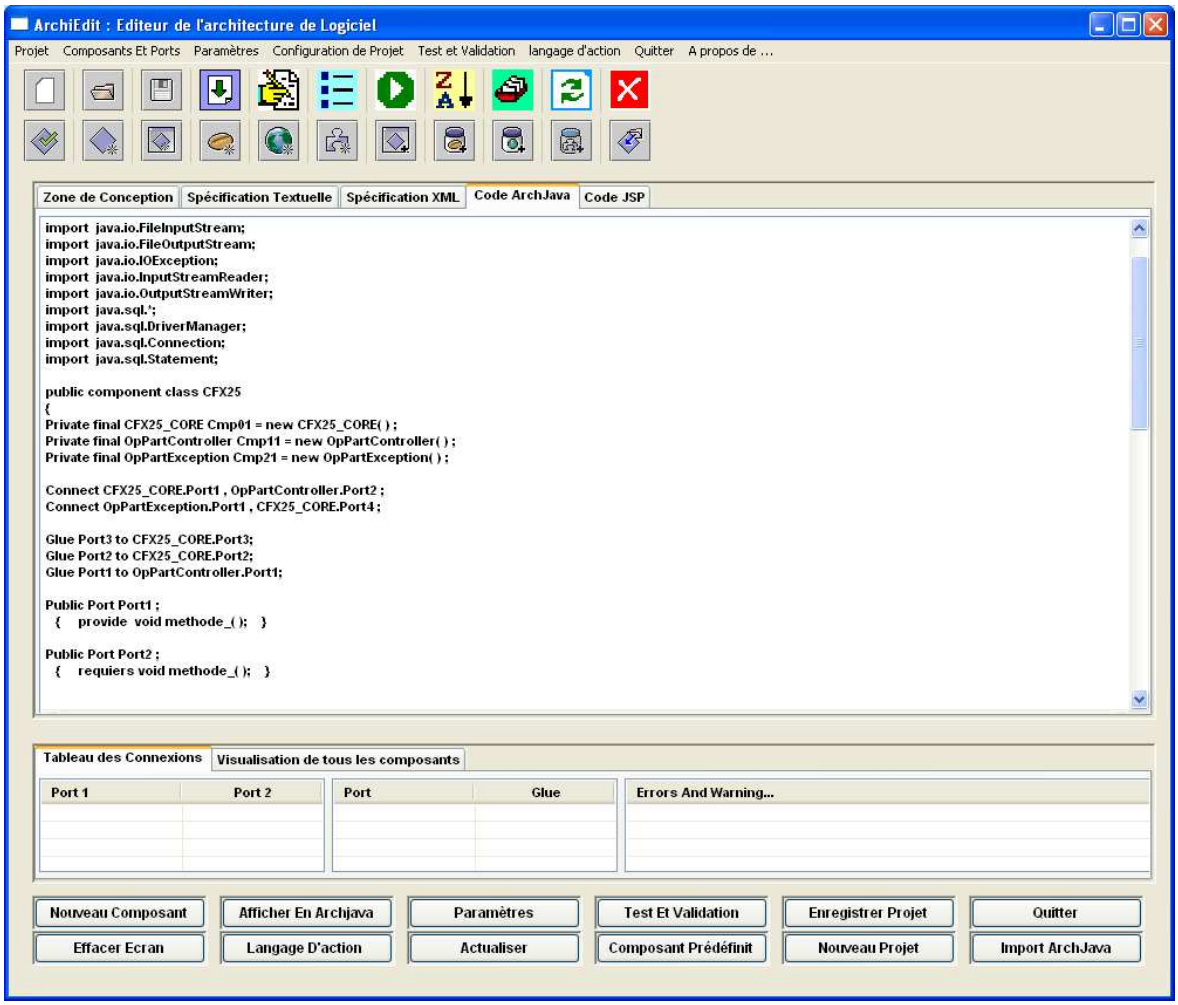

Figure 6.14 : Le code ArchJava de composant CFX25 avec IASASTUDIO.

3.3. Spécification XML :

Pour la spécification XML on a utilisé un ensemble de balise de code xml pour définir notre architecture, ces balises sont illustrées dans le tableau ci – dessous :

Les balises XML	Significations
<Architecture>…..</Architecture>	L'architecture.
<Component Type = '' Width = '' Height = '' X = '' Y = '' Name = ''> </Component >	Les composan.
<Port Type = '' Height = '' Width = '' X = '' Y = '' Name = " />	Les ports.
<AccessPoint Type = '' Action Name = '' ParamaterType =	Les points d'accès.

' ' />	
<Connector Type = 'Binding' Name = ' '> <ConnectedPort>Component1->Port1</ConnectedPort> <ConnectedPort> Component2- >Port2</ConnectedPort></Connector>	Les connecteurs.
<DelegatedPort Type = ' ' Height = ' ' width = ' ' Name = ' '> <PortRef> Component1->Port3</PortRef> </DelegatedPort>	Les connecteurs de délégations.

Tableau 6.1 : signification des balise XML.

La figure 6.15 montre un exemple de code XML associé à cette l'architecture.

```
<Architecture>
<Composite Type = 'CFX25' Name = 'CFX25' >
<Component Type = 'CFX25_CORE' Width = '120' Height = '60' X = 312 Y = 68 Name =
'CFX25_CORE'>
<Port Type = 'ServerPort' Height = '10' Width = '10' X = '487' Y = '78' Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint' Action Name = 'Action_1' ParamaterType = 'Int' />
</Port>
<Port Type = 'ServerPort' Height = '10' Width = '10' X = '477' Y = '90' Name = 'Port2'/>
<AccessPoint Type = 'ServerActionPoint' Action Name = 'Action_2' ParamaterType = 'Int' />
</Port>
<Port Type = 'ServerPort' Height = '10' Width = '10' X = '467' Y = '102' Name = 'Port3'/>
<AccessPoint Type = 'ServerActionPoint' Action Name = 'Action_3' ParamaterType = 'Int' />
</Port>
<Connector Type = 'Binding' Name = 'Connector1'>
<ConnectedPort>CFX25_CORE->Port1</ConnectedPort>
……<DelegatedPort Type = 'DelegatedPort' Height = '10' width = '10' Name = 'Port2'>
<PortRef>CFX25_CORE->Port2</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort' Height = '10' width = '10' Name = 'Port1'>
<PortRef>OpPartController->Port1</PortRef>
</DelegatedPort>
</Composite>
</Architecture>
```

Figure 6.15 : Spécification XML.

4. Mis à jour de la liste des types de composant à réaliser :

Après l'étude de la stabilité nous arrivons à la fin de la 1er itération, nous définissons donc les futurs composants à réaliser, dans notre cas la liste des nouveaux composants à réaliser est composée de : CFX25_CORE.

Etape 2 – 2^{em} ITERATION:

1. choix de composant de la liste des types de composant :

Soit donc à réaliser le composant CFX25_CORE (le seul composant de la liste).

2. Elaboration de la vue interne de CFX25_CORE :

La figure 6.16 montre la vue interne de composant composite CFX25_CORE.

2.1. Détermination des composants nécessaires à la réalisation de CFX25_CORE :

Après l'étude et l'analyse nous avons décidé de décomposer le projet en 8 composants :

- Composant Acquisition.
- Composant Préparation.
- Composant Facturation.
- Composant Base des données.
- Composant Alarme.
- Composant Calcule des coûts.
- Composant coûts des services.
- Composant base de fichier.

L'architecture générale de Composant CFX25_CORE est représentée par la figure 6.16.

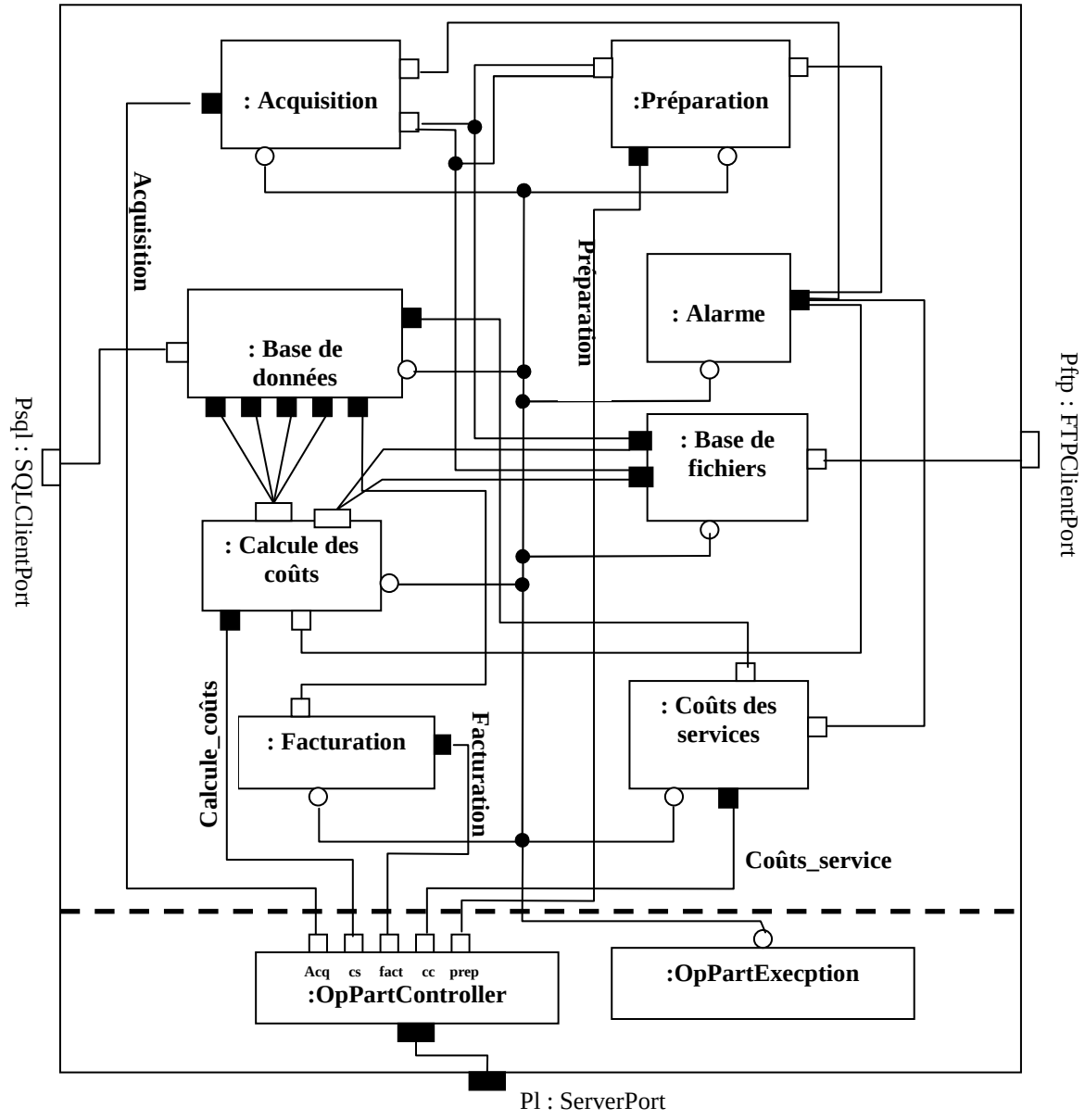

Figure 6.16 : Architecture globale du Composant CFX25_CORE.

La définition des rôles de chaque composant est présenté comme suit :

- ✓ Le Composant Acquisition se chargera de la récupération des fichiers de Tickets de Taxation (TT) en provenance d'une des stations de supervision du réseau X25. La récupération se fait via 2 stations de supervision (SVR). Les 2 stations de supervision reçoivent, simultanément, et au fil de l'eau, les Tickets de Taxation émis par les commutateurs X25. Sur une station de supervision, le fichier de stockage des TT au fil de l'eau ne peut pas être lu directement par le centre de

facturation : il faut passer par un fichier intermédiaire, qui représente l'image de ce fichier de stockage, pris à un instant t.

- ✓ Le composant Préparation a pour but de préparer les tickets de taxation, reçus des réseaux X25 de l'opérateur, pour le sous système « calcule des coûts ».

  La préparation des tickets consiste à :

  - Contrôler s'ils sont exempts d'erreurs dans leur format initial.
  - Transformer les tickets dans le format interne du système et indépendant de tout réseau X25.
  - Contrôler le cas où un fichier de ticket provient d'une station de secours. Dans ce cas il est possible d'avoir des doublons. Il faut donc réaliser une opération de Dé – doublonnage des TT récupérés à partir des deux stations de supervision.

- ✓ La finalité de composant calcule des coûts est d'obtenir en détails les coûts de toutes les connexions aux réseaux de l'opérateur dans le but de pouvoir facturer par la suite les clients concernés. La prochaine étape consiste à valoriser chaque ticket de taxation en se basant sur un plan tarifaire qui aurait été établi au préalable. Cette étape devra se terminer par la production de ticket contenant les informations nécessaires pour être prises en charge par une opération de calcul de la facture. Le ticket dans cette nouvelle forme (nous parlons de tickets enrichi) sera alors stocké dans une base de données dédiée au système de gestion commerciale des produits de X25.

- ✓ L'opération «Coût de service» est l'étape précédant la facturation. Elle est indépendante de l'opération « calcule des coûts » et donc des tickets de taxation. Elle prend en compte les caractéristiques des accès du client. Elle détermine notamment s'il faut éditer une facture pour un client donné et qu'elle est la période prise en compte pour la facturation. Elle prépare les données pour la facturation des accès des clients à facturer.

  Elle consiste donc à :

- Analyse des données d'abonnement.
- Valorisation des mises en service associées au client.
- Valorisation des modifications d'abonnement associées au client.
- Valorisation des abonnements associés au client.

✓ Le Composant Facturation à pour rôle :

- Calcul des valeurs en fonction des données clients (facturation au forfait, réduction, location de modem, vitesse de la ligne, facture détaillée …) et des TT.
- Edition des factures en fonction des données calculées.
- Stockage des factures telles qu'elles sont éditées.
- Constitution d'un fichier récapitulatif des données calculées. Ce fichier sera récupéré par le système comptable.

✓ Le Composant base des données est occupé de toutes les opérations sur les bases des données (insertion, suppression, modification, sélection).
✓ Le composant base des fichiers est occupé e toutes les opérations sur les fichiers (création, suppression, modification, consultation).
✓ Le composant alarme à comme rôle la gestion des alarmes émis par les autres sous composants.

La vue interne du composant CFX25_CORE est composée de 8 composant ordinaire et d'un composant charge de la gestion des exceptions. Dans la suite nous allons définir selon l'approche IASA, la vue externe de chacun de ces composant.

2.2. Définition de la vue externe des sous composants :

Nous présentons dans ce qui suit la vue externe des sous composant préparation, calcule des coûts, coûts des services et facturation.

➢ *Le composant préparation :*

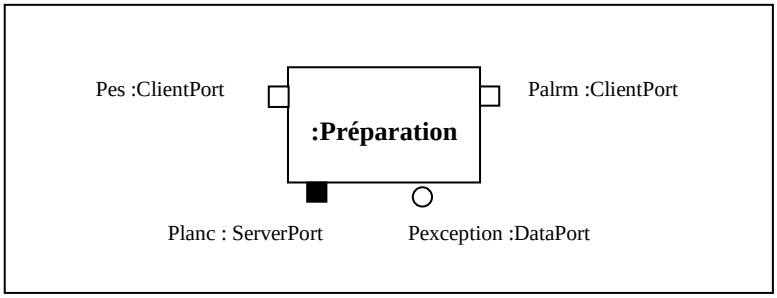

Figure 6.17 : La vue externe de composant préparation.

Ce composant possède quatre ports :

- Un ServerPort pour lancer le composant.
- Un ClientPort pour les opérations sur les fichiers.
- Un ClientPort pour la gestion des alarmes.
- Un DataPort pour le transfert des exceptions.

➢ *Le composant calcule des coûts :*

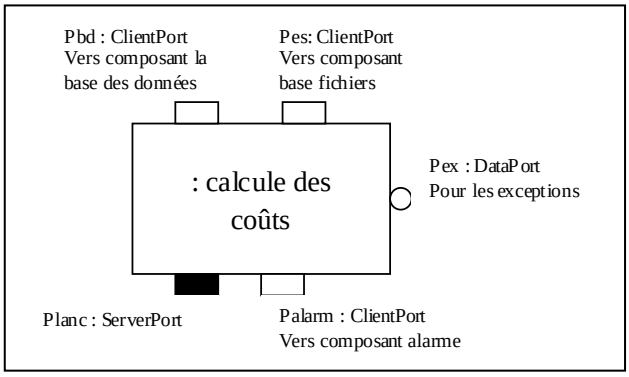

Figure 6.18 : La vue externe de composant calcule des coûts.

Ce composant possède les ports suivants:

- Un ServerPort pour lancer le composant.
- Un ClientPort pour les opérations sur les fichiers.
- Un ClientPort pour les opérations sur les bases des données.
- Un ClientPort pour la gestion des alarmes.
- Un DataPort pour le transfert des exceptions.

