THÈSE

présentée par

Karim MEGZARI

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

Spécialité : Informatique

REFINER : Environnement logiciel pour le raffinement d'architectures logicielles fondé sur une logique de réécriture

---

Soutenue publiquement le** 17 décembre 2004 **devant le jury composé de :

Richard MCCLATCHEY	Président	Professeur à l'Université de West of England, CCCS, Bristol, Royaume Uni
Nacer BOUDJLIDA	Rapporteur	Professeur à l'Université Henri Poincaré de Nancy
François JACQUENET	Rapporteur	Professeur à l'Université Jean Monnet de Saint-Etienne
Flavio OQUENDO	Directeur	Professeur à l'Université de Savoie
Ilham ALLOUI	Co-encadrant	Maître de Conférences à l'Université de Savoie

préparée au sein du LISTIC
Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance
ESIA – Université de Savoie

REMERCIEMENTS

Ce travail a été réalisé au sein du LISTIC (Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance) à l'Ecole Supérieure d'Ingénieurs d'Annecy (LISTIC/ESIA).

Je voudrais tout particulièrement remercier :

M. François JACQUENET et M. Nacer BOUDJLIDA pour avoir accepté d'être rapporteurs de cette thèse. Je leur suis reconnaissant pour leur lecture attentive ainsi que pour les critiques constructives qu'ils ont faites à l'égard de ce travail.

M. Richard MCCLATCHEY pour m'avoir fait l'honneur de présider le jury de cette thèse.

M. Flavio OQUENDO pour avoir dirigé mes travaux durant ces années de thèse et l'année du DEA ainsi que pour l'ensemble des conseils qu'il m'a prodigués et sa patience dans les « passages à vide ». Son expérience m'a été grandement profitable et les discussions que nous avons eues ont été d'une importance indéniable dans mon cursus.

Mme Ilham ALLOUI pour sa bienveillance tout au long de ces années. Son action a été prépondérante tant au démarrage qu'à l'aboutissement de ce travail. Je la remercie d'avoir accepté d'être mon co-encadrant, pour ses remarques, ses nombreux conseils, son aide dans la rédaction de cette thèse avec sa lecture et ses corrections minutieuses, sa disponibilité aux moments importants et le témoignage fréquemment renouvelé de sa confiance.

J'adresse mes remerciements aux membres du groupe « génie logiciel et méta-modélisation » pour l'ambiance agréable, le soutien, l'aide et la bonne humeur qui y régnaient. Un vif merci particulièrement à Valérie pour son soutien logistique et sa précieuse collaboration à tous les niveaux administratifs.

Je remercie M. Philippe Bolon, directeur du LISTIC , qui m'a accueilli avec chaleur et m'a offert les moyens nécessaires à la concrétisation de mon projet de thèse.

Je salue les membres des projets européens PIE et ArchWare et du projet régional e-Alliance pour les critiques apportées à ce travail et les échanges fructueux que nous avons eus.

Je remercie mon père qui, dès mon enfance, m'a donné une force magique et immense me conduisant à la réussite. Son soutien qui a commencé par mon encadrement en mathématiques et a fini par l'élaboration de ma thèse en passant par ses conseils, ses aides et financières et morales, je ne saurais trouver l'expression requise pour le remercier. Tout ce que j'ai à te dire cher Père, c'est que ton rêve est réalisé, je te dédie exclusivement mon doctorat du fond de mon cœur.

Une seule ligne ne saurait transcrire ma gratitude envers ma mère, ma sœur Nora et mon petit frère Si-mohammed qui m'ont offert leur amour et leur soutien pendant toutes ces années d'études.

Il est évident que je n'aurais pas pu terminer cette thèse sans l'appui indéfectible de mon épouse Btissam. Je l'en remercie, pour son amour sans faille, du fond de mon cœur.

Merci, Lamiae, Lina, Hamza, Rachid, Sidi-hafed, Elhadja, Siham, Si-mohammed d'exister et pour votre soutien sans faille.

Je tiens à remercier mon cousin Chakib et sa femme pour leur confiance et pour leur soutien précieux qui restera gravé dans ma mémoire. A celles et ceux, qui par leurs actions, leurs témoignages, leurs concessions, ont permis la réalisation de ce travail, qu'ils trouvent ici le témoignage de ma reconnaissance.

À mon père,

TABLE DES MATIERES

TABLE DES FIGURES

Chapitre 1 : Introduction générale

Chapitre 1 : Introduction générale

Nous introduisons brièvement dans ce chapitre le domaine des architectures logicielles ainsi que les approches dites « dirigées par les modèles » pour le développement des logiciels. Parmi ces approches, figure le développement de logiciel centré-architecture que nous avons choisi comme contexte et que nous présentons pour situer la problématique abordée dans cette thèse : le raffinement des architectures logicielles. Nous terminons ce chapitre par la présentation des objectifs que nous nous sommes fixés.

1. Introduction au domaine des architectures logicielles

Avec la révolution informatique, les logiciels devenus plus complexes et évoluant sans cesse, leur développement exige plus de labeur et un coût plus élevé. En effet, les utilisateurs, de plus en plus exigeants, attendent de leurs logiciels une grande sûreté d'utilisation, un nombre de services de plus en plus important, et le respect de certaines contraintes comme un faible coût ou la rapidité de livraison et de maintenance. Ceci explique la taille et la complexité croissantes des logiciels. Une conséquence directe est que la structure globale du système logiciel, c'est-à-dire, *l'architecture logicielle* est devenue un problème central de conception. Le rôle de l'architecte logiciel a alors été créé par plusieurs organismes pour assurer l'intégrité globale et les caractéristiques critiques des systèmes et des processus de leur développement.

Le rôle de l'architecte n'est pas seulement de créer une bonne architecture, mais également de créer une stratégie technique pour le développement et la maintenance du système. Une architecture logicielle est communément construite à partir de composants logiciels qui ont des capacités de calcul et qui ont la possibilité d'interagir souvent par le biais d'envoi et de réception de messages. Ces interactions peuvent être décrites par des protocoles.

En réalité, il n'existe pas de définition universelle de l'architecture logicielle. En effet, on peut trouver dans la littérature de nombreuses définitions qui, bien qu'elles présentent des similitudes, correspondent à autant de vues différentes du domaine. Parmi ces dernières, nous citons :

- ***Une architecture logicielle est l'organisation fondamentale d'un système, incarnée dans ses composants, leurs relations chacun avec les autres et avec l'environnement, et les principes guidant sa conception et son évolution.*** *[Standard IEEE, 2000]*

- ***L'architecture logicielle comprend la structure des composants d'un logiciel/système, les relations entre eux et les principes et directives régissant leur conception (design) et évolution au cours du temps.*** *[Software Engineering Institute, 1994]*

- ***L'architecture logicielle [est un niveau de conception qui] implique la description des éléments à partir desquels les systèmes sont construits, les interactions entre ces éléments, les patterns (patrons) qui guident leur composition et les contraintes sur ces patterns.*** *[M.Shaw et D.Garlan, 1996]*

L'architecture logicielle concerne plus particulièrement la conception de systèmes logiciels complexes et de grande ou encore de familles de produits logiciels. Elle décrit l'ensemble des composants (unités de calcul) qui la composent, donne la définition de leur protocole de communication, permet l'assemblage de ces composants et prend en compte les structures d'accueil nécessaires à leur déploiement et à l'exploitation du système résultant.

Dans le paragraphe suivant, nous positionnons les architectures logicielles dans le cadre des approches dirigées par les modèles dont MDA et le développement centré architecture sont deux représentants

2. Approches dirigées par les modèles

Les approches dirigées par les modèles ont pour objectif de fournir les méthodes et les outils fondés sur la transformation de modèles pour concevoir des applications indépendamment des plates-formes d'exécution. La récente initiative Model Driven Architecture (MDA) [Poo 01] de l'OMG et le développement de logiciels centré-architecture sont deux représentants de telles approches. En effet, le MDA est une architecture permettant un développement logiciel fondé sur des modèles centrés-objet (UML, MOF, etc.) et leurs transformations (raffinement, simulation, génération de code, etc.). Dans l'approche centrée-architecture nous nous intéressons à un type de transformation particulier : le raffinement des architectures logicielles en partant d'un niveau abstrait de détail jusqu'à arriver à un niveau concret proche de l'implémentation, ceci en nous basant en particulier sur les éléments architecturaux (composant, connecteur, etc.).

Dans le reste de ce document, nous nous focalisons sur l'approche centrée-architecture. Comme le montre la figure Fig. I.1., le cycle de vie classique d'un logiciel démarre par l'analyse des besoins et la définition du cahier des charges puis continue par la conception (celle-ci est subdivisée en conception architecturale et conception détaillée) ; ensuite viennent l'implémentation (codage) et le test. Les phases suivantes concernent l'évolution du logiciel à savoir son intégration, sa livraison, son exploitation, sa maintenance et finalement son retrait du marché.

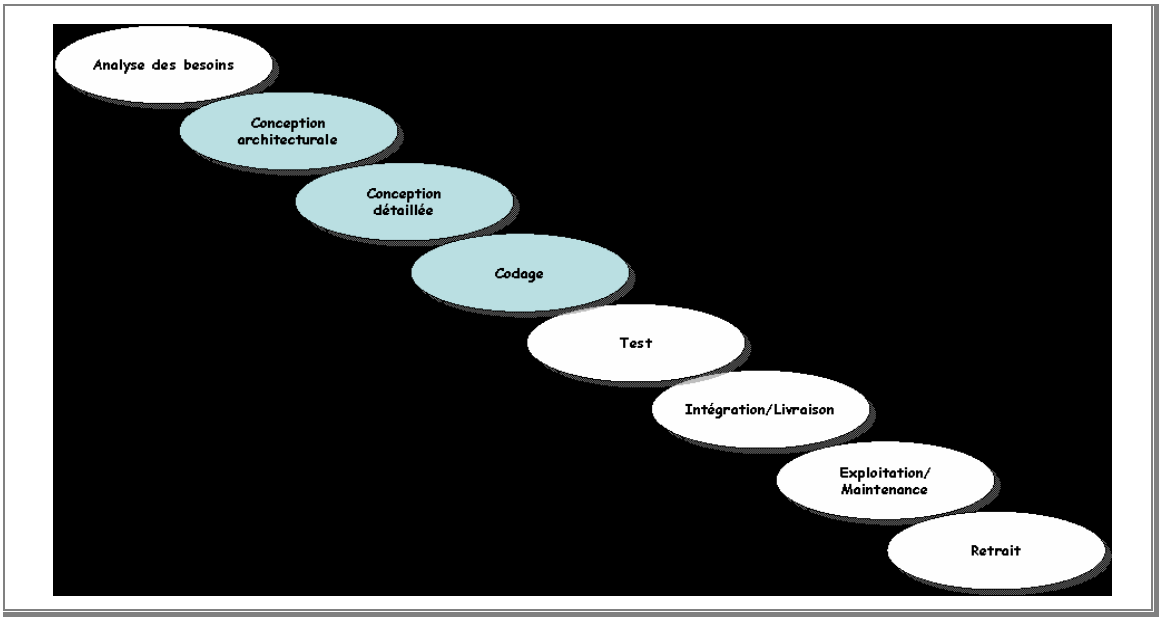


Figure I.1. : Cycle de vie d'une application logicielle

L'objectif du développement centré-architecture est de considérer l'architecture du logiciel au centre des différentes phases du cycle de vie. Contrairement au cycle de vie classique au cours duquel les architectures logicielles sont essentiellement créées et manipulées dans les phases de conception architecturale et détaillée ainsi que la phase de codage, le développement centré-architecture propose

qu'en plus le test et la maintenance portent non plus sur le logiciel directement mais sur son architecture. Pour ce faire, le processus de développement centré-architecture repose sur les rôles suivants : l'architecte, le développeur et l'analyste.

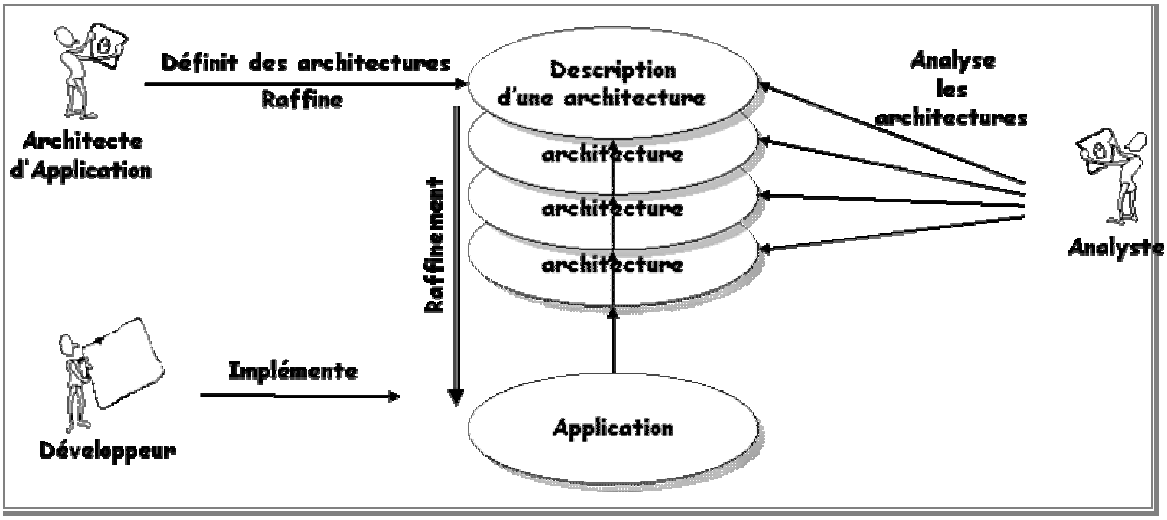


Figure I.2. : Processus de développement centré-architecture

L'architecte a pour charge de concevoir et de décrire l'architecture qui servira de base au développement de l'application logicielle. Ce rôle inclut le raffinement de l'architecture d'un niveau abstrait de détails vers un niveau plus concret. Le raffinement d'une architecture s'effectue jusqu'à ce que l'architecture soit suffisamment détaillée pour pouvoir être implémentée sans ajouter de nouvelles informations.

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

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

Dans cette approche, le raffinement de l'architecture logicielle occupe une place prépondérante. Il consiste à rendre plus déterministe une archietcture abstraite, et ce en plusieurs étapes. Dans la littérature, plusieurs approches sont proposées pour le raffinement où l'on distingue généralement le raffinement dit horizontal (souvent appelé décomposition) qui précise une spécification à un même niveau d'abstraction du raffinement dit vertical qui implique certaines parties de la spécification à différents niveaux d'abstraction. Le raffinement horizontal ajoute de nouvelles parties à la spécification tandis que le raffinement vertical est souvent décidé par des choix liés à la plateforme d'implémentation. Cependant toutes les approches ne proposent pas le même type de raffinement. En effet, dans certaines approches telles que Z [Spi 92] et la méthode B classique [Abr 03] [TrS 00] [Ste 98], le raffinement est fondé sur l'affaiblissement des préconditions de l'abstraction et le renforcement de ses postconditions, ce qui conduit à une description plus déterministe qui représente un comportement possible de l'abstraction. A l'opposé, d'autres approches comme VDM [Jon 90, 93] sont fondées sur le principe de réduction, c'est-à-dire, le renforcement des préconditions des abstractions et l'affaiblissement de leurs postconditions. Par conséquent, le raffinement ici est une opération de simplification.

Dans le cadre de nos travaux, les deux types de raffinement seront pris en compte. L'objectif est de fournir à l'architecte les mécanismes et outils pour à la fois pouvoir rendre déterministe une

specification et pouvoir la simplifier au besoin. C'est dans ce contexte que nous traitons dans cette thèse les problèmes liés au raffinement des architectures logicielles, en particulier ceux concernant le support informatique à fournir aux architectes et aux développeurs.

3. Problématique

Le raffinement des architectures logicielles repose sur le développement de descriptions de ces architectures en partant d'un niveau abstrait de détails vers des niveaux de plus en plus concrets pouvant aboutir à une implantation dans un langage de programmation cible. Le raffinement peut être très simple, par exemple : un architecte peut raffiner un type prédéfini pour décrire les types de données requis par l'architecture logicielle qu'il est entrain de construire. Le raffinement peut également être très complexe, par exemple : le raffinement d'une architecture monolithique (un seul composant) en une architecture plus complexe comprenant plusieurs composants avec tout ce que ceci implique comme changements structurels et comportementaux pour obtenir la nouvelle architecture. En plus d'un support pour effectuer ces transformations, ce dernier type de raffinement requiert la préservation dans la nouvelle architecture des propriétés (structurelles, comportementales et non fonctionnelles) de l'architecture initiale. L'on parle alors de raffinement correct d'architectures logicielles.

Concernant l'existant, plusieurs langages et méthodes ont été proposés pour le raffinement de logiciels en général cependant : (a) très peu d'entre eux prennent en compte les concepts architecturaux du système en termes de structure et de comportement. Soit les approches proposées ne sont pas dédiées aux architectures logicielles soit elles reposent sur le raffinement d'un seul aspect (soit structurel soit comportemental mais pas les deux); (b) ils ne sont pas suffisamment voire pas du tout supportés par des outils logiciels qui faciliteraient la tâche aux architectes de systèmes logiciels. Le raffinement se fait manuellement dans la plupart des cas.

Or la puissance des approches proposées ne devrait pas dépendre uniquement de la richesse du langage et/ou de la méthode utilisés, mais également du support fourni aux utilisateurs à travers les environnements qui permettent de les mettre en œuvre. Un environnement logiciel pour le raffinement des architectures logicielles devrait permettre à l'utilisateur de raffiner en plusieurs étapes une description abstraite d'architecture en descriptions de plus en plus concrètes pouvant mener jusqu'à la génération de l'application dans un langage de programmation cible. L'intérêt d'un tel outillage est celui de pouvoir automatiser des processus complets de raffinement et par-là même intégrer des outils pour la vérification de la correction du raffinement.

Par rapport à l'approche MDA qui offre un cadre général pour la génération de code à partir de transformations successives manuelles ou semi-automatisées de modèles, notre objectif est d'automatiser le processus de raffinement de logiciels en focalisant sur les aspects architecturaux.

4. Objectifs de la thèse

Les travaux menés jusqu'à présent sur le développement centré-architecture de logiciels portent essentiellement sur la spécification formelle des architectures. Le raffinement des architectures logicielles en général, et plus particulièrement, des outils logiciels pour le support du raffinement formel d'architectures restent un point ouvert de recherche.

Dans cette thèse, notre objectif est de fournir un support informatique au processus de raffinement d'architectures logicielles à travers un environnement logiciel, nommé Refiner, fondé sur la logique de réécriture. Le choix de cette dernière est motivé par le fait qu'elle soit bien adaptée à la description de systèmes concurrents et leur raffinement par le mécanisme de réécriture et également par le fait qu'elle soit bien supportée par des outils logiciels.

L'environnement Refiner que nous proposons permet un développement "centré-architecture" de systèmes logiciels. A chaque niveau de la hiérarchie de raffinement, Refiner supporte des actions de raffinement des modèles architecturaux existants, et au niveau concret, supporte en plus la génération de code. La sémantique de ces actions, définie dans la logique de réécriture permet d'assurer par construction un raffinement correct à différents niveaux dans la hiérarchie de raffinement.

Notre proposition a été développée et validée dans le cadre du projet européen *ArchWare*[Arc 02] et dans le cadre du projet thématique région *e-Alliance*[Eal 01].

5. Organisation du rapport

Le rapport de thèse est organisé en trois parties. La première contient les deux premiers chapitres introduisant la problématique et présente le cadre et l'état de l'art relatif à notre problématique. La deuxième partie porte sur notre proposition de solution et ses fondements : l'environnement logiciel de raffinement d'architectures logicielles *Refiner*. Elle englobe le troisième chapitre qui introduit le langage de raffinement utilisé *Archware ARL* et le quatrième chapitre qui présente notre formalisation de *ArchWare ARL* dans la logique de réécriture.
La troisième partie est consacrée à la validation de notre proposition. Elle contient le cinquième chapitre présentant l'implantation des outils de l'environnement *Refiner* et le sixième la validation du *Refiner* à travers d'une étude de cas effectuée dans le cadre du projet thématique région *e-Alliance*.
Enfin, la conclusion propose une synthèse et un bilan du travail effectué ainsi qu'un ensemble de perspectives liées à la poursuite du travail, à de nouvelles applications ainsi qu'à des thèmes potentiels de recherche.

Chapitre 2 : Etat de l’Art

Chapitre 2 : Etat de l'Art

Dans une première partie de ce chapitre nous introduisons les concepts de base liés à la description des architectures logicielles. Cette dernière repose sur l'utilisation de langages de description d'architectures (ADL) que nous illustrons à l'aide du langage appelé Wright [All 97][ADG 97][AG 97][AG 94]. Dans une deuxième partie, nous présentons une synthèse sur les ADL les plus généralement utilisés. Ensuite, dans une troisième partie nous présentons parmi ces différents travaux ceux qui traitent le raffinement des architectures logicielles à savoir SADL [MQR 95][MoR 97] et Rapide [RDT 97][LKA 95][LV95] [Tea 94]. Nous complétons ce chapitre par une brève description de travaux non liés aux architectures mais qui portent sur le raffinement et nous terminons par un bilan en rapport avec la problématique présentée dans le chapitre 1.

1. Langages de description d'architectures

1.1. Modèles de base

Les ADLs focalisent sur la structure de l'application globale à un haut niveau d'abstraction plutôt que sur les détails d'implémentation [Gar 00]. Pour cela ils fournissent une syntaxe concrète et un cadre conceptuel pour la modélisation d'une *architecture conceptuelle* du système logiciel. Celle-ci repose généralement sur les modèles de base suivants :

- ***les composants*** représentant des unités de calcul ou de stockage ;
- ***les connecteurs*** qui sont des composants particuliers employés pour modéliser les interactions entre les composants ainsi que les règles qui régissent ces interactions ;
- ***les configurations architecturales*** qui regroupent des composants et connecteurs en des graphes connexes décrivant la structure architecturale.

A titre d'exemple, l'ADL Wright fondé sur l'algèbre de processus CSP [Hoa 85] comporte les modèles de base suivants :

- le modèle de composant référencé comme *composant* dont les interfaces sont référencées comme *ports*. Leur sémantique est formellement spécifiée en CSP ;
- le modèle de connecteur référencé comme *connecteur*. L'interface d'un connecteur est nommée *rôle ;*
- les connections entre les ports d'un composant et les rôles d'un connecteur sont appellées *attachements* et sont explicitées dans la configuration de l'architecture nommée *configuration*.

Les modèles de base fournis par les ADL servent à spécifier des architectures logicielles telles que celle englobant deux filtres connectés par un tube dont la description abstraite en Wright est donnée ci-après :

System Example
  Component Filter
    port provide *[provide protocol]*
    port request *[request protocol]*
    spec *[Filter specification]*
  Connector Pipe
    role input *[input protocol]*
    role output *[output protocol]*
    glue *[glue protocol]*
  Instances

```
        f1, f2, f3 : Filter ;
        p1, p2 : Pipe Connector.
    Attachments
        f1.provide as p1.input ;
        f2.request as p1.output ;
        f2.provide as p2.input ;
        f3.request as p2.output.
end.
```

Dans cette spécification, la première partie décrit des types de composants et de connecteurs. Le type d'un composant est formé d'un ou plusieurs ports considérés comme autant de points d'interaction avec le système, et d'une spécification de son fonctionnement. Les ports constituent l'interface du composant, c'est-à-dire l'ensemble des fonctionnalités que l'on décide d'observer ; la spécification demandée pour chaque type de composant précise les relations entre ces fonctionnalités. Le type d'un connecteur est formé quant à lui de rôles décrivant le comportement local des composants en jeu (i.e. les événements des ports que l'on décide d'observer) et d'une colle (« glue ») décrivant la coordination des rôles. La principale distinction entre un composant et un connecteur dans Wright est qu'un connecteur ne possède pas de fonctionnalité propre, c'est-à-dire qu'un connecteur est une spécification d'un composant qui n'effectue pas de calcul. Les instances servent à la configuration du système qui est donnée par les *attachements* qui relient les connecteurs aux composants en associant les rôles aux ports. Le principe de l'intégrité de la communication est vérifié par ce langage puisque les seules connections autorisées entre les composants sont celles définies par leurs interfaces.

En faisant l'analogie avec le domaine de l'électronique, un connecteur peut être comparé à un câble reliant des composants. Dans ce cas les rôles correspondent aux prises mâles avec un certain format, la colle décrit la manière dont l'information sera gérée par ce câble et les ports correspondent aux prises femelles. Un point important à préciser dans cette analogie est qu'il existe plusieurs types de prises femelles pouvant être branchées sur ce câble. En effet les prises femelles compatibles doivent avoir au minimum des orifices complémentaires des plots proposés par le connecteur, rien n'empêche qu'un composant ait des prises femelles plus complexes ou même plusieurs prises femelles qui ne sont pas toutes utilisées. Cette possibilité permet aux composants de communiquer avec plusieurs composants à la fois en reliant leurs prises par différents connecteurs.

Les spécifications données dans les composants (*Filter specification)* et dans la glue (*glue protocol*) sont écrites en CSP, dont elles n'utilisent que le sous-ensemble suivant de constructions :

- *événements* : e. Ces événements montrent les aspects de la modélisation auxquels on s'intéresse, ils peuvent transmettre des données ($e!x$) ou en recevoir ($e?y$).
- *préfixe* : $e \rightarrow P$. Cela signifie que le processus Q attend l'action e puis se comporte comme le processus P. On note § le processus d'arrêt avec succès.
- *choix externe* (déterministe) : $P \,[]\, Q$. Le choix entre P et Q est effectué par l'environnement de ce processus.
- *choix interne* (non-déterministe) : $P \prod Q$. Le choix est fait par le processus lui-même.
- *composition parallèle* : $P \,||\, Q$. Cet opérateur synchronise les actions par noms, celles qui ne sont pas communes peuvent avoir lieu indépendamment.

Pour des fins d'illustration, un exemple plus complet portant sur la description en Wright d'un proxy de compression fondé sur une architecture « tube-filtre » (« pipe-filter ») est donné dans le sous-paragraphe suivant.

1.2. Exemple du Proxy de compression

Afin d'améliorer la performance des transactions des navigateurs du Web des réseaux lents, fondés sur Unix, une des possibilités est de créer un serveur HTTP qui compresse les données envoyées sur le réseau et les décompresse à la lecture. Dans ce contexte, le but du proxy de compression est d'ajouter le programme de compression/décompression Gzip au standard HTTP. Ce dernier comporte différentes fonctionnalités pouvant être rangées dans des filtres qui interagissent par le biais d'appels de fonctions. En effet, la transformation des données par le standard HTTP se fait par l'application de fonctions sur ce flot de données, qui passe successivement d'un filtre à un autre. L'on peut considérer dans ce cas que les filtres procèdent par lecture/écriture des données à travers ce flot. On dira par exemple que le filtre F2 lit des données dès que le filtre F1 appelle une fonction présente dans l'interface de F2. Le filtre F1 a aussi la possibilité de fermer le flot. Puisque le moyen de communication est basé sur l'appel de fonctions, tous les filtres doivent être contenus dans un même processus Unix.

Le programme Gzip peut également être considéré comme un filtre, mais au niveau d'un processus Unix ; il utilise donc l'interface standard entrée/sortie d'Unix (pipe). Une différence importante entre Gzip et les filtres du serveur HTTP est que Gzip choisit quand il veut lire tandis que les filtres du serveur HTTP sont obligés de lire lorsqu'un filtre de niveau supérieur leur envoie des données. Pour pouvoir assembler le proxy de compression à partir des filtres du serveur HTTP et du programme Gzip, il est nécessaire de créer un adaptateur (cf. Fi. II.1.) qui sert à coordonner les interfaces. Cet adaptateur agira comme un pseudo-filtre qui communiquera avec les filtres par un appel de fonction et avec Gzip par des tubes.

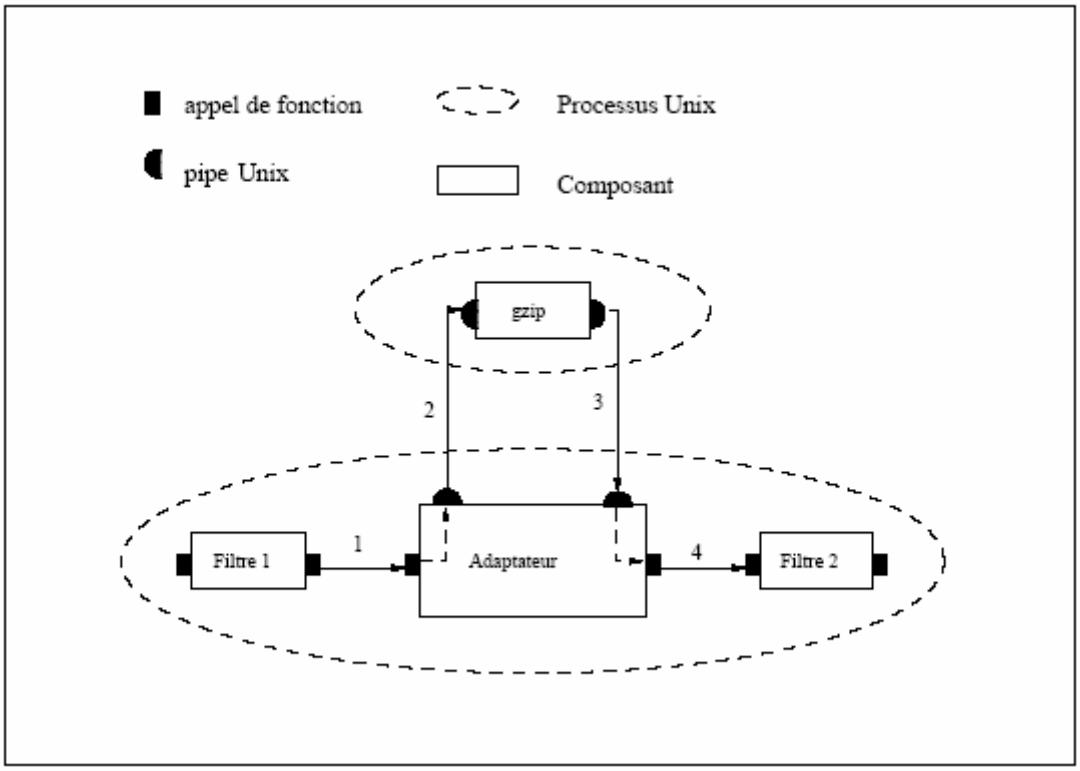


Figure II.1. : Proxy de compression

En considérant la description du problème donnée précédemment, l'on peut dégager quelques idées pour la construction de la spécification. L'on peut partir de deux types de composants : un pour décrire les filtres, un autre pour décrire Gzip. Les interactions entre les composants se passent de deux façons : par

lecture/écriture entre les filtres et l'adaptateur et par des tubes (pipes) entre Gzip et l'adaptateur. L'on peut synthétiser ces deux formes de communications dans un unique type de connecteur en considérant que l'adaptateur gère ces interactions. Dans un esprit de simplification, le comportement des interactions entre Gzip et l'adaptateur sous forme de tube ne sera pas précisé. Il sera considéré comme une simple entrée/sortie sans tampon. Nous détaillons dans ce qui suit la spécification des différents éléments architecturaux du proxy de compression.

- **Les filtres**

Nous pouvons définir deux ports pour un filtre : un port de *lecture* composé d'un événement (*lit*) associé à l'appel d'une fonction du filtre et de la réponse à un événement (*fermeture*) émis par le filtre précédent ainsi qu'un port d'*écriture* composé d'un événement (*écrit*) associé à l'appel d'une fonction du filtre suivant et d'un autre événement signifiant la fin de la transmission (*fin*). La spécification du filtre est simplement esquissée, car pour mettre en évidence le problème architectural, une spécification précise apporterait trop de détails. On ne précise que l'événement *calcul* représentant la fonctionnalité du filtre.
La différence entre le choix déterministe ([]) et le choix non-déterministe ($\prod$) dans la description du filtre reflète le fait que les filtres sont obligés de lire dès qu'on leur soumet des données (choix []) ; En revanche, ils ont la possibilité de choisir quand ils veulent écrire (choix $\prod$).

Component Filtre =
  port Lecture = lit ?x → Lecture [] fermeture → §
  port Écriture = écrit!y → Écriture $\prod$ fin → §
  spec Filtre = lit ?x → calcul → écrit ?y → Filtre [] lit ?x → fermeture → calcul →
        écrit !y → fin → §
end.

- **Gzip**

Pour le composant Gzip, nous pouvons également définir deux ports : un pour l'entrée et un pour la sortie des données du programme. Contrairement aux filtres du protocole HTTP, **Gzip** choisit indépendamment quand il veut lire ou écrire : le choix est non-déterministe dans les deux cas. En plus de la transmission de données entre l'adaptateur et Gzip, les événements *end*$_i$ et *end*$_o$ servent à synchroniser la fin des transmissions avec l'adaptateur. On modélise la possibilité qu'a Gzip d'interrompre la transmission des données de façon autonome par des actions particulières (*fin*$_i$ et *fin*$_o$). Dans la spécification ci-après du composant Gzip on ne s'intéresse pas à l'algorithme de compression mais uniquement à la manière dont se déroulent la lecture et l'écriture des données.

Component Gzip =
  port Entrée = (in!x → Entrée [] end$_i$ → §) $\prod$ fin$_i$ → §
  port Sortie = (out ?y → Sortie $\prod$ end$_o$ → §) $\prod$ fin$_o$ → §
  spec Gzip =
    let P = (Sortie.out?y → P $\prod$ end$_o$ → Gzip) $\prod$ fin$_o$ → Gzip
    in (Entrée.in ?x → Gzip []end$_i$ → P) $\prod$ Entrée. fin$_i$ → P)
end.

- **L'adaptateur**

L'adaptateur joue le rôle du connecteur dans cette architecture. En effet, son comportement est totalement non fonctionnel, son utilité est seulement de présenter une prise dans le protocole HTTP sur laquelle viendra se greffer Gzip. C'est par lui que transitent toutes les données et que se fait le lien entre les deux processus Unix impliqués dans la conception du système. L'on peut définir pour ce connecteur quatre rôles (Un, Deux, Trois et Quatre). La glue contient le protocole de communication créé pour le proxy de

compression. L'événement *prêt* signale la transition de l'adaptateur entre l'écriture des données dans Gzip et la lecture des données provenant de Gzip.

Les connecteurs dans le langage Wright servent en plus à identifier le nom des actions des composants qui seront mises en communication, c'est pourquoi on retrouve dans l'écriture de ce composant toutes les actions utilisées dans les spécifications précédentes.

Connecteur Adaptateur =
  role Un = écrit ?y → Lecture [] fin → §
  role Deux = in !x → Sortie [] **endi** → §
  role Trois = out ?y → Sortie []**endo** → §
  role Quatre = lit !x → Écriture ⨅ fermeture → §
  glue =
    let Q = Trois.out ?y →Q [] **endo** → Quatre.lit!x → glue
    in let P = Deux.in!x ! P ⨅ **endi** → prépare → Q
    in Un.écrit ?y → P
 spec …
end.

- **Le proxy de compression**

Une fois les types de composants et les types de connecteurs présentés, la spécification du proxy de compression s'écrit :

System Compressing Proxy
  Composant Filtre [spécification du type des filtres]
  Composant Gzip [spécification du type de Gzip]
  Connector Adaptateur [spécification du type de l'adaptateur]

Instances
f1, f2 : Filtre ;
A : Adaptateur ;
G : Gzip.
Attachments
  f1.Écriture **as** A.Lecture ;
  A.Sortie **as** G.Entrée ;
  G.Sortie **as** A.Entrée ;
  A.Écriture **as** f2.Lecture.
end.