➢ *Le composant coûts des services :*

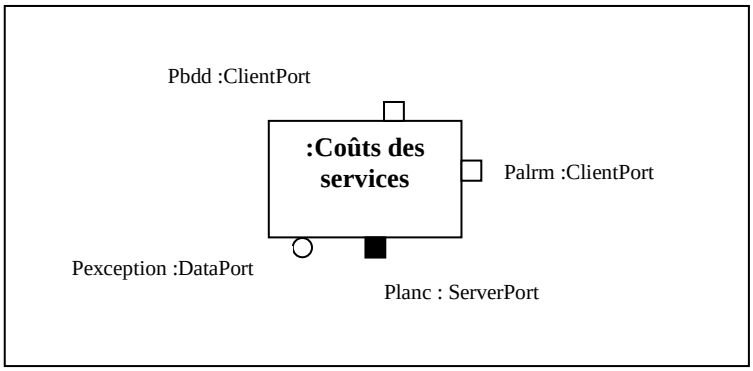

Figure 6.19 : La vue externe de composant coûts des services.

Ce composant possède les ports suivants:

- Un ServerPort pour lancer le composant.
- Un ClientPort pour les opérations sur les bases des données.
- Un ClientPort pour la gestion des alarmes.
- Un DataPort pour le transfert des exceptions.

➢ *Le composant facturation :*

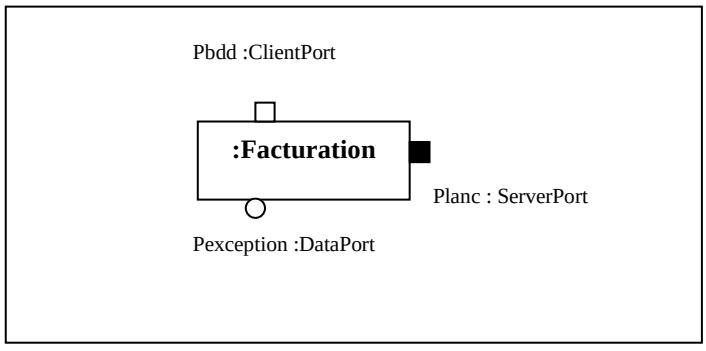

Figure 6.20 : La vue externe de composant facturation.

Ce composant possède les ports suivants:

- Un ServerPort pour lancer le composant.
- Un ClientPort pour les opérations sur les bases des données.
- Un DataPort pour le transfert des exceptions.

2.3. Comportement du composant de contrôle :

Le contrôleur (OpPartController) aura le comportement décrit par les instructions suivantes :

```
Acq.request Acquisition  // doit attendre la fin de la réalisation du
                         //  Service requis.
prep.request  preparation ;
cc.request Calcule_coûts ;
cs.request coûts_services ;
fact.request facturation ;
```

Figure 6.21 : le comportement de OpPartController.

La figure ci – dissous (figure 6.22) montre le comportement de OpPartController avec IASASTUDIO.

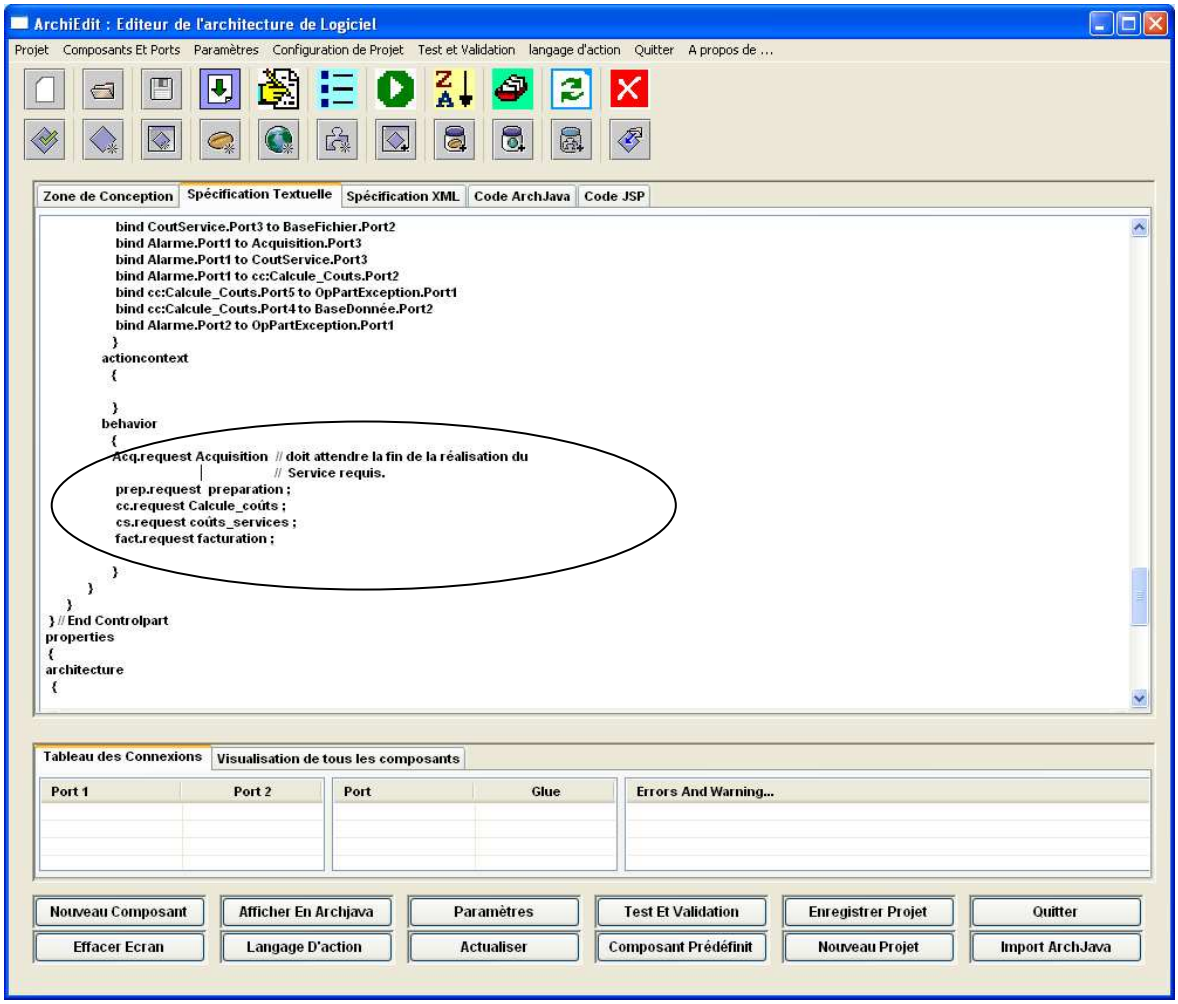

Figure 6.22 : le comportement de OpPartController avec IASASTUDIO.

Remarque : La logique de ces actions nécessite que tous les ports soient synchrones.

La figure 6.23 montre les règles appliquées au niveau des ports.

```
ServerPort Port1
   {
    accesspoint
            {
              ServerActionPoint Point1  Nom_Methode(0,SYNCHRONE);
              DataPoint Point2  Nom_Methode(0,SYNCHRONE);
            }
    behavior
    {
     rules  régle1,
     rule régle1
       {
        precondition : // Aucune précondition nécessaire
        pattern :
        postcondition :
       }
```

Figure 6.23 : Les règles appliquées au niveau des ports.

La figure 6.24 montre les règles appliquées au niveau des ports avec l'outil IASASTUDIO.

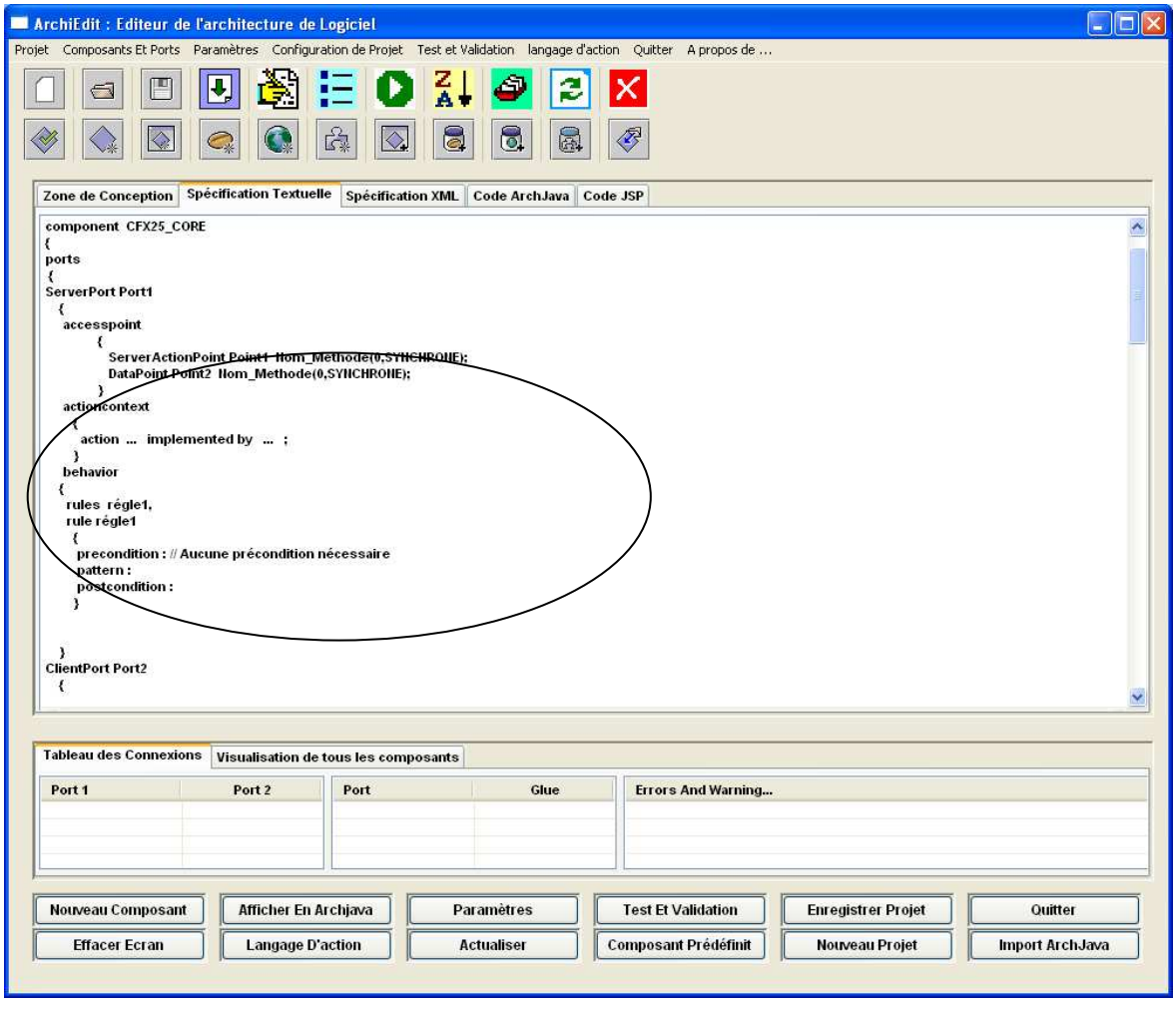

Figure 6.24 : Les règles appliquées au niveau des ports.

2.4. Etude de la stabilité :

Il suffit d'appliquer la technique des enveloppes et le marquage des ports, le résultat de cette opération est montré dans la figure 6.25.

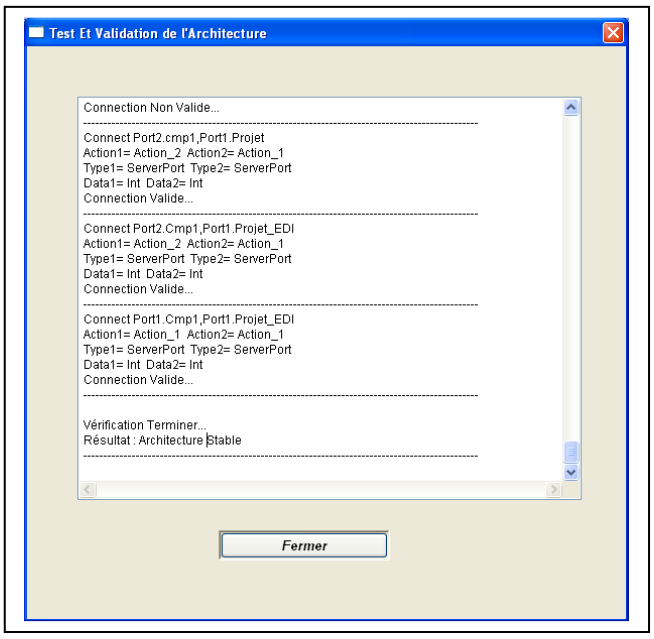

Figure 6.25 : Résultat de la vérification de la stabilité.

3. Réalisation avec l'outil IASASTUDIO :

3.1. Schéma générale de l'architecture :

Le schéma global de l'architecture est représenté dans la figure ci – dissous (Figure 6.26) :

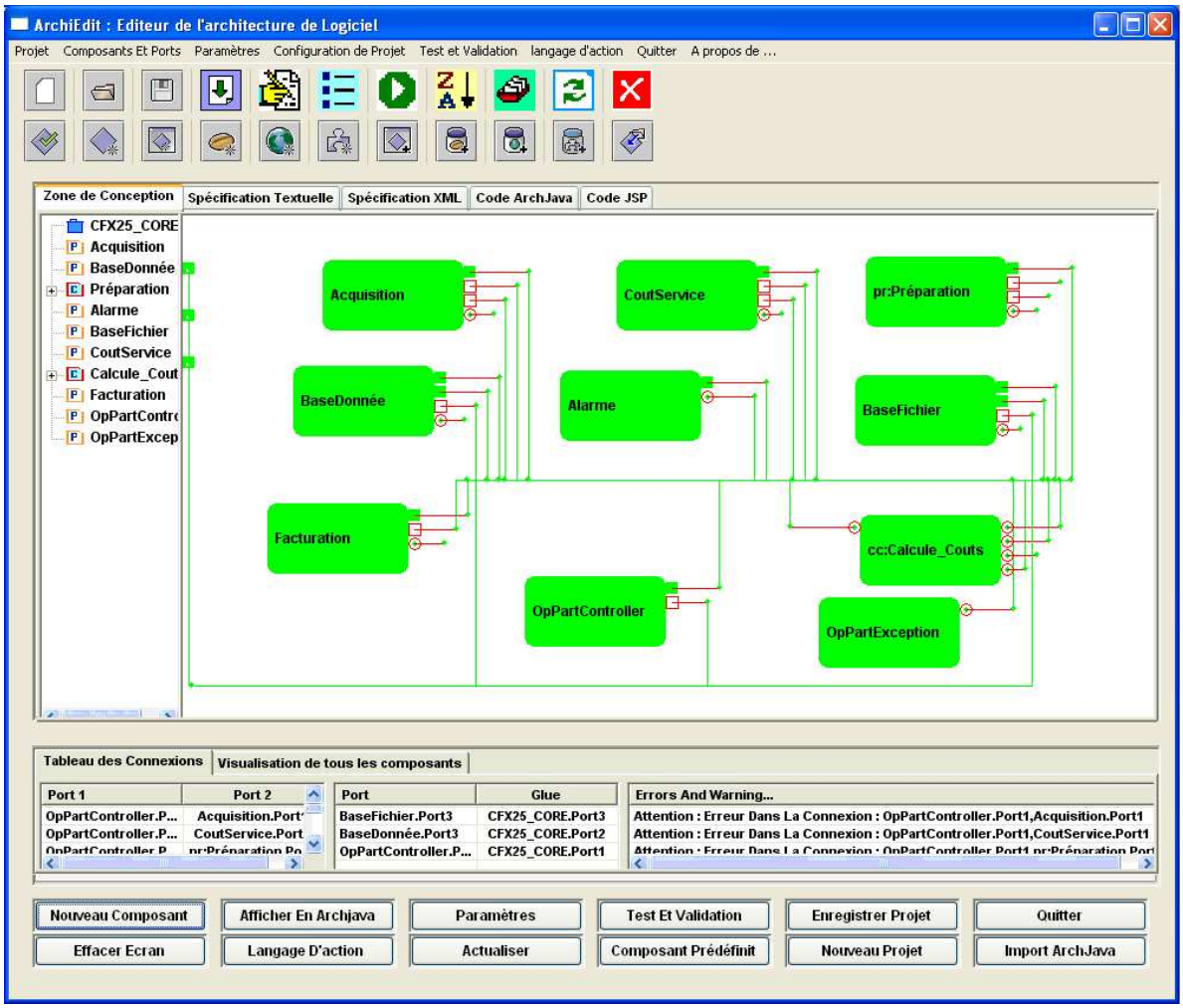

Figure 6.26 : Schéma global de l'architecture.