Les connections entre les composants se font par une identification du nom des actions entre les composants et le connecteur. Cette identification a lieu dans le connecteur.

Maintenant que nous avons introduit les architectures logicielles au travers du langage Wright, nous présentons dans le paragraphe suivant une comparaison de différents langages de descriprtion d'architectures selon des critères définis dans [Gar 00] [MeT 97] [Cle 96]. A ces critères qui portent sur les caractéristiques architecturales des éléments architecturaux (i.e., composant, connecteur et configuration), nous avons rajouté les aspects liés au raffinement des architectures logicielles, l'objet de notre problématique.

2. Synthèse sur les ADL

Nombreux langages de description d'architectures (ADL) ont été proposés dans la littérature, parmi les plus connus : ACME [GMW 97], Aesop [GAO 94], C2-SADEL [Med 96][MTW 96][MRT 99], Darwin[EP 93][MDE 95][MK 96], MetaH [BiV 93] [BEJ 96], UniCon [Zel 96], PADL [BCD 00] et [CPT 97], Rapide, SADL, et Wright. Comme l'objectif d'un ADL est de fournir les moyens pour une spécification explicite, plusieurs caractéristiques pour les composants et de connecteurs sont souhaitables. Parmi ces caractéristiques, nous nous sommes intéressés aux : *interfaces, types, sémantique, raffinement, contraintes, et propriétés non fonctionnelles*. Concernant les caractéristiques souhaitables des configurations architecturales, nous citerons : *composition, raffinement, dynamisme* et *contraintes*. Dans ce qui suit, chacune de ces caractéristiques est introduite avec le positionnement des ADL sus-cités.

Les interfaces de composants/connecteurs *:* l'interface d'un composant est la description de l'ensemble des services offerts et requis par le composant, ceci sous la forme de signature de méthodes, de types d'objets envoyés, d'exceptions et de contexte d'exécution. L'interface d'un composant est un moyen d'expression de ses liens ainsi que ses contraintes avec l'extérieur. L'interface d'un connecteur définit les points d'interaction entre connecteurs et composants. Elle ne décrit pas des services fonctionnels comme ceux du composant mais des mécanismes de connection entre composants.

Tous les ADL cités jusqu'ici supportent les interfaces de composants même si elles diffèrent dans la terminologie et le type d'informations qu'elles spécifient. Par exemple chaque point d'interface de ACME, Aesop, SADL et Wright est un *port*. Dans ces ADL, les ports sont typés et nommés. Dans Wright, la sémantique particulière d'un port est spécifiée en CSP comme protocole d'interaction.

Seuls les ADL qui supportent explicitement la modélisation des connecteurs, indépendamment des configurations dans lesquelles ils sont utilisés, supportent la spécification des interfaces des connecteurs. Dans ACME, Aesop, C2, SADL, UniCon et Wright les connecteurs sont définis explicitement. Pour ACME, Aesop, UniCon et Wright, les points d'interfaces des connecteurs sont nommés *rôles*. La connection explicite des ports de composants avec les rôles de connecteurs est exigée dans une configuration architecturale. Dans Rapide et MetaH, les connecteurs sont des connections modélisées en ligne, et ne peuvant pas être nommés ni sous-typés ni réutilisés. Ce ne sont pas des entités de première classe. Les connecteurs dans Darwin sont des liaisons également spécifiées en ligne, dans le contexte de configuration seulement.

Le type d'un composant/connecteur **:** le type d'un composant est un concept représentant l'implantation des fonctionnalités fournies par le composant. En fournissant un moyen pour décrire, de manière explicite, les propriétés communes à un ensemble d'instances d'un même composant, la notion d'un type de composant introduit un classificateur qui favorise la compréhension d'une architecture et sa conception. Il permet la réutilisation d'instances de même fonctionnalité soit dans une même architecture, soit dans des architectures différentes. Le type d'un connecteur définit les points d'interaction entre connecteurs et composants.

Tous les ADL mentionnés distinguent les types de composants et les instances. De plus, à l'exception de MetaH et UniCon, tous les autres ADL fournissent des systèmes de types de composants extensibles.

La sémantique d'un composant/connecteur : la sémantique d'un composant est exprimée en partie par son interface cependant celle-ci telle qu'elle est décrite ci-dessus ne permet pas de préciser complètement le comportement du composant. La sémantique doit être enrichie par un modèle plus complet et plus abstrait permettant de spécifier les aspects dynamiques ainsi que les contraintes liées à l'architecture. Ce modèle doit garantir une projection cohérente de la spécification abstraite de l'architecture vers la description de son implémentation et vers chacune des descriptions intermédiaires au cours des différentes étapes du raffinement de l'architecture. De même que 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 des différents passages du niveau de description abstraite vers le niveau de l'implantation.

Un ADL englobe généralement une théorie formelle qui permet de caractériser des architectures. En effet cette sémantique influence l'adéquation de l'ADL pour la modélisation de genres particuliers de systèmes (par exemple les systèmes concurrents) ou des aspects particuliers d'un système donné. Comme exemples de théories de spécifications formelles nous citerons : les algèbres de processus (CSP, CCS), les algèbres de processus "dynamiques" (π-Calcul et PA, POSET), les logiques (logique des prédicats, logique de réécriture) et les types de données algébriques (OBJ…). Wright est basé sur CSP pour modéliser les ports et les connecteurs par des protocoles d'interaction, et Darwin utilise le π-Calcul pour la définition formelle de l'ADL de Darwin (le π-Calcul n'est pas une partie du langage mais il est utilisé pour définir sa sémantique). Rapide est basé sur la théorie des ensembles d'événements partiellement ordonnés (POSET) qui fournit une sémantique opérationnelle.

Les contraintes d'un composant/connecteur : les contraintes définissent les limites d'utilisation d'un composant et ses dépendances intra-composants. Une contrainte est une propriété devant être obligatoirement vérifiée sur un système ou sur une de ses parties. Si elle est violée, le système est considéré comme incohérent. Les contraintes permettent ainsi de décrire de manières explicite les dépendances des parties internes d'un composant comme par exemple la spécification de la synchronisation entre composants d'une même application. Concernant les connecteurs, les contraintes permettent de définir les limites d'utilisation du protocole de communication associé ; par exemple, le nombre maximum de composants interconnectés à travers un connecteur peut être fixé et correspond alors à une contrainte.

Raffinement d'un composant/connecteur : Certains ADL comme Wright, ACME et MetaH modélisent les composants et les connecteurs à un niveau élevé d'abstraction et ne supposent pas ou ne prescrivent pas de relation particulière entre une description architecturale et une implémentation. Ils ne fournissent aucun cadre de raffinement. Dans [Gar 00], ces langages sont qualifiés d'indépendants de l'implémentation ("*implementation independent ADL*"). D'autre part, plusieurs ADL comme UniCon et MetaH exigent un degré beaucoup plus élevé de fidélité entre une architecture et son implémentation. Des composants modélisés par ces langages sont liés directement à leurs implémentations. Ces langages sont qualifiés de langages contraints par l'implémentation (*"implementation constraining ADL*"). Ils ne fournissent pas non plus de cadre de raffinement. D'autres approches comme C2-SADEL permettent un raffinement mais en une seule étape mettant en correspondance des modèles de composants en tant que paramètres formels vers des composants implantés en tant que paramètres effectifs. Contrairement aux approches précédentes, SADL et Rapide fournissent des dispositifs pour le raffinement des composants (interfaces) et connecteurs à travers plusieurs niveaux d'abstraction. Ces mécanismes peuvent être employés pour faire évoluer des composants en reportant des décisions de conception.

***Les propriétés non fonctionnelles d'un composant/connecteur* :** les propriétés non fonctionnelles (liées par exemple à la sûreté, la performance et la portabilité) doivent être exprimées à part, permettant ainsi une séparation dans la spécification du composant des aspects fonctionnels (aspects métiers de l'application) et des aspects non fonctionnels ou techniques (aspects transactionnels, de cryptographie, de qualité de service, etc.). Cette séparation permet la simulation du comportement d'un composant à l'exécution dès la phase de conception, et de la vérification de la validité de l'architecture logicielle par rapport à l'architecture matérielle et l'environnement d'exécution.

Les propriétés non fonctionnelles d'un connecteur concernent tout ce qui ne découle pas directement de la sémantique du connecteur. Elles spécifient les besoins qui viennent s'ajouter à ceux déjà existants et qui favorisent une implantation correcte du connecteur. La spécification de ces propriétés est importante puisqu'elle permet de simuler le comportement à l'exécution, l'analyse, la définition des contraintes et la sélection des connecteurs. Rapide par exemple fournit des constructions pour définir des prototypes exécutables d'architectures et Darwin pour la production de descriptions d'interconnections de modules pour les composants de logiciels existants. Cependant, dans les deux cas, il n'est pas possible de vérifier des propriétés comme la performance, l'absence d'interblocage, la sécurité, etc.

La composition : la définition de la configuration d'une application doit permettre la modélisation et la représentation de la composition à différents nivaux de détail. La notion de configuration spécifie une application par composition hiérarchique. Ainsi un composant peut être composé d 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 partition, allant du niveau le plus général formé par les composants et les connecteurs, jusqu'au détail de chaque composant et de chaque connecteur primitif. La composition facilite en plus la réutilisation car on peut réutiliser des composants existants pour en fournir d'autre.

Plusieurs ADL fournissent les dispositifs explicites pour supporter la composition hiérarchique : les "templates" de ACME, représentations dans Aesop, composants composites dans Darwin et UniCon, architecture de composant interne dans C2-SADEL, Macros de MetaH, et correspondances dans Rapide. D'autres ADL tels que SADL et Wright permettent en principe la composition hiérarchique mais ne fournissent aucune construction détaillée pour la supporter.

***Le raffinement d'une configuration* :** les ADL ont besoin de supporter la conception et le développement incrémental permettant l'ajout de détails à une architecture à travers entre autres le rajout, la suppression et le remplacement de composants, de connecteurs ainsi que des topologies de configurations.
Parmi les ADL étudiés, seuls Rapide et SADL que nous détaillerons par la suite fournissent des dispositifs supportant le raffinement de configurations à travers différents niveaux d'abstraction.

Dynamisme d'une configuration : le dynamisme permet de modifier l'architecture au cours de l'exécution du système. Il est important surtout pour les systèmes critiques tels que le trafic aérien, les systèmes d'information publics. En effet, l'arrêt de tels systèmes pour des mises à jour peut conduire à des retards inacceptables et par conséquent à des situations très coûteuses. Pour supporter et prendre en compte les modifications de l'architecture au moment de l'exécution du système, quelques ADL fournissent des dispositifs spécifiques pour modéliser les changements ainsi que des techniques dynamiques pour effectuer ces modifications.

La majorité des configurations existantes des ADL est statique. Les exceptions sont C2-SADEL, Darwin et Rapide. Cependant Darwin et Rapide supportent uniquement la manipulation dynamique des contraintes d'architecture où tous les changements d'exécution doivent être connus à priori. Darwin permet la réplication d'exécution des composants via l'instanciation dynamique et la configuration conditionnelle. Il fournit seulement le support pour la communication unidirectionnelle avec un

composant dynamiquement créé : il permet au composant de demander des services d'autres composants. Rapide quant à lui supporte la configuration conditionnelle et la génération dynamique des événements. Les règles de connection de Rapide emploient des configurations de POSET pour produire de nouveaux ensembles d'événements dynamiques. C2-SADEL, par sa notation de construction d'architecture, spécifie un ensemble d'opérations pour l'insertion, le déplacement d'un composant dans une architecture en cours d'exécution.

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

Ceci termine notre synthèse sur les ADL. Avant de nous attarder sur les ADL qui fournissent un cadre pour le raffinement des architectures logicielles, nous détaillons dans ce qui suit les directions ainsi que les différents types de raffinement.

3. Le raffinement des architectures logicielles

Nous rappelons qu'une architecture logicielle est la description abstraite d'un système logiciel. Le raffinement d'une architecture logicielle permet d'obtenir une description plus « concrète » du système, c'est-à-dire une architecture de plus bas niveau pouvant être proche d'une implémentation. Le raffinement d'architectures logicielles constitue une base de développement des systèmes logiciels corrects. Pour cela, le raffinement d'une architecture logicielle doit garantir que l'architecture logicielle « concrète » est valide au regard de l'architecture logicielle initiale. Ce raffinement conduit à définir de nouveaux composants, des nouvelles inter-connections, etc. Ceci se fait au cours du processus de développement centré-architecture où sont opérées des transformations en plusieurs étapes successives :

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

En effet, les systèmes complexes ne peuvent être architecturés dans une étape unique. Ils doivent donc être raffinés progressivement depuis la construction d'un modèle abstrait jusqu'à l'obtention d'un degré de détail souhaité. Nous pouvons alors distinguer deux directions pour une transformation : « verticale » et « horizontale ». Le raffinement horizontal souvent appelé décomposition n'engendre pas de changement de niveau d'abstraction mais qui ajoute de nouvelles parties à la spécification. Le raffinement vertical quant à lui concerne certaines parties de la spécification et est décidé par des choix liés à la plateforme d'implémentation. Il change le niveau d'abstraction sans pour autant remettre en cause la spécification initiale. Les deux directions du raffinement architecturales sont détaillées dans la suite de cette section.

3.1. Directions du raffinement : horizontal et vertical

Une architecture concrète d'un système logiciel de grande taille est souvent développée à travers une hiérarchie "verticale" d'architectures relatives. Cette dernière est une séquence linéaire de deux ou plusieurs architectures qui diffèrent par certains aspects, en partant d'une même architecture de départ. Prenant à titre d'exemple, une architecture abstraite contenant des composants fonctionnels reliés par des connections de flots de données. A un niveau concret, cette architecture peut être implantée dans une architecture en termes de procédures, de connections de contrôle et de variables partagées. Les étapes de raffinement vertical rajoutent de plus en plus de détails aux modèles abstraits jusqu'à l'obtention d'une

description concrète du modèle architectural. Un raffinement vertical typiquement augmente le déterminisme tout en impliquant des propriétés du modèle abstrait. Un raffinement "horizontal" quant à lui consiste à appliquer différentes actions de raffinement sur différentes parties de la même architecture abstraite en conservant le même niveau d'abstraction. Un exemple de ce type de raffinement est l'explosion d'un composant abstrait en plusieurs sous-composants dans le même niveau d'abstraction.

Formellement, nous pouvons exprimer le raffinement comme une relation notée ► reliant deux différents modèles, le premier est le modèle abstrait, le second est le modèle raffiné :

ModèleAbstrait ► *ModèleRaffiné*

La relation de raffinement est par construction réflexive et également transitive :

ModèleAbstrait ► $ModèleRaffiné_1$ ► $ModèleRaffiné_2$ ► ... ► *ModèleConcret*

En effet, un modèle concret est un modèle abstrait raffiné en plusieurs étapes :

ModèleAbstrait ► *ModèleConcret*

Il peut être considéré comme un autre modèle architectural adéquat à l'implémentation.

La relation de raffinement peut être définie selon deux points de vue :

- l'un *externe* où le raffinement relie seulement les comportements observables et les ports des éléments architecturaux (par des connections libres),
- l'autre *interne* où le raffinement relie les comportements sans restriction à l'observabilité c'est-à-dire que les connections restreintes sont également observables.

Les formes de raffinement définissent la manière de raffiner (comment ?) une architecture logicielle. Que le raffinement soit effectué d'un point de vue interne ou externe, ces formes sont au nombre de quatre comme détaillé ci-après.

3.2. Formes de raffinement des éléments architecturaux

Une relation de raffinement d'un point de vue externe ou interne s'établit sous l'une des quatre formes suivantes :

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

3.2.1. Raffinement d'un comportement

Un modèle architectural est le raffinement du comportement («behaviour refinement ») d'un autre modèle donné si toutes les propriétés de ce dernier sont impliquées par les propriétés de comportement du premier. Le raffinement du comportement relie les modèles architecturaux de même ensemble de ports.

Le modèle raffiné (plus concret) impose un comportement plus contraignant et des attributs de qualité plus contraignants ou supplémentaires à ceux imposés par le modèle donné (plus abstrait).

Pour vérifier qu'une étape de développement est un raffinement de comportement il est suffisant de vérifier que l'expression du comportement « behaviour » du modèle raffiné implique celui du modèle abstrait donné, c'est-à-dire, que tous les comportements possibles du modèle raffiné sont également des comportements possibles pour le modèle abstrait bien que le modèle raffiné soit plus déterministe que le modèle abstrait. Dans le cas du raffinement externe du comportement, l'implication se limite aux comportements observables alors que dans le cas du raffinement interne du comportement, l'implication concerne tous les comportements (observables et restreints).

Pour exprimer ces aspects, la relation de raffinement est spécialisée comme suit avec**[b]** dénotant le raffinement externe du comportement et **]b[** son raffinement interne :

ModèleAbstrait **[b]►** *ModèleRaffiné*
ModèleAbstrait **]b[►** *ModèleRaffiné*

3.2.2. Raffinement de l'interface

Un modèle architectural est un « raffinement d'interface » d'un autre modèle donné si c'est un raffinement de comportement modulo le « mapping » des ports entre les deux modèles. Le raffinement d'interface relie les modèles architecturaux de différents ensembles de ports car il peut relier différents ensembles de connections libres. Rappelons que, le modèle raffiné (plus concret) peut imposer un comportement plus contraignant et des attributs de qualités plus contraignants ou supplémentaires à ceux imposés par le modèle donné (plus abstrait).

Pour vérifier qu'une étape de développement est un raffinement d'interface il est suffisant de vérifier que l'expression du comportement du modèle raffiné implique celui du modèle abstrait donné modulo le « mapping » des ports c'est-à-dire tous les comportements possibles du modèle raffiné modulo le « mapping » des ports sont également des comportements possibles pour le modèle abstrait modulo « mapping » des ports bien que le modèle raffiné soit plus déterministe que le modèle abstrait.

Pour exprimer ces aspects, la relation de raffinement est spécialisée comme suit avec **[p]** dénotant le raffinement de port qui est toujours un raffinement externe de l'architecture.

ModèleAbstrait **[p]►** *ModèleRaffiné*

3.2.3. Raffinement d'une structure

Un modèle architectural est un « raffinement de structure » d'un autre modèle donné si c'est un raffinement de comportement modulo le « mapping » des éléments architecturaux des deux modèles. Le raffinement de structure relie les modèles architecturaux de même ensemble de ports. Pour vérifier qu'une étape de développement est un raffinement de structure il est suffisant de vérifier que l'expression du comportement « behaviour » du modèle raffiné implique celui du modèle abstrait donné modulo le « mapping » des éléments architecturaux, c'est-à-dire, tous les comportements possibles du modèle raffiné modulo le « mapping » de structure sont également des comportements possibles pour le modèle abstrait modulo le « mapping » de structure. Comme dans les cas précédents, le modèle raffiné est plus déterministe que le modèle abstrait.

Pour exprimer ces aspects, la relation de raffinement est spécialisée comme suit avec **]s[** dénotant le raffinement de structure qui est contrairement au port toujours un raffinement interne :

ModèleAbstrait **]s[►** *ModèleRaffiné*

3.2.4. Raffinement de données

Un modèle architectural est un « raffinement de données » d'un autre modèle donné si c'est un raffinement de comportement modulo le « mapping » des données des deux modèles. Le raffinement de données relie les modèles architecturaux de même ensemble de ports. Le modèle raffiné (plus concret) impose un comportement contraint et des attributs de qualités plus contraignants ou supplémentaires à ceux imposés par le modèle donné (plus abstrait).

Pour qu'une étape de développement soit un raffinement de données il est suffisant de vérifier que l'expression du comportement du modèle raffiné implique celui du modèle abstrait donné modulo le « mapping » des données. C'est-à-dire, tous les comportements possibles du modèle raffiné modulo le « mapping » données sont également des comportements possibles pour le modèle abstrait modulo le « mapping » données. Le modèle raffiné est plus deterministe que le modèle abstrait.

Pour exprimer ces aspects, la relation de raffinement est spécialisée comme suit, avec **[d]** le raffinement externe de données et **]d[** le raffinement interne :

ModèleAbstrait **[d]►** *ModèleRaffiné*
ModèleAbstrait **]d[►** *ModèleRaffiné*

L'ensemble de ces formes de raffinement établit la base du raffinement architectural. En effet, on peut appliquer ces différentes formes de raffinement aussi bien aux composants et aux connecteurs qu'aux architectures. Ces formes de raffinement seront utilisées comme critère de comparaison des différents ADL qui proposent un cadre de raffinement pour les architectures logicielles. C'est l'objet du reste du chapitre.

3.3. Les ADL et le raffinement

Nous avons introduit précédemment un certain nombre d'ADL à travers des critères de comparaison portant sur les propriétés architecturales des éléments architecturaux fournis. Nous avons ensuite formalisé la relation de raffinement architectural en distinguant quatre formes possibles. Dans ce sous-paragraphe, nous nous intéressons à quatre ADL qui proposent plus ou moins un cadre de raffinement.

C'est le cas de [EnV 04] où est proposée une méthodologie pour la transformation d'une architecture abstraite en une architecture plus concrète tout en préservant les attributs non-fonctionnels. En effet, le processus de raffinement est guidé à la fois par les attributs non-fonctionnels (sûreté, performance, etc.) et la spécificité de l'infrastructure d'implémentation. Le mécanisme utilisé est celui des annotations rattachées aux éléments architecturaux. En fait, aucun outil ni formalisation du mécanisme de transformation ne sont fournis. PADL [BCD 00] propose également une notion de raffinement mais celui-ci consiste seulement à remplacer des composants existants par d'autres sans pour cela changer la structure globale de l'architecture. Il impose cependant que le nouveau composant remplaçant soit conforme à la spécification abstraite du composant remplacé. En effet, tout comme Wright, aucune primitive de base n'est proposée pour le raffinement des autres éléments architecturaux (connecteurs, configuration). En réalité, à notre connaissance, en dehors de RAPIDE et de SADL, aucun ADL permettant le raffinement en termes architecturaux n'a été proposé jusqu'à maintenant. Ces deux langages sont présentés de manière détaillée dans ce qui suit et afin de pouvoir les comparer, nous utiliserons deux exemples relativement simples : l'architecture X/Open et l'architecture d'un compilateur. Un tableau récapitulatif regroupera les résultats de la comparaison en fin de chapitre.

3.3.1. Rapide

Rapide est un langage de description d'architecture fondé sur les événements concurrents. Il est spécifiquement conçu pour la vérification, la simulation et la validation d'architectures logicielles distribuées.

3.3.1.1 Concepts de base

Les concepts de base du langage Rapide sont les suivants : l'*événement*, le modèle *composant* et le modèle *architecture*. Le modèle connecteur n'est pas référencé par Rapide par contre leur abstraction est permise à travers des comportements de connections complexes dans des composants connecteurs.

- L'*événement* est une information transmise, c'est-à-dire une demande de service. Il permet de construire des expressions appelées *patrons d'évnements* (*event patterns)* qui caractérisent les événements circulant entre composants. La construction de ces expressions se fait en utilisant des opérateurs qui définissent les dépendances entre événements. Parmi ces opérateurs on trouve l'opérateur de dépendance causal ($A \rightarrow B$), l'opérateur de dépendance ($A \parallel B$), l'opérateur de différence ($A \sim B$) et l'opérateur de simultanéité (*A and B*). Ainsi, l'événement correspond à une information permettant de spécifier le comportement d'une application.
- Le modèle *composant* ou module est défini par une interface. Cette dernière est constituée d'un ensemble de services fournis et un ensemble de services requis. Les services sont de trois types : (a) les services *Provides* fournis par le composant appelé de manière synchrone par d'autres composants, (b) les services *Requires* demandés par le composant appelés de manière synchrone (c) les *Actions* qui correspondent à des appels asynchrones entre composants ; deux types d'actions existent : *in* et *out* qui sont respectivement des événements acceptés et envoyés par un composant. Le modèle composant contient également une section de description du comportement (clause *behavior*) qui correspond au fonctionnement observable du composant comme, par exemple, l'ordonnancement des événements ou des appels aux services. De cette façon, l'environnement de Rapide peut simuler le fonctionnement de l'application. Pour la communication des composants proprement dite, on utilise des constructeurs : le constructeur *action* pour spécifier une communication asynchrone et le constructeur *function* pour spécifier une communication synchrone. Chacun de ces constructeurs peut être utilisé avec les mots-clés *extern* ou *public* qui permettent de définir respectivement un port de sortie et un port d'entrée.
- Le modèle *architecture* contient la déclaration des instances de composants et des règles de connections entre ces instances. Toutes les instances sont déclarées sous forme de variables. La règle d'interconnection est composée de deux parties : la partie gauche qui contient une expression d'événements qui doit être vérifiée, la partie droite qui contient une expression d'événements qui doivent être déclenchés lorsque l'expression de la partie gauche est vérifiée. Les parties gauche et droite peuvent être connectées par trois types d'opérateurs : (a) l'opérateur *to* connecte deux expressions d'événements simples. Il ne peut y avoir qu'un événement possible vers un composant, (b) l'opérateur ||> connecte deux expressions quelconques. Dès que la partie gauche est vérifiée, tous les événements contenus dans la partie droite sont déclenchés. Ils sont envoyés vers l'ensemble des destinataires désignés dans cette expression, (c) l'opérateur => est identique au précédent mais l'ordre de l'évaluation des règles est contrôlé. Un déclenchement de cette règle est causalement dépendant des déclenchements antérieurs de cette même règle. Cet opérateur de connection est appelé opérateur « pipe-line ». Les contraintes (clause *constraint*) peuvent être utilisées pour décrire l'architecture. Elles permettent de restreindre le comportement de l'architecture en définissant des patrons d'événements à appliquer pour certaines connections entre composants.

3.3.1.2 Mécanismes de raffinement dans Rapide

Rapide utilise des patrons d'événements qui servent à définir la mise en correspondance entre les architectures de différents niveaux d'abstraction.

Rapide propose trois mécanismes permettant d'accomplir un raffinement, chacun offrant un niveau différent de détail :

- au niveau de la spécification, *les règles réactives* associées aux interfaces constituent un moyen très concis de spécifier le comportement d'un module,
- afin de fournir plus de détails sur la façon dont le module fonctionne, *une sous-architecture* associée à l'interface peut être nécessaire pour spécifier la structure interne, au moyen de sous-composants et de connections,
- enfin, Rapide fournit un *ensemble de constructions de langage procédural* qui permet d'implémenter complètement un module de manière conventionnelle.

Ces mécanismes permettent uniquement un raffinement de comportement de l'architecture. Le raffinement structurel n'est pas possible dans Rapide.

La mise en correspondance est une relation entre deux architectures ou entre systèmes (i.e. instances d'architectures) et architectures. L'idée principale des correspondances est de définir comment des événements dans un système sont reliés aux événements dans un autre. Dans Rapide, la vérification de la « correction » se fait a posteriori et de manière informelle en comparant les événements générés et leur enchaînement par simulation des deux architectures : l'architecture de base et celle qui l'a raffine. Si la trace des événements générés par la première est incluse dans celle générée par la seconde, la relation de raffinement est alors établie entre les deux archietctures.

Rapide ne fournit pas de mécanisme spécifique pour un raffinement horizontal ou vertical. Les mises en correspondance indiquent uniquement quels patrons d'événements sont à comparer. Comme il n'y a aucune construction structurale, il est seulement possible d'observer un raffinement comportemental et des propriétés fondées sur les ensembles partiellement ordonnés observés d'événements. Cette méthode utilise plusieurs sous-langages pour décrire des aspects différents de la spécification architecturale, tels que les interfaces ou les modules objets, mais Rapide ne fournit de moyen ni pour générer du code, ni pour réutiliser le résultat d'étapes de raffinement. Par ailleurs, il n'est pas possible de distinguer des niveaux d'abstraction différents.

L'exemple qui suit montre l'utilisation des correspondances pour relier un système particulier à une architecture de référence X/Open dans le but de tester si le système satisfait des contraintes formelles définies de X/Open.

3.3.1.3 Exemple de l'architecture X/Open

Une architecture X/Open comme le montre la figure Fig. II.1 comporte usuellement une instance du composant de l'application *AP* (« Application Program »), une instance du composant gestionnaire de transaction *TM* (« Transaction Manager ») et un ensemble de gestionnaires de ressource *RMs* (« Ressources Managers »). Ces différents éléments s'échangent les services nommés *AX*, *TX* et *AR* à travers des connections et par l'envoi d'événements. Le service *TX* est partagé par *TM* avec *AP,* le service *AR* est partagé par *AP* avec chacun des *RMs* et le service *XA* est partagé par *TM* avec chacun des *RMs*.

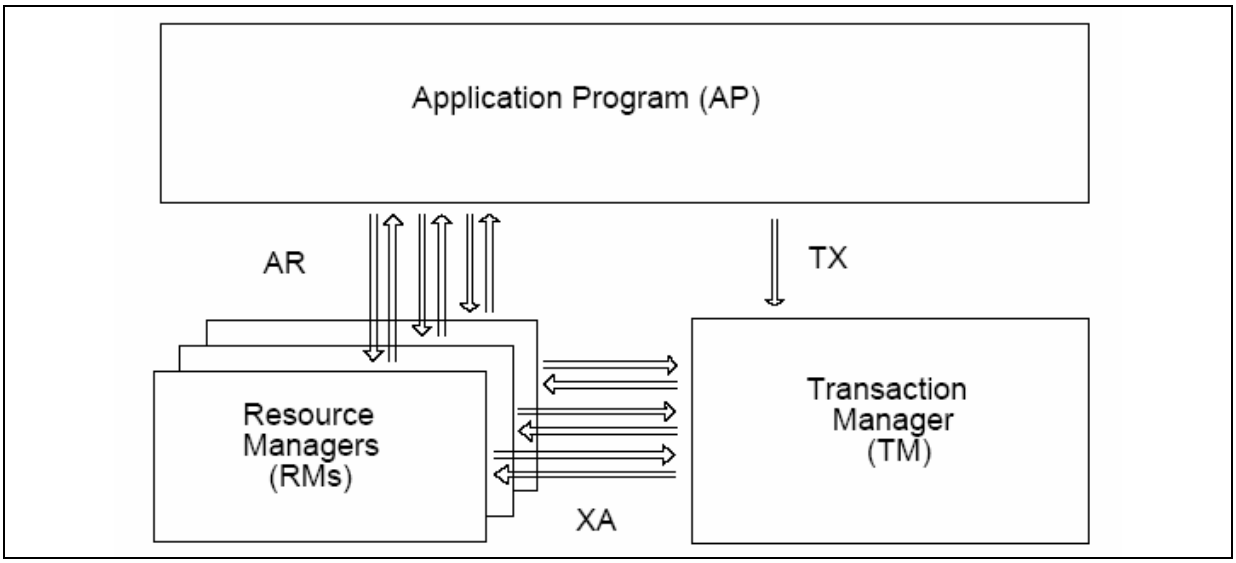


Figure II.2. : Architecture X/Open

Une première spécification de ce système peut être écrite dans Rapide de la façon suivante :

```
Architecture X/Open_Architecture (NumRMs : Integer)
       return  X/Open is
        AP : Application_Program(NumRMs);
        TM : Transaction_Manager(NumRMs);
        RMs : array(Integer) of  Resource_Manager;
Connect
       AP.TX to TM.TX;
      for   i : Integer   in  1..NumRMs   generate
                TM.XA(i)  to RMs[i].XA;
                AP.AR(i)  to RMs[i].AR;
      end  generate
…
…
end  architecture X/Open_Architecture;
```

Les règles de connection (clause *connect*) relient les services des différents composants (interfaces). La première règle porte sur les services de l'application et ceux du gestionnaire de transaction. Elle définit un même ensemble de connections de base (*open(), close(), begin_call(?x), commit_call(?x), roll_back_call(?x)*) pour les deux composants présentés dans la figure II.2 par les services AR, TX et XA . Les deuxième et troisième règles de connections utilisent la boucle (*for … generate … end generate*) pour relier (*NumRMs* fois) les services de *TM* avec ceux de chacun des *RMs* et les services de *AP* avec ceux de chacun des *RMs*.

Parmi les contraintes de l'architecture X/Open figure l'atomicité des transactions qui exige que les effets de toutes les opérations sur les ressources *RMs* soient ou bien effectués de façon permanente (*commit*) ou bien défaits (*rollback*). Cette contrainte d'atomicité est spécifiée dans une architecture Rapide comme contrainte sur tous les *RMs* impliqués. Un exemple de résultat d'une simulation de l'exécution de l'architecture est montré dans Fig. II.3. où sont représentés les différents événements (dont le nom est

préfixé par le nom du service en question, le suffixe *ret* exprime un retour à un appel *call*) impliqués dans une transaction de même que leur enchaînement dans le temps.

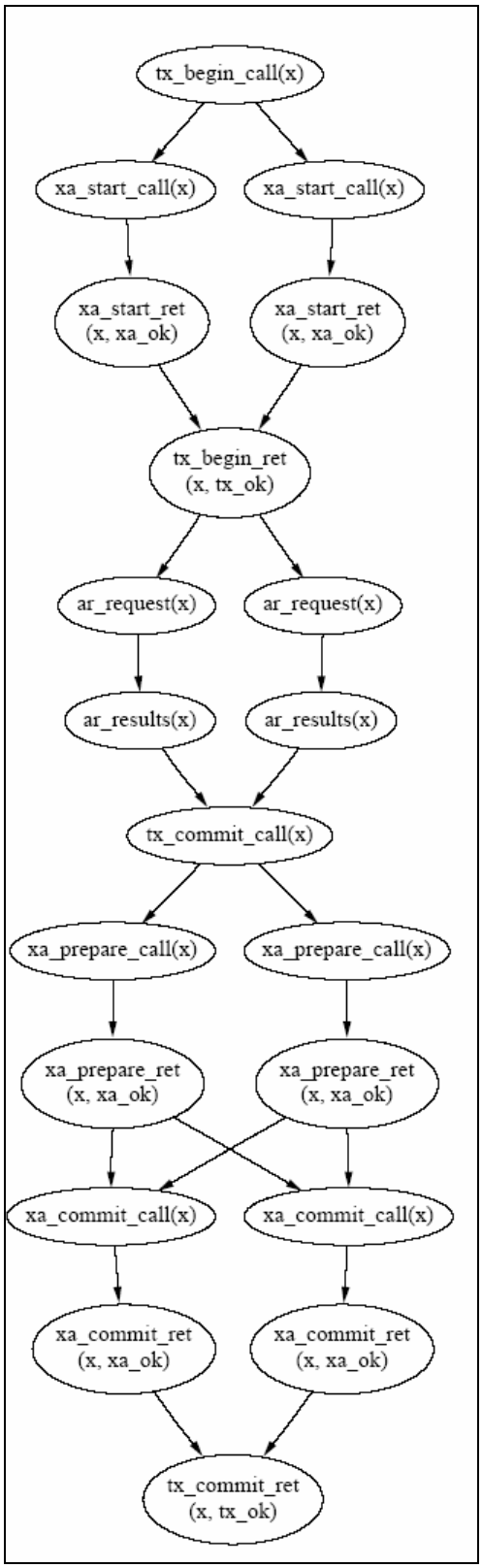


Figure II.3. : Résultat d'une simulation de l'exécution de l'architecture X/Open

L'architecture d'un système X/Open définie précédemment était centralisée. Nous voulons maintenant la raffiner pour obtenir un système X/Open distribué (*Combined_System*) comme celui présenté dans Fig. II.4. Il s'agit en fait de décomposer AP en Scheduling *AP* et *Billing AP*, deux composants qui intergiront et qui échangeront chacun des services avec un *TM* et plusieurs *RMs*.