3.2. Spécification Textuelle :

La spécification textuelle possède la forme suivante (figure 6.27):

```
package  CFX25_CORE;
import  Package1,Package2;
component  CFX25_CORE
{
ports
 {
……
……
  components
    {
     Acquisition  Nom_Instance0;
     BaseDonnée  Nom_Instance1;
     pr:Préparation  Nom_Instance2;
     Alarme  Nom_Instance3;
     BaseFichier  Nom_Instance4;
     CoutService  Nom_Instance5;
```

```
      cc:Calcule_Couts  Nom_Instance6;
      Facturation  Nom_Instance7;
      OpPartController  Nom_Instance8;
      OpPartException  Nom_Instance9;
     }
  connexions
     {
       DelegateConnector Connecteur0 { map BaseFichier.Port3 , Port3 };
       DelegateConnector Connecteur1 { map BaseDonnée.Port3 , Port2 };
       DelegateConnector Connecteur2 { map OpPartController.Port2 , Port1 };
     }
 } // End PartOperative
controlpart
 {
  components
     {
      OpPartController opPartController ;
      ActiveOpPartException OpPartException ;
     }
  connexions
     {
     TupeConnexion NomConnecteru
         {
            Binding
              {
               bind OpPartController.Port1 to Acquisition.Port1
               bind OpPartController.Port1 to CoutService.Port1
               bind OpPartController.Port1 to pr:Préparation.Port1
               bind OpPartController.Port1 to cc:Calcule_Couts.Port1
               ……
               ……
               bind Alarme.Port1 to cc:Calcule_Couts.Port2
               bind cc:Calcule_Couts.Port5 to OpPartException.Port1
               bind cc:Calcule_Couts.Port4 to BaseDonnée.Port2
……
……
         deploy  Facturatio  as  PROCESS  in  ENV2 ;
        }
  }
 }
} //End of component
```

Figure 6.27 : La spécification textuelle de composant CFX25_CORE.

Le code complet est dans l'annexe 1 code 7.4.

3.3. Génération de code ArchJava :

La figure 6.28 montre la spécification ArchJava de composant CFX25_CORE.

```
public component class CFX25_CORE
{
Private final Acquisition Cmp01 = new Acquisition( ) ;
Private final BaseDonnée Cmp11 = new BaseDonnée( ) ;
Private final pr:Préparation Cmp21 = new pr:Préparation( ) ;
Private final Alarme Cmp31 = new Alarme( ) ;
Private final BaseFichier Cmp41 = new BaseFichier( ) ;
Private final CoutService Cmp51 = new CoutService( ) ;
Private final cc:Calcule_Couts Cmp61 = new cc:Calcule_Couts( ) ;
Private final Facturation Cmp71 = new Facturation( ) ;
Private final OpPartController Cmp81 = new OpPartController( ) ;
Private final OpPartException Cmp91 = new OpPartException( ) ;

Connect OpPartController.Port1 , Acquisition.Port1 ;
Connect OpPartController.Port1 , CoutService.Port1 ;
Connect OpPartController.Port1 , pr:Préparation.Port1 ;
Connect OpPartController.Port1 , cc:Calcule_Couts.Port1 ;
……
……
Connect CoutService.Port3 , BaseFichier.Port2 ;
Connect Alarme.Port1 , Acquisition.Port3 ;
Connect Alarme.Port1 , CoutService.Port3 ;
Connect Alarme.Port1 , cc:Calcule_Couts.Port2 ;
Connect Alarme.Port2 , OpPartException.Port1 ;

Glue Port3 to BaseFichier.Port3;
Glue Port2 to BaseDonnée.Port3;
Glue Port1 to OpPartController.Port2;

Public Port Port1 ;
  {    provide  void methode_( );   }

Public Port Port2 ;
  {    requiers void methode_( );   }

Public Port Port3 ;
  {    requiers void methode_( );   }

public void Execute()
 { // Fonction permet de lancer les traitements des sous composants;   }
}

public component class Acquisition
 {
   public port Port1
        {provide Int  Action_1 (); }
   public port Port2
        {requiers Int  Action_2 (); }
   public port Port3
        {requiers Int  Action_3 (); }
   public port Port4
        {provide Int  Action_4 (); }
  public void Acquisition()
  {
  ……..
public component class OpPartException
 {
   public port Port1
        {provide Int  Action_1 (); }
  public void OpPartException()
  {
    }
 }
```

Figure 6.28 : La spécification ArchJava de composant CFX25_CORE.

4. Mis à jour de la liste des types à réaliser :

Après l'étude de la stabilité nous arrivons à la fin de la 2^{em} itération, nous définissons donc les futurs composants à réaliser, dans notre cas la liste des nouveaux composant à réaliser est composée de :
{Acquisition, préparation, calcule des coûts, coûts des services, facturation, base des données, base des fichiers, alarme}

Etape 2 – 3^{em} ITERATION:

1. choix de composant de la liste des types de composant :

Soit donc à réaliser le composant préparation

1.1. Elaboration de la vue externe de préparation:

La vue externe de composant préparation est présentée à la figure 6.17.

2. Elaboration de la vue interne du composant de préparation

La vue interne de composant préparation est présentée à la figure 6.29.

2.1. Détermination des composants nécessaires à la réalisation de préparation:

Dans ce sous système Les tickets sont contrôlés, décodés, harmonisés et dé doublonnés. Ce sous – système a pour but d'uniformiser le format de tout les TT et de ne garder que les champs utiles au centre de facturation, il prépare les tickets de taxation, reçus des réseaux X25 de l'opérateur, pour le sous – système «calcules des coûts ».

Chaque ticket passe par toutes les phases CONTROLE, DECODAGE, HARMONISATION et JOURNALISATION, DE-DOUBLONNAGE. Le sous – système préparation s'appuie sur les grammaires de descriptions de format des divers constructeurs.

- ✓ La phase de contrôle vérifie la structure syntaxique des tickets. Si un ticket ne possède pas un format reconnaissable par l'application en référence à la grammaire associée, il est rejeté.
- ✓ La phase de décodage, harmonisation transforme les champs codés utiles à l'harmonisation, applique les règles d'harmonisation et de simplification et génère un ticket au format unique lisible par le sous – système « calcule des coûts ».
- ✓ La phase de dé – doublonnage élimine les doublons éventuellement présents dans les fichiers harmonisés.

Chaque ticket subit en premier lieu un contrôle syntaxique global avant d'être décodé et harmonisé. Les champs utiles des tickets sont récupérés et décodés. Et en dernier lieu, les tickets sont harmonisés en un ticket unique à l'aide de règles d'harmonisation et à partir des champs utiles.

Les tickets dont les champs ne sont conformes à la grammaire correspondante sont rejetés dans un fichier dédié stocké dans un répertoire dédié aux TT rejetés.

La figure 6.29 montre le schéma qui représente la vue interne de composant préparation :

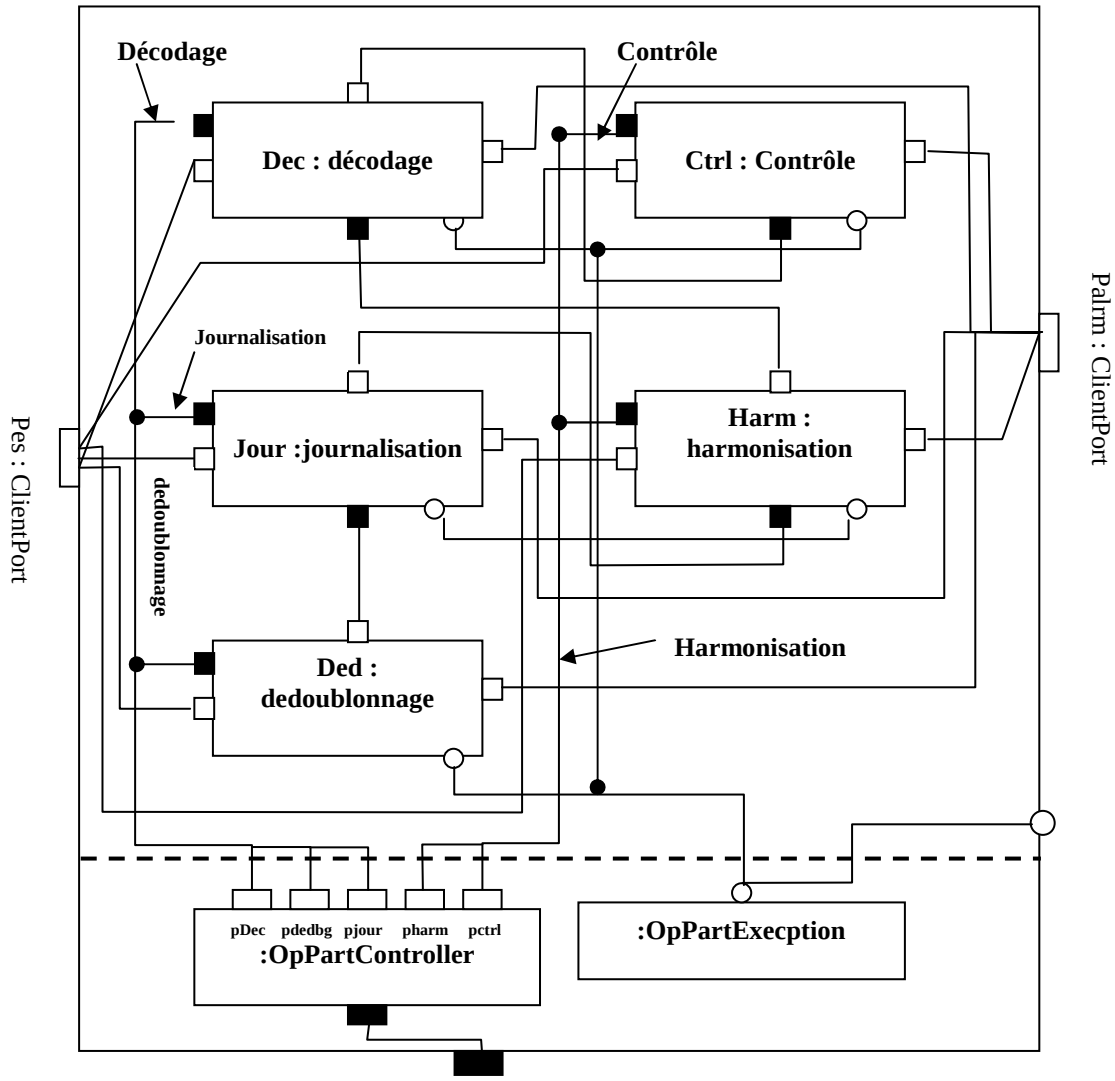

Figure 6.29 : La vue interne de composant préparation

2.2. Définition de la vue externe des sous composants :

Nous présentons dans ce qui suit la vue externe des sous composant décodage, harmonisation, dé doublonnage et journalisation.

➢ *Le composant décodage :*

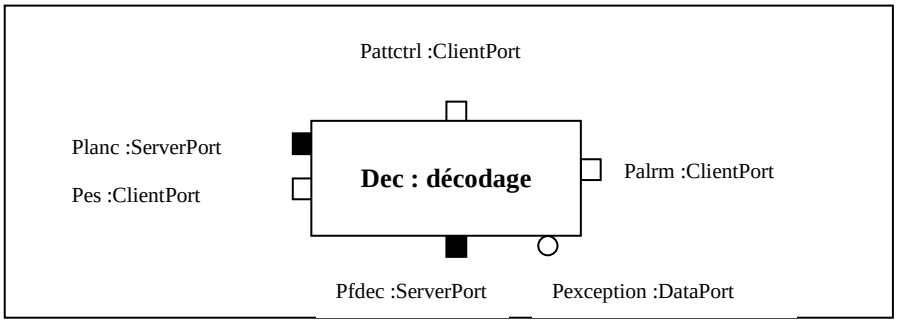

Figure 6.30 : La vue externe de composant décodage.

Ce composant possède les ports suivants :

- Un ServerPort pour lancer le composant.
- Un ServerPort pour indiquer la fin de l'opération de décodage.
- Un ClientPort pour les opérations sur les fichiers.
- Un ClientPort pour la vérification de l'opération de contrôle.
- Un ClientPort pour la gestion des alarmes.
- Un DataPort pour le transfert des exceptions.

➢ *Le composant journalisation :*

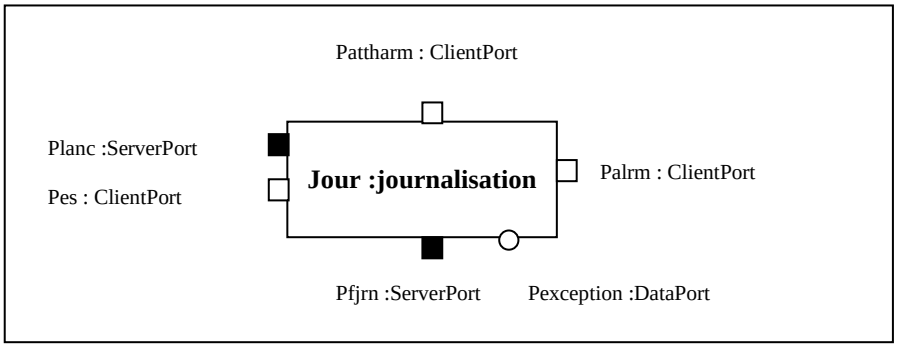

Figure 6.31 : La vue externe de composant journalisation.

Ce composant possède les ports suivants :

- Un ServerPort pour lancer le composant.
- Un ServerPort pour indiquer la fin de l'opération de journalisation.
- Un ClientPort pour les opérations sur les fichiers.

- Un ClientPort pour la vérification de l'opération de harmonisation.
- Un ClientPort pour la gestion des alarmes.
- Un DataPort pour le transfert des exceptions.

➢ *Le composant dé doublonnage :*

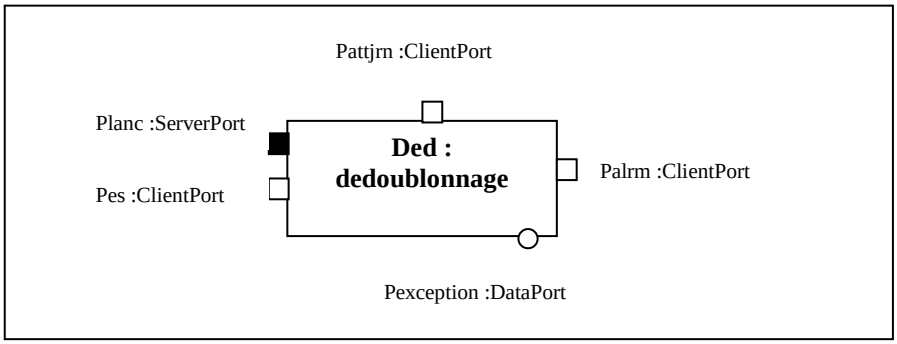

Figure 6.32 : La vue externe de composant ddoublonnage.

Ce composant possède les ports suivants :

- Un ServerPort pour lancer le composant.
- Un ClientPort pour les opérations sur les fichiers.
- Un ClientPort pour la vérification de l'opération de journalisation.
- Un ClientPort pour la gestion des alarmes.
- Un DataPort pour le transfert des exceptions.

➢ *Le composant harmonisation :*

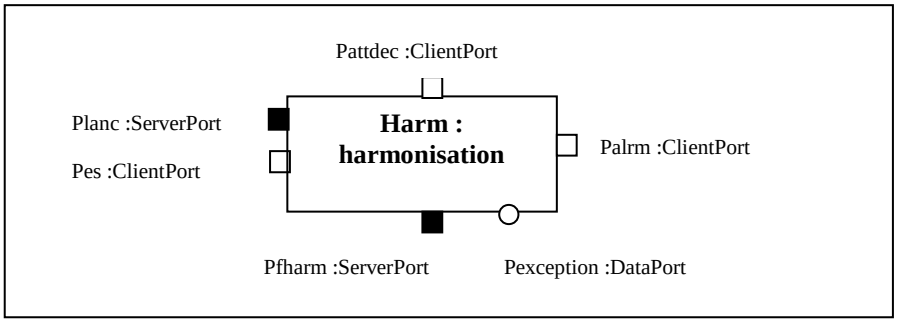

Figure 6.33 : La vue externe de composant harmonisation.

Ce composant possède les ports suivants :

- Un ServerPort pour lancer le composant.
- Un ServerPort pour indiquer la fin de l'opération de harmonisation.
- Un ClientPort pour les opérations sur les fichiers.
- Un ClientPort pour la vérification de l'opération de décodage.
- Un ClientPort pour la gestion des alarmes.
- Un DataPort pour le transfert des exceptions.

2.3. Comportement du composant de contrôle :

Le contrôleur (OpPartController) aura le comportement décrit par les instructions suivantes :

pctrl.request Contrôle.
pDec.request Décodage.
pharm.request harmonisation.
pdedbg.request dedoublonnage.
pjour.request Journalisation.