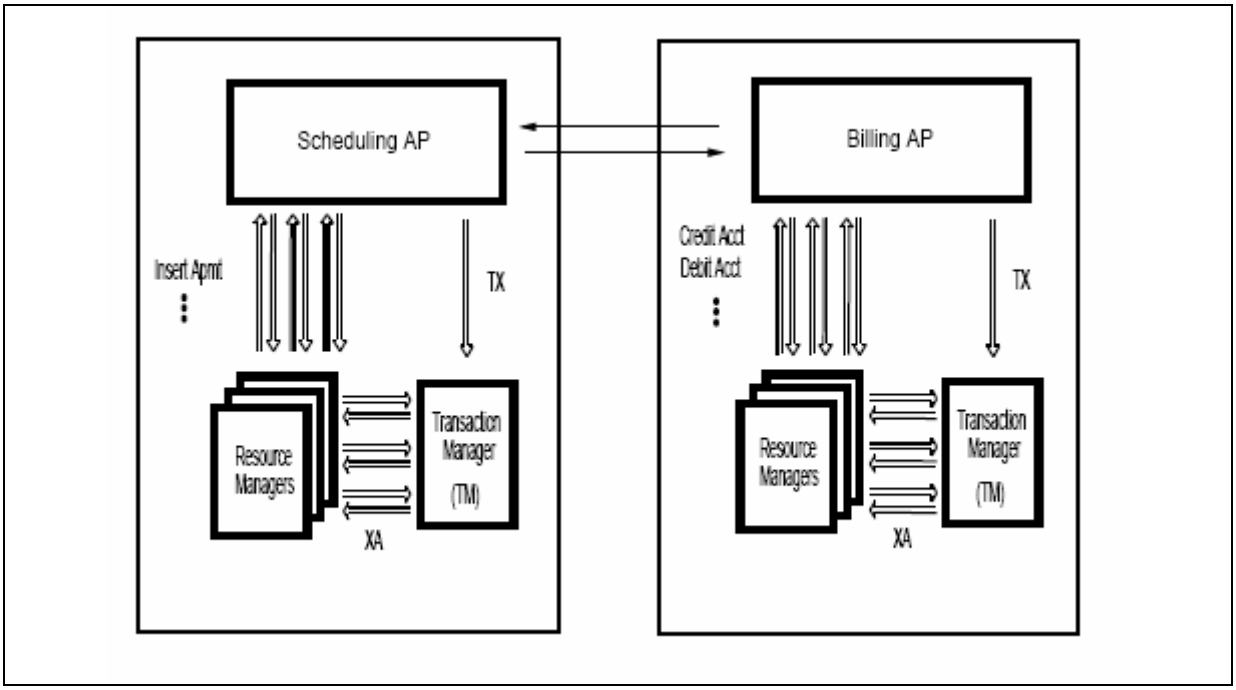


Figure II.4. : Architecture X/Open distribuée (Combined System)

Afin de définir des mises en correspondance entre différentes architectures, Rapide fournit la syntaxe suivante :

```
map ::=
        map name from name to name is
                { declaration }
        rules
                { rule }
         end  [ map ] [name ]  ';'
rule ::=
          patron  '=>'  { map_statement ';'} ';'
map_statement ::=  [ {state_assignment} ] [ restricted_pattern ';']
```

Les name indiqués après *from* et après *to* sont les noms des architectures ou interfaces (composants) objets de la mise en correspondance. Le *map* peut déclarer des objets locaux dans la partie *declaration*. La partie *rule* met en correspondance les patrons événements de deux architectures.

Ainsi, le patron de mise en correspondance de l'architecture X/Open centralisée avec l'architecture X/Open distribuée (Comnined System) peut être défini dans Rapide comme suit :

```
map Correspond from Combined_System to X/Open_Architecture1 is
        function Commun(u, v : xid) return xid is …
        A : array(xid) of  xid  is …
       rules
            -- recognize related transactions
             ( ?y, ?z) in xid
               Schedule_Request ( ) → ( TX.begin_ret(?y)
               and  ( Bill_Request ( ) → TX.begin_ret(?z) ))
               A[ ?y] := Common( ?y, ?z) ;
               A[ ?z] := A[ ?y] ;
              -- map prepare ret events
               (?x  in  xid)  (?rc in return_code)
               XA.prepare_ret (?x, rc)   => XA.prepare_ret (A[ ?x], rc) ;
              -- map commit call events
               (?x  in  xid)  (?rc in return_code)
               XA.commit_ret (?x, rc)   => XA.commit_ret (A[ ?x], rc) ;
        end
```

Par exemple, la règle *Schedule_Request ( ) → (TX.begin_ret(?y)* ***and*** *(Bill_Request ( ) → TX.begin_ret(?z)))* met en correspondance les services *Schedule_Request etBill_Request* apportés par la nouvelle architecture (distribuée) avec le service *begin_ret* défini dans l'ancienne architecture (centralisée).

Si *X* et *Y* sont des architectures et *M* est une mise en correspondance entre eux, un appel *M(X,Y)* résulte en un système qui s'exécute en transformant des *posets* d'événements produits par *X* par des *posets* des événements produits par *Y*, lesquels *posets* seront vérifiés contre les contraintes de *Y*. La simulation de l'éxécution de la nouvelle architecture (cf. Fig. II.5.) montre que le nouveau comportement inclut le comportement de la première architecture. La relation de raffinement est alors établie entre les deux architectures (centralisée et distribuée) de X/Open.

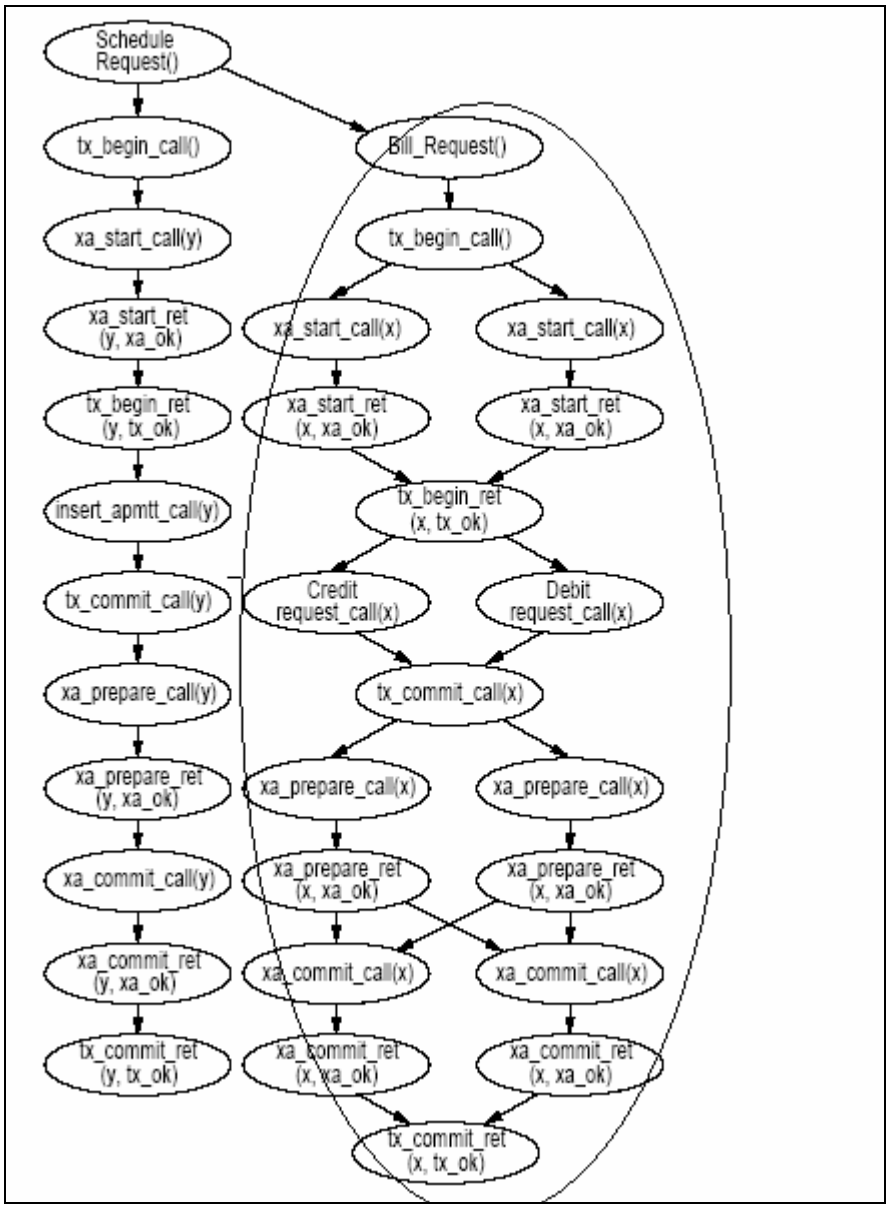


Figure II.5. : Résultat d'une simulation de l'exécution de X/Open distribuée (Combined System)

3.3.2. SADL

SADL ("Structural Architecture Description Language") a été conçu pour décrire la structure architecturale d'un système logiciel en termes de composants et de connecteurs ainsi que pour le raffinement de styles architecturaux [MQR 95] SADL permet de définir des patrons de raffinement réutilisables et pouvant être composés. Deux architectures mises en correspondance par le raffinement doivent permettre d'effectuer exactement les mêmes communications entre composants. L'objectif n'est pas le raffinement de comportement mais celui de l'expression de hiérarchies d'architectures avec jusqu'à un certain point la préservation de propriétés. Le raffinement dans SADL ne conduit pas à la génération de code. Il y a une distinction plutôt claire entre décomposition (en fait la description de composants composites statiques) et le raffinement de style, ce qui pourrait être apparenté aux raffinements horizontal et vertical respectivement.

3.3.2.1 Concepts de base

Les concepts de base de SADL sont *la mise en correspondance explicite* entre les architectures, les architectures génériques, les styles architecturaux et *les patrons de raffinement* des architectures. Un patron de raffinement est présenté dans une table contenant deux schémas d'architectures, une association des éléments abstraits et concrets, et si nécessaires des contraintes sur un ou les deux schémas. Par convention, une variable de schéma qui se produit dans le schéma concret doit désigner le même élément, modulo le renommage.

3.3.2.2 Mécanismes de Raffinement

SADL permettant de supporter des patrons de raffinement correct, la notion de « vérification correcte » est étendue aux règles de transformations des architectures pour la construction incrémentale de la hiérarchie. En effet, une règle de transformation ajoutant des détails à une description d'architecture est non seulement correcte une fois qu'elle conduit (ou produit) un raffinement de description correcte de l'architecture mais la preuve qu'une règle est correcte implique souvent que les preuves de quelques étapes de raffinement sont correctes.

La composition est une méthodologie de développement évitant des descriptions architecturales non homogènes, mais les remplacements doivent être simultanés et le raffinement est complexe. En pratique, il est difficile de trouver des modes de composition préservant un raffinement correct. En effet, il n'est généralement pas possible d'inférer de nouveaux faits sur l'architecture abstraite composite depuis une architecture concrète composite.

Dans SADL, il existe deux formes de composition d'architectures :

- La composition horizontale : utilisée pour composer des instances de patrons de raffinement pour former une architecture composite de raffinement. Elle est également utilisée pour composer des architectures existantes dans des architectures plus grandes.
- La composition verticale : utilisée pour enchaîner une séquence d'architectures correctes (une hiérarchie d'architecture), permettant de conclure que l'architecture concrète est correcte si elle respecte l'architecture la plus abstraite dans la hiérarchie.

Contrairement à l'approche compositionnelle qui a pour objectif la vérification complète de la hiérarchie, l'approche incrémentale a pour objectif d'éviter l'introduction des erreurs durant le processus de développement. A chaque décision de conception, la hiérarchie est transformée pour refléter l'état courant de la conception. Si toutes les transformations de la hiérarchie préservent son exactitude et si le point de départ est un raffinement relativement correct de la hiérarchie, alors la hiérarchie résultante complète est garantie d'être correcte par construction.

Des transformations peuvent être vues comme étant appliquées à la hiérarchie entière plutôt qu'à différentes descriptions dans la hiérarchie produite par le raffinement par composition. Le niveau concret réalisé est également un raffinement correct de la description de niveau abstrait. Une règle de transformation de hiérarchie d'architectures est correcte si et seulement si le résultat de chaque application de raffinement à une hiérarchie est correct (toutes les étapes de raffinement étant correctes). Les règles de transformations correctes préservent l'exactitude de la hiérarchie.

3.3.2.3 L'exemple du compilateur

Afin d'illustrer le raffinement d'architectures dans SADL, nous prenons l'exemple d'un compilateur dont l'architecture est montrée dans Fig. II.6 et dont la spécification est présentée dans Fig. II.7.

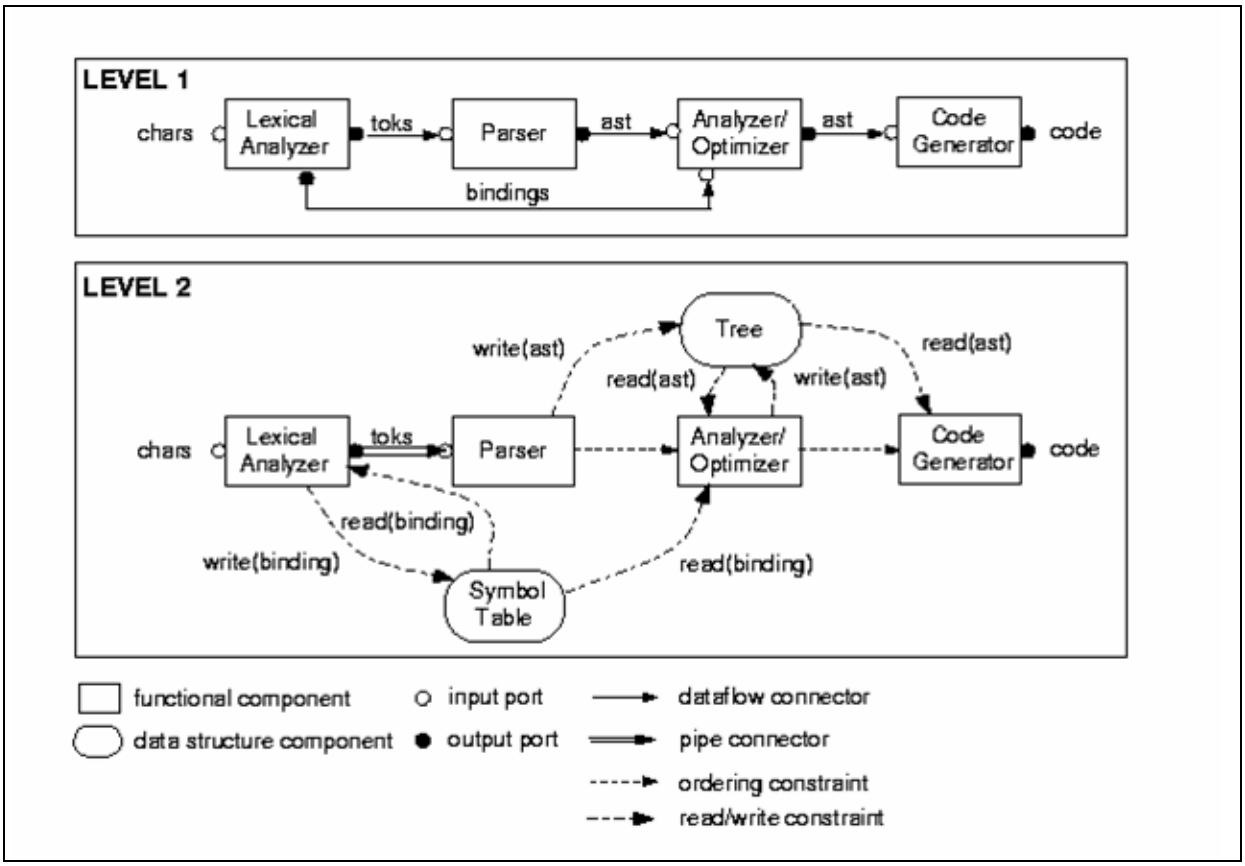


Figure II.6. : Architecture d'un compilateur

L'architecture du compilateur est considérée à travers deux niveaux de raffinement : le premier montre les différents composants et leurs connections, le second montre la première architecture raffinée par les structures de données. La spécification en SADL de l'architecture du premier niveau est donnée ci-après.

```
compiler_L1: ARCHITECTURE
    [char_iport: SEQ(character) -> code_oport: code]
  IMPORTING character, code, token, binding, ast FROM compiler_types
  IMPORTING Function FROM Functional_Style
  IMPORTING Dataflow_Channel, Connects FROM Dataflow_Style
BEGIN
COMPONENTS
  lexical_analyzer: Function
     [char_iport: SEQ(character)
         -> token_oport: SEQ(token), bind_oport: SEQ(binding)]
  parser: Function
     [token_iport: SEQ(token) -> base_ast_oport: ast]
  analyzer_optimizer: Function
     [base_ast_iport: ast, bind_iport: SEQ(binding)
         -> full_ast_oport: ast]
  code_generator: Function [full_ast_iport: ast -> code_oport: code]
CONNECTORS
  token_channel:    Dataflow_Channel<SEQ(token)>
  bind_channel:     Dataflow_Channel<SEQ(binding)>
  base_ast_channel: Dataflow_Channel<ast>
  full_ast_channel: Dataflow_Channel<ast>
CONFIGURATION
  token_flow: CONNECTION
    = Connects(token_channel, token_oport, token_iport)
  bind_flow: CONNECTION
    = Connects(bind_channel, bind_oport, bind_iport)
  base_ast_flow: CONNECTION
    = Connects(base_ast_channel, base_ast_oport, base_ast_iport)
  full_ast_flow: CONNECTION
    = Connects(full_ast_channel, full_ast_oport, full_ast_iport)
END compiler_L1
```

Figure II.7. : Spécification de l'architecture du compilateur en SADL

Afin d'obtenir une architecture *arch_L2* du niveau 2 à partir d'une architecture *arch_L1*, une règle de mise en correspondance est définie selon la syntaxe suivante :

arch_map : **MAPPING FROM** arch_L1 **TO** arch_L2
BEGIN
comp - → (new_comp)
conn - → (new_comp!subcomp)
port -- → ( )
END

Le composant *comp* de l'architecture du niveau 1 est mis en correspondance avec *new_comp* de l'architecture du niveau 2 à travers la règle "*comp - → (new_comp)*". Le connecteur *conn* de l'architecture

de niveau 1 est raffiné par le sous-composant *new_comp* (sous-composant de *subcomp*). Le *port* du niveau 1 est éliminé de l'architecture du niveau 2.

Pour le raffinement du compilateur, les règles de mise en correspondance des deux architectures *compiler_L1* et *compiler_L2* sont les suivantes :

```
compiler_map: MAPPING FROM compiler_L1 TO compiler_L2
BEGIN
  lexical_analyzer --> (lexical_analyzer_module)
  bind_oport       --> ( )
  base_ast_oport   --> ( )
  base_ast_iport   --> ( )
  bind_iport       --> ( )
  full_ast_oport   --> ( )
  full_ast_iport   --> ( )
  token_channel    --> (token_pipe)
  bind_channel     --> (lexical_analyzer_module!symbol_table)
  base_ast_channel --> (abstract_syntax_tree)
  full_ast_channel --> (abstract_syntax_tree)
  bind_flow        --> (lexical_analyzer_module!write_bind, read_bind)
  base_ast_flow    --> (write_base_ast, read_base_ast,
                            precedence_1, precedence_2)
  full_ast_flow    --> (write_base_ast, read_base_ast,
                            precedence_1, precedence_2)
END compiler_map
```

Par application de ces règles nous obtenons l'architecture plus « concrète » suivante :

```
compiler_L2: ARCHITECTURE [char_iport: SEQ(character) -> code_oport: code]
  IMPORTING character, code, token, binding, ast FROM compiler_types
  IMPORTING Function FROM Functional_style
  IMPORTING Pipe, Finite_Stream, Connects FROM Process_Pipeline_style
  IMPORTING Variable, Reads, Writes FROM Shared_Memory_style
  IMPORTING Start_After_Finish_Of FROM Batch_Sequential_style
BEGIN
COMPONENTS
  lexical_analyzer_module: ARCHITECTURE
          [char_iport: SEQ(character) -> token_oport: Finite_Stream(token)]
     EXPORTING lexical_analyzer, symbol_table
     IMPORTING character, token, binding FROM compiler_types
     IMPORTING Function FROM Functional_style
     IMPORTING Finite_Stream FROM Process_Pipeline_style
     IMPORTING Variable, Reads, Writes FROM Shared_Memory_style
  BEGIN
  COMPONENTS
     lexical_analyzer: Function
        [char_iport: SEQ(character) -> token_oport: Finite_Stream(token)]
     symbol_table: Variable(SEQ(binding))[ -> ]
  CONFIGURATION
     write_bind: CONSTRAINT = Writes(lexical_analyzer, symbol_table)
     read_bind: CONSTRAINT = Reads(lexical_analyzer, symbol_table)
  END lexical_analyzer_module
  parser: Function[token_iport: Finite_Stream(token) -> ]
  analyzer_optimizer:   Function[ -> ]
  code_generator:       Function[ -> code_oport: code]
  abstract_syntax_tree: Variable(ast)[ -> ]
CONNECTORS
  token_pipe: Pipe<Finite_Stream(token)>
CONFIGURATION
  token_flow: CONNECTION
    = Connects(token_pipe, token_oport, token_iport)
  read_bind: CONSTRAINT
    = Reads(analyzer_optimizer, lexical_analyzer_module!symbol_table)
  write_base_ast: CONSTRAINT = Writes(parser, abstract_syntax_tree)
  read_base_ast: CONSTRAINT
    = Reads(analyzer_optimizer, abstract_syntax_tree)
  write_full_ast: CONSTRAINT
    = Writes(analyzer_optimizer, abstract_syntax_tree)
  read_full_ast: CONSTRAINT
    = Reads(code_generator, abstract_syntax_tree)
  precedence_1: CONSTRAINT
    = Starts_After_Finish_Of(analyzer_optimizer, parser)
  precedence_2: CONSTRAINT
    = Starts_After_Finish_Of(code_generator, analyzer_optimizer)
END compiler_L2
```

Concrètement à Rapide, le raffinement dans SADL concerne les aspects structurels et non comportementaux.

3.4. Autres langages pour le raffinement

En dehors des ADL, il existe en génie logiciel plusieurs méthodes formelles pour le raffinement [Bol 04]. Parmi les méthodes les plus connues et aussi les plus utilisées, il y a : B, VDM et Z.

3.4.1. B

La méthode B tient une place tout à fait particulière puisqu'elle bénéficie d'outils commerciaux utilisés dans l'industrie du logiciel. Le raffinement dans la méthode B est décrit explicitement par l'usage de machines introduites par les mots-clés "REFINEMENT" ou "IMPLEMENTATION", et dans lesquelles sont décrites de façon plus poussée les mêmes opérations que dans la machine abstraite raffinée. Chaque raffinement repose alors sur une relation bien précise, établie par l'invariant de la machine qui raffine. Néanmoins, chaque étape nécessite que les préconditions des opérations s'affaiblissent et que leurs postconditions se renforcent : le comportement qui résultera du raffinement doit donc être un comportement possible de l'abstraction. Des outils apportent une aide aux utilisateurs pour effectuer les preuves ou automatiser la génération du code à partir d'implémentations.

3.4.2. VDM

La méthode de développement VDM ("Vienna Development Method"), antérieure à B, a été conçue à l'origine pour la spécification formelle et le développement des aspects fonctionnels de systèmes. C'est une méthode formelle orientée modèle fondée sur une sémantique dénotationelle, destinée à supporter le raffinement pas à pas de modèles abstraits en implémentations concrètes, mais sans mener à un code exécutable. Une spécification en VDM consiste en une description d'états (pouvant contenir des prédicats d'invariant et d'initialisation), une collection de définitions de domaines (auxquelles des invariants peuvent être ajoutés), une collection de définitions de constantes, une collection d'opérations et une collection de fonctions. La signification d'une spécification en VDM peut être vue comme un ensemble de modèles et la sémantique dénotationnelle est basée sur la construction de l'ensemble des modèles possibles. La relation de raffinement de VDM établit qu'une implémentation doit avoir une fonctionnalité dont il est possible de dire qu'elle implémente la construction spécifiée. Autrement dit, le comportement de l'abstraction doit être un comportement possible de la spécification concrète qui la raffine. Il ne s'agit pas d'un raffinement vertical, même s'il est basé sur le raffinement de fonctions et d'opérations, car des détails peuvent être ajoutés à la spécification de cette manière. Ce raffinement à la fois comportemental et compositionnel préserve les propriétés internes des spécifications abstraites entre autres à travers des invariants.

3.4.3. Z

La notation de spécification formelle Z est fondée sur la théorie des ensembles de Zermelo-Fraenkel [Spi 92] et sur la logique des prédicats du premier ordre. Z supporte des raffinements d'opérations, de données et de données fonctionnelles : les préconditions sont affaiblies et les opérations deviennent plus déterministes à chaque raffinement. En effet, en plus de ce que fait VDM, Z effectue une distinction entre les raffinements d'opérations, de données et de données fonctionnelles. Par conséquent, le raffinement recouvre un peu plus de choses. Par ailleurs, il est contraint par le cadre formel car comme dans la méthode B, le raffinement vertical est plutôt limité à la modification des corps d'opérations, en exprimant plus précisément la manière dont le résultat est obtenu. L'établissement de la relation de raffinement repose sur la vérification de conditions logiques ; il s'agit à nouveau d'un test a posteriori, mais les obligations de preuve constituent un moyen propice à la préservation de propriétés. En réalité, la relation de raffinement de Z suit l'idée opposée à celle de VDM et de la réduction en B : elle s'exprime comme l'affaiblissement des préconditions de l'opération abstraite et le renforcement de ses postconditions (l'opération qui raffine l'autre est plus déterministe). Par conséquent, le raffinement résulte en un comportement possible de l'abstraction. Enfin, tout comme dans le cas de VDM, le raffinement dans Z est compositionnel, sans distinction claire entre les niveaux d'abstraction et ne conduit pas à la génération de code exécutable.

3.5. Outils pour le raffinement

Concernant les outils logiciels pour supporter le raffinement, plusieurs ADL transforment directement des modèles architecturaux en code exécutable via la compilation. Darwin, MetaH, et UniCon réalisent ceci de la même manière que les MIL (« Modelling Interaction Languages ») [Gar 00] : les composants architecturaux sont implémentés dans un langage de programmation et la description architecturale sert à assurer l'interconnection appropriée ainsi que les différentes transmissions entre eux.

Contrairement aux ADL précédents, SADL et Rapide fournissent un support pour le raffinement d'architectures à travers des niveaux multiples d'abstraction. Le support de SADL est partiel car il exige des preuves manuelles pour les constructions de mise en correspondance entre un modèle abstrait et un modèle architectural plus concret. Rapide quant à lui, supporte les correspondances d'événements entre des architectures individuelles grâce à un sous-langage exécutable. En effet ces correspondances sont compilées par un outil rendant possible la vérification que les événements générés pendant la simulation de l'architecture concrète satisfont les contraintes de l'architecture abstraite.

En conclusion, dans RAPIDE le raffinement est supporté seulement par un outil de simulation offrant des mécanismes de mise en correspondance a posteriori de deux architectures à travers leur exécution. Par ailleurs aucune primitive n'est à proprement parler fournie pour le raffinement architectural et seul le raffinement des comportements est supporté par simulation. SADL, conçu spécialement pour le raffinement des architectures logicielles, est également supporté par un seul outil d'interprétation de patrons de raffinement qui met en correspondance différentes architectures avec la contrainte que l'architecture concrète soit une implantation fidèle de l'architecture abstraite. Ceci rend le raffinement rigide dans le sens où il ne permet pas de différer des décisions de conception. Contrairement à RAPIDE, le raffinement dans SADL est uniquement structurel.

4. Bilan

En résumé, nous avons dans ce chapitre présenté différents ADL dont deux seulement prennent en charge le raffinement des architectures. Dans RAPIDE seul le raffinement des comportements est supporté par simulation. Dans SADL le raffinement est uniquement structurel. Aucun d'entre eux cependant ne fournit à vraiment parler de support logiciel pour le raffinement. La comparaison de ces deux langages par rapport au raffinement des différents éléments architecturaux est présentée dans le tableau ci-dessous.

Langages	***Raffinement d'un Composant ou Connecteur***			***Raffinement d'architecture***			
	Comportement	***Port***	***Données***	***Comportement***	***Port***	***Données***	***Structure***
SADL	*Non*	*Non*	*Non*	*Non*	*Non*	*Non*	*Oui*
Rapide	*Oui*	*Non*	*Non*	*Oui*	*Non*	*Non*	*Non*

La comparaison de ces deux langages par rapport aux mécanismes de raffinement et outils de vérification de raffinement est présentée dans le tableau qui suit.

Langages	***Mécanismes de Raffinement***	***Outil pour le raffinement***	***Outil pour la vérification du Raffinement***
SADL	*Patrons de Raffinement et Mapping*	***Non***	*Oui, mais partiel*
Rapide	*Patrons d'événements et Mapping*	***Non***	*Par simulation*

Notre objectif est par conséquent de fournir un environnement logiciel, permettant de supporter le raffinement d'architectures logicielles à travers multiples niveaux d'abstraction, avec prise en charge des différents éléments architecturaux (composant, connecteur, etc.) et en permettant aussi bien le raffinement de comportement que celui de structure. Cet environnement étant construit autour du langage de description et de raffinement d'architecture appelé *ArchWare ARL,* nous présentons celui-ci dans le troisième chapitre avant de présenter l'environnement *Refiner* fondé sur la logique de réécriture.

En fait, l'environnement *Refiner* que nous proposons dans cette thèse permet un développement "centré-architecture" de systèmes logiciels depuis un niveau abstrait (proche de la spécification du cahier des charges) jusqu'à un niveau concret (proche d'un langage de programmation). Ceci en s'appuyant sur des actions de raffinement de base fournies pour le langage *Archware ARL* ainsi que sur le modèle de processus de raffinement dédié à ce dernier. A chaque niveau de la hiérarchie de raffinement, *Refiner* supporte la réutilisation de modèles architecturaux existants, et au niveau concret, supporte en plus la génération de code.

Chapitre 3 : ArchWare ARL : Language de Description et de Raffinement d'Architectures Logicielles

Chapitre 3 : ArchWare ARL : Language de Description et de Raffinement d'Architectures logicielles

Dans ce chapitre, nous présentons le langage *ArchWare ARL* [Oqu 03] autour duquel nous avons construit l'environnement *Refiner*. Ce dernier est un langage formel permettant la description et le raffinement d'architectures logicielles. Il permet de prendre en considération le raffinement fondé sur les termes architecturaux, i.e. *données, comportements, connections et structures* à travers des mécanismes que nous présentons également. ArchWare ARL permet le raffinement progressif (*stepwise refinement*) d'une architecture abstraite en une architecture concrète prévue pour l'implémenter. Une étape de raffinement implique l'application d'une *action de raffinement* qui fournit une solution correcte d'une transformation architecturale.

Les actions de raffinement sont exprimées par des *pré-conditions*, des *transformations* et des *post-conditions*. Les pré-conditions sont des conditions qui doivent être satisfaites dans un modèle architectural abstrait avant l'application d'une action de raffinement. Les post-conditions doivent être satisfaites dans le modèle obtenu par l'application d'une action de raffinement sur un modèle architectural abstrait. Les deux modèles (le modèle abstrait et le modèle résultat du raffinement) peuvent contenir des concepts différents, voire même différents styles architecturaux [CLO 03]. Une transformation exprimée par une action de raffinement montre comment on peut transformer une architecture satisfaisant les pré-conditions vers un modèle moins abstrait satisfaisant les post-conditions. Un *modèle de transformation* architecturale est dit correct si et seulement si il satisfait les pré et les post-conditions, il peut être appliqué sans preuve pour le développement des architectures concrètes spécifiques depuis des architectures abstraites. Dans certains cas, pour prouver que le raffinement est correct, des hypothèses sont rajoutées, par exemple, pour vérifier si un comportement est un raffinement d'un autre comportement, nous rajoutons des hypothèses sur les valeurs reçues via les connections d'entrées libres. Ces hypothèses sont des obligations de preuve déchargées au moment de l'application, éventuellement en utilisant l'outil d'analyse ArchWare AAL [AGM 02] afin d'établir si oui ou non un raffinement est correct.

Les actions de raffinement fournies par *ArchWare ARL* sont compositionnelles aussi bien "horizontalement" que "verticalement", permettant un développement et un raisonnement progressifs. En effet, une architecture concrète d'un système logiciel de grande taille est souvent développée à travers une hiérarchie "verticale" des architectures relatives. La hiérarchie d'une architecture est une séquence linéaire de deux ou plusieurs architectures qui diffèrent selon plusieurs aspects. Prenons à titre d'exemple, une architecture abstraite contenant des composants fonctionnels reliés par des connections de flots de données. Cette dernière peut être implantée dans une architecture concrète en termes de procédures, connections de contrôle et des variables partagées. En général, une architecture abstraite est d'une petite taille et facile à comprendre. Le raffinement "horizontal" consiste en l'application de différentes actions de raffinement sur différentes parties de la même architecture abstraite en conservant le même niveau d'abstraction.

En s'appuyant sur les actions de raffinement définies par un architecte et les modèles de transformation, plus précisément de raffinement, *Archware ARL* permet un développement rapide "centré architecture" d'un système logiciel depuis un niveau abstrait (proche de la spécification du cahier de charge) vers un niveau très concret (proche d'un langage de programmation). A chaque niveau (de la hiérarchie) de raffinement, *ArchWare ARL* permet la réutilisation des modèles architecturaux existants et au niveau concret permet la génération centrée-architecture de code. Le reste de ce chapitre est dédié à la modélisation et la description des éléments architecturaux dans *ArchWare ARL* qui seront suivies d'une présentation détaillée de l'approche de raffinement avec *ArchWare ARL*.

1. Description d'architectures dans ArchWare ARL

Les *architectures* (souvent appelées ailleurs configurations ou systèmes) dans *ArchWare ARL* se composent d'éléments architecturaux qui peuvent être des *composants* et des *connecteurs*. Les composants fournissent les fonctionnalités de base du système tandis que les connecteurs relient un ensemble de composants. Composants et connecteurs peuvent être eux-mêmes décrits par des architectures. Ils ont un comportement interne ainsi que des *ports* pour interagir avec leur environnement. Les ports permettent des *connections d'entrée* (incoming) et/ou des *connections de sortie* (*outgoing*). Les composants et les connecteurs sont en réalité modélisés de la même manière, la différence est essentiellemlent d'ordre méthodologique.

1.1. Modèle architecture

Une architecture peut être décrite dans *ArchWare ARL* en utilisant la syntaxe suivante :

```
archetype architectureDefId is architecture {
  types is { typeDeclarations }
  ports is { portDeclarations }
  behaviour is compose { compositionOfComponentsUsingConnectors }
    }
```

où :

- *architectureDefId* est un identifiant unique de la définition d'une architecture,
- *typeDeclarations* est l'ensemble des déclarations explicites de types présentant des alias de nouveaux types dans la portée de *architectureDefId*. Ils peuvent être utilisés dans la déclaration des ports et du comportement,
- *portDeclarations* est l'ensemble des déclarations de ports dans la portée de *architectureDefId* utilisés pour grouper des connections,
- *compositionOfComponentsUsingConnectors* est un ensemble d'éléments architecturaux de *architectureDefId* (ces éléments communiquent via des connections de ports). Rappelons que, les éléments architecturaux peuvent être des composants et des connecteurs, eux-mêmes décrits par des architectures.