Figure 6.34 : Le comportement de *OpPartController*.

Remarque : La logique de ces actions nécessite que tous les ports soient synchrones.

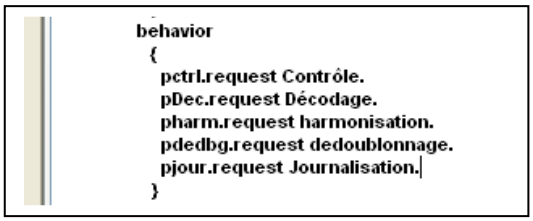

Figure 6.35 : Le comportement de *OpPartController*
Avec IASASTUDIO.

2.4. Etude de la stabilité :

Il suffit d'appliquer la technique des enveloppes et le marquage des ports.
Le résultat est montré dans la figure suivante :

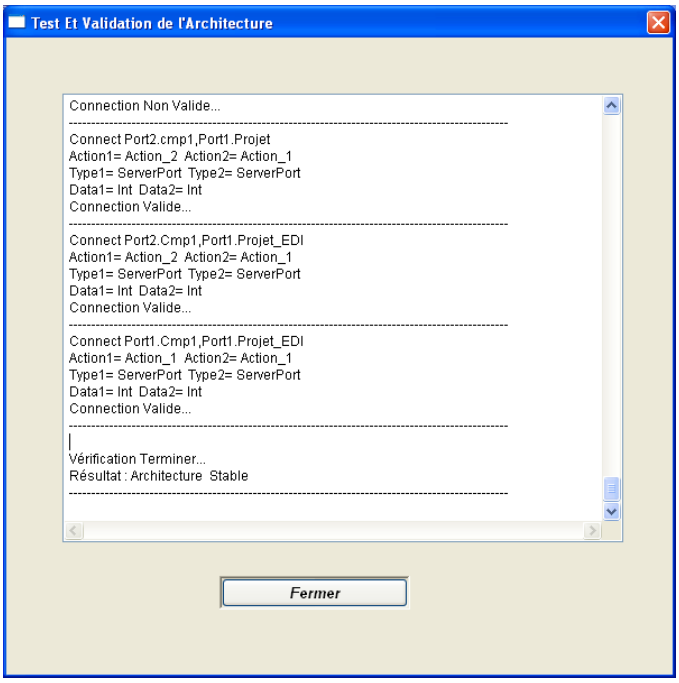

Figure 6.36 : Résultat de test de la stabilité.

3. Réalisation avec l'outil IASASTUDIO :

3.1. Schéma générale de l'architecture :

La figure 6.37 montre le schéma général de l'architecture.

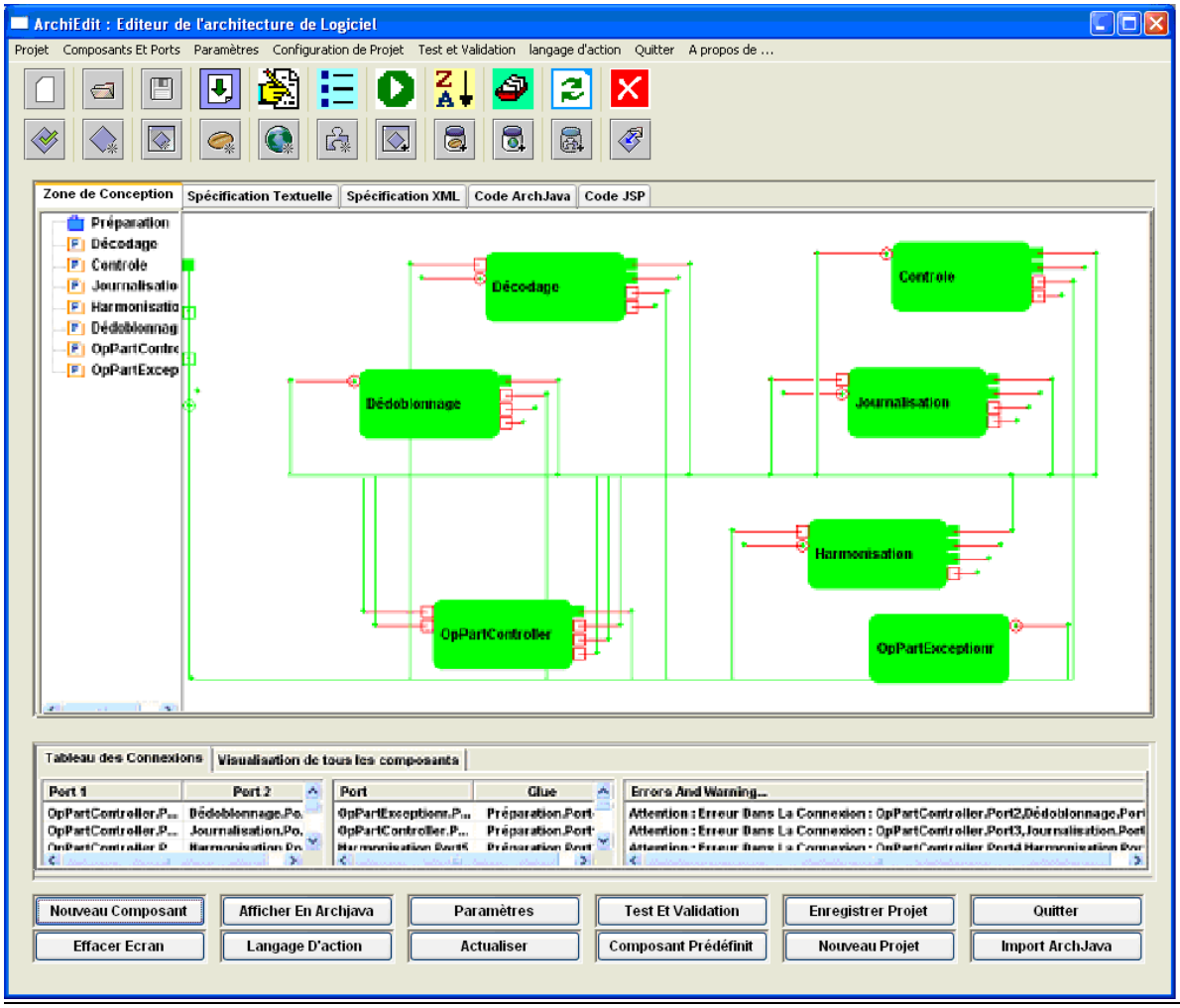

Figure 6.37 : Schéma général de l'architecture.

3.2. Génération du code SEAL décrivant l'architecture

La figure 6.38 montre le code SEAL correspondant aux divers aspects de l'architecture réalisée à l'aide de IASASTUIO. Le code complet est disponible en annexe (annexe 1 code 7.6).

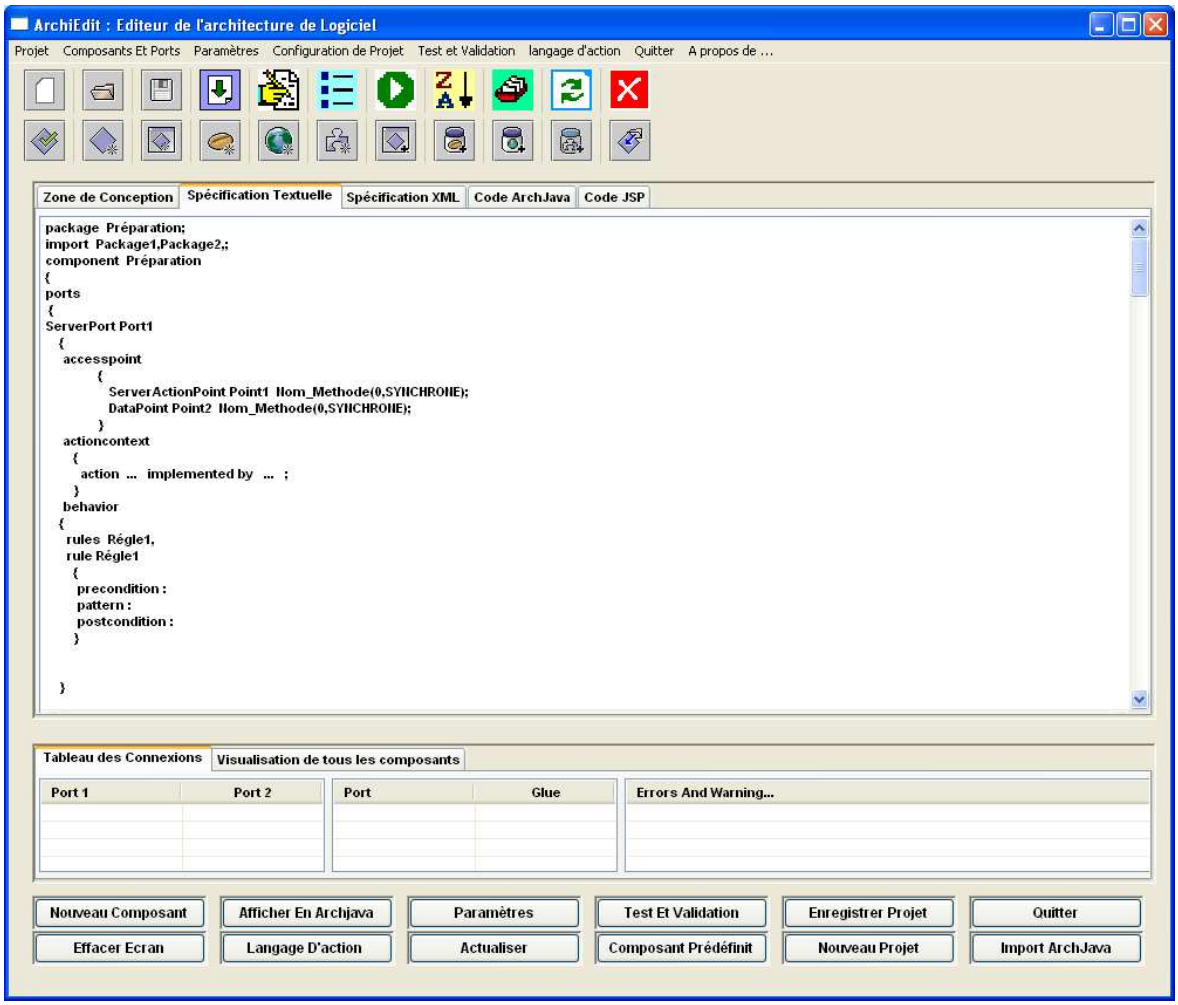

Figure 6.38 : Génération du code SEAL décrivant l'architecture.

3.3. Génération de code ArchJava :

Le code ArchJava qui correspond à l'architecture du composant préparation est le suivant :

```
import  java.io.BufferedReader;
import  java.io.BufferedWriter;
import  java.io.File;
import  java.io.FileInputStream;
import  java.io.FileOutputStream;
import  java.io.IOException;
import  java.sql.Connection;
import  java.sql.Statement;

public component class Préparation
{
Private final Décodage Cmp01 = new Décodage( ) ;
Private final Controle Cmp11 = new Controle( ) ;
Private final Journalisation Cmp21 = new Journalisation( ) ;
Private final Harmonisation Cmp31 = new Harmonisation( ) ;
Private final Dédoblonnage Cmp41 = new Dédoblonnage( ) ;
Private final OpPartController Cmp51 = new OpPartController( ) ;
Private final OpPartExceptionr Cmp61 = new OpPartExceptionr( ) ;
```

```
Connect OpPartController.Port2 , Dédoblonnage.Port1 ;
Connect OpPartController.Port3 , Journalisation.Port1 ;
Connect OpPartController.Port4 , Harmonisation.Port1 ;
Connect OpPartController.Port5 , Décodage.Port1 ;
Connect OpPartController.Port6 , Controle.Port1 ;

Glue Port4 to OpPartExceptionr.Port1;
Glue Port1 to OpPartController.Port1;
Glue Port3 to Harmonisation.Port5;
Glue Port3 to Dédoblonnage.Port2;
Glue Port2 to Décodage.Port3;
Glue Port2 to Controle.Port3;
Glue Port2 to Décodage.Port5;

Public Port Port1 ;
  {    provide  void methode_( );    }

Public Port Port2 ;
  {    requiers void methode_( );    }

Public Port Port3 ;
  {    requiers void methode_( );    }

Public Port Port4 ;
  {    provide  void methode_( )     }

public void Execute()
 { // Fonction permet de lancer les traitements des sous composants;   }
}

……...
……...
……...

  public void OpPartController()
  {

  }
 }
public component class OpPartExceptionr
 {
   public port Port1
        {provide Int  Action_1 (); }
  public void OpPartExceptionr()
  {

  }
 }
```

Figure 6.39 : Le code ArchJava qui correspond à l'architecture

Du composant préparation.

3.4. Spécification XML :

La figure 6.40 montre le code XML correspondant aux divers aspects de l'architecture réalisée à l'aide de IASASTUIO. Le code complet est disponible en annexe (annexe 1 code 7.8).

```
<Architecture>
<Composite Type = 'Préparation'  Name = 'Préparation' >
<Component Type = 'Décodage'  Width = '120'  Height = '60'   X = 259  Y = 34  Name = 'Décodage'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '434'  Y = '44'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '424'  Y = '56'  Name = 'Port2'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '414'  Y = '68'  Name = 'Port3'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '404'  Y = '80'  Name = 'Port4'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_4'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '194'  Y = '44'  Name = 'Port5'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_5'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '204'  Y = '56'  Name = 'Port6'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_6'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'Controle'  Width = '120'  Height = '60'   X = 607  Y = 25  Name = 'Controle'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '782'  Y = '35'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
…….
…….
…….
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port3'>
<PortRef>Dédoblonnage->Port2</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port2'>
<PortRef>Décodage->Port3</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port2'>
<PortRef>Controle->Port3</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port2'>
<PortRef>Décodage->Port5</PortRef>
</DelegatedPort>
</Composite>
</Architecture>
```

Figure 6.40 : Le code XML qui correspond à l'architecture
Du composant préparation.

4. Mise à jour de la liste des types à réaliser :

Après l'étude de la stabilité nous arrivons à la fin de la 3^{em} itération, nous définissons donc les futurs composants à réaliser, dans notre cas la liste des nouveaux composants à réaliser reste la même (effacer les composant déjà traiter) puisque il y a pas de nouveaux types à ajouter (tous les nouveaux composants sont primitifs) la liste est donc composée de :
{Acquisition, calcule des coûts, coûts des services, facturation, base des données, base des fichiers, alarme}.

ETAPE 2 – 4ème ITERATION :

1. choix de composant de la liste des types de composant :

Soit donc à réaliser le composant « calcule des coûts ».

1.1. Elaboration de la vue externe de composant calcule des coûts :

La figure 6.18 montre la vue externe de composant calcule des coûts.

2. Elaboration de la vue interne de composant calcule des coûts :

La vue interne de composant calcule des coûts est représenté par la figure 6.41.

2.1. Détermination des composants nécessaires à la réalisation de composant calcule des coûts :

Deux sous composants sont nécessaires à la réalisation de ce composant :
Réconciliation et Usage_Traitement : Ces deux sous composant sont primitifs et correspondent à des implémentations directes de leur traitement. Le but est donc d'obtenir en détails, les coûts de toutes les communications nationales et internationales effectuées sur le réseau de l'opérateur dans le but de pouvoir facturer par la suite les abonnées (clients) concernés par réseau X25.
Le sous composant « *Usage_Traitement* » génère des données servant à la facturation des communications. Ces données sont :

- la pondération du volume échangé pendant la communication en fonction du temps passé sur chaque tranche horaire,
- la durée totale de la communication,
- Le coût réel pour le volume sur chaque tranche tarifaire, tenant compte du type de tarification de l'accès (forfait complet, forfait durée, réel).
- Le coût réel pour la durée, coût tenant compte du type de tarification de l'accès (forfait complet, forfait durée, réel).
- le coût pour le volume pour la facture détaillée, coût ne tenant pas compte du type de tarification de l'accès (forfait complet, forfait durée, réel)
- le coût pour la durée pour la facture détaillée, coût ne tenant pas compte du type de tarification de l'accès (forfait complet, forfait durée, réel).

Le sous composant «*Réconciliation*» stocke les valeurs obtenues par la valorisation et les tickets de taxation proprement dite. Avant la phase de stockage, les tickets sont analysés pour déterminer s'ils représentent une communication complète. Si ce n'est pas le cas ils sont réconciliés avec les tickets déjà stockés.