Un port est déclaré en termes de connections d'entrée et de connections de sortie de la façon suivante :

```
archetype portDefId is port {
  incoming is { inConnections }
  outgoing is { outConnections }
} assuming { -- assumption on the port protocol }
```

où:

- *inConnections* est un ensemble de connections libres de *portDefId* utilisées pour les entrées (input),
- *outConnections* est un ensemble de connections libres de *portDefId* utilisées pour les sorties (output),
- une hypothèse « assumption » sur le protocol (comportement) du port peut être exprimée.

1.2. Modèle composant

Un composant peut être décrit dans ArchWare ARL de la manière suivante :

```
archetype componentDefId is component {
  types is { typeDeclarations }
  ports is { portDeclarations }
  behaviour is { componentBehaviour }
}
```

où :

- *componentDefId* est un identifiant unique de la définition d'un composant,
- *typeDeclarations* est l'ensemble des déclarations explicites des types présentant des alias des nouveaux types dans la portée de *componentDefId.* Les types peuvent être utilisés dans la déclaration des ports et du comportement,
- *portDeclarations* est l'ensemble des déclarations de ports dans la portée de *componentDefId* utilisés pour regrouper des connections (cf. modèle d'architecture),
- *componentBehaviour* est l'expression d'un comportement de *componentDefId* (le comportement d'un composant atomique représente le niveau « feuille » d'un comportement hiérarchiquement défini d'une architecture « arbre »).

Il est important de noter que dans *ArchWare ARL*, une architecture est elle-même un composant d'une architecture englobante. En effet, une architecture est perçue de l'extérieur comme étant un composant atomique d'un composant composite.

1.3. Modèle connecteur

Un connecteur peut être décrit dans ArchWare ARL en utilisant la syntaxe suivante :

```
archetype connectorDefId is connector {
  types is { typeDeclarations }
  ports is { portDeclarations }
  behaviour is { connectorBehaviour }
        }
```

où:

- *connectorDefId* est un identificateur unique de la définition d'un connecteur,
- *typeDeclarations* est l'ensemble des déclarations explicites des types présentant des alias des nouveaux types dans la portée de *connectortDefId.* Ces types peuvent être utilisés dans la déclaration des ports et du comportement,
- *portDeclarations* est l'ensemble des déclarations des ports dans la portée de *connectorDefId* utilisés pour grouper des connections,
- *connectorBehaviour* est l'expression d'un comportement de *connectorDefId* représentant "la colle" qui relie différents composants.

Il est également important de noter dans *ArchWare ARL* qu'un connecteur est lui même un composant d'architecture. En effet, un connecteur atomique est perçu de l'extérieur comme étant un composant atomique. Ainsi, les connecteurs composites sont des architectures puisqu'ils peuvent lier un ensemble de connecteurs entre eux.

2. Approche de Raffinement

Après la présentation des modèles architecturaux de base fournis par *ArchWare ARL*, nous présentons dans cette section, l'approche de raffinement dans ArchWare ARL ainsi que les différentes formes des relations de raffinement supportées à savoir : le raffinement des *comportements*, des *connections*, des *structures* et des *données*.

Les modèles architecturaux peuvent être raffinés progressivement depuis des descriptions abstraites (haut niveau) vers des descriptions concrètes (bas niveau), de telle sorte que les descriptions concrètes soient prouvées en tant qu'un raffinement correct des descriptions abstraites. L'approche *ArchWare ARL* pour le raffinement des architectures est fondée sur des transformations formelles des descriptions d'architectures. Elle fournit une sémantique compositionnelle pour le raffinement des éléments architecturaux, et par conséquent, ceux-ci sont raffinés, puis recomposés pour la construction d'architectures concrètes. Un des principes de base de l'approche du raffinement architectural dans *ArchWare ARL*, est la sous-spécification (« underspecification»(, c'est-à-dire, qu'à un « haut niveau » d'abstraction, on spécifie un élément architectural tout en laissant certains aspects non spécifiés. La diminution de cette sous-spécification passe par l'établissement de relations de raffinement des comportements, des connections, des structures, et des données des éléments architecturaux.

Le comportement externe d'un élément architectural est le comportement observé par son environnement. Le comportement interne concerne l'expression interne du comportement dans la portée de l'élément architectural. La structure d'un élément architectural est définie en termes d'éléments architecturaux et leurs connections (liaisons), ceci dans la portée de l'élément architectural d'un point de vue interne. Les connections d'un élément architectural sont les points d'interaction entre l'élément architectural et son environnement. De ce fait, le raffinement externe (cf. chapitre II, 3.1) concerne le comportement observable et les connections libres alors que le raffinement interne concerne le comportement interne et les connections attachées.

Dans *ArchWare ARL*, les raffinements internes et externes peuvent être exprimés. Tandis que les raffinements externes sont liés seulement aux comportements observables et aux connections libres des composants ou connecteurs, les raffinements internes permettent le raffinement de la structure interne du comportement d'un composant ou connecteur par un comportement primitif ou composé, i.e. un comportement décrit lui-même par une architecture. Le raffinement de connections traite les transformations des connections d'un élément architectural.

3. Relation de raffinement architectural en ArchWare ARL

Dans ArchWare ARL une relation de raffinement d'un point de vue externe ou interne s'établit sous les quatre formes présentées dans le chapitre 3. :

- raffinement du comportement,
- raffinement du port,
- raffinement de la structure,
- raffinement des données.

La forme fondamentale du raffinement dans *ArchWare ARL* est le raffinement du comportement. Le raffinement des ports, structures et données implique le raffinement du comportement modulo les ports, structures et données raffinées respectivement. Le raffinement architectural est alors une combinaison de ces quatre types de raffinement. Par exemple, un architecte peut d'abord définir une architecture abstraite

en utilisant seulement le type prédéfini *Any*, ensuite il peut appliquer un « raffinement de données » sur cette architecture afin de présenter la base et les types de données, puis procéder au « raffinement des ports » sur cette dernière architecture pour avoir des ports avec des connections « de grain plus fins » supportant des données de différents types, et enfin appliquer un «raffinement de structure» sur la structure du comportement composite en ajoutant de nouveaux connecteurs « de grain fin » et ainsi de suite.

4. Actions de raffinement des architectures logicielles

Le raffinement d'une architecture s'effectue en plusieurs étapes. Une étape de base est définie en termes d'actions de raffinement de base visant à transformer une architecture. Cette section définit les actions de raffinement de base fournies par ArchWare ARL et la notation pour leur application dans le cadre d'un processus de raffinement progressif.

4.1. Relation de raffinement d'architectures

Un modèle d'architecture peut être raffiné vers un modèle d'architecture plus concret. Ceci établit entre l'ensemble des modèles architecturaux une relation de raffinement binaire notée ► qui rappelons le est réflexive et transitive (cf. chapitre 2, section 3) :

architectureDef ∈ ***architecture*** ► *architectureRef* ∈ ***architecture***

ou plus explicitement :

archetype *architectureDefId* ***is architecture {***
types is { *typeDeclarations* ***}***
ports is { *portDeclarations* ***}***
behaviour is { *compositionOfComponentsUsingConnectors* ***}***
}
►
archetype *architectureRefId* ***is architecture {***
types is { *typeDeclarations'* ***}***
ports is { *portDeclarations'* ***}***
behaviour is { *compositionOfComponentsUsingConnectors'* ***}***
}

où :

- *typeDeclarations* et *portDeclarations* sont des déclarations de types et de ports avant le raffinement et *typeDeclarations'* et *portDeclarations'* sont des déclarations de types et de ports après le raffinement.
- *compositionOfComponentsUsingConnectors* est le comportement composite avant le raffinement et *compositionOfComponentsUsingConnectors'* est le nouveau comportement composite après le raffinement.

Une étape de raffinement s'effectue par l'application d'une action de raffinement que l'on peut exprimer en utilisant la syntaxe suivante :

on *architectureDefId* ***action*** *refinementActionId* ***is refinement (*** *actionParameters* ***) {***
pre is { *actionPreconditions* ***}***
post is { *actionPostconditions* ***}***
transformation is { *refinementActions* ***}***
} assuming { *properties* ***} as {*** *mixfix* ***}***

où :

- *actionPreconditions* sont les conditions nécessaires à satisfaire avant d'appliquer l'action de raffinement,
- *actionPostconditions* sont les conditions à satisfaire après l'application de l'action de raffinement,
- *refinementActions* sont les actions de raffinement à appliquer pour exécuter une transformation depuis l'*architectureDefId* vers une nouvelle architecture raffinée,
- *properties* sont les conditions que l'on suppose vraies et qui doivent être vérifiées (des obligations de preuve) afin d'appliquer les actions de raffinement,
- *mixfix* définit la notation d'une action dans une forme mixfix

Notons que *actionPreconditions, actionPostconditions et properties* sont des expressions logiques.

4.2. Différentes actions de base pour le raffinement

Nous présentons dans cette section les différentes actions de base fournies par *ArchWare ARL* pour le raffinement de modèles d'architecture, de composant et de connecteurs.

Nous pouvons regrouper les opérations liées aux actions de raffinement en trois catégories :

Les opérations de raffinement similaires sur tous les éléments architecturaux :

- ***Add*** : ajoute un élément ou un sous-ensemble d'élément à un ensemble,
- ***Remove*** : supprime un élément ou un sous-ensemble d'éléments d'un ensemble,
- ***Replace*** : remplace un élément par un élément,
- ***Becomes*** : remplace un sous-ensemble d'éléments par un autre sous-ensemble d'éléments.

Les opérations de raffinements spécifiques aux composants et connecteurs :

- ***Explodes*** : explose (décompose) des composants (ou connecteurs) sous forme de plusieurs composants (ou connecteurs) dans une architecture, composant ou connecteur,
- ***Implodes*** : implose (compose) des composants (ou connecteurs) sous forme de plusieurs composants (ou connecteurs) dans une architecture, composant ou connecteur.

Les opérations de raffinement spécifiques aux connections :

- ***Unifies*** : unifie des connections ou des ports
- ***Separates***: sépare des connections ou des ports

Ces trois catégories regroupent la liste suivante des actions de base :

pour le raffinement de données :

- ajout d'une déclaration (ou un ensemble de déclarations) de type dans une architecture,
- suppression d'une déclaration (ou un ensemble de déclarations) de type dans une architecture,
- remplacement d'une déclaration de types dans une architecture,
- transformation des déclarations de types d'une architecture,

- ajout de déclarations de types à un composant,
- suppression de déclarations de types d'un composant,
- remplacement de déclarations de types d'un composant,
- transformation de déclarations de types d'un composant,

- ajout de déclarations de types à un connecteur,
- suppression de déclarations de types d'un connecteur,
- remplacement de déclarations de types d'un connecteur,
- transformation de déclarations de types d'un connecteur

pour le raffinement de ports :

- ajout de ports dans une architecture,
- suppression de ports dans une architecture,
- remplacement de ports dans une architecture,
- transformation de ports dans une architecture,

- ajout de ports dans un composant,
- suppression de ports d'un composant,
- remplacement de ports d'un composant,
- transformation de ports d'un composant,

- ajout de ports dans un connecteur,
- suppression de ports d'un connecteur,
- remplacement de ports d'un connecteur,
- transformation de ports d'un connecteur,

- ajout et suppression de connections de sortie dans une architecture,
- ajout et suppression des connections d'entrée dans une architecture,
- remplacement de connections d'entrée et de sortie d'une architecture,
- transformation de connections de sortie et d'entrée d'une architecture,

- ajout et suppression de connections de sortie d'un connecteur,
- ajout et suppression de connections d'entrée d'un connecteur,
- remplacement de connections d'entrée et de sortie dans un connecteur,
- transformation de connections d'entrée et de sortie dans un connecteur,

- ajout et suppression de connections de sortie d'un composant,
- ajout et suppression de connections d'entrée d'un composant,
- remplacement de connections d'entrée et de sortie dans un composant,
- transformation de connections d'entrée et de sortie dans un composant,

pour le raffinement de comportements :

- transformation du comportement d'une architecture,
- raffinement du comportement d'un composant,

- raffinement du comportement d'un connecteur,
- transformation du comportement d'un composant dans une architecture,
- transformation du comportement d'un connecteur dans une architecture,

pour le raffinement de structures :

- ajout et suppression de composants dans une architecture,
- remplacement de composants dans une architecture,
- explosion et implosion des composants dans une architecture,
- ajout et suppression de connecteurs dans une architecture,
- explosion ou implosion de connecteurs dans une architecture,

- unification et séparation de connections dans une architecture,
- unification et séparation de connections internes et externes.

Ces actions, appliquées étape par étape, permettent le raffinement incrémental d'une architecture. Comme nous l'avons indiqué auparavant, les notions de composants et connecteurs sont méthodologiques. Pour éviter des répétitions, nous présenterons dans ce qui suit uniquement les actions de raffinement d'architectures. Nous montrerons également comment l'on peut appliquer ces actions en partant d'une architecture vide, c'est-à-dire complètement non spécifiée (aucune déclaration de type, ni port, ni comportement). Cette architecture peut être décrite dans ArchWare ARL de la façon suivante :

```
archetype architectureDefId is architecture {
  types is { deferred }
  ports is { deferred }
  behaviour is { deferred }
}
```

ou bien par convention syntaxique comme suit :

```
archetype architectureDefId is architecture { deferred }
```

avec *deferred* un mot réservé signifiant que la spécification des éléments concernés est différée.

4.2.1. Actions pour l'ajout et la suppression de déclarations de types dans une architecture

Les déclarations de types peuvent être ajoutées à une architecture si les noms des types présentés sont nouveaux. L'action pour l'ajout d'une déclaration de type à une architecture est définie de la manière suivante :

```
on a : architecture action addTypeArchRef is refinement (t : type ) {
  pre is { a::types excludes? t }
  post is { a::types includes? t }
  } as { a::types includes t }
```

où le prédicat "*c includes? e*" est évalué à vrai si *e* appartient à *c* sinon à faux. Le prédicat "*c excludes? e*" est évalué à vrai si *e* n'appartient pas à *c* sinon à faux. La présentation des nouvelles déclarations de types ne change pas le comportement de l'architecture. Pour appliquer ou exprimer qu'une action de raffinement est un « ajout de déclarations de types », la notation suivante est utilisée. Plusieurs types peuvent être ajoutés à la fois.

archetype *architectureRefId* ***refines*** *architectureDefId*
using { types includes t_0 *....* ***and types includes*** t_n ***}***

Des déclarations de types d'une architecture seront enlevées si elles ne sont pas utilisées dans le comportement de l'architecture. De ce fait, le comportement de l'architecture n'est pas modifié. Cette action est définie par :

```
on a : architecture action removeTypeArchRef is refinement (
  t : type ) {
  pre is { a::types includes? t }
  post is { a::types excludes? t }
  } as { a::types excludes t }
```

Pour appliquer ou exprimer qu'une action de raffinement est une « suppression de déclarations de types », la notation suivante est utilisée, plusieurs types pouvant être supprimés à la fois :

archetype *architectureRefId* ***refines*** *architectureDefId*
using { types excludes t_0 *....* ***and types excludes*** t_n ***}***

4.2.2. Action pour le remplacement d'une déclaration de type dans une architecture

Des déclarations de types d'une architecture seront remplacées si les noms des types présentés sont de nouveaux noms. Le remplacement des déclarations de types ne changera pas le comportement de l'architecture si les types remplacés sont équivalents (sémantiquement, c'est équivalent à un renommage) :

```
on a : architecture action replaceTypeArchRef is refinement (
  et : type, nt : type ) {
        pre is { a::types includes? et and a::types excludes? nt }
        post is { a::types excludes? et and a::types includes? nt }
  } as { a::types replaces et by nt }
```

Pour appliquer ou exprimer qu'une action de raffinement est un « remplacement de déclaration de type », la notation suivante est utilisée :

archetype *architectureRefId* ***refines*** *architectureDefId*
using { types replaces t_0 ***by*** nt_0 ***}***

4.2.3. Action pour la transformation de déclarations de types d'une architecture

Une action de transformation de déclarations de types est définie de la manière suivante :

```
on a : architecture action transformTypeArchRef is refinement (
                                        et : set[type], rt : set[type] ) {
          pre is { a::types includes? et and a::types excludes? rt }
          post is { a::types excludes? et and a::types includes? rt }
  } as { a::types where { et } becomes { rt }
```

Plusieurs types sont susceptibles d'être transformés à la fois. Ces transformations doivent préserver toutes les propriétés de l'architecture la plus abstraite. Afin de permettre la vérification de telles propriétés sur le type remplaçant, des fonctions de correspondance (mapping) entre le nouveau type et l'ancien seront définies dans la partie hypothèses (*assuming*).

```
on a : architecture action transformTypeArchRef is refinement (
  et : set[type], rt : set[type] ) {
```

pre is { a::types includes? et and a::types excludes? rt }
post is { a::types excludes? et and a::types includes? rt }
} as { a::types where { et } becomes { rt }
assuming { --** *mapping functions: mapping(rt)=et* **}
}

Pour appliquer ou exprimer qu'une action de raffinement est une « transformation de déclarations de types, la notation suivante est utilisée :

***archetype** architectureRefId **refines** architectureDefId*
using { types where {**et_{00}, ... et_{0n}**} by {**rt_{00}, ... rt_{0n} **} }

4.2.4. Actions pour l'ajout et la suppression de ports dans une architecture

Un port peut être ajouté à une architecture en tant que nouveau port s'il est présenté par un nouveau nom. Cette action est définie de la manière suivante :

on** a **: architecture action** addPortArchRef **is refinement (**p **: port) {
pre is { a::ports excludes? p }
post is { a::ports includes? p }
} as { a::ports includes** p **}

Le comportement d'une architecture n'est pas modifié dans ce cas. Pour exprimer qu'une action de raffinement est un « ajout de ports », plusieurs ports pouvant être ajoutés à la fois, la notation suivante est utilisée :

***archetype** architectureRefId **refines** architectureDefId*
using { ports includes** p_0 .... **and ports includes** p_n **}

Un port dans une architecture pourrait être supprimé s'il n'est pas utilisé dans le comportement d'une architecture. Ainsi, le comportement reste inchangé :

on** a **: architecture action** removePortArchRef **is refinement (**p **: port) {
pre is { a::ports includes? p and a::behaviour::ports excludes? p }
post is { a::ports excludes? p }
} as { a::ports excludes** p **}

Pour exprimer qu'une action de raffinement est une « suppression de ports », la notation suivante est utilisée, plusieurs ports pouvant être supprimés à la fois :

***archetype** architectureRefId **refines** architectureDefId*
using { ports excludes** p_0 .... **and ports excludes** p_n **}

L'ajout et la suppression de ports sont des opérations complémentaires. Les actions suivantes permettent la gestion des connections dans la portée des ports d'une architecture.

4.2.5. Actions pour le remplacement de ports d'une architecture

Un port d'architecture pourrait être remplacé par un nouveau port si ce dernier est de type équivalent au premier. Le remplacement d'un port revient à vérifier l'équivalence entre les types, en supprimant le port existant et en ajoutant le nouveau port de manière atomique. Le port est également remplacé dans l'expression de comportement. L'ensemble des connections d'un port remplacé doit être un sous-ensemble de l'ensemble des connections du nouveau port. L'action de remplacement est définie comme suit :

on** a **: architecture action** replacePortArchRef **is refinement (**p **: port,** np **: port) {

```
        pre is { a::ports includes? p and a::ports excludes? np and np includes? p }
        post is { a::ports excludes? p and a::ports includes? np }
    } as { a::ports replaces p by np }
```

Le comportement d'une architecture n'est pas modifié si le port est remplacé par un port équivalent. Sémantiquement, en renommant un port et en lui ajoutant de nouvelles connections non utilisées, l'ancienne définition et la nouvelle sont équivalentes.

Pour appliquer ou exprimer qu'une action de raffinement est un « remplacement de ports », la notation suivante est utilisée :

```
archetype architectureRefId refines architectureDefId
using { ports replaces p0 by np0 }
```

4.2.6. Actions pour la transformation de ports d'une architecture

L'action de transformation de ports est définie de la manière suivante :

```
on a : architecture action transformPortArchRef is refinement (
                                        ep : set[port], rp : set[port] ) {
        pre is { a::ports includes? ep and a::ports excludes? rp }
        post is { a::ports excludes? ep and a::ports includes? rp }
    } as { a::ports where { ep } becomes { rp }
```

Plusieurs ports peuvent être transformés à la fois. Ces transformations doivent préserver toutes les propriétés de l'architecture la plus abstraite. Pour cela de même que pour les types, des fonctions de correspondance seront définies dans la partie hypothèses afin de permettre la vérification de ces propriétés :

```
on a : architecture action transformPortArchRef is refinement (
                                            ep : set[port], rp : set[port] ) {
        pre is { a::ports includes? ep and a::ports excludes? rp }
        post is { a::ports excludes? ep and a::ports includes? rp }
    } as { a::ports where { ep } becomes { rp }
          assuming { -- mapping functions: mapping(rp)=ep }
    }
```

Pour exprimer qu'une action de raffinement est une « transformation de ports », la notation suivante est utilisée :

```
archetype architectureRefId refines architectureDefId
using { ports where {pt00, ....pt0n} by {rp00, .... rp0n } }
```

4.2.7. Actions pour l'ajout et la suppression de connections de sortie dans une architecture

Une connection libre pourrait être ajoutée comme nouvelle connection de sortie à un port si elle est définie en tant que connection avec un nouveau nom :

```
on a : architecture action addOutputConnectionArchRef is refinement (
                                        p : port, oc : connection ) {
        pre is { a::p::outgoing excludes? oc }
        post is { a::p::outgoing includes? oc }
```

} as { a::p::outgoing includes** oc **}

Le comportement d'une architecture demeure alors inchangé. Pour exprimer qu'une action de raffinement est un « ajout de connections de sortie dans une architecture », la notation suivante est utilisée, plusieurs connections peuvent être ajoutées à la fois :

***archetype** architectureRefId **refines** architectureDefId*
using {** p_0 **:: outgoing includes** OC_0 .... **and** p_n**:: outgoing includes** OC_n **}

Une connection de sortie pourrait être supprimée d'une architecture si elle n'est pas utilisée dans son comportement. Ce dernier ne subit alors aucun changement :

on** a **: architecture action** removeOutputConnectionArchRef **is refinement (
*p **: port,** oc **: connection) {***
pre is { a::p::outgoing includes? oc and
a::behaviour::connections excludes? p::oc }
post is { a::p::outgoing excludes? oc }
} as { a::p::outgoing excludes** oc **}

Pour exprimer qu'une action de raffinement est une « suppression de connections de sortie d'une architecture », la notation suivante est utilisée, plusieurs connections pouvant être supprimées à la fois :

***archetype** architectureRefId **refines** architectureDefId*
using {** p_0 **:: outgoing excludes** OC_0 .... **and** p_n**:: outgoing excludes** OC_n **}

L'ajout et la suppression des connections de sortie sont des opérations complémentaires, préservant le comportement d'une architecture.

4.2.8. Actions pour l'ajout et la suppression de connections d'entrée d'une Architecture

Comme pour les connections de sorties, les connections d'entrée peuvent aussi être ajoutées ou supprimées dans une architecture.
Une connection libre pourrait s'ajouter comme une nouvelle connection d'entrée à un port si elle est définie avec un nouveau nom :

on** a **: architecture action** addInputConnectionArchRef **is refinement (
*p **: port,** ic **: connection) {***
pre is { a::p::incoming excludes? ic }
post is { a::p::incoming includes? ic }
} as { a::p::incoming includes** ic **}

Le comportement d'une architecture reste alors inchangé. Pour exprimer qu'une action de raffinement est un « ajout de connections d'entrée d'une architecture », la notation suivante est utilisée, plusieurs connections pouvant être ajoutées à la fois :

***archetype** architectureRefId **refines** architectureDefId*
using {** p_0 **:: incoming includes** IC_0 .... **and** p_n**:: incoming includes** IC_n **}

Une connection d'entrée pourrait être supprimée d'une architecture si elle n'est pas utilisée dans le comportement de l'architecture. De ce fait, le comportement n'est pas modifié :

on** a **: architecture action** removeInputConnectionArchRef **is refinement (
*p **: port,** ic **: connection) {***

pre is { a::p::incoming includes? ic and
a::behaviour::connections excludes? p::ic }
post is { a::p::incoming excludes? ic }
} as { a::p::incoming excludes ic }

Pour exprimer qu'une action de raffinement est une « suppression de connections d'entrée d'une architecture », la notation suivante est utilisée, plusieurs connections peuvent être supprimées à la fois :

***archetype** architectureRefId **refines** architectureDefId*
using { p0 :: incoming excludes IC0 …. and pn:: incomingexcludes ICn }

Ainsi, l'ajout et la suppression des connections d'entrées sont des opérations complémentaires, préservant le comportement d'une architecture. Les actions pour l'ajout et la suppression de connections d'entrée et de sortie de ports sont des actions de base pour exécuter des raffinements incrémentaux d'architecture, par exemple, l'on ajoute dans un premier temps une nouvelle connection de sortie à une architecture et on raffine par la suite son comportement pour utiliser cette nouvelle connection en vue de fournir un résultat plus contraint.

4.2.9. Actions pour le remplacement de connections d'entrée et de sortie dans une architecture

Les connections de sortie et d'entrée d'une architecture peuvent être remplacées. Une connection libre peut être remplacée par une nouvelle connection si la nouvelle connection est d'un type équivalent à celui de la première et a un nouveau nom, inconnu de l'architecture initiale. Un remplacement de connection consiste à vérifier des équivalences de types, en supprimant une connection existante et en ajoutant une nouvelle connection de manière atomique. Il faudrait noter que la connection est également remplacée dans l'expression du comportement.

Pour les connections de sortie, le remplacement est défini comme suit :

on a : architecture action replaceOutputConnectionArchRef is refinement (
p : port, oc : connection, nc : connection) {
pre is { a::p::outgoing includes? oc and a::p::outgoing excludes? nc }
post is { a::p::outgoing excludes? oc and a::p::outgoing includes? nc }
} as { a::p::outgoing replaces oc by nc }

Pour les connections d'entrée, le remplacement est défini de manière analogue comme suit :

on a : architecture action replaceInputConnectionArchRef is refinement (
p : port, ic : connection, nc : connection) {
pre is { a::p::incoming includes? ic and a::p::incoming excludes? nc }
post is { a::p::incoming excludes? ic and a::p::incoming includes? nc }
} as { a::p::incoming replaces ic by nc }

Le comportement d'une architecture demeure alors inchangé car la connection est remplacée par une connection équivalente. Sémantiquement, elle est équivalente par renommage. Par convention syntaxique, s'il y a seulement un port dans une architecture, le nom du port sera optionnel (facultatif) dans la désignation des connections. De même, si aucun port n'est explicitement créé dans une architecture alors un port anonyme sera implicitement créé.

4.2.10. Actions pour la transformation de connections d'entrée/sortie dans une architecture

L'action de transformation de connections d'entrée/sortie dans une architecture est définie comme suit :

```
on a : architecture action transformConnectionArchRef is refinement (
  ec : set[connection], rc : set[connection] ) {
        pre is { a::connections includes? ec and a::connections excludes? rc }
       post is { a::connections excludes? ec and a::connections includes? rc }
} as { a::connections where { ec } becomes { rc }
```

Plusieurs connections peuvent être transformées à la fois. Ces transformations doivent préserver toutes les propriétés de l'architecture la plus abstraite. La désignation des connections a besoin de la désignation des ports, lorsqu'il y a plus d'un port dans l'architecture.

Des fonctions de correspondance seront définies afin de permettre la vérification des propriétés.

```
on a : architecture action transformConnectionArchRef is refinement (
                ec: set[connection], rc: set[connection] ) {
        pre is { a::connections includes? ec and a::connections excludes? rc }
        post is { a::connections excludes? ec and a::connections includes? rc }
} as { a::connections  where { ec } becomes { rc }
                        assuming { -- mapping functions: mapping(rc)=ec }
}
```

4.2.11. Actions de transformation du comportement d'une architecture

Le raffinement d'une architecture peut s'opérer en transformant son comportement. Soit *a* une architecture. Si le comportement de *a* est raffiné pour être *ba* (ceci est traduit par l'expression, *a* est une abstraction de *ba*), le résultat sera un raffinement extérieurement visible de *a*. Le prédicat "*a abstracts? ba*" est évalué à vrai si *ba* est un raffinement de comportement de *a*. Le prédicat "*is?*" sera évalué à vrai si les deux arguments sont les mêmes. Cette action est définie comme suit :

```
on a : architecture action refineBehaviourArchRef is refinement (
                                        ba : abstraction ) {
   pre is { a::behaviour abstracts? ba }
   post is { a::behaviour is? ba }
} as { a::behaviour becomes ba }
```

Dans certains cas, afin de vérifier que le comportement *ba* est un raffinement de comportement, quelques hypothèses sur les valeurs reçues via des connections d'entrée sont nécessaires. L'action de raffinement sera utilisée en conséquence avec des hypothèses :

```
on a : architecture action refineBehaviourArchRef is refinement (
                                    ba : abstraction ) {
        pre is { a::behaviour abstracts? ba }
        post is { a::behaviour is? ba }
}
        as { a::behaviour becomes ba
assuming { -- assumption on free input connections }
}
```

Contrairement aux actions de raffinement d'architectures précédemment présentées qui laissent inchangé le comportement, cette action permet d'opérer un véritable raffinement du comportement.

4.2.12. Actions pour la transformation d'un comportement de composant dans une Architecture

Une architecture peut être raffinée par la transformation d'un comportement de ses composants. Soit *c* un composant d'une architecture *a*. Si le comportement de *c* est raffiné par *bc*, le résultat est un raffinement extérieurement visible depuis une architecture *a*. Cette action a la définition suivante :

```
on a : architecture action behaviourArchRef is refinement (
                    c : component, bc : abstraction ) {
        pre is { a::components includes? c and c::behaviour abstracts? bc }
        post is { c::behaviour is? bc }
  } as { c::behaviour becomes bc }
```

Dans certains cas, afin de vérifier que le comportement *bc* est un raffinement du comportement de *c* certaines hypothèses sur les valeurs reçues via les connections d'entrée sont nécessaires. Dans ce cas également, l'action de raffinement est utilisée avec des hypothèses :

```
on a : architecture action behaviourArchRef is refinement (
                                c : component, bc : abstraction ) {
        pre is { a::components includes? c and c::behaviour abstracts? bc }
        post is { c::behaviour is? bc }
  } as { c::behaviour   becomes bc
                        assuming { -- assumption on free input connections }
  }
```

Cette action conduit également à un véritable raffinement du comportement.

4.2.13. Actions pour l'ajout et la suppression de composants dans une architecture

Un composant pourra être ajouté à une architecture sans le changement de son comportement s'il n'y a aucune unification (ou attachement) de ses connections avec les connections des autres composants de l'architecture ou avec les connections de l'architecture elle-même :

```
on a : architecture action addComponentArchRef is refinement (
                                        c : component ) {
        pre is { a::components excludes? c }
        post is { a::components includes? c }
  } as { a::components includes c }
```

Le comportement d'une architecture n'est pas modifié puisque les composants ajoutés ne sont pas en interaction avec les autres composants de l'architecture ou de l'environnement de l'architecture. Par exemple, afin de raffiner une architecture, on peut ajouter un nouveau composant sans connections, puis par la suite ajouter des connections d'entrée et de sortie et enfin raffiner le comportement du composant ajouté en utilisant les actions de raffinement précédemment définies.

De la même façon, des composants peuvent être supprimés. En effet, un composant pourrait être supprimé d'une architecture, sans que le comportement de celle-ci ne change, s'il n'a pas de connections unifiées (attachées) avec les connections d'autres composants de l'architecture ou avec les connections de l'architecture elle-même :

```
on a : architecture action removeComponentArchRef is refinement (
  c : component ) {
        pre is { a::components includes? c and c::connections is? set() }
        post is { a::components excludes? c }
  } as { a::components excludes c }
```

Ainsi, l'ajout ou la suppression des composants sont des opérations complémentaires qui préservent le comportement d'une architecture puisque les composants ajoutés ou supprimés ne sont pas attachés.

4.2.14. Actions pour le remplacement de composants dans une Architecture

Un composant pourrait être remplacé dans une architecture sans impact sur le comportement de celle-ci s'il n'a pas de connections unifiées avec les connections des autres composants ou avec les connections de l'architecture elle-même :

```
on a : architecture action replaceComplementArchRef is refinement (
  c : component, nc : component ) {
        pre is { a::components includes? c and a::components excludes? nc }
        post is { a::components excludes? c and a::components includes? nc }
  } as { a::components replaces c by nc }
```

Le comportement de l'architecture reste inchangé puisque le composant remplacé n'est pas en interaction avec les autres composants de l'architecture ou avec l'environnement de celle-ci.

4.2.15. Actions pour l'explosion et l'implosion des composants dans une architecture

En utilisant les actions précédentes, un architecte peut décrire une architecture de façon hiérarchique, c'est-à-dire en décrivant des architectures où les composants et les connecteurs peuvent être eux-mêmes décrits comme étant des architectures. Ainsi, un composant atomique peut être remplacé par un composant composite, décrit en tant qu'architecture, en utilisant le remplacement.
Une fois l'architecture décrite, un architecte peut exploser (éclater) un composant qui est décrit lui-même par une architecture, ceci en utilisant l'action définie ci-après :

```
on a : architecture action explodeComponentArchRef is refinement (
  c : component ) {
        pre is { a::components includes? c }
        post is { a::components excludes? c  and
           a::components includes? c::components and
           a::connectors includes? c::connectors }
  } as { a::components explodes c }
```

L'explosion d'un composant composite *c* dans une architecture *a* produit une nouvelle architecture dont tous les composants et connecteurs sont inclus dans l'architecture *a* mais *c* lui-même en est exclu.

Il est utile de noter que cette action change la portée des composants et des connecteurs du moment que les composants et les connecteurs de *c* deviennent des composants et connecteurs de *a*. Afin d'interdire le changement du comportement de *a*, tous les noms restreints de *c* doivent être différents des noms de *a*. Si nécessaire, un renommage peut être effectué en utilisant la clause : ***where {*** *newName* ***renames*** *currentName* ***}***

L'action complémentaire de l'explosion d'un composant est l'implosion d'un ensemble de composants et connecteurs en une sous architecture. La sous-architecture est incorporée comme nouveau composant à l'aide de l'action suivante :

```
on a : architecture action implodeComponentArchRef is refinement (
  sc : set[constituent], nc : component ) {
        pre is { a::constituents includes? sc and
           sc::outgoing includes? nc::outgoing and
           sc::incoming includes? nc::incoming }
        post is {  a::components includes? nc and
            nc::behaviour is? sc and
            a::constituents excludes? sc }
```

} as { a::components implodes { sc } as nc }

L'implosion d'un ensemble de composants et connecteurs en tant que sous-architecture produit un nouveau composant composite avec un comportement présenté par une composition d'un ensemble de composants/connecteurs. Les composants et les connecteurs qui ont été implosés sont exclus de l'architecture et deviennent des composants et des connecteurs de la sous-architecture du composant composite.

De façon similaire, les actions de base « ajout, suppression, remplacement et transformation » d'un connecteur dans une architecture sont les mêmes que celles pour les composants dans une architecture sauf qu'elles portent sur les connecteurs.