La figure ci – dissous montre le schéma qui représente la vue interne de composant calcule des coûts :

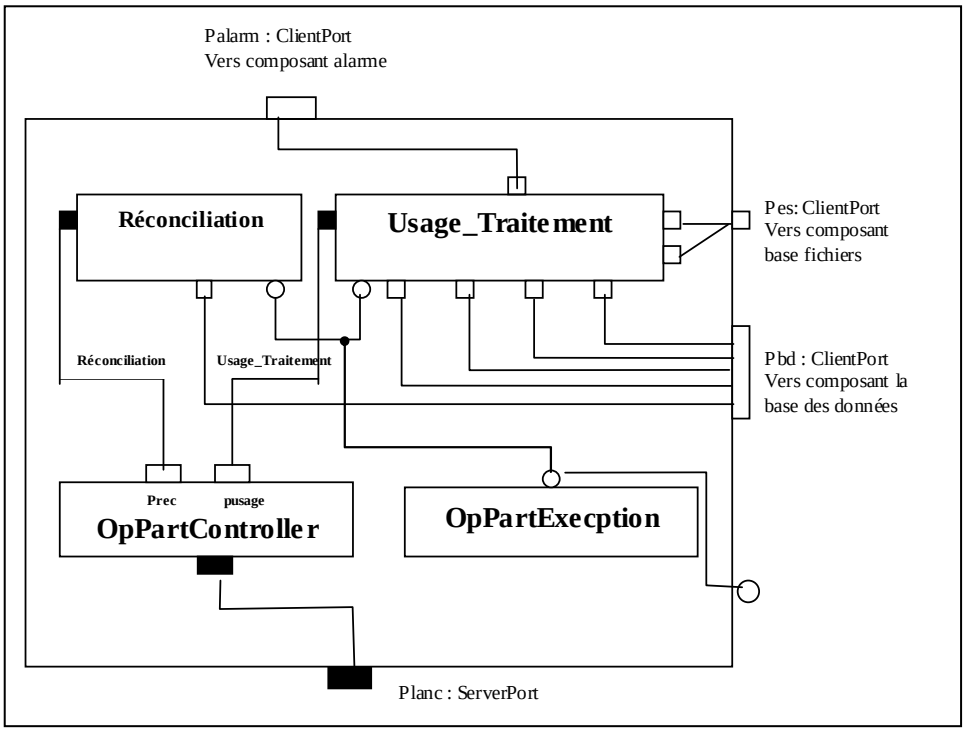

Figure 6.41 : La vue interne de composant calcule des coûts.

2.2. Définition de la vue externe des sous composants :

La vue externe des sous composant est représentée par le schéma global du composant (figure 6.41).

2.3. Comportement du composant de contrôle :

Le contrôleur (OpPartController) aura le comportement décrit par les instructions suivantes :

Prec.request Réconciliation
Pusage.request Usage_Traitement

Figure 6.42 : le comportement de *OpPartController*.

2.4. Etude de la stabilité :

Il suffit d'appliquer la technique des enveloppes et le marquage des port.

3. Réalisation avec l'outil IASASTUDIO :

3.1. Schéma générale de l'architecture :

La figure 6.43 montre l'architecture de composant calcule des coûts.

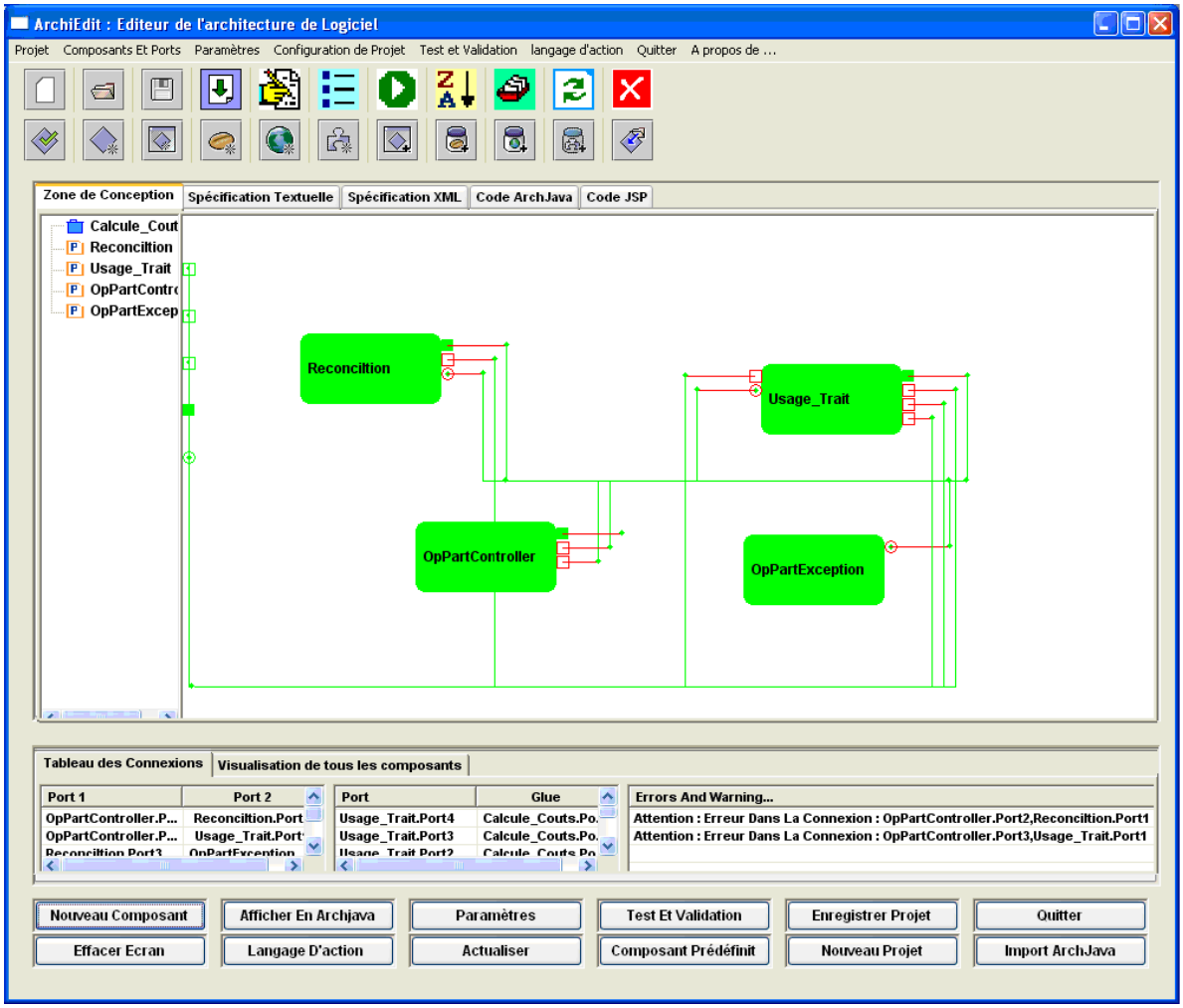

Figure 6.43 : Schéma général de composant calcule des coûts.

3.2. Spécification Textuelle :

Le code SEAL complet correspondant aux divers aspects de l'architecture réalisée à l'aide de IASASTUIO est disponible en annexe (annexe 1 code 7.9).

3.3. Génération de code ArchJava :

Le code ArchJava complet correspondant aux divers aspects de l'architecture réalisée à l'aide de IASASTUIO est disponible en annexe (annexe 1 code 7.10).

3.4. Spécification XML :

Le code XML complet correspondant aux divers aspects de l'architecture réalisée à l'aide de IASASTUIO est disponible en annexe (annexe 1 code 7.11).

4. Mis à jour de la liste des types à réaliser :

Après l'étude de la stabilité nous arrivons à la fin de la 4^{em} itération, nous définissons donc les futurs composants à réaliser, dans notre cas la liste des nouveaux composants à réaliser reste la même (effacer les composant déjà traiter) puisque il y a pas de nouveaux types à ajouter (tous les nouveaux composants sont primitifs) la liste est donc composée de :
{Acquisition, coûts des services, facturation, base des données, base des fichiers, alarme}.

En refait le même travail pour chaque composant composite de la liste ci – dissus, en obtient à la fin le schéma général de l'architecture de l'application CFX25 et la spécification détaillé de tout les composants et les sous composants ainsi que le code ARCJAVA qui correspond à l'architecture.

6.7. Application des instructions de langage d'action SEAL :

On peu applique les différents instruction de langage SEAL décrit précédemment, on peut prendre les instruction suivante et on les appliquent sur le composant composite Préparation :

```
Cmp1 = CreateComponent (Controle) ;
Cmp2 = CreateComponent (Décodage) ;
FOR  ( i=1 ;i<10 ;i++)
{
Connecteur1=CreateConnecteur(Controle.port3,Décodage.port1);
} ;
WHILE(i:<:10)
{
Connecteur1=CreateConnecteur(Controle.port3,Décodage.port1) ;
} ;
IF(i=5;i:<:10)
{
Connecteur1=CreateConnecteur(Controle.port3,Décodage.port1) ;
} ;
```

Figure 6.44 : Les instructions de langage SEAL.

On saisie ce langage a l'aide de l'outil IASASTUDIO comme il est montré dans la figure suivante :

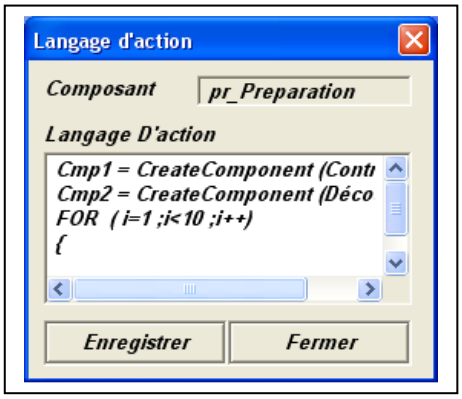

Figure 6.45 : Saisie des instructions du langage SEAL

On lance l'exécution de langage par l'outil IASASTUDIO et on peut bien suivre la traçabilité d'exécution.

Figure 6.46 : Choix de comportement à exécuter.

Et on fin on peut bien suivre l'exécution de ce langage d'action et les résultat a la fin, le code complète est bien présenter dans l'annexe à la fin de ce document.

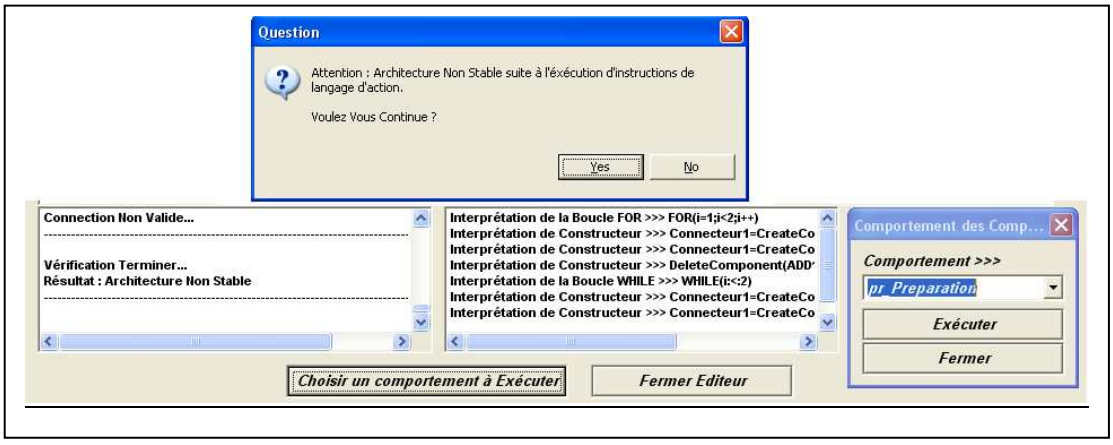

Figure 6.47 : Résultat de l'exécution.

CONCLUSION

La complexité croissante des systèmes et leur évolution de plus en plus rapide ont motivé un intérêt accru pour les modèles, les techniques et les méthodes de description de l'architecture d'un système logiciel. La description architecturale qui a émergé au sein de la communauté de recherche « Architectures logicielles » et l'approche de modélisation orientée objet sont les deux principales approches permettant de modéliser et de décrire l'architecture d'un système logiciel. Ces deux approches ont plusieurs points communs, elles se basent sur les mêmes concepts qui sont l'abstraction et les interactions entre les entités. [26]

Les architectures statiques ne permettent pas de répondre à un nombre important de besoins, notamment des besoins d'adaptabilité, de résistance aux pannes, de mise à jour à chauds de logiciel, d'optimisation de ressources, d'évolution structurelle ou comportementale. La prise en charge de tels besoins ne peut être accomplie que dans le contexte des architectures dynamiques. La spécification d'architectures dynamiques permet de planifier à la conception les changements topologiques qui auront lieu à l'exécution.

Dans ce travail, nous avons présenté une approche pour la description, la spécification et la validation de comportement d'une architecture logiciel dynamique, cette approche nous a permis de mettre au point un langage de spécification d'architecture logicielle dynamique. Ce langage, appelé SEAL est de type langage action. Il s'inspire de *Precise Action Semantic* de UML et rentre dans le contexte des actions de recherche dont le but est d'arriver à la spécification de modèles exécutables.

La définition du modèle de composant IASA repose sur l'étude des différents ADLs, modèles de composant abstraits académiques ou issus des organismes de normalisation. On a proposé à l'architecte un modèle de haut niveau de l'architecture qui se concentre sur les éléments caractéristiques d'une description d'architecture. Cette abstraction lui permettre de travailler avec un formalisme simple, facilement compréhensible par l'ensemble des acteurs d'un projet informatique. La technique de spécification que nous avons proposés est en fait un renforcement du modèle informel à base de boite noire et connexions par des mécanismes pour le rendre plus précis et rigoureux. Cette technique de spécification a une grande simplicité et elle est capable de

prendre en charge de manière la plus directe possible le modèle mental de l’architecte sous ses deux aspects, structurel et comportemental.

La spécification des comportements c’est à dire les interactions et le comportement de la partie opérative dans un composant composite se fait dans le langage d’action SEAL, Celui ci est doté d’une grande puissance d’expression due principalement à la notion de contexte (*ActionContext*). La version actuelle de l’interpréteur de SEAL ne s’intéresse qu’à l’exécution d’un composant composite, plus exactement du composant *BehavioralComponent* dans le but de montrer la validité des opérations de transformation architecturale comme l’ajout ou la suppression des composants et de connexions. Ce langage à été définit donc dans le but de montrer la validité des opérations de transformation architecturale et l’évolution structurelle de l’architecture. Il repose sur des contextes fondamentaux et sur tous les types prédéfinis relatifs aux composants, ports, point d’accès et connecteurs.

Nous avons proposés aussi des solutions aux problèmes que nous avons cités. Ces propositions sont faites dans le contexte de l'approche IASA pour l'architecture logicielle. On a aussi développés un environnement de conception et validation de l'évolution structurelle d'une architecture. Cette validation se fait par l'exécution d'architecture à un haut niveau d'abstraction. L'évaluation du langage d'action SEAL ainsi que l'environnement de conception réalisé a été montré et bien discuté dans ce travail. Cette évaluation s'est faite dans le contexte de la conception et réalisation d'une application de gestion commerciale des produits d'un réseau X25.

Les perspectives se résument dans les points suivants :

- ✓ Il est nécessite d’avoir un modèle de connecteurs comme élément de base des ADL.

- ✓ Il es nécessaire aussi de faire un recensement sur les différents domaines de conception des connecteurs et de déterminer l’espace de mise en œuvre des connecteurs.

- ✓ En architecture logicielle, tous les connecteurs ne sont pas des connecteurs d'interaction où il y a nécessairement échanges d'informations. A titre d'exemple, il y a des connecteurs de contrôle de l'implémentation, des connecteurs montrant une dépendance entre un composant et un autre. Ces connecteurs apparaissent dans un domaine bien précis de conception.

- ✓ Une des évolutions attendues est le remplacement des achats de logiciels informatiques par des locations de ces mêmes logiciels pour le temps nécessaire à leur utilisation. Cette évolution doit être mise en rapport avec le concept de Web 2.0

Annexe 1

Code 7.1 :

```
package  CFX25;
import  Package1,Package2,;
component  CFX25
{
ports
 {
ServerPort Port1
    {
     accesspoint
               {
                  ServerActionPoint Point1  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point2  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  régle1,régle2,régle3,
      rule régle1
        {
         precondition :
         pattern :
         postcondition :
        }

      rule régle2
        {
         precondition :
         pattern :
         postcondition :
        }

      rule régle3
        {
         precondition :
         pattern :
         postcondition :
        }


    }
ClientPort Port2
    {
     accesspoint
               {
                  ClientActionPoint Point3  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point4  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  régle1,
      rule régle1
        {
         precondition :
         pattern :
         postcondition :
        }


    }
ClientPort Port3
    {
     accesspoint
               {
                  ClientActionPoint Point5  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point6  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
```

```
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  régle1,régle2,
      rule régle1
        {
         precondition :
         pattern :
         postcondition :
        }

      rule régle2
        {
         precondition :
         pattern :
         postcondition :
        }


    }
 } // End of Port
connector
{
  connector TypeNomConnector
          {
              roles Liste_des_roles;
             architectures
               {

               }
             actioncontext
               {

               }
             behavior
               {

               }
          }
}
operativepart
 {
   components
      {
       CFX25_CORE  Nom_Instance0;
       OpPartController  Nom_Instance1;
       OpPartException  Nom_Instance2;
      }
   connexions
      {
         DelegateConnector Connecteur0 { map CFX25_CORE.Port3 , Port3 };
         DelegateConnector Connecteur1 { map CFX25_CORE.Port2 , Port2 };
         DelegateConnector Connecteur2 { map OpPartController.Port1 , Port1 };
      }
 } // End PartOperative
controlpart
 {
   components
      {
        OpPartController opPartController ;
        ActiveOpPartException OpPartException ;
      }
   connexions
      {
       TupeConnexion NomConnecteru
            {
                Binding
                   {
                    bind CFX25_CORE.Port1 to OpPartController.Port2
                    bind OpPartException.Port1 to CFX25_CORE.Port4
                   }
                actioncontext
                   {

                   }
                behavior
```

```
                    {

                    }
               }
          }
 } // End Controlpart
properties
 {
architecture
  {
     environment ENV1
        {
           machine 10.1.1.155 ;
           os UNIX ;
        }
     environment ENV2
        {
           machine Localhost ;
           os UNIX ;
        }
     environment ENV3
        {
           machine 10.1.25.254 ;
           os WINDOWS ;
        }
  }

deployment
  {
    deploymentcase{THREA, PROCESS, COMPOSITE};
    deploymentmap map1
          {
            deploy  CFX25_CORE  as  PROCESS  in  10.1.1.155 ;
            deploy  OpPartController  as  PROCESS  in  localhost ;
          }
  }

 }
} //End of component
```

Code 7.2 :

```
import  java.io.BufferedReader;
import  java.io.BufferedWriter;
import  java.io.File;
import  java.io.FileInputStream;
import  java.io.FileOutputStream;
import  java.io.IOException;
import  java.io.InputStreamReader;
import  java.io.OutputStreamWriter;
import  java.sql.*;
import  java.sql.DriverManager;
import  java.sql.Connection;
import  java.sql.Statement;

public component class CFX25
{
Private final CFX25_CORE Cmp01 = new CFX25_CORE( ) ;
Private final OpPartController Cmp11 = new OpPartController( ) ;
Private final OpPartException Cmp21 = new OpPartException( ) ;

Connect CFX25_CORE.Port1 , OpPartController.Port2 ;
Connect OpPartException.Port1 , CFX25_CORE.Port4 ;

Glue Port3 to CFX25_CORE.Port3;
Glue Port2 to CFX25_CORE.Port2;
Glue Port1 to OpPartController.Port1;

Public Port Port1 ;
   {     provide  void methode_( );    }

Public Port Port2 ;
   {     requiers void methode_( );    }

Public Port Port3 ;
   {     requiers void methode_( );    }
```

```
public void Execute()
  { // Fonction permet de lancer les traitements des sous composants;   }
}

public component class CFX25_CORE
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {provide Int  Action_2 (); }
    public port Port3
          {provide Int  Action_3 (); }
    public port Port4
          {provide Int  Action_4 (); }
   public void CFX25_CORE()
   {

   }
 }
public component class OpPartController
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
   public void OpPartController()
   {

   }
 }
public component class OpPartException
 {
    public port Port1
          {provide Int  Action_1 (); }
   public void OpPartException()
   {

   }
 }
```

Code 7.3 :

```
<Architecture>
<Composite Type = 'CFX25'  Name = 'CFX25' >
<Component Type = 'CFX25_CORE'  Width = '120'  Height = '60'   X = 312  Y = 68  Name =
'CFX25_CORE'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '487'  Y = '78'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '477'  Y = '90'  Name = 'Port2'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '467'  Y = '102'  Name = 'Port3'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '457'  Y = '114'  Name = 'Port4'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_4'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'OpPartController'  Width = '120'  Height = '60'   X = 184  Y = 307  Name =
'OpPartController'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '359'  Y = '317'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '349'  Y = '329'  Name = 'Port2'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'OpPartException'  Width = '120'  Height = '60'   X = 578  Y = 331  Name =
'OpPartException'>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '753'  Y = '341'  Name = 'Port1'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
</Component>
<Connector Type = 'Binding'  Name = 'Connector1'>
<ConnectedPort>CFX25_CORE->Port1</ConnectedPort>
<ConnectedPort>OpPartController->Port2</ConnectedPort>
```

```
</Connector>
<Connector Type = 'Binding'  Name = 'Connector2'>
<ConnectedPort>OpPartException->Port1</ConnectedPort>
<ConnectedPort>CFX25_CORE->Port4</ConnectedPort>
</Connector>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port3'>
<PortRef>CFX25_CORE->Port3</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port2'>
<PortRef>CFX25_CORE->Port2</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port1'>
<PortRef>OpPartController->Port1</PortRef>
</DelegatedPort>
</Composite>
</Architecture>
```

Code 7.4 :

```
package  CFX25_CORE;
import  Package1,Package2,;
component  CFX25_CORE
{
ports
 {
ServerPort Port1
    {
     accesspoint
               {
                  ServerActionPoint Point1  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point2  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  régle1,
      rule régle1
        {
         precondition :
         pattern :
         postcondition :
        }


    }
ClientPort Port2
    {
     accesspoint
               {
                  ClientActionPoint Point3  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point4  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle1,Régle2,
      rule Régle1
        {
         precondition :
         pattern :
         postcondition :
        }

      rule Régle2
        {
         precondition :
         pattern :
         postcondition :
        }
```

```
    }
ClientPort Port3
    {
     accesspoint
               {
                  ClientActionPoint Point5  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point6  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle1,Régle2,Régle3,
      rule Régle1
        {
         precondition :
         pattern :
         postcondition :
        }

      rule Régle2
        {
         precondition :
         pattern :
         postcondition :
        }

      rule Régle3
        {
         precondition :
         pattern :
         postcondition :
        }


    }
 } // End of Port
connector
{
  connector TypeNomConnector
          {
              roles Liste_des_roles;
             architectures
               {

               }
             actioncontext
               {

               }
             behavior
               {

               }
          }
}
operativepart
 {
   components
      {
       Acquisition  Nom_Instance0;
       BaseDonnée  Nom_Instance1;
       pr:Préparation  Nom_Instance2;
       Alarme  Nom_Instance3;
       BaseFichier  Nom_Instance4;
       CoutService  Nom_Instance5;
       cc:Calcule_Couts  Nom_Instance6;
       Facturation  Nom_Instance7;
       OpPartController  Nom_Instance8;
       OpPartException  Nom_Instance9;
      }
   connexions
      {
         DelegateConnector Connecteur0 { map BaseFichier.Port3 , Port3 };
```

```
         DelegateConnector Connecteur1 { map BaseDonnée.Port3 , Port2 };
         DelegateConnector Connecteur2 { map OpPartController.Port2 , Port1 };
      }
 } // End PartOperative
controlpart
 {
   components
      {
        OpPartController opPartController ;
        ActiveOpPartException OpPartException ;
      }
   connexions
      {
       TupeConnexion NomConnecteru
            {
                Binding
                   {
                    bind OpPartController.Port1 to Acquisition.Port1
                    bind OpPartController.Port1 to CoutService.Port1
                    bind OpPartController.Port1 to pr:Préparation.Port1
                    bind OpPartController.Port1 to cc:Calcule_Couts.Port1
                    bind OpPartController.Port1 to Facturation.Port1
                    bind Facturation.Port2 to BaseDonnée.Port1
                    bind Acquisition.Port2 to BaseFichier.Port1
                    bind BaseFichier.Port2 to Acquisition.Port3
                    bind CoutService.Port2 to Facturation.Port1
                    bind CoutService.Port3 to BaseFichier.Port2
                    bind Alarme.Port1 to Acquisition.Port3
                    bind Alarme.Port1 to CoutService.Port3
                    bind Alarme.Port1 to cc:Calcule_Couts.Port2
                    bind cc:Calcule_Couts.Port5 to OpPartException.Port1
                    bind cc:Calcule_Couts.Port4 to BaseDonnée.Port2
                    bind Alarme.Port2 to OpPartException.Port1
                   }
                actioncontext
                   {

                   }
                behavior
                   {

                   }
            }
      }
 } // End Controlpart
properties
 {
architecture
  {
     environment ENV1
        {
           machine 10.1.1.155 ;
           os UNIX ;
        }
     environment ENV2
        {
           machine Localhost ;
           os WINDOWS ;
        }
     environment ENV3
        {
           machine 192.168.0.1 ;
           os WINDOWS ;
        }
  }

deployment
  {
    deploymentcase{THREA, PROCESS, COMPOSITE};
    deploymentmap map1
          {
            deploy  Acquisition  as  PROCESS  in  ENV1 ;
            deploy  Facturatio  as  PROCESS  in  ENV2 ;
          }
  }

 }
```

```
} //End of component
```

Code 7.5 :

```
import  java.io.BufferedReader;
import  java.io.BufferedWriter;
import  java.io.File;
import  java.io.FileInputStream;
import  java.io.FileOutputStream;
import  java.io.IOException;
import  java.io.InputStreamReader;
import  java.io.OutputStreamWriter;
import  java.sql.*;
import  java.sql.DriverManager;
import  java.sql.Connection;
import  java.sql.Statement;

public component class CFX25_CORE
{
Private final Acquisition Cmp01 = new Acquisition( ) ;
Private final BaseDonnée Cmp11 = new BaseDonnée( ) ;
Private final pr:Préparation Cmp21 = new pr:Préparation( ) ;
Private final Alarme Cmp31 = new Alarme( ) ;
Private final BaseFichier Cmp41 = new BaseFichier( ) ;
Private final CoutService Cmp51 = new CoutService( ) ;
Private final cc:Calcule_Couts Cmp61 = new cc:Calcule_Couts( ) ;
Private final Facturation Cmp71 = new Facturation( ) ;
Private final OpPartController Cmp81 = new OpPartController( ) ;
Private final OpPartException Cmp91 = new OpPartException( ) ;

Connect OpPartController.Port1 , Acquisition.Port1 ;
Connect OpPartController.Port1 , CoutService.Port1 ;
Connect OpPartController.Port1 , pr:Préparation.Port1 ;
Connect OpPartController.Port1 , cc:Calcule_Couts.Port1 ;
Connect OpPartController.Port1 , Facturation.Port1 ;
Connect Facturation.Port2 , BaseDonnée.Port1 ;
Connect Acquisition.Port2 , BaseFichier.Port1 ;
Connect BaseFichier.Port2 , Acquisition.Port3 ;
Connect CoutService.Port2 , Facturation.Port1 ;
Connect CoutService.Port3 , BaseFichier.Port2 ;
Connect Alarme.Port1 , Acquisition.Port3 ;
Connect Alarme.Port1 , CoutService.Port3 ;
Connect Alarme.Port1 , cc:Calcule_Couts.Port2 ;
Connect cc:Calcule_Couts.Port5 , OpPartException.Port1 ;
Connect cc:Calcule_Couts.Port4 , BaseDonnée.Port2 ;
Connect Alarme.Port2 , OpPartException.Port1 ;

Glue Port3 to BaseFichier.Port3;
Glue Port2 to BaseDonnée.Port3;
Glue Port1 to OpPartController.Port2;

Public Port Port1 ;
   {     provide  void methode_( );    }

Public Port Port2 ;
   {     requiers void methode_( );    }

Public Port Port3 ;
   {     requiers void methode_( );    }

public void Execute()
  { // Fonction permet de lancer les traitements des sous composants;   }
}

public component class Acquisition
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {provide Int  Action_4 (); }
   public void Acquisition()
   {
```

```
   }
 }
public component class BaseDonnée
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {provide Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {provide Int  Action_4 (); }
   public void BaseDonnée()
   {

   }
 }
public component class pr:Préparation
 {
    public port null
    public port null
    public port null
    public port null
 }
public component class Alarme
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {provide Int  Action_2 (); }
   public void Alarme()
   {

   }
 }
public component class BaseFichier
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {provide Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {provide Int  Action_4 (); }
   public void BaseFichier()
   {

   }
 }
public component class CoutService
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {provide Int  Action_4 (); }
   public void CoutService()
   {

   }
 }
public component class cc:Calcule_Couts
 {
    public port null
    public port null
    public port null
    public port null
    public port null
 }
public component class Facturation
 {
    public port Port1
          {provide Int  Action_1 (); }
```

```
    public port Port2
          {requiers Int  Action_2 (); }
    public port Port3
          {provide Int  Action_3 (); }
   public void Facturation()
   {

   }
 }
public component class OpPartController
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
   public void OpPartController()
   {

   }
 }
public component class OpPartException
 {
    public port Port1
          {provide Int  Action_1 (); }
   public void OpPartException()
   {

   }
 }
```

Code 7.6 :

```
package  Préparation;
import  Package1,Package2,;
component  Préparation
{
ports
 {
ServerPort Port1
    {
     accesspoint
               {
                  ServerActionPoint Point1  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point2  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle1,
      rule Régle1
        {
         precondition :
         pattern :
         postcondition :
        }


    }
ClientPort Port2
    {
     accesspoint
               {
                  ClientActionPoint Point3  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point4  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle2,
      rule Régle2
        {
```

```
         precondition :
         pattern :
         postcondition :
        }


    }
ClientPort Port3
    {
     accesspoint
               {
                  ClientActionPoint Point5  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point6  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle3,
      rule Régle3
        {
         precondition :
         pattern :
         postcondition :
        }


    }
PeerPort Port4
    {
     accesspoint
               {
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle1,Régle2,Régle3,Régle4,
      rule Régle1
        {
         precondition :
         pattern :
         postcondition :
        }

      rule Régle2
        {
         precondition :
         pattern :
         postcondition :
        }

      rule Régle3
        {
         precondition :
         pattern :
         postcondition :
        }

      rule Régle4
        {
         precondition :
         pattern :
         postcondition :
        }


    }
 } // End of Port
connector
{
  connector TypeNomConnector
          {
```

```
             roles Liste_des_roles;
            architectures
              {

              }
            actioncontext
              {

              }
            behavior
              {

              }
         }
}
operativepart
 {
   components
      {
       Décodage  Nom_Instance0;
       Controle  Nom_Instance1;
       Journalisation  Nom_Instance2;
       Harmonisation  Nom_Instance3;
       Dédoblonnage  Nom_Instance4;
       OpPartController  Nom_Instance5;
       OpPartExceptionr  Nom_Instance6;
      }
   connexions
      {
         DelegateConnector Connecteur0 { map OpPartExceptionr.Port1 , Port4 };
         DelegateConnector Connecteur1 { map OpPartController.Port1 , Port1 };
         DelegateConnector Connecteur2 { map Harmonisation.Port5 , Port3 };
         DelegateConnector Connecteur3 { map Dédoblonnage.Port2 , Port3 };
         DelegateConnector Connecteur4 { map Décodage.Port3 , Port2 };
         DelegateConnector Connecteur5 { map Controle.Port3 , Port2 };
         DelegateConnector Connecteur6 { map Décodage.Port5 , Port2 };
      }
 } // End PartOperative
controlpart
 {
   components
      {
        OpPartController opPartController ;
        ActiveOpPartException OpPartException ;
      }
   connexions
      {
       TupeConnexion NomConnecteru
            {
                Binding
                   {
                    bind OpPartController.Port2 to Dédoblonnage.Port1
                    bind OpPartController.Port3 to Journalisation.Port1
                    bind OpPartController.Port4 to Harmonisation.Port1
                    bind OpPartController.Port5 to Décodage.Port1
                    bind OpPartController.Port6 to Controle.Port1
                   }
                actioncontext
                   {

                   }
                behavior
                   {

                   }
            }
      }
 } // End Controlpart
properties
 {
architecture
  {
     environment Controle
        {
           machine Localhost ;
           os UNIX ;
        }
```

```
     environment
        {
           machine Localhost ;
           os UNIX ;
        }
     environment
        {
           machine Localhost ;
           os UNIX ;
        }
  }

deployment
  {
    deploymentcase{THREA, PROCESS, COMPOSITE};
    deploymentmap map1
          {
            deploy  Controle  as  PROCESS  in  localhost ;
            deploy  Harmonisation  as  THREAD  in  10.1.1.155 ;
          }
  }

 }
} //End of component
```