4.2.16. Actions pour l'unification et la séparation de connections dans une architecture

Les connections d'entrée et de sortie de composants et connecteurs d'une architecture peuvent être unifiées (ou attachées). Une connection libre peut être unifiée (ou attachée) avec une connection libre si les deux sont de types équivalents et si elles jouent des rôles complémentaires d'entrée/sortie. Le prédicat "*unifies? with*" sera évalué à vrai si les connections sont unifiées (attachées) ou sont les mêmes.
L'unification est définie comme suit :

```
on a : architecture action unifyConnectionArchRef is refinement (
  cs : constituent, p : port, oc : connection,
  cd : constituent, q : port, ic : connection ) {
        pre is { a::constituents includes? cs and a::constituents includes? cd and
          cs::p::outgoing includes? oc and cd::q::incoming includes? ic }
        post is { a::connections unifies? cs::p::oc with cd::q::ic }
  } as { a::connections unifies cs::p::oc with cd::q::ic }
```

De manière duale, la séparation de connections est définie comme suit :

```
on a : architecture action separateConnectionArchRef is refinement (
  cs : constituent, p : port, oc : connection,
  cd : constituent, q : port, ic : connection ) {
  pre is { a::constituents includes? cs and a::constituents includes? cd and
        cs::p::outgoing includes? oc and cd::q::incoming includes? ic }
  post is { a::connections separates? cs::p::oc from cd::q::ic }
} as { a::connections separates cs::p::oc from cd::q::ic }
```

Dans ces deux derniers cas, le comportement de l'architecture modifiée doit être un raffinement du comportement de l'architecture avant modification.

4.2.17. Actions pour l'unification et la séparation de connections externes et internes

Une connection externe d'une architecture peut être unifiée (attachée) avec une connection interne d'un composant ou connecteur de cette architecture. Une connection libre pourra être unifiée (attachée) avec une autre connection libre si elles sont de types équivalents et elles ont le même mode d'entrée/sortie.

Une unification est définie comme suit :

```
on a : architecture action unifyExternalConnectionArchRef is refinement (
  ep : port, ec : connection, c : constituent, ip : port, ic : connection ) {
        pre is {  a::constituents includes c and
          ( a::ep::outgoing includes ec and c::ip::outgoing includes ic ) xor
          ( a::ep::incoming includes ec and c::ip::incoming includes ic ) }
        post is { a::connections unifies? a::ep::ec with c::ip::ic }
  } as { a::connections unifies a::ep::ec with c::ip::ic }
```

Une séparation est exprimée par :

```
on a : architecture action separateExternalConnectionArchRef is refinement (
```

ep **: port,** *ec* **: connection,** *c* **: constituent,** *ip* **: port,** *ic* **: connection) {**
pre is { *a*::**constituents includes?** *c* **and**
(*a*::*ep*::**outgoing includes?** *ec* **and** *c*::*ip*::**outgoing includes?** *ic* **) xor**
(*a*::*ep*::**incoming includes?** *ec* **and** *c*::*ip*::**incoming includes?** *ic* **) }**
post is { *a*::**connections separates?** *a*::*ep*::*ec* **from** *c*::*ip*::*ic* **}**
} as { *a*::**connections separates** *a*::*ep*::*ec* **from** *c*::*ip*::*ic* **}**

Dans ces deux cas, le comportement de l'architecture modifiée doit être un raffinement du comportement de l'architecture avant modification.

5. Etude de cas : DAS (Data Acquisition System)

Cette section présente une étude de cas pour illustrer l'utilisation de ArchWare ARL dans la définition et le raffinement d'une architecture indépendamment de la plateforme d'implémentation. L'étude de cas détaillée ci-après porte sur la modélisation et le raffinement d'un système d'acquisition de données (DAS). Le processus de raffinement consistera dans un premier temps à établir à partir d'une spécification initiale, une architecture abstraite de l'application, puis, d'autres étapes suivront qui permettront de raffiner de manière incrémentale cette architecture pour enfin obtenir une architecture concrète.

5.1. Modélisation du système DAS

Le DAS peut être modélisé de manière abstraite comme une architecture (cf. Fig. III.1) qui reçoit des données via une connection *in*. Les valeurs reçues via *in* se composent de paires constituées chacune d'une clef et de données à stocker sous cette clef. Parallèlement à cela, le système répond aux demandes de données correspondant à une certaine clef reçue via la connection *key*, par l'envoi de(s) la donnée(s) stockée(s) sous cette clef via la connection *data*.

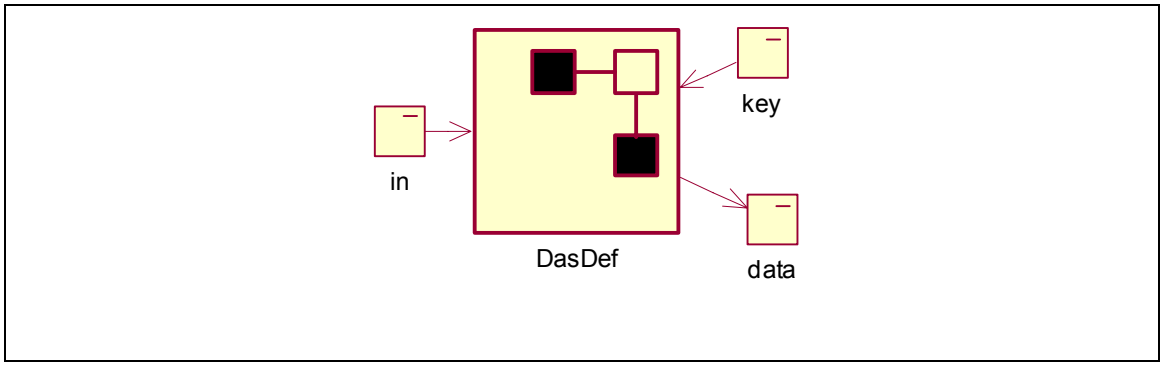


Figure III.1. : Modèle abstrait DAS

En utilisant *ArchWare ARL*, l'architecte pourrait spécifier le premier modèle abstrait de DAS comme suit :

```
archetype DasDef is architecture {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { in is connection(Entry). key is connection(Key) }
  outgoing is { data is connection(Data) }
  behaviour is { deferred }
}
```

A ce niveau d'abstraction, le comportement n'est pas encore spécifié « deferred ». *Key* est l'ensemble de toutes les clefs possibles pour la base de données et *Data* est l'ensemble de toutes les valeurs possibles de données. *Entry* est le type *tuple[Key, Data],* i.e. l'ensemble de toutes les entrées possibles pour la base de données. Aucun port n'a été explicitement déclaré à ce niveau.

5.2. Etapes de raffinement pour obtenir une architecture abstraite

L'architecte pourrait raffiner le modèle précédent pour obtenir une première architecture abstraite de DAS. La procédure de raffinement à ce niveau serait de partitionner le modèle en termes de composants et de connecteurs. La figure (Fig. III.2.) suivante illustre cette architecture abstraite.

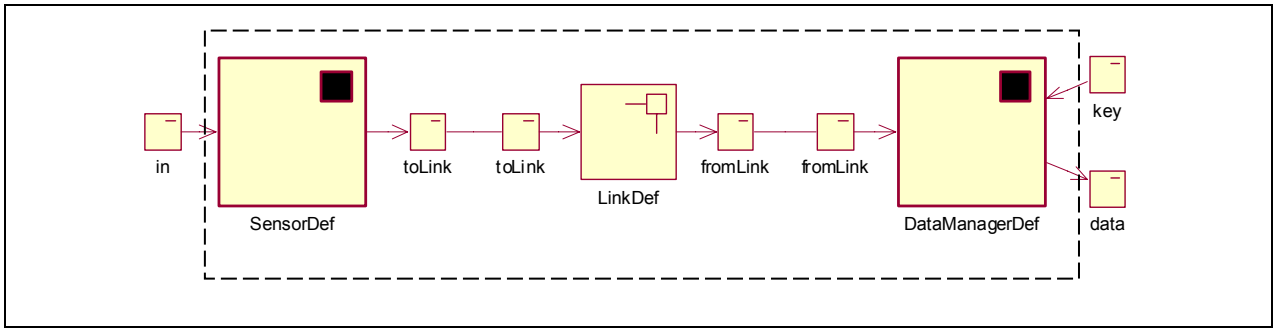


Figure III.2. : Définition d'une architecture abstraite pour le DAS

Après la partition, l'architecture comporte deux composants : *SensorDef* et *DataManagerDef*. Les données brutes de l'environnement subissent d'abord quelques transformations dans *SensorDef* et sont ensuite expédiées via le connecteur *LinkDef* à *DataMangerDef* qui les stocke dans sa base de données interne.

Les deux composants *SensorDef* et *DataManagerDef* ainsi que le connecteur *LinkDef* sont décrits dans *ArchWare ARL* de la façon suivante :

```
archetype SensorDef is component {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { in is connection(Entry) }
  outgoing is { toLink is connection(Entry) }
  behaviour is { deferred }
}

archetype DataManagerDef is component {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { key is connection(Key). fromLink is connection(Entry) }
  outgoing is { data is connection(Data) }
  behaviour is { deferred }
}

archetype LinkDef is connector {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { toLink is connection(Entry) }
  outgoing is { fromLink is connection(Entry) }
  behaviour is { deferred }
}
```

Dans cette spécification, seuls les aspects structurels sont modélisés. Les comportements restent non spécifiés, c'est-à-dire que leur description est différée (deferred). En utilisant *SensorDef*, *LinkDef* et *DataManagerDef*, le modèle *DasDef* précédent peut être raffiné en définissant des aspects structurels de son architecture abstraite dans *ArchWare ARL* :

```
archetype DasArchDef refines DasDef using {
  behaviour is compose { SensorDef and LinkDef and DataManagerDef }
}
```

Cette description structurelle de l'architecture abstraite peut être encore raffinée pour aboutir à un modèle de l'architecture abstraite qui englobe des aspects structuraux et comportementaux. Elle peut être décrite dans *ArchWare ARL* comme suit :

```
archetype DasArchAbs refines DasArchDef using {
  SensorDef::behaviour becomes abstraction() {
      process is function(d: Data) : Data { deferred }.
      replicate   via in receive entry.
                  via toLink send tuple(entry::key, process(entry::data))
  }.
  DataManagerDef::behaviour becomes abstraction() {
      database is location(Set() : Entry).
      replicate choose {
                        via fromLink receive entry.
                        database := database' including(entry)
                  or    via key receive queryKey : Key.
                        via data send (database' selecting(d | d::key == queryKey)
                  }
  }.
  LinkDef::behaviour becomes abstraction() {
      replicate   via toLink receive entry. via fromLink send entry
  }
}
```

La fonction *process : Data → Data* effectue le traitement sur les données dans *SensorDef*. La base de données est modélisée comme un emplacement qui stocke un ensemble de tuples de type *Entry*. En donnant une clef, une valeur de donnée sera recherchée. S'il n'y a pas encore de donnée stockée sous la clef, la base de données retourne une valeur nulle.

5.3. Etapes de raffinement pour obtenir une architecture plus concrète

L'architecture abstraite, précédemment définie est raffinée pour obtenir une architecture plus concrète. Ceci peut être réalisé en réduisant le temps de transmission pour les entrées : pour chaque entrée, *SensorDef* compressera les données pour les transmettre et *DataManagerDef* décompressera ces dernières pour les stocker. Les données compressées sont sensées être moins importantes en taille que les données elles-mêmes et le gain au niveau du temps de transmission compense le temps de compression et décompression des données.

A ce niveau, l'architecte n'est pas intéressé par les aspects algorithmiques pour le calcul de la compression de données. Il considère que les données compressées sont elles-mêmes un élément de *Data,* et qu'il y a une fonction *compress : Data → Data* qui compresse les données. Une autre fonction *uncompress : Data → Data* décompresse les données. Il suppose que pour toutes les données : *uncompress(compressed(data)) = data*.

Afin de raffiner l'architecture, plusieurs actions sont à effectuer : *SensorDef* peut être étendu par un compresseur. Pour chaque nouvelle entrée, les données liées à une clef sont compressées et expédiées. Le *DataManagerDef* pourrait être étendu par un décompresseur. Il lit l'entrée, décompresse les données reçues et stocke les données principales et relatives dans sa base de données.
A ce niveau, un raffinement architectural consisterait à introduire deux composants, *Compressor* et *Uncompressor* pour respectivement compresser et décompresser les données. Ce raffinement peut être effectué à l'aide des actions de raffinement définies dans *ArchWare ARL* :

1ère étape : ajout de composants
Comme première étape, l'architecte pourrait introduire deux nouveaux composants vides, *Compressor* et *Uncompressor*. Ils ne devraient pas être reliés à un autre composant de l'architecture La figure (Fig. III.3.) suivante illustre l'architecture après la première étape de raffinement.

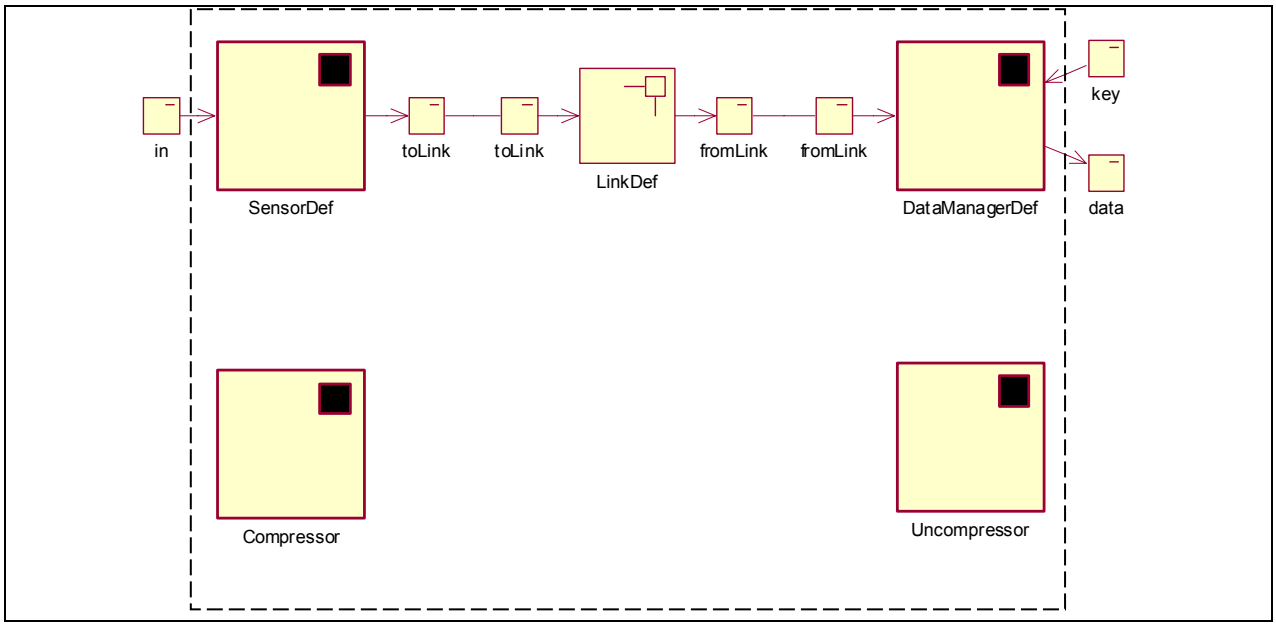


Figure III.3. : Ajout des nouveaux composants non-connectés

Dans *ArchWare ARL*, cette action de raffinement est décrite comme suit :

```
archetype DasRef1 refines DasArchAbs using {
  components includes { Compressor is component { deferred } }.
  components includes { Uncompressor is component { deferred } }
}
```

2ème étape: ajout des connections de sortie

L'architecte pourrait maintenant ajouter une connection de sortie *toCoLink* au *Compressor* et une connection de sortie *UDLink* au *Uncompressor*. Il peut également ajouter une connection de sortie *toCSLink* à *sensorDef*. La figure suivante (Fig. III.4.) illustre l'architecture après cette étape de raffinement.

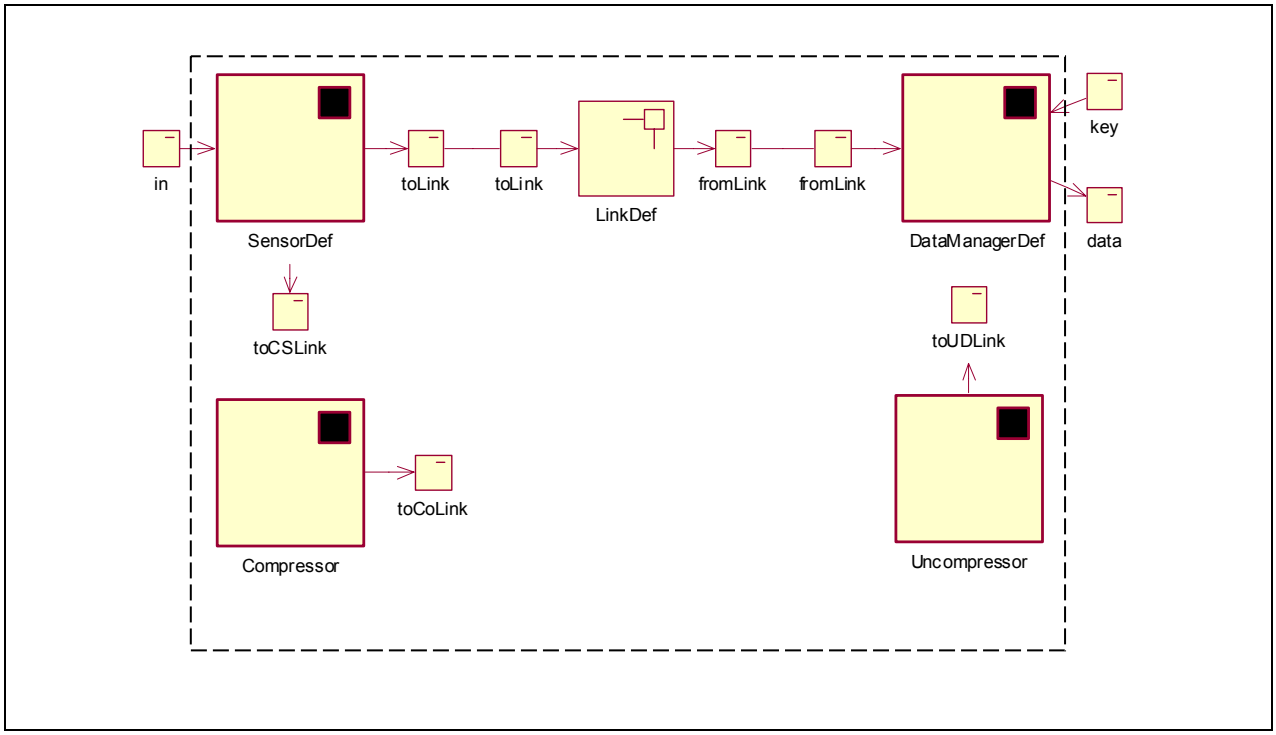


Figure III.4. : Ajout des connections de sortie à des composants

Dans *ArchWare ARL*, ce raffinement peut être exprimé de la façon suivante :

```
archetype DasRef2 refines DasRef1 using {
  Compressor::types includes
        { Key is Any. Data is Any. Entry is tuple[Key, Data] }.
  Uncompressor::types includes
        { Key is Any. Data is Any. Entry is tuple[Key, Data] }.
  SensorDef::types includes
        { Key is Any. Data is Any. Entry is tuple[Key, Data] }
}

archetype DasRef3 refines DasRef2 using {
  Compressor::outgoing includes { toCoLink is connection(Entry) }.
  Uncompressor::outgoing includes { toUDLink is connection(Entry) }.
  SensorDef::outgoing includes { toCSLink is connection(Entry) }
}
```

Puisque ces connections sont limitées à l'intérieur de l'architecture et ne sont pas reliées à des composants au moment de l'application de l'action, les conditions pour l'addition des connections de sortie sont satisfaites. Notons que le comportement des composants qui ont été ajoutés n'est pas du tout défini. Le comportement du système lui-même reste inchangé, puisque les nouvelles connections ne sont pas utilisées dans l'architecture.

3ème étape : ajout des connections d'entrées

L'architecte pourrait maintenant ajouter une connection d'entrée *fromCSLink* à *Compressor* et une connections d'entrée *fromCoLin* à *Uncompressor*. Il peut également ajouter une connection *fromUDLink* à *DataManagerDef*. La figure (Fig. III.5.) suivante illustre l'architecture après cette nouvelle étape de raffinement.

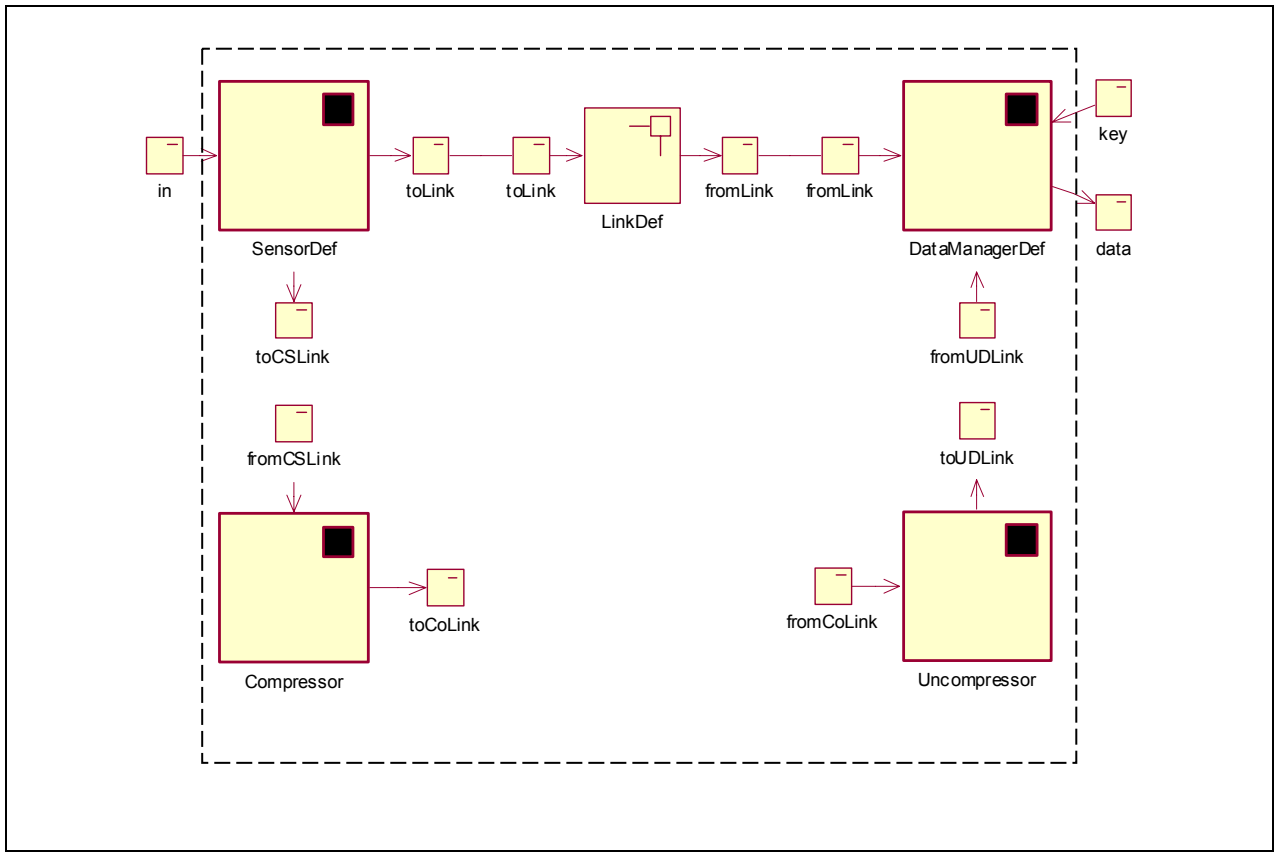


Figure III.5. : Ajout des connections d'entrée à des composants

Dans *ArchWare ARL*, ce raffinement peut être exprimé comme suit :

```
archetype DasRef4 refines DasRef3 using {
  Compressor::incoming includes { fromCSLink is connection(Entry) }
  Uncompressor::incoming includes { fromCoLink is connection(Entry) }
  DataManagerDef::incoming includes { fromUDLink is connection(Entry) }
}
```

4ème étape : ajouter et connecter les connecteurs

Comme le montre la figure Fig. III.6., l'architecte peut maintenant ajouter des connecteurs : *CoLink* entre *Compressor* et *Uncompressor*, *CSLink* entre *Compressor* et *SensorDef* et *DDLink* entre *Uncompressor* et *DataManagerDef*.

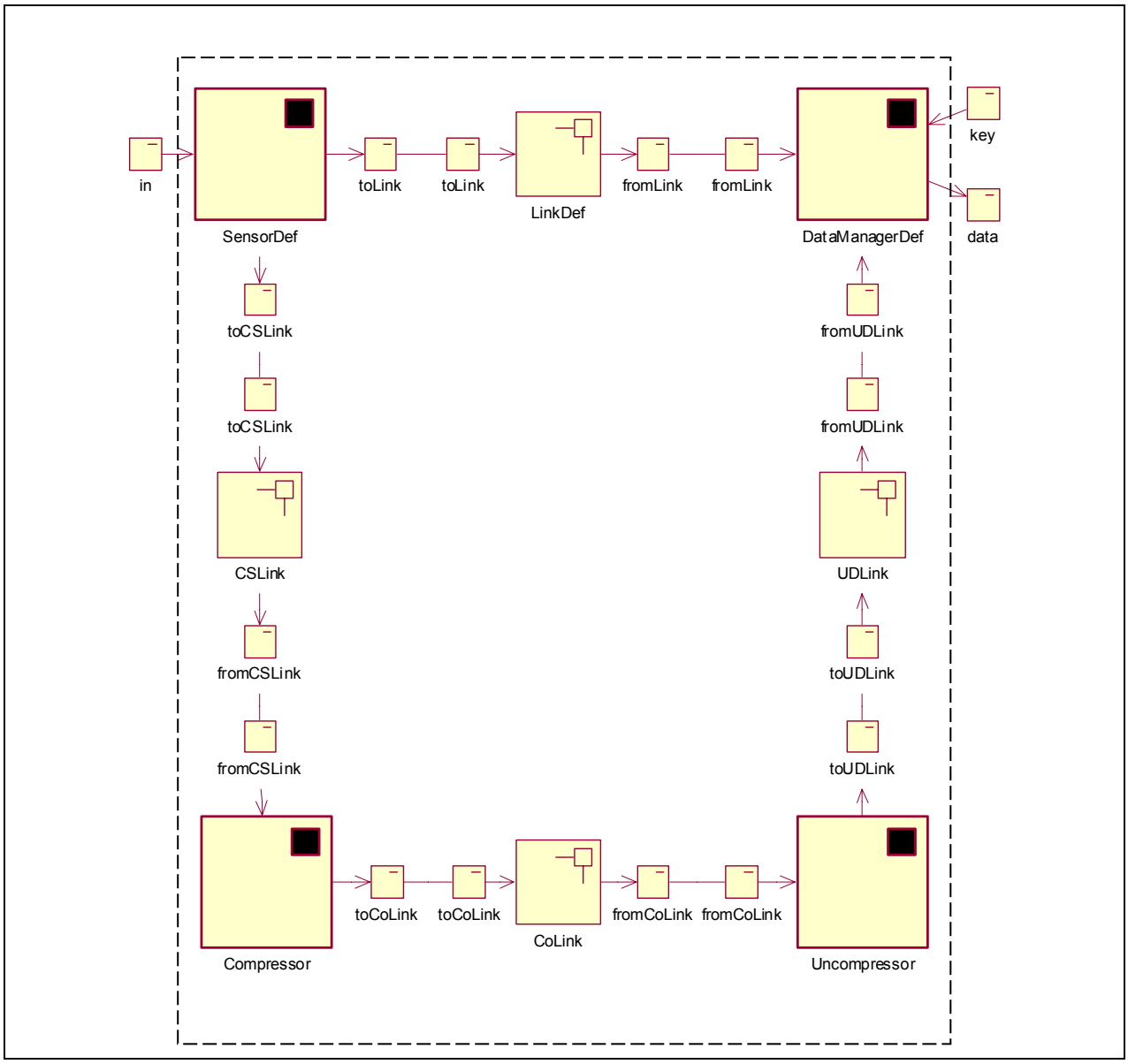


Figure III.6. : Rajouter et connecter des connecteurs

Dans *ArchWare ARL*, ce raffinement peut être décrit comme suit :

```
archetype DasRef5 refines DasRef4 using {
  connectors includes {
  CoLink is connector {
    types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
    incoming is { toCoLink is connection(Entry) }
    outgoing is { fromCoLink is connection(Entry) }
    behaviour is { deferred }
  }.
  CSLink is connector {
    types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
    incoming is { toCSLink is connection(Entry) }
    outgoing is { fromCSLink is connection(Entry) }
    behaviour is { deferred }
  }.
  UDLink is connector {
    types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
    incoming is { toUDLink is connection(Entry) }
    outgoing is { fromUDLink is connection(Entry) }
    behaviour is { deferred }
  }.
  connections unifies CoLink::toCoLink with Compressor::toCoLink.
  connections unifies CoLink::fromCoLink with Uncompressor::fromCSLink.
  connections unifies CSLink::toCSLink with SensorDef::toCSLink.
  connections unifies CSLink::fromCSLink with Compressor::fromCSLink.
  connections unifies UDLink::toUDLink with Uncompressor::toUDLink.
  connections unifies UDLink::fromUDLink with DataManagerDef::toUDLink
}
```

Naturellement, l'ajout des connecteurs et des composants, suivent les mêmes étapes de base : ajout des connecteurs sans connections, des connections de sortie et des connections d'entrée. Le comportement du système lui-même reste inchangé puisque les connections *toCSLink* et *fromUDLink* supplémentaires de *SensorDef* et *DataManagerDef* ne sont pas encore utilisées dans leurs comportements respectifs.

5ème étape : raffinement des comportements des composants

L'architecte pourrait maintenant raffiner le comportement des composants supplémentaires. Les composants *Compressor* et *Uncompressor* par l'action suivante :

```
archetype DasRef6 refines DasRef5 using {
  Compressor::behaviour is abstraction() {
      compress is function(d : Data) : Data { deferred }.
      replicate   via fromCSLink receive entry.
                  via toCoLink send tuple(entry::key, compress(entry::data))
  }
  Uncompressor::behaviour is abstraction() {
      uncompress is function(d : Data) : Data { deferred }.
      replicate   via fromCoLink receive entry.
                  via toUDLink send tuple(entry::key, uncompress(entry::data))
  }
}
```

Le compresseur applique la fonction de compression aux données reçues de sa connection d'entrée pour envoyer les données compressées via sa connection de sortie. Le décompresseur applique la fonction *uncompress* aux données reçues de sa connection d'entrée pour envoyer les données non compressées via sa connection de sortie. La structure du système demeure sans changement.

6ème étape : raffinement des comportements des connecteurs ajoutés

L'architecte pourrait maintenant raffiner le comportement des connecteurs supplémentaires pour porter les données par *CSLink,* les données compressées par *CoLink* et les données non compressés par *UDLink* Ceci est accompli en raffinant le comportement de ces connecteurs comme suit :

```
archetype DasRef7 refines DasRef6 using {
  CSLink::behaviour is abstraction() {
      replicate via toCSLink receive entry : Entry. via fromCSLink send entry
  }.
  CoLink::behaviour is abstraction() {
      replicate via toCoLink receive entry : Entry. via fromCoLink send entry
  }.
  UDLink::behaviour is abstraction() {
      replicate via toUDLink receive entry : Entry. via fromUDLink send entry
  }
}
```

7ème étape : raffinement des comportements des composants existants

L'architecte pourrait maintenant raffiner le comportement des composants existants de l'architecture abstraite afin de tenir compte des nouveaux composants et connecteurs présentés. Ceci est accompli en raffinant le comportement de *SensorDef* et *DataManagerDef* :

```
archetype DasRef8 refines DasRef7 using {
  Sensor refines SensorDef using {
      connections replaces toLink by toCSLink
  }.
  DataManager refines DataManagerDef using {
      connections replaces toLink by toUDLink
  }
}
```

L'architecte suppose que la compression et la décompression des données traitées via *CoLink* rapporte les mêmes données que celles transmises par *Link*. Formellement, ceci s'exprime comme suit :

$\forall d : Data \bullet uncompress(compress(d)) = d$

c'est-à-dire qu'afin de garantir que le comportement de la nouvelle architecture raffine le comportement de l'architecture abstraite, l'architecte suppose vraie cette propriété. Cette hypothèse est exprimée en *ArchWare ARL* de la façon suivante :

```
archetype DasRef8bis refines DasRef8 assuming {
  property dataIntegrity is { forall d : Data | uncompress(compress(d)) = d }
}
```

Elle sera vérifiée après l'application de l'action de raffinement. La figure suivante (Fig. III.7.) illustre l'architecture résultant des dernières étapes de raffinement.

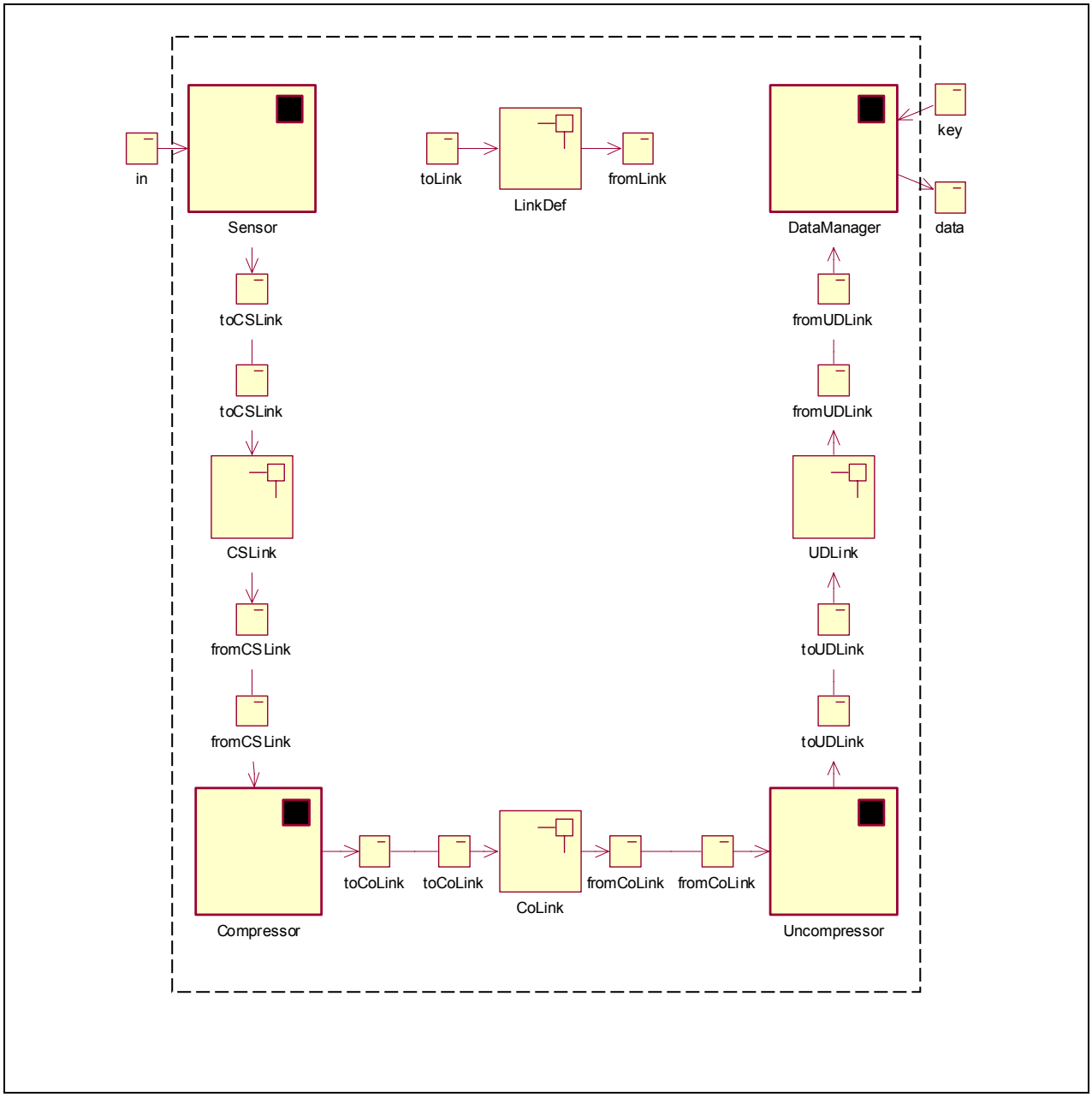


Figure III.7. : Reconnecter les connections de SensorDef et DataManagerDef

8ème étape : supprimer les connecteurs déconnectés

Après ce raffinement, le connecteur *LinkDef* ne sera pas utilisé et l'architecte peut alors le supprimer en utilisant l'action suivante :

```
archetype aDasRef9 refines aDasRef8bis using {
  connectors excludes LinkDef
}
```

La figure suivante (Fig. III.8.) illustre l'architecture résultant de cette étape de raffinement.

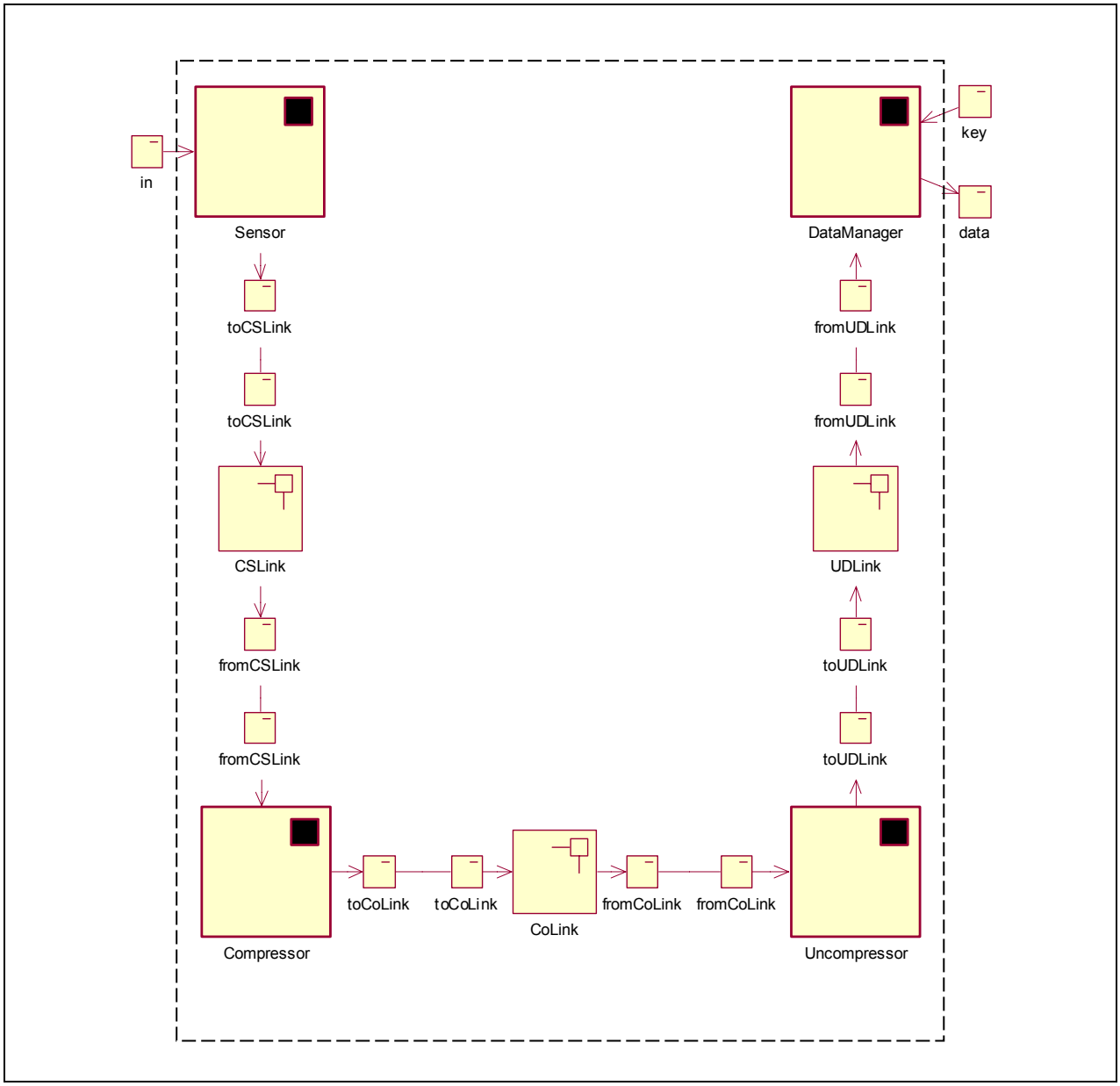


Figure III.8. : Suppression d'un connecteur

9ème étape : implosion des sous-architectures comme étant un seul composant

Dans cette étape (finale) de raffinement, afin d'obtenir une architecture concrète indépendante de la plateforme, l'architecte implosera *SensorDef* et *Compressor* en un seul composant, de même pour *DataManagerDef* et *Uncompressor*. La figure suivante (Fig. III.9.) illustre l'architecture avant cette étape de raffinement.

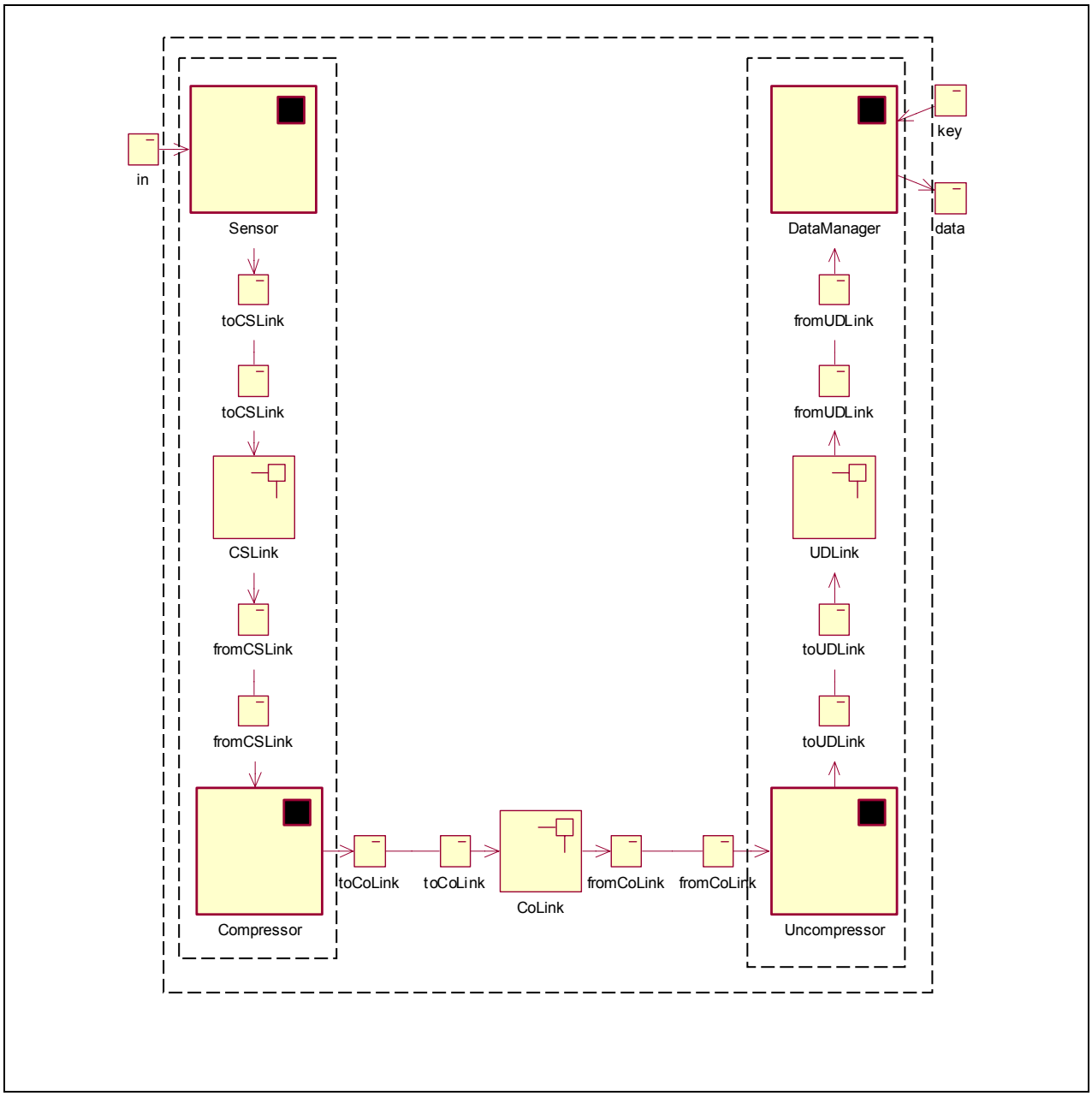


Figure III.9. : Implosion des sous-architectures

L'implosion s'exprime comme suit :

```
archetype DasArchConc refines DasRef9 using {
  components implodes { Sensor and Compressor } as CoSensor and
  components implodes { DataManager and Uncompressor } as CoDataManager
}
```

La figure suivante (Fig. III.10.) montre l'architecture après cette étape de raffinement.

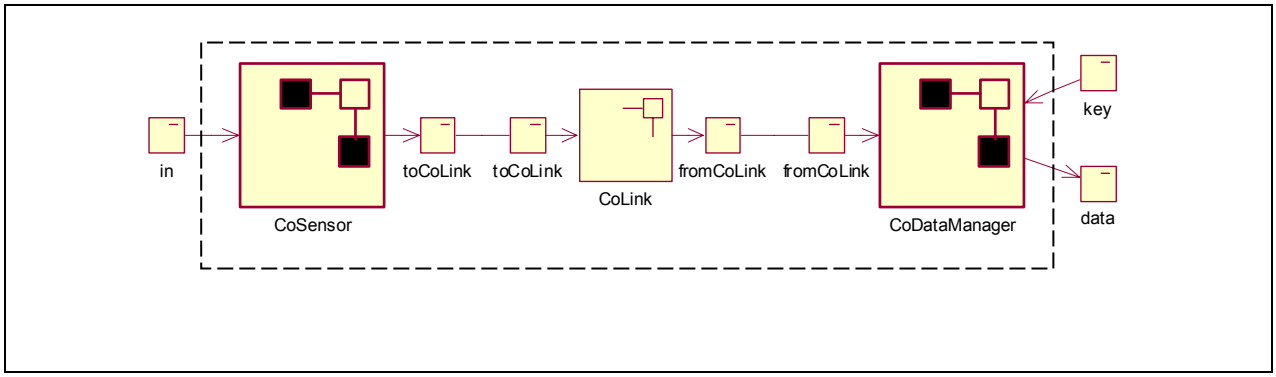


Figure III.10. : Définition d'une architecture concrète

Description de l'architecture raffinée

L'application de toutes ces actions de raffinement conduit au fur et à mesure des étapes à une architecture concrète. L'architecture obtenue par raffinement est équivalente à l'architecture suivante dans *ArchWare ARL :*

```
archetype DasArchConc is architecture {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { in is connection(Entry). key is connection(Key) }
  outgoing is { data is connection(Data) }
  behaviour is compose { CoSensor and CoLink and CoDataManager }
}
```

Les deux composants composites *CoSensor* et *CoDataManager* et le connecteur *CoLink* sont décrits en *ArchWare ARL* de la façon suivante :

```
archetype CoSensor is component {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { in is connection(Entry) }
  outgoing is { toCoLink is connection(Entry) }
  behaviour is compose { Sensor and Compressor }
}

archetype CoDataManager is component {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { key is connection(Key). fromCoLink is connection(Entry) }
  outgoing is { data is connection(Data) }
  behaviour is compose { DataManager and Uncompressor }
}

archetype CoLink is connector {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { toCoLink is connection(Entry) }
  outgoing is { fromCoLink is connection(Entry) }
  behaviour is abstraction() {
        replicate via toCoLink receive entry. via fromCoLink send entry
        }
}
```

Les sous-composants et le sous-connecteur du composant composite *CoSensor* sont décrits en *ArchWare ARL* comme suit :

```
archetype Sensor is component {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { in is connection(Key) }
  outgoing is { toCSLink is connection(Entry) }
  behaviour is abstraction() {
        process is function(d: Data) : Data { deferred }.
        replicate     via in receive entry.
                      via toCSLink send tuple(entry::key, process(entry::data))
        }
}

archetype Compressor is component {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { fromCSLink is connection(Entry) }
  outgoing is { toCoLink is connection(Entry) }
  behaviour is abstraction() {
        compress is function(d : Data) : Data { deferred }.
        replicate     via fromCSLink receive entry.
                      via toCoLink send tuple(entry::key, compress(entry::data))
        }
}

archetype CSLink is connector {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { toCSLink is connection(Entry) }
  outgoing is { fromCSLink is connection(Entry) }
  behaviour is abstraction() {
        replicate via toCSLink receive entry. via fromCSLink send entry
        }
}
```

Les sous-composants et le sous-connecteur du composant composite *CoDataManager* sont décrits en *ArchWare ARL* de la manière suivante :

```
archetype DataManager is component {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { key is connection(Key). fromCoLink is connection(Entry) }
  outgoing is { data is connection(Data) }
  behaviour is abstraction() {
        database is location(Set() : Entry).
        replicate choose {
                via fromLink receive entry.
                database := database' including(entry)
            or  via key receive queryKey : Key.
                via data send (database' selecting(d | d::key == queryKey)
            }
  }
}

archetype Uncompressor is component {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { fromCoLink is connection(Entry) }
  outgoing is { toUDLink is connection(Entry) }
  behaviour is abstraction() {
        uncompress is function(d : Data) : Data { deferred }.
        replicate   via fromCoLink receive entry.
                    via toUDLink send tuple(entry::key, uncompress(entry::data))
        }
}

archetype UDLink is connector {
  types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
  incoming is { toUDLink is connection(Entry) }
  outgoing is { fromUDLink is connection(Entry) }
  behaviour is abstraction() {
        replicate via toUDLink receive entry. via fromUDLink send entry
        }
}
```

6. Conclusion

En résumé, *ArchWare ARL* est un langage dédié au raffinement d'architectures logicielles avec d'une part la prise en compte du raffinement des données, comportements, ports et structures, et d'autre part, la préservation des propriétés architecturales à travers le processus de raffinement. Le noyau du langage est un ensemble d'opérations de modification d'architectures supportant les transformations architecturales. *ArchWare ARL* à l'instar des méthodes formelles B et Z, fournit un ensemble d'opérations architecturales compositionnelles pour la transformation d'architectures logicielles. Les raffinement horizontal et vertical sont tous les deux possibles dans *ArchWare ARL*.

Du point de vue processus, le raffinement d'une architecture abstraite vers une architecture concrète en vue de son implantation se fait en plusieurs étapes successives. Chaque étape de raffinement implique l'application d'une action de raffinement qui fournit une solution correcte d'une transformation architecturale. Les actions de raffinement dans *ArchWare ARL* sont exprimées par des *pré-conditions*, *transformations* et *post-conditions*. Les pré-conditions sont des conditions qui doivent être satisfaites dans un modèle architectural abstrait avant l'application d'une action de raffinement. Les post-conditions doivent être satisfaites dans le résultat obtenu par l'application d'une action de raffinement. Une transformation exprimée par une action de raffinement montre comment on peut transformer une architecture satisfaisant les pré-conditions en une architecture moins abstraite satisfaisant les post-conditions. Un modèle de transformation architecturale est dit correct si et seulement si il satisfait les pré- et les post-conditions ; il peut alors être appliqué sans preuve pour le développement des architectures concrètes spécifiques depuis des architectures abstraites. Pour prouver que le raffinement est correct, des "hypothèses" sont rajoutées. Ces hypothèses sont des obligations de preuve déchargées au moment de l'application et utilisées par exemple par l'outil d'analyse *ArchWare AAL* [AGM 02] (développé dans le cadre du projet *ArchWare)* pour établir si un raffinement est correct ou non.

Dans le chapitre qui suit, nous présentons l'environnement logiciel *Refiner* que nous proposons pour automatiser les processus de raffinement dédiés au langage *ArchWare ARL*. *Refiner* est fondé sur la logique de réécriture [Mes 98] [MOM 93].

Chapitre 4 : Formalisation du Refiner dans la logique de réécriture

Chapitre 4 : Formalisation du Refiner dans la logique de réécriture

La logique de réécriture est un cadre sémantique général pour spécifier des systèmes concurrents et des langages [MaM 96][Mes 98][ChO 01]. C'est également un cadre logique pour représenter et exécuter différentes logiques et langages [Mes 02][Ven 02]. Dans ce chapitre, nous nous focalisons sur la spécification de la sémantique formelle et opérationnelle en logique de réécriture, du langage de raffinement des architectures *ArchWare ARL*, présenté dans le chapitre précédent. Nous commençons d'abord par l'introduction des différents concepts et fondements de la logique de réécriture puis nous présenterons la formalisation dans celle-ci de l'environnement logiciel *Refiner* dédié au langage *ArchWare ARL*.

1. La logique de réécriture

1.1. Présentation

La logique en général est une méthode de raisonnement correct pour quelques classes d'entités. Pour la logique équationnelle, les entités sont les ensembles, les fonctions entre ces ensembles et la relation d'identité entre les éléments. Pour la *logique de réécriture*, les entités sont les systèmes concurrents ayant des états, et qui évoluent à travers des transitions. Elle est considérée également comme une logique de changement concurrent qui traite l'état et les calculs concurrents.

Formellement, la logique de réécriture est représentée par une théorie de réécriture *TR = (S, R)* où :
S est une signature qui décrit une structure particulière pour les états du système (multi-ensemble, arbre binaire, etc.) qui peuvent être distribués, et, *R* les règles de réécritures, qui indiquent les transitions possibles dans l'état distribué, engendrées par les transformations locales concurrentes. Les règles de réécriture $t \rightarrow t'$ dans *R* ne sont pas des équations car au niveau du calcul, elles sont interprétées comme des règles de transitions locales dans un système concurrent (logiquement elles sont interprétées comme des règles d'inférence).

Dans la logique de réécriture les unités de base de la spécification sont appelées *théories* qui sont de deux types : les *théories fonctionnelles* et les *théories systèmes*.

Les *théories fonctionnelles* sont des théories dans la logique équationnelle d'appartenance appelée *MEL* (*MemberShip Equational Logics*) qui est équivalente à la logique de Horn. Les phrases dans *MEL* sont des égalités de type `t=t'` ou bien des assertions d'appartenance de la forme `t:s` exprimant que `t` est de sorte `s`. Une telle logique étend la logique équationnelle des sortes ordonnées (*order-sorted OBJ3*) et contient ainsi les *sortes*, les relations *sous-sorte*, la *surcharge des opérateurs*, et la *définition des opérations*.

A titre d'exemple, supposons que `t, t', t'', t1,…, tn, t1',…, tn'` soient des termes quelconques, `x` une variable et `f` une opération d'arité `n`. Nous pouvons alors définir dans une théorie fonctionnelle des règles (interprétées ici comme des règles d'inférence) dites de « remplacement d'égaux par les égaux » de la façon suivante :

- *Réflexivité :* `t=t`,
- *Symétrie :* `t=t' => t'=t`,
- *Transitivité :* `t=t' & t'=t'' => t=t''`,
- *Substitution :* `t=t' => t [t''/x] = t' [t''/x]`,

- *Congruence :* `t1=t1' & ……& tn=tn' => f(t1,…, tn) = f(t1',…,tn').`

Les *théories systèmes* quant à elles spécifient le modèle initial *TR0* d'une théorie de réécriture *TR*. Ce modèle initial est un système de transitions dont :

- les états sont des classes d'équivalence *[t]* des termes *t* modulo les équations *E* définies dans *R*,
- les transitions sont des « α-Preuves » de la forme : $[t] \rightarrow [t']$.

Nous détaillons dans la suite les différents concepts utilisés dans les théories fonctionnelles et systèmes avant de présenter notre formalisation du langage ARL. Nous ne présentons cependant que les concepts dont nous nous sommes servis. Plus de détails pourront être trouvés dans [CDELMMQ 99] [MFSPNJC 03].

1.2. Concepts de base

Dans ce qui suit nous introduisons d'abord les éléments communs aux *théories fonctionnelles* et aux *théories systèmes* : *sortes, sous-sortes, opérations et variables.* Nous présenterons ensuite la définition de la syntaxe et de la sémantique spécifiques à chacun des types de théories.

1.2.1. Sortes

La première chose à déclarer dans une spécification sont les *sortes* de données et les opérations correspondantes. Les sortes sont partiellement ordonnées via une relation de sous-sorte. Une sorte est déclarée en utilisant le mot-clé **`sort`** suivi d'un identificateur (le nom de sorte) suivi d'un espace et un point :

`sort <Sort> .`

Les sortes multiples seront déclarées en utilisant le mot-clé **`sorts`** :

`sorts <Sort`$_1$**`>`** … **`<Sort`**$_n$**`>.`**

1.2.2. Sous-sortes

La relation de sous-sorte **`subsort`** sur les sortes est équivalente à une relation de sous-ensemble sur un ensemble d'éléments (relation d'inclusion). Les sous-sortes sont déclarées en utilisant le mot-clé **`subsort`** ou **`subsorts`**. La déclaration :

`subsort <Sort`$_1$**`>  <  <Sort`**$_2$**`> .`**

signifie que la sorte **`<Sort`**$_1$**`>`** est une sous-sorte de **`<Sort`**$_2$**`>`**. Par exemple, les déclarations :

```
subsort Zero < Nat .
subsort NzNat < Nat .
```

signifient que les sortes **`Zero`** (contenant seulement la constante 0) et **`NzNat`** (les nombres naturels non nuls) sont des sous-sortes de **`Nat`** (les nombres naturels). On peut déclarer plusieurs relations de sous-sorte en utilisant **`subsorts`** :

`subsorts <Sort`$_1$**`>`** … **`<Sort`**$_2$**`>  <`** … **`<  <Sort`**$_n$**`> .`**

Par conséquent les déclarations de l'exemple précédent peuvent être faites en une seule ligne :

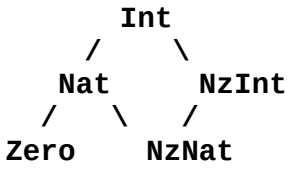
```
subsorts Zero NzNat < Nat .
```

On peut également exprimer la relation **subsort** comme suit :

```
sorts NzInt Int .
subsorts NzNat < NzInt Nat < Int .
```

où **NzNat et Int** sont les ensembles respectivement des entiers naturels non nuls et des entiers.

Un ensemble de déclarations de sous-sortes définit un ordre partiel sur l'ensemble de sortes. Toutefois, pour que cette déclaration soit correcte, l'utilisateur doit éviter les cycles dans les déclarations de sous-sortes. Graphiquement, nous pouvons visualiser cet ordre partiel de la façon suivante :



1.2.3. Opérations

Dans une théorie, une opération est déclarée à l'aide du mot-clé **op** suivi de son nom, suivi de deux point, suivis d'une série de sortes qui constituent ses arguments (arité), suivis de **->**, suivi des sortes des résultats (co-arité), optionnellement suivis de la déclaration d'attributs (présentés plus loin) suivis d'un espace et un point. Le schéma général a la forme suivante :

op <OpName> : <Sort$_0$>. . . <Sort$_k$> -> <Sort>[<OperatorAttributes>] .

Voici quelques exemple de déclarations d'opérations :

```
op zero : -> Zero .
op s_ : Nat -> NzNat .
op sd : Nat Nat -> Nat .
ops _+_ _*_ : Nat Nat -> Nat .
```

Si l'arité des arguments est vide, l'opération est appelée une constante comme c'est le cas pour le **Zero**. Le nom d'opération est une chaîne de caractères. Les espaces soulignés (_) jouent un rôle spécial dans ces chaînes de caractères. En effet, si aucun espace souligné ne se produit dans la chaîne de caractères de l'opération comme dans le cas de **sd** alors l'opérateur est déclaré sous la forme *préfixée*. Si les espaces soulignés se produisent dans la chaîne de caractères, alors leur nombre doit coïncider avec le nombre de sortes déclarées comme arguments de l'opérateur. L'opérateur est alors sous la forme dite *mixfix* où le nième espace souligné indique la position de l'argument de la nième sorte dans les expressions formées avec cet opérateur. Dans l'exemple précédent les opérateurs **s_**, **_+_**, et **_*_** sont sous la forme *mixfix*.

Il peut y avoir tous les autres caractères avant ou après n'importe lequel de ces espaces soulignés. Si aucun caractère n'apparaît, nous dirons que l'opérateur a été déclaré avec la syntaxe vide. Par exemple, nous pourrions déclarer une sorte **NatSeq** séquence des nombres naturels avec la syntaxe vide comme suit :

```
sort NatSeq .
subsort Nat < NatSeq .
```

```
op __ : NatSeq NatSeq -> NatSeq [assoc] .
```

L'attribut **assoc** dans l'exemple ci-dessus est un attribut déclarant que la concaténation est associative. Avec cette déclaration d'opérateur nous pouvons écrire des termes comme :

```
zero (s zero) (s s zero)
```

Des opérations ayant les mêmes arité et co-arité peuvent être déclarées simultanément en employant **ops** comme mot-clé et en donnant les sortes non vides après le mot-clé **ops** et avant deux points, comme pour les déclarations de **_+_** et **_*_** dans l'exemple plus haut.

1.2.4. Surcharge d'opérateurs

Les opérateurs dans la logique de réécriture peuvent être surchargés, cest-à-dire que nous pouvons avoir plusieurs déclarations d'opérations pour le même opérateur avec différentes arités et co-arités. Considérons l'extension de notre théorie de nombres avec une nouvelle sorte **Nat3** (les nombres naturels modulo 3), les constantes 0, 1, et 2 de sorte **Nat3**, et deux autres déclarations d'opérateur pour **_+_**.

```
op _+_ : NzNat Nat -> NzNat .
sort Nat3 .
ops 0 1 2 : -> Nat3 .
op _+_ : Nat3 Nat3 -> Nat3 .
```

L'opérateur **_+_** est surchargé et a trois déclarations jusqu'à maintenant. Cependant, il y a deux types de surcharge dans l'exemple. La déclaration de _+_ avec le premier argument **NzNat** est un exemple de la surcharge de **subsort** où les deux opérateurs sont sensés avoir le même comportement (l'opérateur sur la sous-sorte est la restriction de l'opérateur sur la sorte plus grande). Le point principal de telles déclarations est de fournir plus d'informations sur la sorte, par exemple le résultat de l'ajout d'un nombre différent de zéro à n'importe quel nombre est non nul.

1.2.5. Variables

Les variables peuvent être déclarées à tout moment avec une syntaxe comportant un identificateur (le nom de la variable), deux points et un autre identificateur (sa sorte). Par exemple, **N:Nat** déclare une variable nommée **N** de sorte **Nat**.

La portée d'une déclaration de variable est déterminée «à la volée » par l'occurrence de la déclaration. Ainsi, une variable doit être accompagnée de sa sorte. Une variable peut également être déclarée dans une théorie en utilisant le mot clé **var** suivi d'un identificateur (le nom de variable) suivi des deux points avec l'espace avant et après les deux points, suivi d'un identificateur (sa sorte), suivi d'un espace et d'un point :

```
var N : Nat .
```

La portée d'une telle déclaration est la théorie **NAT** des entiers naturels. Elle a pour effet de remplacer « à la volée » des occurrences de **N** par **N:Nat** . Des variables de la même sorte peuvent être déclarées en utilisant le mot-clé **vars** :

```
vars M N : Nat .
```

Les conventions syntaxiques pour les noms de variables dans les déclarations de variables sont les mêmes que celles des opérations et constantes.

1.2.6. Théories Fonctionnelles

Les théories fonctionnelles définissent les sortes de données et les opérations sur ces données à travers des théories équationnelles. Les sortes de données se composent des éléments qui peuvent être appelés par des termes. Deux termes dénotent le même élément si et seulement si ils appartiennent à la même classe d'équivalence déterminée par les équations. La sémantique d'une théorie fonctionnelle est son algèbre initiale. On suppose que les théories fonctionnelles ont la propriété suivante : l'application répétée des équations comme règles de simplification atteint par la suite un terme à laquelle aucune autre équation ne s'applique. Le résultat, appelé *forme canonique*, est le même quelque soit l'ordre de l'application des règles. Ainsi chaque classe d'équivalence a un représentant normal, sa forme canonique, qui peut être calculé par simplification équationnelle.

Comme nous l'avons dit précédemment, la logique équationnelle sur laquelle les théories fonctionnelles sont fondées est la logique équationnelle d'appartenance (MEL) qui fournit entre autres les concepts de sorte, relation sous-sorte, surcharge d'opérateur et assertions d'appartenance à une sorte. Une théorie fonctionnelle est déclarée selon la syntaxe suivante :

fmod <Théorie Name> is [<DeclarationsAndStatements>] endfm

Par exemple :

```
fmod NUMBERS is
...
endfm
```

déclare une théorie fonctionnelle appelée **NUMBERS**. Les points représentent les déclarations et les expressions qui peuvent apparaître dans la théorie. Les déclarations incluent l'importation d'autres théories fonctionnelles, les sortes, les sous-sortes, et des déclarations d'opérations. Les expressions incluent des axiomes équationnels et d'appartenance.

1.2.7. Equations inconditionnelles

Les équations sont déclarées en utilisant le mot-clé **eq**, suivi par un terme (la partie gauche), le signe d'égalité, puis un terme (la partie droite), et optionnellement suivi par des attributs encadrés par des crochets, terminé par un espace et un point. Ainsi le schéma général est le suivant :

eq <Term$_1$> = <Term$_2$> [<**StatementAttributes**>]**.**

Les termes **t** et **t'** dans une équation **t = t'** doivent être de même sorte. Pour que l'équation s'exécute, chaque variable qui apparaît dans **t'** doit apparaître dans **t**.

Nous pouvons par exemple ajouter des équations relatives à l'addition dans la théorie **NUMBERS** :

```
vars N M : Nat .
eq N + zero = N .
eq N + s M = s (N + M) .
```

avec *s* défini précédemment comme opérateur unaire.

1.2.8. Axiomes d'appartenance inconditionnelle

Les axiomes d'appartenance sont déclarés avec le mot-clé **mb** suivi par un terme, suivi par deux points, suivis par une sorte suivie par un point. Comme les équations, les axiomes d'appartenance peuvent « optionnellement » être suivis par des attributs désignés par « statements ». Pour illustrer ceci, nous

considérons la théorie **3*NAT** avec la déclaration des nombres de base "peano" comme dans la théorie **NUMBERS** et une nouvelle sorte **3*Nat**. En effet, **3*Nat** correspond aux multiples de 3 et est exprimée en utilisant la déclaration de sous-sorte suivante : **Zero < 3*Nat < Nat** et l'expression d'appartenance **mb(s s s M3) : 3*Nat** pour **M3** une variable de sorte **3*Nat**.

```
fmod 3*NAT is
sort Zero Nat .
subsort Zero < Nat .
op zero : -> Zero [ctor] .
op s_ : Nat -> Nat [ctor] .
sort 3*Nat .
subsorts Zero < 3*Nat < Nat .
var M3 : 3*Nat .
mb (s s s M3) : 3*Nat .
endfm
```

1.2.9. Equations et axiomes d'appartenance conditionnelle

Les conditions dans les équations conditionnelles et d'appartenance se composent de différentes équations **t=t0** et d'appartenance **t:s**. Une condition peut être une équation simple, une appartenance simple, ou une conjonction d'équations et d'axiomes d'appartenance en utilisant la conjonction binaires de liaison **/** qui est associative.
Ainsi la forme générale des équations conditionnelles (ceq) et des axiomes d'appartenance (cmb) est la suivante :

```
ceq  <Term-1> = <Term-2>
if <EqCondition-1>/\... /\ <EqCondition-k >[<StatementAttributes>] .

cmb  <Term> : <Sort >
if <EqCondition-1>/\ ... /\<EqCondition-k>[<StatementAttributes> ] .
```

De plus, la syntaxe concrète de la condition équationnelle "**EqCondition**" a trois variantes :

- équations ordinaires **t = t'**,
- équations d'affectation **t:=t'**,
- équations booléennes dites abrégées de la forme **t**, avec **t** un terme dans la sorte **Bool** abrégeant l'équation **t = true**.

Ainsi, ces conditions peuvent apparaître dans une équation :

```
(N == zero) = true
(M =/= s zero) = true
(N > zero or M =/= s zero) = true
```

1.2.10. Confluence et terminaison des équations

La logique de réécriture ne spécifie pas dans quel ordre les équations seront appliquées en calculant la forme normale d'un terme. Par conséquent les équations doivent être confluentes et mener à une terminaison. Par exemple, l'exécution suivante de l'axiome d'appartenance n'est pas confluente dans la logique de réécriture :

```
eq E:Element in E:Element, S:Set = true .
eq E:Element in S:Set = false .
```

dans le cas où l'élément serait dans l'ensemble donné, la première règle renvoie vrai, tandis que la deuxième règle renvoie faux. Une solution au problème exige un deuxième opérateur **_in?_.**

```
eq  E:Element in? E:Element, S:Set = true .
eq  E:Element in S:Set = E:Element in? S:Set = true .
```

La définition est confluente, puisque seulement une équation est donnée pour chaque opérateur. Si **E** est contenu dans **S**, la proposition **E in? S** est vraie, et l'expression entière devient vraie. Si **E** n'est pas contenu dans **S**, la valeur de **E in? S** est fausse ce qui a comme conséquence le renvoi de la valeur *false*.

1.2.11. Attributs

Les déclarations d'opérateur peuvent inclure des attributs qui fournissent des informations additionnelles à l'opérateur : sémantique, syntaxique, pragmatique, etc. Tous ces attributs sont déclarés dans une seule paire de crochets, après la sorte du résultat et avant le point. Nous abordons chacune des catégories d'attributs dans ce qui suit.

1.2.12. Attributs équationnels

Les attributs équationnels représentent le moyen pour déclarer certains genres d'axiomes qui permettent à la logique de réécriture d'utiliser les équations d'une manière intégrée. La logique de réécriture supporte les attributs équationnels suivants :

- **assoc (associativité),**
- **comm (commutativité),**
- **idem (idempotence),**
- **id: <Term> (identité).**

Ces attributs sont permis seulement pour les opérateurs binaires dont les arguments appartiennent au même composant. Un opérateur peut être déclaré avec n'importe quelle combinaison de ces attributs qui peuvent apparaître dans n'importe quel ordre dans la déclaration.

Sémantiquement, la déclaration d'un ensemble d'attributs équationnels pour un opérateur est équivalente à déclarer les équations correspondantes pour l'opérateur. Du point de vue fonctionnement, l'utilisation des attributs équationnels évite des problèmes d'arrêt et mène à une évaluation beaucoup plus efficace des termes contenant cet opérateur. En fait, l'effet de déclarer des attributs équationnels permet de calculer les classes d'équivalence modulo les équations. Par exemple, si une équation de commutativité, $x + y = y + x$, est déclarée comme équation ordinaire, elle produira facilement une boucle infinie de simplifications. Si on la déclare avec l'attribut **comm**, ce comportement faisant une boucle ne se produit pas. Dans notre exemple de nombres nous pouvons ajouter **nil** pour exprimer le vide et raffiner la déclaration de la concaténation des séquences des nombres, de sorte que la concaténation soit associative avec l'identité nil.

```
op nil : -> NatSeq .
op __ : NatSeq NatSeq -> NatSeq [assoc id: nil] .
```

1.2.13. Constructeurs

Supposant que les équations dans une théorie fonctionnelle se terminent, alors chaque terme dans la théorie (un terme sans variables) sera simplifié dans une forme canonique modulo certains attributs équationnels. Les constructeurs sont les opérateurs apparaissant sous telle forme canonique. Les opérateurs qui " disparaissent " après la simplification équationnelle s'appellent dans ce cas « fonctions

définies ». Par exemple, des constructeurs typiques dans une sorte **Nat** sont **Zero** et **s_**, tandis que dans la sorte **Bool**, **true** et **false** sont les seuls constructeurs.