Code 7.7 :

```
import  java.io.BufferedReader;
import  java.io.BufferedWriter;
import  java.io.File;
import  java.io.FileInputStream;
import  java.io.FileOutputStream;
import  java.io.IOException;
import  java.io.InputStreamReader;
import  java.io.OutputStreamWriter;
import  java.sql.*;
import  java.sql.DriverManager;
import  java.sql.Connection;
import  java.sql.Statement;

public component class Préparation
{
Private final Décodage Cmp01 = new Décodage( ) ;
Private final Controle Cmp11 = new Controle( ) ;
Private final Journalisation Cmp21 = new Journalisation( ) ;
Private final Harmonisation Cmp31 = new Harmonisation( ) ;
Private final Dédoblonnage Cmp41 = new Dédoblonnage( ) ;
Private final OpPartController Cmp51 = new OpPartController( ) ;
Private final OpPartExceptionr Cmp61 = new OpPartExceptionr( ) ;

Connect OpPartController.Port2 , Dédoblonnage.Port1 ;
Connect OpPartController.Port3 , Journalisation.Port1 ;
Connect OpPartController.Port4 , Harmonisation.Port1 ;
Connect OpPartController.Port5 , Décodage.Port1 ;
Connect OpPartController.Port6 , Controle.Port1 ;

Glue Port4 to OpPartExceptionr.Port1;
Glue Port1 to OpPartController.Port1;
Glue Port3 to Harmonisation.Port5;
Glue Port3 to Dédoblonnage.Port2;
Glue Port2 to Décodage.Port3;
Glue Port2 to Controle.Port3;
Glue Port2 to Décodage.Port5;

Public Port Port1 ;
   {     provide  void methode_( );    }

Public Port Port2 ;
   {     requiers void methode_( );    }

Public Port Port3 ;
   {     requiers void methode_( );    }

Public Port Port4 ;
   {     provide  void methode_( )     }
```

```
public void Execute()
  { // Fonction permet de lancer les traitements des sous composants;   }
}

public component class Décodage
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {provide Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {requiers Int  Action_4 (); }
    public port Port5
          {requiers Int  Action_5 (); }
    public port Port6
          {provide Int  Action_6 (); }
   public void Décodage()
   {

   }
 }
public component class Controle
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {provide Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {requiers Int  Action_4 (); }
    public port Port5
          {provide Int  Action_5 (); }
   public void Controle()
   {

   }
 }
public component class Journalisation
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {provide Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {requiers Int  Action_4 (); }
    public port Port5
          {requiers Int  Action_5 (); }
    public port Port6
          {provide Int  Action_6 (); }
   public void Journalisation()
   {

   }
 }
public component class Harmonisation
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {provide Int  Action_2 (); }
    public port Port3
          {provide Int  Action_3 (); }
    public port Port4
          {requiers Int  Action_4 (); }
    public port Port5
          {requiers Int  Action_5 (); }
    public port Port6
          {provide Int  Action_6 (); }
   public void Harmonisation()
   {
```

```
   }
 }
public component class Dédoblonnage
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {requiers Int  Action_4 (); }
    public port Port5
          {provide Int  Action_5 (); }
   public void Dédoblonnage()
   {

   }
 }
public component class OpPartController
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {requiers Int  Action_4 (); }
    public port Port5
          {requiers Int  Action_5 (); }
    public port Port6
          {requiers Int  Action_6 (); }
   public void OpPartController()
   {

   }
 }
public component class OpPartExceptionr
 {
    public port Port1
          {provide Int  Action_1 (); }
   public void OpPartExceptionr()
   {

   }
 }
```

Code 7.8 :

```
<Architecture>
<Composite Type = 'Préparation'  Name = 'Préparation' >
<Component Type = 'Décodage'  Width = '120'  Height = '60'   X = 259  Y = 34  Name =
'Décodage'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '434'  Y = '44'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '424'  Y = '56'  Name = 'Port2'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '414'  Y = '68'  Name = 'Port3'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '404'  Y = '80'  Name = 'Port4'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_4'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '194'  Y = '44'  Name = 'Port5'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_5'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '204'  Y = '56'  Name = 'Port6'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_6'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'Controle'  Width = '120'  Height = '60'   X = 607  Y = 25  Name =
'Controle'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '782'  Y = '35'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
```

```
</Port>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '772'  Y = '47'  Name = 'Port2'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '762'  Y = '59'  Name = 'Port3'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '752'  Y = '71'  Name = 'Port4'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_4'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '542'  Y = '35'  Name = 'Port5'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_5'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'Journalisation'  Width = '120'  Height = '60'   X = 569  Y = 133  Name =
'Journalisation'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '744'  Y = '143'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '734'  Y = '155'  Name = 'Port2'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '724'  Y = '167'  Name = 'Port3'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '714'  Y = '179'  Name = 'Port4'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_4'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '504'  Y = '143'  Name = 'Port5'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_5'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '514'  Y = '155'  Name = 'Port6'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_6'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'Harmonisation'  Width = '120'  Height = '60'   X = 535  Y = 263  Name =
'Harmonisation'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '710'  Y = '273'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '700'  Y = '285'  Name = 'Port2'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '690'  Y = '297'  Name = 'Port3'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '680'  Y = '309'  Name = 'Port4'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_4'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '470'  Y = '273'  Name = 'Port5'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_5'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '480'  Y = '285'  Name = 'Port6'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_6'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'Dédoblonnage'  Width = '120'  Height = '60'   X = 151  Y = 134  Name =
'Dédoblonnage'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '321'  Y = '144'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '311'  Y = '156'  Name = 'Port2'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '301'  Y = '168'  Name = 'Port3'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '291'  Y = '180'  Name = 'Port4'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_4'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '91'  Y = '144'  Name = 'Port5'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_5'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'OpPartController'  Width = '120'  Height = '60'   X = 214  Y = 332  Name =
'OpPartController'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '384'  Y = '342'  Name = 'Port1'/>
```

```
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '374'  Y = '354'  Name = 'Port2'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '364'  Y = '366'  Name = 'Port3'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '354'  Y = '378'  Name = 'Port4'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_4'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '154'  Y = '342'  Name = 'Port5'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_5'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '164'  Y = '354'  Name = 'Port6'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_6'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'OpPartExceptionr'  Width = '120'  Height = '60'   X = 588  Y = 344  Name =
'OpPartExceptionr'>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '758'  Y = '354'  Name = 'Port1'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
</Component>
<Connector Type = 'Binding'  Name = 'Connector1'>
<ConnectedPort>OpPartController->Port2</ConnectedPort>
<ConnectedPort>Dédoblonnage->Port1</ConnectedPort>
</Connector>
<Connector Type = 'Binding'  Name = 'Connector2'>
<ConnectedPort>OpPartController->Port3</ConnectedPort>
<ConnectedPort>Journalisation->Port1</ConnectedPort>
</Connector>
<Connector Type = 'Binding'  Name = 'Connector3'>
<ConnectedPort>OpPartController->Port4</ConnectedPort>
<ConnectedPort>Harmonisation->Port1</ConnectedPort>
</Connector>
<Connector Type = 'Binding'  Name = 'Connector4'>
<ConnectedPort>OpPartController->Port5</ConnectedPort>
<ConnectedPort>Décodage->Port1</ConnectedPort>
</Connector>
<Connector Type = 'Binding'  Name = 'Connector5'>
<ConnectedPort>OpPartController->Port6</ConnectedPort>
<ConnectedPort>Controle->Port1</ConnectedPort>
</Connector>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port4'>
<PortRef>OpPartExceptionr->Port1</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port1'>
<PortRef>OpPartController->Port1</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port3'>
<PortRef>Harmonisation->Port5</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port3'>
<PortRef>Dédoblonnage->Port2</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port2'>
<PortRef>Décodage->Port3</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port2'>
<PortRef>Controle->Port3</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port2'>
<PortRef>Décodage->Port5</PortRef>
</DelegatedPort>
</Composite>
</Architecture>
```

Code 7.9 :

```
package  Calcule_Couts;
import  Package1,Package2,;
component  Calcule_Couts
{
ports
 {
ClientPort Port1
```

```
    {
     accesspoint
               {
                  ClientActionPoint Point1  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point2  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle1,
      rule Régle1
        {
         precondition :
         pattern :
         postcondition :
        }


    }
ClientPort Port2
    {
     accesspoint
               {
                  ClientActionPoint Point3  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point4  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle2,
      rule Régle2
        {
         precondition :
         pattern :
         postcondition :
        }


    }
ClientPort Port3
    {
     accesspoint
               {
                  ClientActionPoint Point5  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point6  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  régle1,
      rule régle1
        {
         precondition :
         pattern :
         postcondition :
        }


    }
ServerPort Port4
    {
     accesspoint
               {
                  ServerActionPoint Point7  Nom_Methode(0,SYNCHRONE);
                  DataPoint Point8  Nom_Methode(0,SYNCHRONE);
               }
     actioncontext
        {
```

```
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle4,
      rule Régle4
        {
         precondition :
         pattern :
         postcondition :
        }


    }
PeerPort Port5
    {
     accesspoint
               {
               }
     actioncontext
        {
          action  ...   implemented by   ...   ;
        }
     behavior
    {
      rules  Régle5,
      rule Régle5
        {
         precondition :
         pattern :
         postcondition :
        }


    }
 } // End of Port
connector
{
  connector TypeNomConnector
          {
              roles Liste_des_roles;
             architectures
               {

               }
             actioncontext
               {

               }
             behavior
               {

               }
          }
}
operativepart
 {
   components
      {
       Reconciltion  Nom_Instance0;
       Usage_Trait  Nom_Instance1;
       OpPartController  Nom_Instance2;
       OpPartException  Nom_Instance3;
      }
   connexions
      {
         DelegateConnector Connecteur0 { map Usage_Trait.Port4 , Port3 };
         DelegateConnector Connecteur1 { map Usage_Trait.Port3 , Port2 };
         DelegateConnector Connecteur2 { map Usage_Trait.Port2 , Port1 };
         DelegateConnector Connecteur3 { map Usage_Trait.Port5 , Port1 };
         DelegateConnector Connecteur4 { map Reconciltion.Port2 , Port2 };
      }
 } // End PartOperative
controlpart
 {
   components
      {
```

```
        OpPartController opPartController ;
        ActiveOpPartException OpPartException ;
      }
   connexions
      {
       TupeConnexion NomConnecteru
            {
                Binding
                   {
                    bind OpPartController.Port2 to Reconciltion.Port1
                    bind OpPartController.Port3 to Usage_Trait.Port1
                    bind Reconciltion.Port3 to OpPartException.Port1
                    bind Usage_Trait.Port6 to OpPartException.Port1
                   }
                actioncontext
                   {

                   }
                behavior
                   {

                   }
            }
      }
 } // End Controlpart
properties
 {
architecture
  {
     environment ENV1
        {
           machine Localhost ;
           os WINDOWS ;
        }
     environment ENV2
        {
           machine 10.1.1.155 ;
           os UNIX ;
        }
     environment
        {
           machine Localhost ;
           os UNIX ;
        }
  }

deployment
  {
    deploymentcase{THREA, PROCESS, COMPOSITE};
    deploymentmap map1
          {
            deploy  Usage_Trait  as  PROCESS  in  localhost ;
            deploy    as  PROCESS  in   ;
          }
  }

 }
} //End of component
```

Code 7.10 :

```
import  java.io.BufferedReader;
import  java.io.BufferedWriter;
import  java.io.File;
import  java.io.FileInputStream;
import  java.io.FileOutputStream;
import  java.io.IOException;
import  java.io.InputStreamReader;
import  java.io.OutputStreamWriter;
import  java.sql.*;
import  java.sql.DriverManager;
import  java.sql.Connection;
import  java.sql.Statement;

public component class Calcule_Couts
{
```

```
Private final Reconciltion Cmp01 = new Reconciltion( ) ;
Private final Usage_Trait Cmp11 = new Usage_Trait( ) ;
Private final OpPartController Cmp21 = new OpPartController( ) ;
Private final OpPartException Cmp31 = new OpPartException( ) ;

Connect OpPartController.Port2 , Reconciltion.Port1 ;
Connect OpPartController.Port3 , Usage_Trait.Port1 ;
Connect Reconciltion.Port3 , OpPartException.Port1 ;
Connect Usage_Trait.Port6 , OpPartException.Port1 ;

Glue Port3 to Usage_Trait.Port4;
Glue Port2 to Usage_Trait.Port3;
Glue Port1 to Usage_Trait.Port2;
Glue Port1 to Usage_Trait.Port5;
Glue Port2 to Reconciltion.Port2;

Public Port Port1 ;
   {     requiers void methode_( );    }

Public Port Port2 ;
   {     requiers void methode_( );    }

Public Port Port3 ;
   {     requiers void methode_( );    }

Public Port Port4 ;
   {     provide  void methode_( );    }

Public Port Port5 ;
   {     provide  void methode_( )     }

public void Execute()
  { // Fonction permet de lancer les traitements des sous composants;   }
}

public component class Reconciltion
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
    public port Port3
          {provide Int  Action_3 (); }
   public void Reconciltion()
   {

   }
 }
public component class Usage_Trait
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
    public port Port4
          {requiers Int  Action_4 (); }
    public port Port5
          {requiers Int  Action_5 (); }
    public port Port6
          {provide Int  Action_6 (); }
   public void Usage_Trait()
   {

   }
 }
public component class OpPartController
 {
    public port Port1
          {provide Int  Action_1 (); }
    public port Port2
          {requiers Int  Action_2 (); }
    public port Port3
          {requiers Int  Action_3 (); }
   public void OpPartController()
   {
```

```
   }
 }
public component class OpPartException
 {
    public port Port1
          {provide Int  Action_1 (); }
   public void OpPartException()
   {

   }
 }
```

Code 7.11 :

```
<Architecture>
<Composite Type = 'Calcule_Couts'  Name = 'Calcule_Couts' >
<Component Type = 'Reconciltion'  Width = '120'  Height = '60'   X = 100  Y = 100  Name =
'Reconciltion'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '275'  Y = '110'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '265'  Y = '122'  Name = 'Port2'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '255'  Y = '134'  Name = 'Port3'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'Usage_Trait'  Width = '120'  Height = '60'   X = 492  Y = 126  Name =
'Usage_Trait'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '667'  Y = '136'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '657'  Y = '148'  Name = 'Port2'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '647'  Y = '160'  Name = 'Port3'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '637'  Y = '172'  Name = 'Port4'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_4'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '427'  Y = '136'  Name = 'Port5'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_5'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '437'  Y = '148'  Name = 'Port6'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_6'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'OpPartController'  Width = '120'  Height = '60'   X = 198  Y = 260  Name =
'OpPartController'>
<Port Type = 'ServerPort'  Height = '10' Width = '10'  X = '373'  Y = '270'  Name = 'Port1'/>
<AccessPoint Type = 'ServerActionPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '363'  Y = '282'  Name = 'Port2'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_2'  ParamaterType = 'Int'  />
</Port>
<Port Type = 'ClientPort'  Height = '10' Width = '10'  X = '353'  Y = '294'  Name = 'Port3'/>
<AccessPoint Type = 'ClientActionPoint'  Action Name = 'Action_3'  ParamaterType = 'Int'  />
</Port>
</Component>
<Component Type = 'OpPartException'  Width = '120'  Height = '60'   X = 477  Y = 271  Name =
'OpPartException'>
<Port Type = 'PeerPort'  Height = '10' Width = '10'  X = '652'  Y = '281'  Name = 'Port1'/>
<AccessPoint Type = 'PeerPoint'  Action Name = 'Action_1'  ParamaterType = 'Int'  />
</Port>
</Component>
<Connector Type = 'Binding'  Name = 'Connector1'>
<ConnectedPort>OpPartController->Port2</ConnectedPort>
<ConnectedPort>Reconciltion->Port1</ConnectedPort>
</Connector>
<Connector Type = 'Binding'  Name = 'Connector2'>
<ConnectedPort>OpPartController->Port3</ConnectedPort>
<ConnectedPort>Usage_Trait->Port1</ConnectedPort>
</Connector>
```

```
<Connector Type = 'Binding'  Name = 'Connector3'>
<ConnectedPort>Reconciltion->Port3</ConnectedPort>
<ConnectedPort>OpPartException->Port1</ConnectedPort>
</Connector>
<Connector Type = 'Binding'  Name = 'Connector4'>
<ConnectedPort>Usage_Trait->Port6</ConnectedPort>
<ConnectedPort>OpPartException->Port1</ConnectedPort>
</Connector>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port3'>
<PortRef>Usage_Trait->Port4</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port2'>
<PortRef>Usage_Trait->Port3</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port1'>
<PortRef>Usage_Trait->Port2</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port1'>
<PortRef>Usage_Trait->Port5</PortRef>
</DelegatedPort>
<DelegatedPort Type = 'DelegatedPort'  Height = '10'  width = '10'  Name = 'Port2'>
<PortRef>Reconciltion->Port2</PortRef>
</DelegatedPort>
</Composite>
</Architecture>
```