Il est tout à fait utile pour différents buts, y compris la correction et la démonstration de théorèmes, d'indiquer quand un opérateur donné est un constructeur. Ceci peut être fait avec l'attribut **ctor**. Par exemple, nous pouvons définir nos déclarations d'opérateurs avec l'information de constructeur comme suit :

```
op zero : -> Zero [ctor] .
op s_ : Nat -> NzNat [ctor] .
op __ : NatSeq NatSeq -> NatSeq [ctor assoc] .
```

1.2.14. Théories Systèmes

Une théorie système de la logique de réécriture spécifie une théorie de réécriture. Une théorie de réécriture a des sortes, et des opérations, et peut avoir des : équations, axiomes d'appartenance, et règles qui peuvent être conditionnelles. Par conséquent, n'importe quelle théorie de réécriture a une théorie équationnelle fondamentale, contenant les équations et les appartenances, plus les règles de réécriture.

Informatiquement, les règles spécifient les transitions concourantes locales qui peuvent avoir lieu dans un système si le patron dans le côté gauche de la règle correspond à un fragment de l'état de système et la condition de la règle est satisfaite. Dans ce cas, la transition indiquée par la règle peut avoir lieu, et le fragment de l'état correspondant est transformé en l'instance correspondante du côté droit. Comme cela a été déjà mentionné auparavant, une théorie système est déclarée comme suit :

```
mod  <Théorie Name> is <DeclarationsAndStatements>  endm
```

Par exemple :

```
mod REFINER is
...
endm
```

où les points de suspension correspondent à toutes les déclarations et expressions dans une théorie :

1. importations,
2. sortes et sous-sortes,
3. opérations,
4. variables,
5. équations et axiomes d'appartenance (conditionnels et non conditionnels),
6. règles de réécriture (conditionnelles et non conditionnelles).

Les déclarations **1-5** sont exactement les mêmes que celles dans les théories fonctionnelles. Les théories systèmes rajoutent les règles de réécriture que nous présentons.

1.2.15. Règles de réécriture

Il n'y a (mathématiquement) aucun comportement dynamique dans des spécifications équationnelles. En effet, dans le modèle mathématique des spécifications équationnelles de même que dans la logique équationnelle, deux expressions sont soit équivalentes soit aucune relation n'existe entre elles. Hors pour les systèmes dynamiques, qui évoluent avec le temps, cet arrangement équationnel n'a pas de sens

mathématique ou logique. Pour illustrer un système dynamique, regardons le modèle simplifié de la vie d'un être humain.

Dans cette exemple, un terme

person(‘Peter, 32)

dénote une personne avec le nom ‘Peter qui a 32 ans. Pour simuler sa vie (et par la suite son vieillissement), le prochain état devrait être :

person(‘Peter, 33)

Une façon de modéliser un système de simulation d’âge serait l’équation :

person(X,N) = person(X,N+1)

Dans un tel système il serait logiquement vrai que

person(‘Peter,32) = person(‘Peter,33)

Ce changement est réversible – de part la propriété de symétrie de la logique équationnelle –de sorte qu’on aura aussi :

person(‘Peter,33)= person(‘Peter,32),

et

person(‘Peter,32) = person(‘Peter,20)

Cependant dans le système que nous modélisons cette propriété intuitivement ne devrait pas se tenir car on ne peut malheureusement diminuer notre âge. Le changement n'est habituellement pas réversible dans les systèmes dynamiques.

En d'autres termes, pour spécifier (ou changer/évoluer) un système dynamique, nous ne pouvons pas employer seulement des équations. En revanche, nous employons la logique de réécriture, qui prolonge des techniques algébriques de spécifications équationnelles pour modéliser les systèmes dynamiques.

1.2.16. Règles non conditionnelles de réécriture

Le problème avec l’utilisation des équations pour modéliser le comportement dynamique est, comme indiqué auparavant, le fait que l'égalité soit symétrique. En logique de réécriture, le comportement dynamique est modélisé par des règles de réécriture. Une règle de réécriture est essentiellement une équation " à sens unique ", et la logique de réécriture est de ce fait une logique équationnelle sans symétrie. Ainsi, le comportement dynamique d'une personne dans notre exemple peut être modélisé par une règle de réécriture :

person(X, N) = > person(X, N+1)

Pour des variables appropriées **X** et **N**. Dans la logique de réécriture un séquent **t = > t ',** est prévu pour signifier que l'état **t** peut changer en (ou évoluer, ou être étendu à) un état **t'**. Les règles de réécriture donnent les changements d’états locaux et atomiques comme dans l'exemple précédent.

1.2.17. Règles étiquetées

Une règle de réécriture peut avoir une étiquette qui appelle l’action ou l’événement qui change l’état. Dans notre exemple, une règle étiquetée s’écrit :

birth_day : person(X,N) => person(X,N+1)

ou

```
getting_older : person(X,N) => person(X,N+1).
```

L'étiquette n'influence pas le calcul/déduction dans la logique de réécriture.

1.2.18. Règles conditionnelles de réécriture

Les règles conditionnelles de réécriture peuvent avoir des conditions très générales impliquant des équations, des appartenances, et autre réécritures ; c'est-à-dire, dans leur notation mathématique elles peuvent être de la forme :

```
l: t  => t'  if  (/\i Ui = Vi)  /\  (/\j Wj : Sj)  /\ (/\k Pk → Qk)
```

Dans leur représentation en logique de réécriture, les règles conditionnelles sont déclarées avec la syntaxe :

```
crl [<etiquette > ]: < Term-1 > => < Term-2 >
 if <Condition-1 /\ .. /\ <Condition-k> [<StatementAttributes >] .
```

Comme dans les équations conditionnelles, la condition peut se composer d'une expression simple ou peut être une conjonction avec le connectif associatif /\. Cependant les conditions sont plus générales, puisqu'en plus des équations et des appartenances, elles peuvent également contenir des expressions de réécriture, pour lesquelles la syntaxe concrète **`t => t'`** est employée. Les équations, les appartenances, et les réécritures peuvent être entremêlées dans n'importe quel ordre. Quant aux théories fonctionnelles, certaines des équations en condition peuvent être des équations d'affectation ou des équations fonctionnelles booléennes abrégées.

2. L'approche par couches de la formalisation de ArchWare ARL en logique de réécriture

Avant de présenter l'approche adoptée pour donner une sémantique formelle au *Refiner* en logique de réécriture, nous introduisons brièvement la démarche théorique de la représentation de *ArchWare ARL* en logique de réécriture.

2.1. Approche théorique

Nous représentons le langage ARL dans la logique de réécriture à travers la correspondance équationnelle et de réécriture :

$$\Psi : \text{ARL} \rightarrow \text{RWLogic}$$

Grâce à la réflexivité, on peut internaliser une telle correspondance en utilisant la sorte *Théorie*. On peut réifier la représentation Ψ ci-dessus par la définition d'un type de données abstrait *Théorie* $_{ARL}$ représentant les théories de ARL.

$$\overline{\Psi}: \text{Théorie}_{\text{ARL}} \rightarrow \text{Théorie}$$

Dans ce cas, on peut exécuter ARL dans la logique de réécriture [RoM 03]. Comme la logique de réécriture est réflexive, on peut écrire :

$$R : \text{Théorie}_{\text{ARL}} \rightarrow \text{Théorie}_{\text{ARL}}$$

R permet de réécrire une spécification de ARL en lui-même. Nous exploitons cette propriété au niveau du raffinement.

2.2. Approche par couches

Pour la construction du *Refiner* dans la logique de réécriture, nous avons choisi une approche par couches, dans laquelle chaque couche est représentée par une ou plusieurs théories. Dans cette approche, les services fournis par une couche sont directement dépendants des services fournis par la couche inférieure. Ainsi, comme le montre la figure Fig. IV.1., *Architecture* est la couche la plus profonde, en fait le cœur du *Refiner*. Les services de ce dernier sont construits autour du noyau.

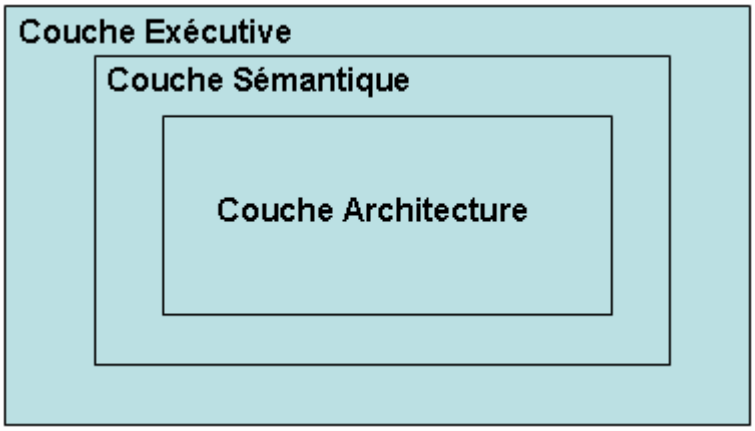


Figure VI.1. : Approche par couches

Nous pouvons distinguer trois couches :

- *La couche Architecture :* c'est une représentation des éléments architecturaux *de ArchWare ARL* dans la logique de réécriture. Elle est composée d'une seule théorie appelée *Architecture.*

- *La couche Sémantique :* c'est une spécification sémantique de *ArchWare ARL* dans la logique de réécriture. Cette couche est une extension de la couche précédente et est composée de 3 théories : *Extractor, Preconditions et PostConditions* qui expriment formellement la sémantique de *ArchWare ARL* en termes de *pré-conditions* et *post-conditions*. En particulier, toutes les actions de base de raffinement seront exprimées dans ces théories.

- *La couche Exécutive :* c'est une spécification exécutable de *ArchWare ARL* dans la logique de réécriture. Cette couche est une extension de la couche précédente et est composée d'une seule théorie appelée *Refiner*. Celle-ci est une théorie système permettant d'appliquer et d'exécuter toutes les actions de raffinement architectural.

Il est important de noter qu'un niveau de services ne peut pas faire appel à une couche plus basse que celle qui lui est immédiatement inférieure. Par exemple, une application se trouvant sur la couche *Exécutive* (la plus externe) n'interfère pas directement avec la couche *Architecture*. Elle appelle des services définis dans la théorie de la couche qui lui est directement inférieure, c'est-à-dire la couche *Sémantique*.

2.2.1. Méthode de construction et structures de données

La méthode de construction de l'environnement logiciel *Refiner* repose sur l'extension des théories déjà définies et l'expression de nouvelles opérations en fonction des théories étendues. La relation d'importation $\subseteq$ entre les cinq théories sus-citées est de type « extension », c'est-à-dire qu'il est possible d'étendre les théories importées.

$Architecture \subseteq Extractor \subseteq Preconditions \subseteq Postcondition \subseteq Refiner$

Quant aux structures de données, nous avons choisi pour la construction de *Refiner* les ensembles et les ensembles d'ensembles. Nous expliquons dans ce qui suit de façon macroscopique le rôle de chaque couche et, de façon microscopique à travers les théories, ses différents aspects et fonctionnalités.

2.2.2. Couche Architecture

La couche architecture comporte une seule théorie appelée *Architecture*. Cette dernière, nous permet d'introduire dans la logique de réécriture des spécifications architecturales en *ArchWare ARL* et de les analyser syntaxiquement. Ainsi, nous introduisons la signature de *ArchWare ARL* en nous basant sur les concepts : sorte, sous-sorte et opérations modulo des attributs.

Comme la logique de réécriture n'a aucune connaissance sur *ArchWare ARL,* toute la grammaire de ce dernier sera exprimée dans cette théorie (notons que les identificateurs de *ArchWare ARL* seront précédés de cotes). Pour cela les règles génériques suivantes sont appliquées lors de la construction de la théorie Architecture à partir de la grammaire EBNF du langage *ArchWare ARL* :

- Les non terminaux doivent être traduits par des sortes et/ou des sous-sortes.
- A chaque règle de production correspond une ou plusieurs opérations.

Le tableau suivant résumé les correspondances entre grammaire EBNF et logique de réécriture.

Règle de Production	**Logique de réécriture**
<NonTerminal>	sort NonTerminal .
[<NonTerminal>]	sort NonTerminal . op deferredNonTerminal -> NonTerminal.
Nombre de possibilités dans une règle de production	Est égal au nombre d'opérations (ou nombre d'opérations + 1)
Règle de production	Une opération de cette forme : op partie_droite -> partie gauche [attributs] *partie droite : présentation mix-fix de la partie droite de la règle de production* *partie gauche : sorte du NonTerminal qui représente la partie gauche de la règle de production*
<NonTerminalSet>	sort NonTerminalSet . subsort NonTerminal < NonTerminalSet . op _._ : NonTerminalSet NonTerminalSet -> NonTerminalSet [id: deferredNonTerminalSet] NB: Le point est l'opérateur de concaténation.

Afin d'illustrer des différentes correspondances, un exemple de définition d'architecture en *ArchWare ARL* et sa représentation en logique de réécriture sont donnés dans le tableau suivant :

```
    <ArchitectureDefinition>:=
                        archetype <ArchitectureName> is <ArchitecturalElement>{
                           types is { [<ADLTypesSet>] }
                           ports is { [<ArchetypePortSet>] }
                           incoming is { [< ADLConnectionSet>] }
                           outgoing is { [< ADLConnectionSet>] }
                           (behaviour is { [<ADLBehaviour>] }
                               | behaviour is compose { [ComponentsConnectorsSet] }{ [ UnifiesSet] })+
                        }
```

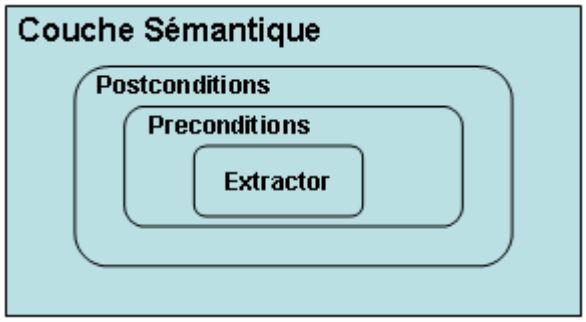
```
fmod  ARLParserArchitecture  is

sorts ArchitectureDefinition   ArchitectureName  ADLTypesSet  ArchetypePortSet  ADLConnectionSet
       ADLBehaviour ComponentsConnectorsSet UnifiesSet .

subsort   Qid < ArchitectureName .
  --- Comme ArchitectureName est un identificateur donc on fait appel à la sorte  Qid prédéfinie dans Maude

op Architecture : -> ArchitecturalElement  .
op Component: ->  ArchitecturalElement  .
op Connector : ->  ArchitecturalElement .

--- La présentation mix-fix de Architecture
op   archetype_is architecture  {types is{_} ports is{_} incoming is{_} outgoing is{_} behaviour is{_} } :
       ArchitectureName ADLTypesSet  ArchetypePortSet  ADLConnectionSet ADLConnectionSet  ADLBehaviour
              -> ArchitectureDefinition [ctor] .

op   archetype_is architecture  {types is{_} ports is{_} incoming is{_} outgoing is{_} behaviour is compose{_} } :
       ArchitectureName ADLTypesSet ArchetypePortSet ADLConnectionSet ADLConnectionSet
        ComponentsConnectorsSet   UnifiesSet
             ->ArchitectureDefinition [ctor] .

op deferredADLTypesSet :  ->  ADLTypesSet .
subsort ADLType < ADLTypesSet .
op _._ :  ADLTypesSet   ADLTypesSet  -> ADLTypesSet [assoc com id: deferredADLTypesSet] .

op deferredArchetypePortSet :  ->  ArchetypePortSet .
subsort ArchetypePort < ArchetypePortSet .
op _._ :  ArchetypePortSet  ArchetypePortSet  -> ArchetypePortSet  [assoc com id: deferredArchetypePortSet] .


op deferredADLConnectionSet : -> ADLConnectionSet .
subsort  ADLConnection < ADLConnectionSet  .
op _._ :   ADLConnectionSet ADLConnectionSet  ->  ADLConnectionSet [assoc com id: deferredADLConnectionSet] .

op deferredADLBehaviour :  ->  ADLBehaviour .

op deferredComponentsConnectorsSet :  ->  ComponentsConnectorsSet .
subsorts    ArchetypeCompDefinition  ArchetypeConnDefinition  <    ComponentsConnectorsSet .
op _._ :   ComponentsConnectorsSet  ComponentsConnectorsSet .
                    -> ComponentsConnectorsSet [assoc com id: deferredComponentsConnectorsSet] .

op deferredUnifiesSet  :  ->  UnifiesSet .
subsort   Unifies <  UnifiesSet .
op _._ :   UnifiesSet  UnifiesSet   -> UnifiesSet [assoc com id: deferredUnifiesSet] .

....

Endfm
```

Cet exemple, s'il ne présente qu'une petite partie de la grammaire du langage *ArchWare ARL*, permet néanmoins de révéler toute la difficulté de sa formalisation en logique de réécriture. Chaque possibilité supplémentaire de construction nécessite d'expliciter sa syntaxe spécifique en utilisant une nouvelle opération, même si celle-ci présentera peu de modifications par rapport à ses voisines.

Comme la logique de réécriture peut représenter plus que la syntaxe d'un langage puisqu'elle peut servir de support à la sémantique [Mes 02], nous l'avons également utilisée pour exprimer la sémantique de *ArchWare ARL* dans l'environnement *Refiner*.

2.2.3. Couche Sémantique

La couche sémantique donne une sémantique formelle dans la logique de réécriture des spécifications de *ArchWare ARL*, en particulier les actions de raffinement architectural. Cette couche est composée des théories : *Extractor, Preconditions et Postconditions.*



Couche Sémantique

2.2.3.1 Théorie Extractor

La théorie Extractor : nous permet de facilement naviguer dans les modèles architecturaux de *ArchWare ARL*. Plus précisément, elle permet d'extraire les types et les éléments architecturaux (port, incoming, outgoing, behaviour, component et connector) d'un modèle d'architecture, d'un modèle de composant ou d'un modèle de connecteur. Ces éléments sont extraits en vue de leur utilisation dans les autres théories. *Extractor* est construit à travers des opérations, des variables et des équations non-conditionnelles et conditionnelles. Dans les tableaux qui suivent, nous présentons les principales opérations ainsi que les équations correspondantes.

Pour extraire des connections de sortie d'un modèle architectural (composant, connecteur ou architecture) :

Opérations Extractor (Connections)	Explication
ExtractsOutputConnectionsArchitecture op ExtractsOutputConnectionsArchitecture : ArchetypeDefinition -> ADLConnectionsSet .	Permet d'extraire l'ensemble des connections d'une architecture. Elle prend comme paramètre la spécification d'une architecture de sorte *ArchetypeDefinition* et retourne un ensemble de connections de sorte *ADLConnectionsSet*
ExtractsOutputConnectionsComponent op ExtractsOutputConnectionsComponent : ArchetypeCompDefinition -> ADLConnectionsSet .	Permet d'extraire l'ensemble des connections d'un composant de sorte *ADLConnectionsSet*
ExtractsOutputConnectionsConnector op ExtractsPortsConnector : ArchetypeConnDefinition -> ADLConnectionsSet .	Permet d'extraire l'ensemble des connections d'un connecteur *ADLConnectionsSet*
ExtractsOutputConnectionsConstituentsArchitecture op ExtractsOutputConnectionsConstituentsArchitecture : ConstituentsName ArchetypeDefinition -> ADLConnectionsSet .	Permet d'extraire l'ensemble des connections d'un composant ou connecteur dans une architecture. Elle prends comme paramètre la spécification d'un constituant (composant ou connecteur) et retourne un ensemble de connections de sorte *ADLConnectionsSet*
ExtractsConnectionsSubSetFromConnectionsSet op ExtractsTypesSubSetFromTypesSet : NameOfADLConnectionsSet ADLConnectionsSet -> ADLConnectionsSet .	Permet d'extraire une déclaration de connection ou un sous-ensemble de déclarations de connections d'un ensemble de déclarations de connections à partir de leurs noms de sortes *NameOfADLConnectionsSet* . Elle prend comme paramètres les noms de connections et l'ensemble de déclarations de connections et retourne le sous-ensemble de déclarations de connections correspondant à ces noms.

Pour extraire des types d'un modèle architectural (composant, connecteur ou architecture), *Extractor* fournit les opérations suivantes :

Opérations Extractor (Types)	Explication
ExtractsTypesDeclarationArchitecture op ExtractsTypesDeclarationArchitecture : ArchetypeDefinition -> ADLTypeSet .	Permet d'extraire l'ensemble des types d'une architecture. Elle prend comme paramètre la spécification d'une architecture de sorte *ArchetypeDefinition* et retourne l'ensemble de types de sorte *ADLTypesSet*
ExtractsTypesDeclarationComponent op ExtractsTypesDeclarationComponent : ArchetypeCompDefinition -> ADLTypeSet.	Permet d'extraire l'ensemble des types d'un composant
ExtractsTypesDeclarationConnector op ExtractsTypesDeclarationConnector : ArchetypeConnDefinition -> ADLTypeSet.	Permet d'extraire l'ensemble des types d'un connecteur
ExtractsTypesDeclarationsConstituentsArchitecture op ExtractsTypesDeclarationConstituentsArchitecture : ConstituentName ArchetypeDefinition -> ADLTypeSet.	Permet d'extraire l'ensemble des types d'un composant ou connecteur dans une architecture. Elle prend comme paramètre la spécification d'un constituant (composant ou connecteur) et retourne un ensemble de connections de sorte *ADLTypeSet*
ExtractsTypesSubSetFromTypesSet op ExtractsTypesSubSetFromTypesSet : TypeNameSet ADLTypeSet -> ADLTypeSet.	Permet d'extraire une déclaration de type ou un sous-ensemble de déclarations de types d'un ensemble de déclarations de types à partir de leurs noms. Elle prend comme paramètre les noms de types et l'ensemble de déclarations de typse et retourne le sous-ensemble de déclarations de types correspondant à ces noms.

En appliquant le même principe on peut extraire à partir des modèles architecturaux des ports, des composants, des connecteurs, des connections d'entrée et des comportements (cf. Annexe A).

2.2.3.2 Théorie Preconditions

La théorie Preconditions : permet d'interroger si un type ou un élément architectural ou un ensemble d'éléments architecturaux (port, type, composant, connecteur, InputConnection, OutputConnection) est présent (*includes?*) ou non (*excludes?*) dans un modèle d'architecture, un modèle composant ou un modèle connecteur. Cette théorie présente la partie pré-conditions des actions de base de raffinement. Elle est construite à travers des variables, des opérations et des équations conditionnelles et non-conditionnelles modulo les équations de *Extractor*. Chaque opération retourne une valeur booléenne (vrai ou faux). Ceci nous permettra de construire (par composition de ces opérateurs) les préconditions des actions de raffinement. En résumé :

- *includes?* : teste la présence d'un élément dans un modèle d'architecture,
- *excludes?* ou *not (includes?)* : teste la non présence d'un élément dans un modèle d'architecture.

Opérations de la théorie Preconditions	Explication
includesTypes? op includesTypes? : TypeDeclarationADL TypeDeclarationADLSet -> Bool . op includesTypes? : TypeDeclarationADLSet TypeDeclarationADLSet -> Bool .	Permet de tester si une déclaration de type ou un Sous-ensemble de déclarations de types appartient ou non à un ensemble de déclarations de types. Elle prend comme paramètres : une déclaration de type de sorte TypeDeclarationADL (ou un ensemble de déclarations de types de sorte TypeDeclarationADLSet) et un ensemble de types et retourne une valeur booléenne de sorte *Bool*
excludesTypes?	*not (includesTypes?)* est équivalent à *excludesTypes?*
includesPorts? op includesPorts? : ArchetypeDefinitionPort ArchetypeDefinitionPortSet -> Bool . op includesPorts? : ArchetypeDefinitionPortSet ArchetypeDefinitionPortSet -> Bool .	De la même manière, cette opération permet de tester si un port ou un sous-ensemble de ports appartient ou non à un ensemble de ports
excludesPorts?	*not (includesPorts?)* est équivalent à *excludesPorts?*
includesConnections? op includesConnections? : ConnectionDeclarationADL ConnectionDeclarationADLSet -> Bool . op includesConnections? : ConnectionDeclarationADLSet ConnectionDeclarationADLSet -> Bool .	Permet de tester si une connection ou un sous-ensemble de connections appartient ou non à un ensemble de connections. Remarque : *not (includesConnections?)* est équivalent à *excludesConnections?*
includesComponents? op includesComponents? : ComponentDeclarationADL ComponentDeclarationADLSet -> Bool . op includesComponents? : ComponentDeclarationADLSet ComponentDeclarationADLSet -> Bool .	Permet de tester si un composant ou un sous-ensemble de composants appartient ou non à un ensemble de composants. Remarque : *not (includesComponents?)* est équivalent à *excludesComponents?*
includesConnectors? op includesConnectors? : ConnectorDeclarationADL ConnectorDeclarationADLSet -> Bool . op includesConnectors? : ConnectorDeclarationADLSet ConnectorDeclarationADLSet -> Bool .	Permet de tester si un connecteur ou un sous-ensemble de connecteurs appartient ou non à un ensemble de connecteurs. Remarque : *not (includesConnectors?)* est équivalent à *excludesConnectors?*

Vu que l'ensemble des déclarations de connections de l'opération *ConnectionDeclarationSet* est commutatif (cf. couche Architecture, théorie *Architecture 2.2.2*), on peut parcourir de manière récursive

l'ensemble des éléments, dans notre cas des connections, pour voir si un élément appartient à l'ensemble ou pas. Le principe est simple, on compare l'élément recherché avec le premier élément, si ce dernier ne coïncide pas avec le premier élément, on passe au deuxième élément (en éliminant le premier) et ainsi de suite. La condition d'arrêt est la fin de parcours de l'ensemble (la comparaison avec le dernier élément). C'est ce qui est exprimé dans les équations attachées à l'opération *includesConnections?* Pour décider si la connection appartient ou non à un architecture.

Pour décider si une connection (ou ensemble de connections) appartient à une architecture on fait appel aux services de l'opération *extractsConnectionsArchitecture* pour extraire les déclarations de connections de l'architecture puis on applique l'opération *includesConnections?* pour décider si la connection appartient ou non à une architecture.

Pour décider si une connection (ou ensemble de connections) appartient à un composant ou un connecteur dans une architecture alors il faut appeler les deux opérations *extractsConstituentsArchitecture* et *extractsConnectionsArchitecture* puis l'on applique l'opération *includesConnections?* aux éléments extraits pour décider si la connection appartient ou non au composant ou connecteur.

2.2.3.3 Théorie Postconditions

La théorie Postconditions : permet de construire en logique de réécriture toutes les actions de base de raffinement des modèles d'architecture, composant et connecteur. Cette théorie fait appel à toutes les théories précédentes. Les transformations sont implémentées à travers des variables et des équations conditionnelles modulo les opérations de la théorie *Preconditions*.

Rappelons qu'il y a trois types d'opérations liées aux actions de raffinement (cf. chapitre 3) :

Les opérations de raffinement similaires sur tous les éléments architecturaux :

- ***Add*** : ajouter un élément ou un sous-ensemble d'éléments dans un ensemble,
- ***Remove*** : supprimer un élément ou un sous-ensemble d'éléments dans un ensemble,
- ***Replace*** : remplacer un élément par un autre équivalent,
- ***Becomes*** : remplacer un sous-ensemble d'éléments par un autre sous-ensemble d'éléments.

Les opérations de raffinement spécifiques aux composants et connecteurs :

- ***Explodes*** : exploser des composants (ou connecteurs) sous forme de plusieurs composants (ou connecteurs) dans une architecture, composant ou connecteur,
- ***Implodes*** : imploser des composants (ou connecteurs) sous forme de plusieurs composants (ou connecteurs) dans une architecture, composant ou connecteur.

Les opérations de raffinement spécifiques aux connections :

- ***Unifies*** : pour unifier des connections ou des ports
- ***Separates*** : pour séparer des connections ou des ports

La formalisation de *ArchWare ARL* comporte 240 opérations, 268 équations et 48 règles de réécriture (cf. Annexe A). Par conséquent, nous ne présentons dans ce qui suit que le sous-ensemble suivant des actions de base pour le raffinement architectural :

- ajout et suppression de connections de sortie d'une architecture,
- ajout et suppression de connections d'entrée d'une architecture,
- remplacement de connections d'entrée et de sortie d'une architecture,
- transformation de connections d'entrée et de sortie d'une architecture,
- explosion et implosion de composants dans une architecture.

Pour chacune de ces actions, nous présentons d'abord sa définition en ArchWare ARL puis sa traduction dans la logique de réécriture. Les règles de traduction sont les suivantes :

- Les *préconditions* seront mises dans la partie condition des équations conditionnelles.
- Les *postconditions* se font par construction des équations dans la partie équationnelle des équations conditionnelles.

Action pour l'ajout d'une connection de sortie dans une architecture

```
on a : architecture action addOC is refinement (
                p : port, oc : connection ) {
                pre is { a::p::outgoing excludes? oc }
                post is { a::p::outgoing includes? oc }
 } as { a ::p :: outgoing includes oc }
```

```
vars  CL CL1  CL2:  ConnectionDeclarationADL .
vars CLS1 CLS2 : ConnectionDeclarationADLSet .

op  addOC : ConnectionDeclarationADL ConnectionDeclarationADLSet ->  ConnectionDeclarationADLSet   .
ceq addOC(CL1, CLS1) = CLS1    if includesConnections?(CL1, CLS1) .
ceq addOC(CL1,CLS1) = (CLS1 . CL1)    if not (includesConnections?(CL1, CLS1)) .

op  addOC : ConnectionDeclarationADLSet ConnectionDeclarationADLSet-> ConnectionDeclarationADLSet .
eq  addOC(deferredConnections, CLS1) = CLS1 .
eq  addOC((CL2 . CLS2), CLS1) =   addOC(CLS2,addOC(CL2, CLS1)) .
```

L'opération *addOC* prend comme argument une déclaration de connection de sortie *CL1* de sorte ConnectionDeclarationADL ou un ensemble de déclarations de connections de sortie *CLS2* de sorte ConnectionDeclarationADLSet à rajouter dans un ensemble de déclarations de connections de sortie *CLS1* de sorte ConnectionDeclarationADLSet qui représente le deuxième argument, et, retourne un nouvel ensemble de déclarations de connections de sortie qui englobe le tout.

L'opération *addOC* et ses équations permettent de rajouter une déclaration de connection de sortie ou un ensemble de déclarations de connections de sortie dans une architecture. Notons que nous avons utilisé deux fois l'opération *addOC* avec des arguments différents de façon naturelle comme le permet la logique de réécriture. Nous utilisons le mécanisme de récursivité dans l'équation : *eq addOC((CL2, CLS2), CLS1) = addOC(CLS2,addOC(CL2, CLS1))* pour rajouter une connection de sortie *CL2* (de même pour chaque élément de *CLS2*) dans l'ensemble des connections *CLS1*.

Avant d'appliquer l'opération *addOC*, nous faisons appel à l'opération *ExtractsOutputConnections* qui permet d'extraire les connections de sortie d'une architecture. Dans les équations, les préconditions *(pre is {…})* utilisent l'opération *includesConnections?* définie dans la théorie *Preconditions*. Les *postconditions* se font par construction, une fois que l'action est appliquée.

Action pour la suppression d'une connection (de sortie) dans une architecture

```
on a : architecture action removeOC is refinement (
                p : port, oc : connection ) {
                pre is { a::p::outgoing includes? oc and a::behaviour::connections excludes? p::oc }
                post is { a::p::outgoing excludes? oc }
    } as { a::p::outgoing excludes oc }
```

```
var  CL' : ConnectionDeclarationADL .
vars CLS CLS' : ConnectionDeclarationADLSet .

op  removeOC : ConnectionDeclarationADL ConnectionDeclarationADLSet -> ConnectionDeclarationADLSet .
eq  removeOC(CL,deferredConnections) = deferredConnections .
ceq removeOC(CL', (CL . CLS)) = remove(CL', CLS)    if ((CL' == CL) and (includesConnections?( CL',CLS)) .
ceq removeOC(CL', (CL . CLS)) = (CL . remove(CL',CLS))   if ((CL' =/= CL)
                                                          and not (includesConnections?( CL',CLS)) .

op  removeOC : ConnectionDeclarationADLSet ConnectionDeclarationADLSet
                                          -> ConnectionDeclarationADLSet .
eq  removeOC(deferredConnections, CLS) = CLS .
eq  removeOC((CL' . CLS'), CLS) = remove(CLS', remove(CL', CLS)) .
```

L'opération *removeOC* prend comme argument : une déclaration de connection de sortie *CL* de sorte ConnectionDeclarationADL ou un ensemble de déclarations de connections de sortie *CLS'* de sorte ConnectionDeclarationADLSet à supprimer de l'ensemble de déclarations de connections de sortie *CLS* de sorte ConnectionDeclarationADLSet qui représente le deuxième argument, et , retourne un nouvel ensemble de déclarations de connections de sortie qui prend en compte la suppression.

L'opération *removeOC* et ses équations permettent de supprimer une déclaration de connection de sortie ou un ensemble de déclarations de connections de sortie dans une architecture. Les préconditions *(pre is {...})* utilisent l'opération *includesConnections?* définie dans la théorie *Preconditions*. Les *postconditions* se font par construction, une fois que l'action est appliquée. Avant d'appliquer l'opération *removeOC*, nous faisons appel à l'opération *ExtractsOutputConnections* qui permet d'extraire les connections de sortie d'une architecture.

Action pour le remplacement d'une connection de sortie dans une architecture

```
on a : architecture action replaceOC is refinement (
                p : port, oc : connection, nc : connection ) {
                pre is { a::p::outgoing includes? oc and a::p::outgoing excludes? nc }
                post is { a::p::outgoing excludes? oc and a::p::outgoing includes? nc }
 } as { a::p::outgoing replaces oc by nc }
```

```
op  replaceOC: ConnectionDeclarationADL ConnectionDeclarationADL
                                        ConnectionDeclarationADLSet -> ConnectionDeclarationADLSet .

eq  replaceOC(CL1,deferredConnections,(CL1 . CLS1)) = CLS1 .
ceq  replaceOC(CL1, CL2,(CL1 . CLS1)) = (CL2 . CLS1)
                        if (CL2 =/= CL1) and  not(includesConnections?(CL1,CLS1))  .

ceq  replaceOC(CL1, CL2, CLS1) = CLS1
  if (CL2 =/= CL1) and not(includesConnections?(CL1,CLS1))  .
ceq replaceOC(CL1, CL2, (CL1 . CLS1)) = (CL1 . CLS1)   if (CL2 == CL1) .
```

L'opération *replaceOC* et ses équations permettent de remplacer une déclaration de connection de sortie par une autre dans une architecture. Les préconditions *(pre is {...})* utilisent l'opération *includesConnections?* et sa négation définies dans la théorie *Preconditions*. Les postconditions se font par construction, une fois que l'action est appliquée. Avant d'appliquer l'opération *replaceOC*, nous faisons appel à l'opération *ExtractsOutputConnections* qui permet d'extraire les connections de sortie d'une architecture.