Annexe 2

Grammaire de la spécification textuelle:

```
Spécification : :=  statement ;
statement : :=    "package"  NomPackage ;
                  "import"   ListePackage ;
                  "component"  NomTypeComponent
                          '{'
                              SpecificationTypePort ;
                              SpecificationTypeConnector ;
                              SpecificationTypeComponent;
                              SpecificationPorts ;
                              Specificationoperatepart ;
                              Specificationcontrolepart ;
                              Specificationbehavior ;
                                Specificationproprietes ;
                          '}'

ListePackage : := NomPackage (  ', '  NomPackage )* ;
NomPackage  : := Alpha(Alpha | Digit)*
NomConnector : := Alpha(Alpha | Digit)*
NomRole : := Alpha(Alpha | Digit)*
NomInstance : := Alpha(Alpha | Digit)*
NomInteraction : := Alpha(Alpha | Digit)*
NomComponent: := Alpha(Alpha | Digit)*
NomTypeComponent : := Alpha(Alpha | Digit)*
NomMethode  : := Alpha(Alpha | Digit)*
variablelogigue : := Alpha(Alpha | Digit)*
Digit : ='0'….'9'
Alpha: ='a'….'z' | 'A'…..'Z'

SpecificationTypePort : := "port"   '{'   (Définetypeport)*   '}'

                                    Définetypeport : := "port"   NomTypedePort
                                '{'
                                            "accesspoint"    '{'   (DéfineAccessPoint)*    '}'
                                            "actioncontext"  '{'  (DéfineActionContext)*   '}'
                                            "behavior"       '{'      (DéfineBehavior)*    '}
                                 '}'

SpecificationPorts : := "ports"   '{'   (DéfinePorts)*   '}'
                                      DéfinePorts : := TypedePort NomPort
                                '{'
                                            "accesspoint"    '{'   (DéfineAccessPoint)*    '}'
                                            "actioncontext"  '{'  (DéfineActionContext)*   '}'
                                            "behavior"       '{'      (DéfineBehavior)*    '}
                                  '}'

DéfineAccessPoint : := TypedePointAccés NomInstance ListedeParamétres ;
TypedePointAccés : := ClientActionPoint | ServerActionPoint | IntDataPoint | StringDataPoint
| DataPoint | DoublePoint;

NomInstance : := Alpha (Alpha | Digit)*
                                     NomTypedePort: := Alpha (Alpha | Digit)*
                                       TypedePort: := Alpha (Alpha | Digit)*
NomPort : := Alpha(Alpha | Digit)*
ListedeParamétres : :=  '('  Ident  (,Ident)*  ',' Synch  ')' ;
Ident : := in | out | 0 ;
Synch : := SYNCHRONE | ASYNCHRONE ;
```

```
DéfineActionContext : :=  "action"  ListeActions " implemented by "  SignatureMethode ;
                                  |  "action" ListeActions ;

ListeActions : := NomAction (',' NomActions)* ;
NomAction: := Alpha(Alpha | Digit)* ;
SignatureMethode : := InstanceDataPoint NomMethode '(' (ListeParametres) ? ')' ;
InstanceDataPoint: := Alpha(Alpha | Digit)* ;
ListeParametres : := Param (',' Param)* ;
Param : := Alpha (Alpha | Digit)* ;
NomMethode : :=  Alpha (Alpha | Digit)*
DéfineBehavior : := "behavior"
                                '{'
                                    DeclarationVar ;
                                        "rules"  ListeRules ;
                                         (DéfineRules)* ;
                                 '}'

DeclarationVar : := TypeBool variable intialisation ;
intialisation : := ('='  varbool) ;
varbool : := false | true ;
TypeBool : := Boolean ;
Variable : := Alpha (Alpha | Digit)* ;

ListeRules : := NomRule (',' NomRule)* ;
NomRule: := Alpha(Alpha | Digit)* ;
DéfineRules : := "rule" NomRule
                                '{'
                                   "précondition"      ' : '  (ExpressionLogigue)? ;
                                        "pattern"              ' : '  (ActionNormalePanne)?;
                                        "postcondition"    ' : '  (ActionPositionnement)?;
                                        ("failure"              ' : '  ActionPanne ) ? ;
                                '}'
ExpressionLogigue : := (!) ? variablelogigue ( ',' (!) ?variablelogigue)* ;
ActionNormalePanne : := ListeActions ;
ActionPositionnement : := variablelogigue '=' varbool ;
variablelogigue : := Alpha (Alpha | Digit)* ;

Specificationbehavior : := "behavior"
              '{'
                 (BehviorControlpart)*;
                 BehaviorSEAL ;
                 RefFile ;
               '}' ;

                        SpecificationTypeConnector : := "connector"  '{'  (Défineconnector)*  '}'
Défineconnector : := "connector" NomNouveauTypeConnector
                                                              '{'
                                                                 (DéfineRole)* ;
                                                                 DéfineArchitecture ;
                                                                 DéfineContextAction ;
                                                        DéfineComportement ;
                                                              '}' ;

DéfineRole : :=  " roles "  ListeRoles ;
ListeRoles : := NomRole (',' NomRole)* ;
DéfineArchitecture : := "architecture"
                                        '{'
                                           (DeclarConnector)* ;
                                           (AttachConnector)* ;
                                         '}' ;
```

```
DeclarConnector : :=  TypedeConnector ListedeNomInstance ;
ListedeNomInstance : := NomInstance (,'NomInstance)* ;
AttachConnector : := " map " NomRoleAtt "to" NomRoleAtt (,'NomRoleAtt)* ;
NomRoleAtt : :=NomRole | NomInstance  '.' NomRole ;
DéfineContextAction : := " actioncontext "  '{'  ListeAction  '}' ;
DéfineComportement : :="behavior"
                                    '{'  (Interaction)* ;   '}' ;

Interaction : := " interaction  " NomInteraction '{'  (ListeInteraction)*  '}' ;
ListeInteraction : := NomRole '.'  " request " '(' parametre ')' ;
                    |  NomRole '.'  " invoke " '(' parametre ')' ;
                    |  NomRole '.'  " send " '(' parametre ')' ;
                    |  NomRole '.'  " receive " '(' parametre ')' ;
                       "success" ;

parametre : :=Alpha (Alpha | Digit)* ;

Specificationcontrolepart : :="controlpart"
                '{'
                  DéfineComp ;
                  DéfineConnex ;
                '}' ;

DéfineComp : := "components" '{ '    (DéfineInstanceComponent)  *  '}'
DéfineInstanceComponent : := TypeComponent InstanceComponent ;

InstanceComponent : := NomComponent (,'NomComponent)* ;
DéfineConnex : := "connexions" '{ '
                                      (DéfineAllConnexion) * ;
                                       DéfineContextAction ;
                                       DéfineComportement ;
                                    '}' ;

DéfineAllConnexion : :=  TypeConnector  NomConnector '{'  (Binding)*   '}' ;
Binding : := " Binding "  '{'   (bind)*   '}'
Bind : := "bind" " NRole "to" NRole ;
NRole : :=NomRole | NomInstance  '.' NomRole ;
DéfineContextAction : := " actioncontext "  '{'  ListeAction  '}' ;
DéfineComportement : :="behavior"
                                    '{'   (Interaction)* ;   '}' ;

Interaction : := " interaction  " NomInteraction '{'  (ListeInteraction)*  '}' ;
ListeInteraction : := NomRole '.'  " request " '(' parametre ')' ;
                    |  NomRole '.'  " invoke " '(' parametre ')' ;
                    |  NomRole '.'  " send " '(' parametre ')' ;
                    |  NomRole '.'  " receive " '(' parametre ')' ;

                                    Specificationoperatepart : := "operativepart"
                      '{ '
                            (DéfineComponent)* ;
                            (DéfineConnexion)* ;
                      ' } ' ;

DéfineComponent : := "components" '{ '    (DéfineInstanceComponent)  *  '}'
DéfineInstanceComponent : := TypeComponent InstanceComponent ;
InstanceComponent : := NomComponent (,NomComponent)* ;
DéfineConnexion : := "connexions"  '{ '    (DéfineDelegateConnexion)  *  '}'
                    DéfineDelegateConnexion: := "DelegateConnector" NomConnector
                                            '{'  "map"   connector  ','  connector ; '}' ;
```

```
connector : := NomComposant '.' NomPort | NomPort ;

                                        Specificationproprietes : :=  "properties"
                                          '{'
                                            architecture_prop ;
                                            deployment_prop ;
                                          '}' ;

architecture_prop : := "architecture"
                                       '{'    (Environnement)* ; '}' ;

Environnement  : := "environment" NomEnvironment
                                                  '{'
                                                    "machine"  NomDomain ;
                                                     "os"   SystExp ;
                                                   '}'

NomDomain : := Alpha(Alpha | Digit| ' . ' )* ;
NomEnvironment : := Alpha(Alpha | Digit| ' . ' )* ;
Digit : :='0'….'9' ;
Alpha: :='a'….'z' | 'A'…..'Z' ;
SystExp : := Windows | UNIX ;
deployment_prop : := deployment
                                    '{'
                                      deployment_case ;
                                      (deployment_map )*;
                                    '}' ;

deployment_case : :=  "deploymentcase"  '{' case '}' ;
case : := case1 (','case1)* ;
case1 : := THREAD |  PROCESS  |  COMPOSITE ;
deployment_map : := "deploymentmap" NomDeployment '{'  deploy_comp_inst  '}' ;
NomDeployment : := Alpha(Alpha | Digit| ' . ' )* ;
deploy_comp_inst : := "deploy"  NomComposite "as" case1 "in" NomEnvironment ;
NomComposite: := Alpha(Alpha | Digit| ' . ' )* ;
```

REFERENCES

1. Projet ACCORD (Assemblage de composants par contrats en environnement ouvert et réparti) "Etat de l'art sur les Langages de Description d'Architecture" (Juin 2002). Revue technique, Référence : Livrable 1.1-2, France, Date : Juin 2002.

2. Leymonerie Fabien, "ASL : un langage et des outils pour les styles architecturaux, Contribution à la description d'architectures dynamiques", Thèse doctorat, Université de Savoie. France, Décembre 2004.

3. Anne-Marie Déplanche, Sébastien Faucou, "Les langages de description d'architecture *(ADL)* pour le temps réel", Actes de l'École d'Éte Temps Réel (ETR'05), pages 31-46, Nancy, France, 2005.

4. Projet ACCORD (Assemblage de composants par contrats en environnement ouvert et réparti). "Assemblage de composants par contrats, Etat de l'art et de la standardisation". Revue technique de projet, 2002.

5. C.CHAUDET. "p-Space : langage et outils pour la description d'architectures évolutives à composants dynamiques. Formalisation d'architectures logicielles et industrielles". Thèse de doctorat, Université de SAVOIE, France. Décembre 2002.

6. Olivier Barais, "Construire et Maîtriser l'évolution d'une Architecture Logicielle à base de Composants". Thèse de doctorat, Département de formation doctorale en informatique, Ecole doctorale SPI Lille, UFR IEEA, Novembre 2005.

7. Michel Nolard, "Intégrer l'Architecture d'une Application avec son Implémentation grâce à ArchJava". Rapport Interne, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Rue Grandgagnage, 21, Belgium, Mai 2003.

8. http://www.archjava.org.

9. Barais, Laurence Duchien et Lionel Seinturier Olivier, "SafArchie ADL : Construire et Déployer une Architecture Logicielle Typée", Publication LIFL 2004 N°10, laboratoire Fondamentale d'informatique de Lille, Septembre 2004.

10. R. J. Allen, A Formal Approach to Software Architecture, PHD Thesis, Carnegie Mellon University, 1997.

11. Jorge VILLALOBOS, "Fédération de Composants : une Architecture Logicielle pour la Composition par Coordination", Thèse de doctorat, UNIVERSITE JOSEPH FOURIER – GRENOBLE, Juillet 2003.

12. Z. Mammeri, "Introduction au langage de description et de spécification, SDL « Specification and Description Language » ", Thèse de doctorat Université Paul

Sabatier – Toulouse, France, 2001.

13. D.Hubert, G.Sébastien, M.Chokri, "Un langage d'action pour le développement UML de systèmes embarqués temps réel", IDM'05 Premières Journées sur l'Ingénierie Dirigée par les Modèles, Paris, 30 juin, 1 Juillet 2005.

14. D. Bennouar, T.Khammaci and A.Henni, "A Layered Connector Model for A Complex Software System Design", ACIT' The International Arab Conference on Information Technology, Constantine, Algeria, 2004.

15. D. Bennouar, T. Khammaci, A. Henni, "IASA, une approche intégrée pour l'architecture logicielle, Rapport Interne N° 03/07, Laboratoire LRDSI, Département d'Informatique, Université Saad Dahlab, Blida, Janvier 2007.

16. N.SABRI, "Une Architecture à base de Composants CORBA pour des Services Personnalisés", Thèse doctorat, Université d'Evry-Val-d'Essonne, France, Juin 2003

17. Renaud Blanch, "Architecture logicielle et outils pour les interfaces hommes – machines graphiques avancées", Thèse doctorat, L'UNIVERSIT´E PARIS XI, ORSAY, France, Septembre 2005.

18. Alain LE GUENNEC, "Génie Logiciel et Méthodes Formelles avec UML Spécification, Validation et Génération de tests". Thèse de doctorat, Équipe d'accueil : IRISA/PAMPA, École Doctorale : MATISSE, Composante universitaire : IFSIC, Université de Rennes 1, France, Juin 2001.

19. François Pennaneac'h, "UML : de l'action à la réflexion", Thèse de doctorat Université de Rennes 1, France, Novembre 2001.

20. C.TIBERMACINE, "Contractualisation de l'évolution architecturale de Logiciels à base de composants : une approche pour la préservation de la qualité". Thèse de doctorat, Université de Bretagne Sud, France, Octobre 2006.

21. Vincent PORTIGLIATTI,"Contribution à l'allocation dynamique de ressources pour les composants expressifs dans les systèmes répartis", Thèse doctorat, l'u.f.r des sciences et techniques de l'université de franche-comt, France, Décembre 2003.

22. A.Hachichi, C.Martin, G.Thomas, S.Patarin, B.Folliot, "Reconfigurations dynamiques de services dans un intergiciel à composants CORBA CCM". Computer Science, Networking and Internet Architecture, Journal référence: DECOR04 (2004) 159-170.

23. Jean Michel DOUDOUX, "Développons en Java avec Eclipse". Tutorial JAVA et Eclipse, Version électronique, Chapitre 2 – Chapitre 15.

24. N.Kanaoui, T.Khammaci et M.Oussaleh, "Les ADLS : une voie prometteuse pour les architectures logicielle". Actes du congrès Agents, Logiciels, Coopération,

Apprentissage & Activité Humaine, (ALCAA 2003), Bayonne, France, Septembre 2003,

25. Frédéric FONDEMENT, Michel HASSENFORDER, "Création d'un langage d'action pour un logiciel MDA". Rapport de stage Code : DRT GEII. 2 em Année, BJEXION France, Décembre 2002.

26. Tahar Khammaci, Adel Smeda et Mourad Oussalah, "ALBACO : Une architecture logicielle à base de composants et d'objets pour la description de systèmes complexes". Actes du congrès Agents, Logiciels, Coopération, Apprentissage & Activité Humaine, (ALCAA 2003), Bayonne, France, Septembre 2003.

27. F.Peschanski, T.Meurisse, J-P.Briot, "Les Composants Logiciels : Evolution technologique ou nouveau paradigme ? ". Conference OCM 2000, Nantes, France, Mai 2000.

28. Djamal BENNOUAR, Tahar KHAMMACI and Abderrezak HENNI, "Vers la définition d'un modèle de connecteurs logiciels pour la description de systèmes complexes". Journée CNRS-GDR-ALP-OCM, Langages et modèles à Objets (LMO'04), Lille, mars 2004.

29. N.MEDVIDOVIC, R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Software Engineering, v.26 n.1, p.70-93, January 2000.

30. http://www.journaldunet.com/developpeur/tutoriel/cpt/040915-passer-a-uml2.shtml

31. M. Oussalah, A. Smeda et T. Khammaci, ``Software Connectors Reuse in Components based Systems, Proceedings of the 2003 IEEE International Conference on Information Reuse and Integration (IRI2003), Las Vegas, NV, USA, October 2003.

32. G. Zelesnik, The *UniCon Language Reference Manual*, School of Computer Science Carnegie Mellon Universtity, Pittsburgh , Mai 1996.

33. G. Zelesnik, *The UniCon Language User Manual*, School of Computer Science Carnegie Mellon Universtity, Pittsburgh, Juin 1996.

34. Tahar Khammaci, Adel Smeda et Mourad Oussalah, ALBACO : Une architecture logicielle à base de composants et d'objets pour la description de systèmes complexes, LINA – FRE CNRS 2729 - Université de Nantes.

35. T. Khammaci, A. Smeda et M. Oussalah, " Coexistence of Object-oriented Modeling and Architectural Description", Advances in Software Engineering and Knowledge Engineering, S.K. Chang Editor, World Scientific Publishing Co., October 2005, ISBN: 981-256-273-7.

36. G. Wiederhold, “Mediators in the Architecture of Future Information Systems”, IEEE Computer Magazine, vol. 25, no. 3, March 1992, pp.38-49.

37. MARCHETTI C., VERDE L., BALDONI R., « CORBA request portable interceptors: a performance analysis », Proceedings of the 3rd International Symposium on Distributed Objects and Applications, Rome, Italy, September 2001.