Action pour la transformation d'un ensemble de connections de sortie dans une architecture

```
on a : architecture action transformTypeArchRef is refinement (et : set[type], rt : set[type] ) {
                pre is { a::types includes? et and a::types excludes? rt }
                post is { a::types excludes? et and a::types includes? rt }
 } as { a::types where { et } becomes { rt }
```

```
op becomesOC : ConnectionDeclarationADLSet ConnectionDeclarationADLSet
                                           ConnectionDeclarationADLSet -> ConnectionDeclarationADLSet .
ceq becomesOC(CLS1,CLS2,CLS3) = removeOC(CLS1,addOC(CLS2,CLS3))
 if (includesConnections?(CLS1,CLS3))
              and not (includesConnections?(CLS2,CLS3))  .
```

L'opération *becomesOC* et ses équations permettent de remplacer un sous-ensemble de connections de sortie par un autre dans une architecture.

Action pour l'explosion de composants dans une architecture

```
on a : architecture action explodeComponentArchRef is refinement (c : component ) {
                pre is { a::components includes? c }
                post is { a::components excludes? c  and
                          a::components includes? c::components and
                          a::connectors includes? c::connectors }
 } as { a::components explodes c }
```

```
var Q : Qid .  --- identificateur qui présente le nom du composant à exploser dans l'architecture
var  TD  : TypesDeclarationADL .
var  PD : PortsDeclarationADL .
var ID : IncomingconnectionsDeclarationADL .
var OD : OutgoingconnectionsDeclarationADL .
var ADCL :  ArchetypeDefinitionConstituentsSet .

op AComponentsExplodes :
     ArchetypeDefinition Qid  -> ArchetypeDefinitionConstituentsSet .
ceq AComponentsExplodes(
       archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q)
 =     addComponents (ExtractsSubComponentSet(Q,ADCL),
       removeComponents( ExtractsComponent(Q,ADCL),ADCL)))
            if  IncludesComponents?
        (archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q) .

ceq AComponentsExplodes(
    archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q) = ADCL
 if  not(IncludesComponents?
     (archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q)) .
```

L'opération *AcomponentsExplodes* prend comme arguments une architecture de sorte *ArchetypeDefinition* et le nom du composant *Q* de sorte *Qid* à exploser. Elle retourne les sous-composants du composant Q qui seront rajoutés à l'architecture.

En effet, dans un premier temps nous utilisons l'opération ExtractsSubComponentSet(Q,ADCL) pour extraire les sous-composants (du composant à exploser) et l'opération ExtractsComponent(Q,ADCL)pour extraire le composant de la liste des composants de l'architecture, puis nous faisons appel à l'opération *addComponents* pour ajouter les sous-composants à l'architecture en excluant le composant composite lui-même par l'opération *removeComponents*. De la même manière que pour les autres actions de raffinement, nous faisons appel à l'opération *includesComponents?* pour vérifier les préconditions.

Action pour l'implosion de composants dans une architecture

```
on a : architecture action implodeComponentArchRef is refinement (
                sc : set[constituent], nc : component ) {
                pre is { a::constituents includes? sc and
                         sc::outgoing includes? nc::outgoing and
                         sc::incoming includes? nc::incoming }
                post is {  a::components includes? nc and
                           nc::behaviour is? sc and
                           a::constituents excludes? sc }
} as { a::components implodes { sc } as nc }
```

```
var  C : Contenu .
var  ADCL : ArchetypeDefinitionConstituentsSet .
var CD : ComponentDef .

op AComponentsImplodes : Contenu  ArchetypeDefinitionConstituentsSet  ComponentDef ->
ArchetypeDefinitionConstituents .

eq AComponentsImplodes(C,ADCL,CD) = archetype CD is component { behaviour is {
(ExtractsComponentsSet(C,ADCL) . ExtractsConnectorsSet(C,ADCL)) } } .


op AImplodes : Contenu  ArchetypeDefinitionConstituentsSet  ComponentDef ->
ArchetypeDefinitionConstituentsSet .
ceq AImplodes(C,ADCL,CD) =
addComponents((removeComponents(projectCConstituents(AComponentsImplodes(C,ADCL,CD)),
ADCL)),AComponentsImplodes(C,ADCL,CD))
  if isAConstituents?(C,ADCL) .

ceq AImplodes(C,ADCL,CD) = ADCL
  if not(isAConstituents?(C,ADCL)) .
```

L'opération *AComponentsImplodes* permet d'imploser des composants dans un composant dans l'architecture. Elle prend comme arguments les noms des composants à imploser de sorte *Contenu*, l'ensemble des composants de l'architecture *ArchetypeDefinitionConstituentsSet* et le nom du composant où seront implosés ces composants. Elle retourne une nouvelle architecture avec une nouvelle structure, c'est-à-dire, que les composants de l'architecture deviennent eux-mêmes des sous-composants d'un composant de cette architecture. Nous utilisons *addComponnents* et *removeComponents* pour effectuer cette opération. En effet, nous rajoutons les sous-composants dans un composant que nous incluons directement dans l'architecture. Les sous-composants seront supprimés de l'architecture.

L'annexe A contient la formalisation complètes des différentes actions de raffinements de *ArchWare ARL*.

2.2.4. Couche Exécutive

La couche Exécutive comporte une seule théorie système nommée *Refiner*. Cette théorie représente la partie dynamique de l'environnement logiciel que nous proposons. En effet, d'un point de vue formel, le raffinement d'une architecture abstraite en une architecture plus concrète se fait à travers des règles de réécriture conditionnelles et non conditionnelles modulo les équations de la couche sémantique. Des exemples d'application de règles de raffinements sont présentés dans ce qui suit.

Action à appliquer pour l'ajout des connections de sortie dans une architecture

archetype *architectureRefId* ***refines*** *architectureDefId*
using { p_0::outgoing includes *(oc_0 .... . oc_n)*
}

```
var OCLN : ConnectionDeclarationADLSet .

op archetype_refines_using{outgoing includes(_)} in_ :
            ArchitectureDef ArchitectureDef
          ConnectionDeclarationADLSet  ArchetypeDefinition  -> RefAppli .

crl[AddingOutputConnectionsArch] :
     archetype A2 refines A1 using{  outgoing includes(CL1)  }
in (archetype A3 is architecture {AT ports is {APS1} AID outgoing is{OCLN} ABD}) =>  archetype A2 is
architecture { AT ports is {APS1} AID outgoing is {addCO(OCLN,CL1) } ABD } if (A1 == A3) .
```

Pour appliquer l'action de raffinement pour l'ajout de connections de sortie, il suffit de réécrire cette architecture sous la forme d'une autre en appliquant l'opération prédéfinie *addCO*.

Ce raffinement est fait de manière simple à travers la règle de réécriture étiquetée par *crl[AddingOutputConnectionsArch]* qui permet de rajouter une connection de sortie dans la partie « outgoing », c'est-à-dire dans l'ensemble des connections de sortie de l'architecture. …

Action à appliquer pour la suppression des connections de sortie dans une architecture

archetype *architectureRefId* ***refines*** *architectureDefId*
using { p_0::outgoing excludes *(oc_0 .... . oc_n)*
}

```
op archetype_refines_using{ outgoing excludes(_)} in_ :
    ArchitectureDef ArchitectureDef  Contenu
                ArchetypeDefinition  -> RefAppli .

crl[RemovingOutputConnectionsArch] :
      archetype A2 refines A1 using{  outgoing excludes(C1)  }
  in (archetype A3 is architecture {AT ports is {APS1} AID outgoing is{OCLN} ABD}) =>  archetype A2 is
architecture { AT ports is {APS1} AID outgoing is{ removeOC(isAConnectionsSet(C1,OCLN),OCLN)} ABD }
   if (A1 == A3)  .
```

De la même manière que pour le rajout de connections de sortie, pour appliquer l'action de raffinement pour la suppression de connections de sortie, il suffit de réécrire cette architecture sous la forme d'une autre en appliquant l'opération prédéfinie *removeOC*. La règle de réécriture étiquetée par *crl[RemovingOutputConnectionsArch]* permet de le faire.

Action à appliquer pour le remplacement d'une connection de sortie dans une architecture

archetype *architectureRefId* ***refines*** *architectureDefId*
using { p_0***::outgoing replaces*** oc_0 ***by*** nc_0
}

```
op archetype_refines_using{ outgoing replaces_by_} in_ :
  ArchitectureDef ArchitectureDef  Qid  ConnectionDeclarationADL
                ArchetypeDefinition  -> RefAppli .

rl[ReplacingOutputConnections] :
    archetype A2 refines A1 using{  outgoing replaces Q1 by CC1  }
  in (archetype A3 is architecture {AT ports is {APS1} AID outgoing is{OCLN} ABD}) =>  archetype A2 is
architecture { AT ports is {APS1} AID outgoing is{replaceOC(isAConnectionsSet({Q1},OCLN),CC1,OCLN)}
ABD } if (A1 == A3)  .
```

Pour appliquer l'action de raffinement pour le remplacement des connections de sortie, il suffit de réécrire cette architecture sous la forme d'une autre en appliquant l'opération prédéfinie *replaceCO*. La règle de réécriture étiquetée par *crl[RplacingOutputConnectionsArch]* permet de le faire.

Action à appliquer pour la tranformation des connections de sortie dans une architecture

archetype *architectureRefId* ***refines*** *architectureDefId*
using { p_0***::outgoing where {*** oc_{00}***, ..., ***oc_{0n} ***} becomes {*** roc_{00}***, ..., ***roc_{0n} ***}***
}

```
op archetype_refines_using{outgoing where{_}becomes{_}} in_ :
        ArchitectureDef ArchitectureDef
   Contenu ConnectionDeclarationADLSet  ArchetypeDefinition  -> RefAppli   .

 rl[becomesConnectionDeclarationsArch] :
 archetype A2 refines A1 using { outgoing  where {CC} becomes {CL1}}
   in (archetype A3 is architecture {AT ports is {APS1} AID outgoing is {OCLN} ABD})  =>  archetype A2 is
architecture { AT ports is {APS1}  AID outgoing is {becomes(isAConnectionsSet(CC,OCLN),CL1,OCLN)} ABD}
if (A1 == A3)  .
```

De manière similaire à celle des autres actions, nous faisons appel à l'opération *becomes* et la règle de réécriture correspondante.

Action à appliquer pour exploser des composants dans une architecture

archetype *architectureRefId* ***refines*** *architectureDefId*
using { components explodes (c_0 **...** . c_n **) }**

```
op archetype_refines_using{components explodes_} in_ :
   ArchitectureDef ArchitectureDef  ComponentDef
                   ArchetypeDefinition  -> RefAppli  .

rl[explodingComponents] :
    archetype A2 refines A1 using{components explodes AQ}
   in (archetype A3 is architecture {AT APD AID AOD behaviour is {AADCL} })
=>
archetype A1 is architecture {AT APD AID AOD behaviour is {
 (AComponentsExplodes(archetype A2 is architecture {AT APD AID AOD behaviour is {AADCL} } ,AQ))} }
if (A1 == A3)  .
```

De manière similaire à celle des autres actions, nous faisons appel à l'opération *AExplodes* et la règle de réécriture correspondante.

Action à appliquer pour imploser des composants dans une architecture

archetype *architectureRefId* ***refines*** *architectureDefId*

using { components implodes { c_{00}, ..., c_{0m} **} as** cc_0 **}**

```
op archetype_refines_using{components implodes_as_} in_ :
       ArchitectureDef ArchitectureDef Contenu
                  ComponentDef ArchetypeDefinition  -> RefAppli  .

rl[implodingComponents] :
   archetype A2 refines A1 using{components implodes AC as AQ}
    in (archetype A3 is architecture {AT APD AID AOD behaviour is {AADCL} })
       => archetype A2 is architecture {AT APD AID AOD behaviour is
                 {AImplodes(AC,AADCL,AQ)} } if (A1 == A3) .
```

De manière similaire à celle des autres actions, nous faisons appel à l'opération *AImplodes* et la règle de réécriture correspondante.

3. Conclusion

Les travaux menés sur le développement centré-architecture portent essentiellement sur la spécification formelle des architectures logicielles. Le raffinement des architectures logicielles en général, et plus particulièrement, des outils logiciels pour le support du raffinement formel d'architectures restent un point ouvert de recherche. En effet, d'une part et à notre connaissance, en dehors de RAPIDE et de SADL, aucun outil fondé sur les termes architecturaux (i.e. données, comportements, connections et structures) n'a été proposé jusqu'à maintenant. D'autre part, dans ces mêmes approches, un seul outil logiciel est proposé à l'utilisateur. Dans RAPIDE, le raffinement est supporté seulement par un outil de simulation offrant des mécanismes de mise en correspondance a posteriori de deux architectures à travers leur exécution. Par ailleurs aucune primitive (*action de raffinement*) n'est à proprement parler fournie pour le raffinement architectural et seul le raffinement des comportements est supporté par simulation. SADL, conçu spécialement pour le raffinement des architectures logicielles, est également supporté par un seul outil d'interprétation de patrons de raffinement qui met en correspondance différentes architectures avec la contrainte que l'architecture concrète soit une implantation fidèle de l'architecture abstraite. Ceci rend le raffinement rigide dans le sens où il ne permet pas de différer des décisions de conception. Contrairement à RAPIDE, le raffinement dans SADL est uniquement structurel.

L'environnement *Refiner* que nous proposons repose sur les actions de raffinement de base fournies par le langage *Archware ARL* ainsi que sur le modèle de processus de raffinement dédié à ce dernier. Le choix de la logique de réécriture comme fondement à la fois de la sémantique formelle et de la sémantique opérationnelle de *Refiner* permet la description d'architectures logicielles de systèmes concurrents, leur raffinement par le mécanisme de réécriture ainsi que leur exécution grâce aux outils proposés pour cette logique.

Chapitre 5 : Implémentation de l'environnement Refiner

Chapitre 5 : Implémentation de l'environement Refiner

Nous présentons dans ce chapitre l'architecture de l'environnement *Refiner* ainsi que les implémentations réalisées. En particulier, nous présenterons les composants logiciels *RefinerTool* et *Sigma* qui nous permettent de progressivement raffiner une architecture logicielle décrite en *ArchWare ARL* jusqu'à la génération de l'application dans le langage de programmation ProcessWeb/PML [War 89] [BGRSW 94] [ICL 96].

1. Architecture d'implémentation de *Refiner*

Le *Refiner* en tant qu'environnement logiciel centré-architecture, rappelons-le, permet à un architecte de raffiner ses modèles architecturaux jusqu'à la génération d'application dans un langage d'implémentation (cf. Fig. V.1.). Ceci grâce à l'application sur les modèles architecturaux des actions de raffinement présentées et formalisées dans les chapitres précédents.

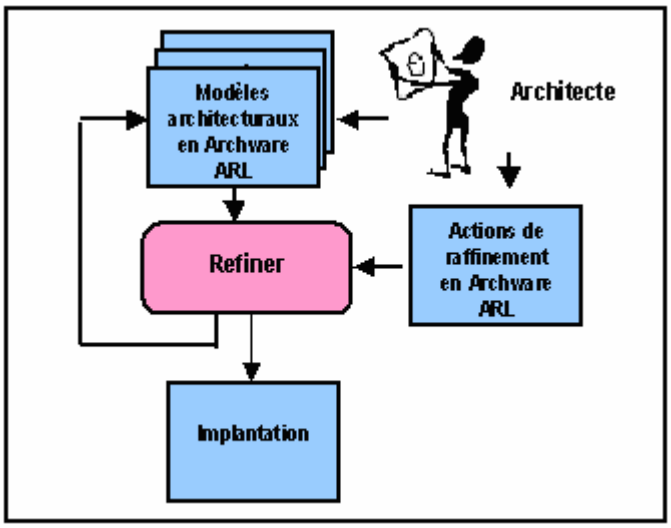


Figure V.1. : L'environement Refiner du point de vue utilisation

Du point de vue implémentation, l'environnement *Refiner* comporte deux composants principaux :

- *RefinerTool* lié directement au raffinement des modèles architecturaux,
- *Sigma* lié à la génération d'applications à partir d'architectures raffinées, et ce, dans différents langages de programmation.

Comme l'indique la figure Fig. V.2., nous pouvons décrire l'architecture de *Refiner* lui-même à l'aide de deux composants (*RefinerTool* et *Sigma*) et un connecteur (*Link*) leur permettant d'interagir par le biais des connections (*fromLink* et *toLink*). Les connections restantes (*fromARL* et *code*) représentent les points d'interaction avec l'architecte de l'application.

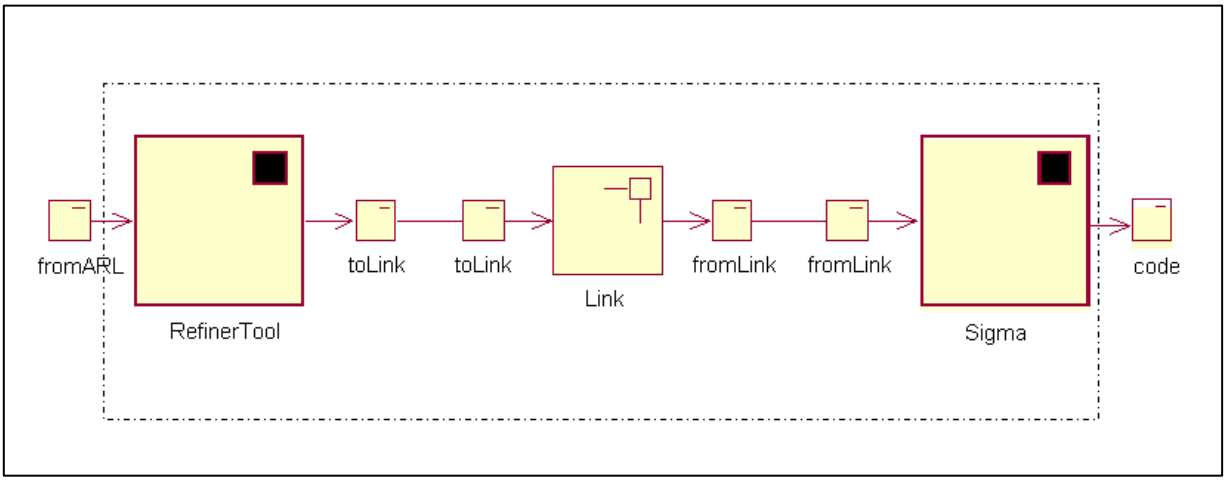


Figure V.2. : Architecture du Refiner

Le comportement de cette architecture peut être décrit comme une composition de trois comportements récursifs (ceux des deux composants *RefinerTool* et *Sigma* et du connecteur *Link*) avec dans celui de *RefinerTool*, une alternative où le choix est fait par l'architecte de soit de nouveau accomplir une étape de raffinement soit de demander la génération du code dans un langage de programmation cible. Le modèle architectural avec les différentes actions de raffinement sont envoyés à travers la connection *fromARL ;* le composant *RefinerTool* en recevant ces paramètres applique les actions de raffinement au modèle architectural et envoie le résultat à travers la connection *toLink.* Lorsque le composant *Sigma* reçoit le modèle architectural raffiné à travers la connection *fromLink*, il génère le code de l'application et renvoie le résultat via la connection *code*.

2. Le composant *RefinerTool*

Pour l'implémentation du composant *RefinerTool*, nous avons choisi l'outil Maude [CDELMMQ 99][MFSPNJC 03] fondé sur la logique de réécriture. En effet, Maude comporte une machine de réécriture exécutable qui nous a permis d'implémenter les 240 opérations, 268 équations et 48 règles de réécriture nécessitées par la formalisation du langage *ArchWare ARL* (cf. Annexe A).

Le choix de l'outil Maude est motivé par le fait qu'il soit un langage réflexif supportant à la fois la spécification et la programmation des logiques équationnelles et de réécriture pour un grand choix d'applications. Maude est influencé d'une façon importante par le langage OBJ3 [MaM 96] qui peut être vu comme un sous-langage de la logique équationnelle. Maude fournit également un filtrage très puissant et modulaire permettant de réécrire des termes modulo des combinaisons différentes de l'associativité, la commutativité, l'identité (à gauche, à droite) et l'idempotence [Mes 98]. Grâce à sa propriété de réflexivité, Maude est considéré par ses réalisateurs [CDELMMQ 99] comme étant un méta-langage [CDELM 98] et un méta-outil [CDEMO 98] pour créer des environnements exécutables destinés à différentes logiques et langages. Nous avons pu donc compiler et exécuter les différentes théories que nous avons construites pour la formalisation de *ArchWare ARL* (cf. Annexe B), lesquelles théories représentent les fonctionnalités du composant logiciel *RefinerTool*. Nous avons également construit pour ce dernier une interface utilisateur assez simple.

2.1. Utilisation du composant *RefinerTool*

L'utilisation de l'interface utilisateur du composant *RefinerTool* est très intuitive puisqu'elle est fondée sur des boutons, un explorateur et des zones de texte (cf. Fig. V.3).

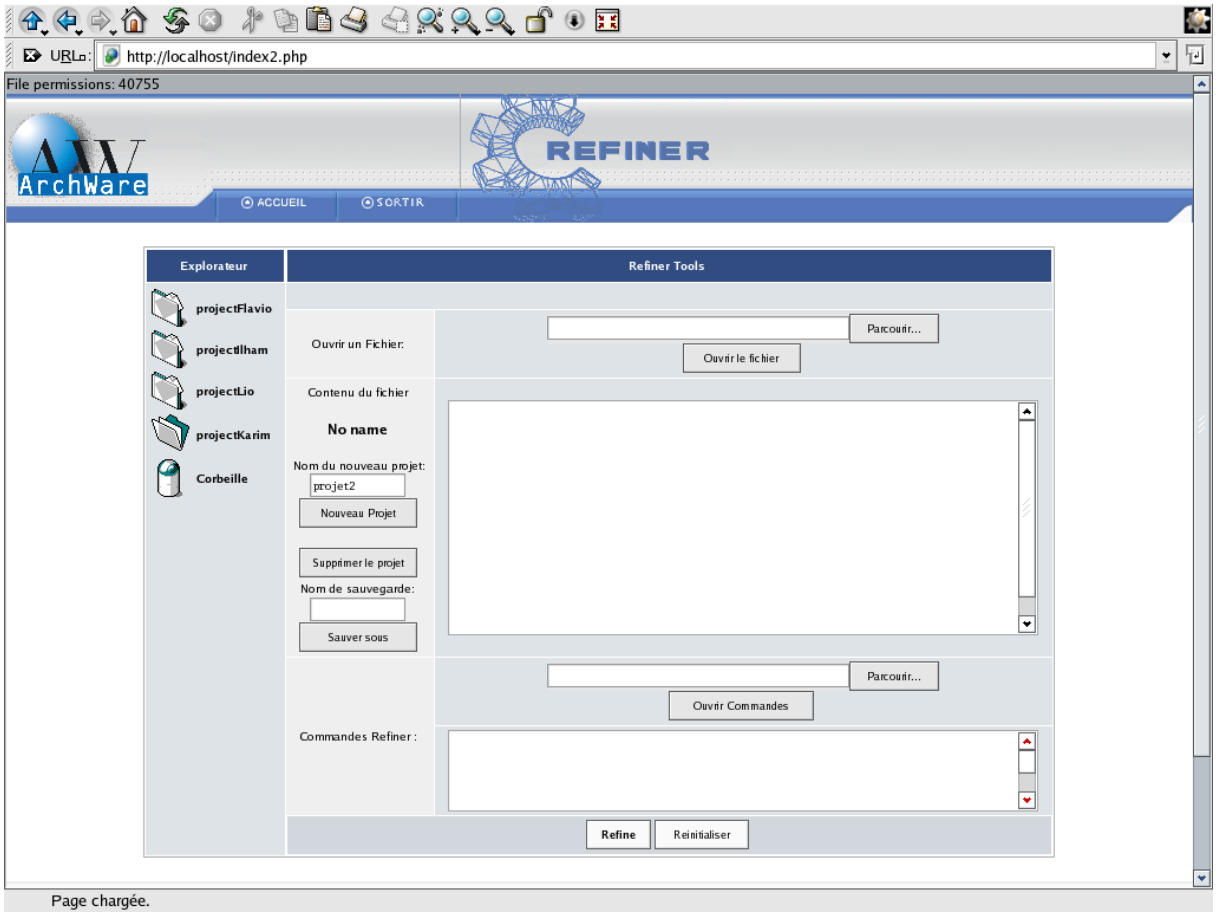


Figure V.3. : Interface graphique du RefinerTool

En effet, l'interface est constituée de diverses parties dont entre autres :

- un explorateur de fichiers,
- une partie boutons (sauver sous, supprimer, etc.),
- une partie zone de texte,
- une partie commandes.

Il existe également d'autres parties qui apparaissent dynamiquement comme l'affichage de résultats de raffinement.

Concernant les entrées au composant *RefinerTool*, elles sont de quatre types, contenues dans des fichiers manipulables à travers l'explorateur (cf. Fig. V.3.) :

- fichier source en *ArchWare ARL*.
- fichier commandes (actions de raffinement) en *ArchWare ARL*.

- fichier résultat de raffinement (généré par le bouton Refine) en *ArchWare ARL*.
- fichier de traduction (généré par le bouton translate) dans le langage de programmation.

L'ouverture d'un fichier source dans l'explorateur engendre automatiquement l'ouverture du fichier de commandes associé dans la zone de texte réservée aux actions.

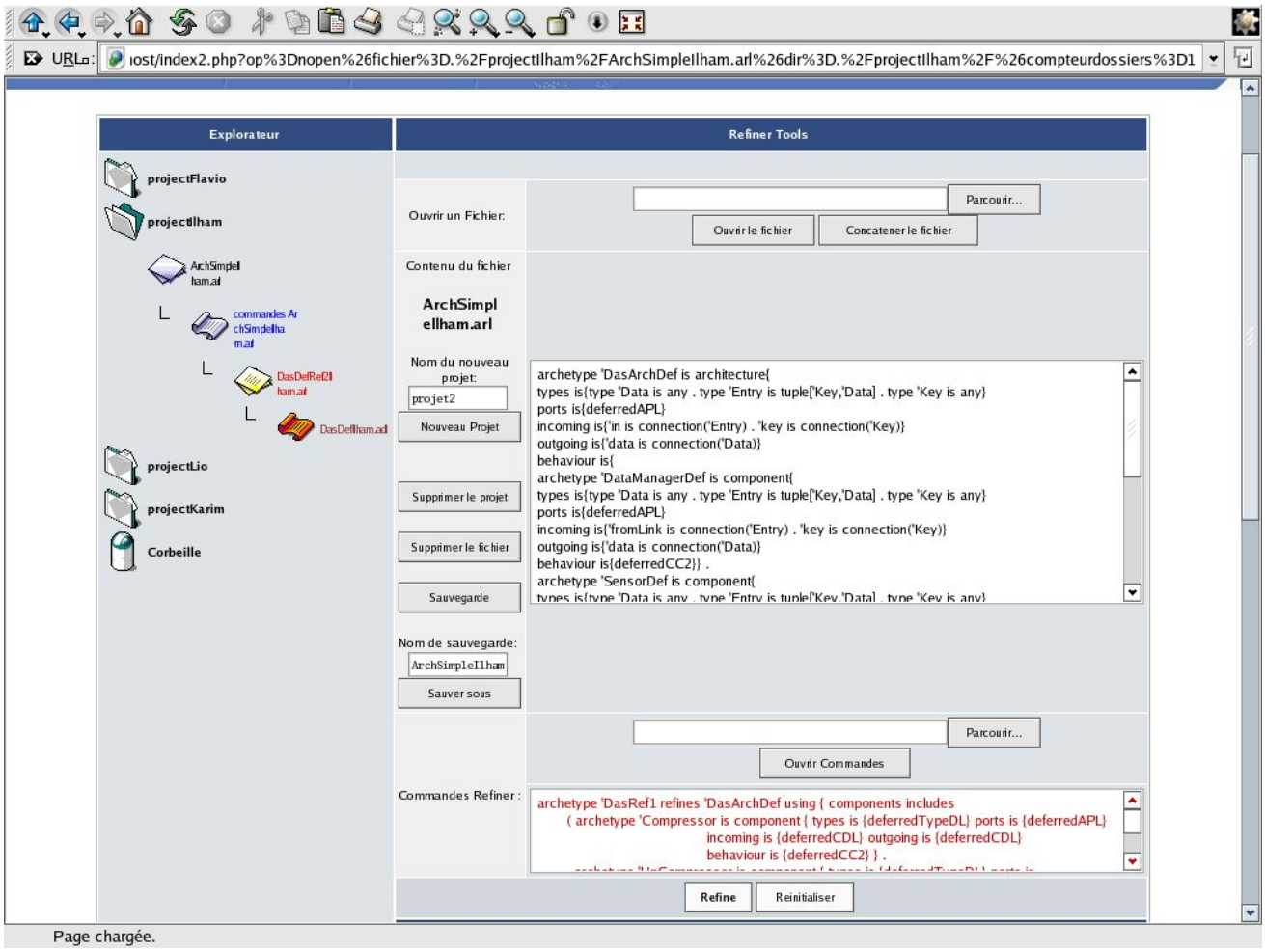


Figure V.4. : Téléchargement du modèle d'architecture et l'action de raffinement

La figure Fig. V.4. illustre un résultat de raffinement de l'architecture appelée *DasRef1* présentée dans le chapitre 3. A partir de là, l'architecte peut soit demander une génération de code (bouton *Translate*) soit accomplir un nouveau raffinement (bouton *Refine*). Le résultat de l'exécution d'une commande peut être sauvegardé à chaque fois.

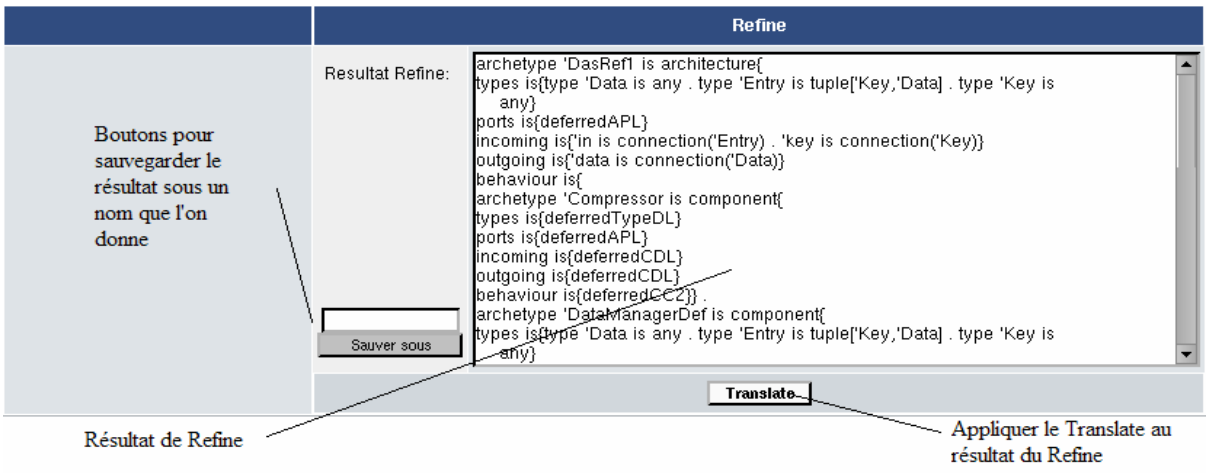


Figure V.5. : Résultat de raffinement

L'utilisation de cette interface nécessite les logiciels suivants :

- système d'exploitation Linux (Redhat 9.X)
- serveur Web (Apache)
- module PHP associé à ce serveur.

3. Le composant *Sigma* de génération d'applications

Le but final de n'importe quelle conception ou modélisation de l'architecture est de produire un système exécutable. Cependant, générer manuellement le code à partir de la spécification peut introduire des erreurs entre une architecture et son implémentation. Il est donc souhaitable sinon impératif d'introduire un outil pour la génération automatique du code à partir d'un modèle de l'architecture. Dans cette optique, nous avons conçu et développé le composant *Sigma* pour produire une bonne partie du code de l'application.

Sigma se veut un composant générique de génération d'applications. Il permet de générer à partir d'une description d'architecture le code d'une application dans divers langages cibles, à l'opposé du compilateur qui produit le code d'un seul langage cible. Par exemple, nous voudrions utiliser le même ADL dans deux applications dont le choix d'implémentation diffère (e. g. Java et PML). Au lieu de développer un compilateur ou traducteur pour chaque langage cible, nous pouvons réduire le coût et le temps en produisant à travers *Sigma* des traducteurs tout en assurant la traçabilité entre la description d'architecture et le code généré. Les principaux concepts utilisés dans Sigma sont ceux de « mapping » et de « pattern de synthèse ».

3.1. Mapping

En général, le mapping permet de relier deux architectures quelconques par une correspondance d'interprétation syntaxique. Dans notre approche le mapping, appelé aussi synthétiseur, définit une correspondance (réécriture dans la logique de réécriture) entre une architecture abstraite (spécifiée par exemple dans *ArchWare ARL*) et une architecture concrète (codée par exemple en *Processweb/PML*). La phase de mapping est comprise dans tous les patterns de synthèse que nous verrons en détail dans le paragraphe suivant. Son exécution (désignée par l'exécution de synthétiseur) conduit à une génération automatique de code cible.

Une définition de mapping est donnée par les règles de production suivantes :

<MappingDefinition>::=

define mapping <MappingName>

from <NomArchitectureAbstraite > **to** <NomArchitectureConcrete >

{ <Mapping> **where** <MappingSet > **}**

<MappingSet>::= <Mapping> **.** <MappingSet> **|** **$**

<Mapping>::= <Règle> **|** **(** <Règle >**)**

| **(**<Règle>**) * |** **(**<Règle>**) +**

<Règle> **::=** <Abstraite> → <Concrète>

<Abstraite> **::=** <Label> **|** **(** <LabelSet> **)**

Une définition de mapping supporte la génération de code exécutable. Les règles ci-dessus ont les significations suivantes :

- <Abstraite> → <Concrète> réécrit la description abstraite d'architecture <Abstraite> en description concrète d'architecture <Concrète>,
- **(**<Règle>**) *** applique la règle de réécriture <Règle> *zéro, une ou plusieurs fois*,
- **(**<Règle>**) +** : applique la règle de réécriture <Règle> *au moins une fois*.
- <Label> : est un identificateur et <LabelSet> : est un ensemble des identificateurs

3.2. Patterns de Synthèse

Les patterns de synthèse représentent les solutions réutilisables que l'on peut composer pour concevoir des applications logicielles. Dans notre approche du composant *Sigma*, un pattern est représenté par deux grammaires EBNF étiquetées et un mapping d'un modèle abstrait vers un modèle concret.

Par convention un terme entre les crochets angulaires ("<", " >") (un non terminal) qui apparaît dans les deux grammaires se rapporte au même terme. Les termes en gras sont des termes terminaux. Des étiquettes (labels) sont employées afin d'étiqueter les termes (terminaux et/ou non terminaux) dans le pattern. On peut étiqueter un ensemble de terminaux ou non terminaux par une seule étiquette en les encadrant de deux #. *Sigma* a été implanté en C (1800 lignes de code) ; le code complet est présenté dans l'annexe C.

4. Génération d'applications en *ProcessWeb/PML*

4.1. Introduction générale à ProcessWeb

ProcessWeb est un environnement logiciel qui fournit un support permettant la définition et l'exécution de modèles de processus. Un processus est la coordination des activités effectuées par des personnes dans des organisations utilisant une variété d'outils. Ce système supporte la collaboration de modèles de processus exécutables. Les utilisateurs de *ProcessWeb* sont connectés au système par un navigateur Web. A la différence d'un serveur Web, ProcessWeb identifie ses utilisateurs individuellement, et les pages Web qu'ils reçoivent sont personnalisées, même si ProcessWeb peut fournir la même information à plusieurs personnes. Les pages Web (au format HTML) représentent les points d'entrée/sortie pour *ProcessWeb*.

Quand un utilisateur se connecte à *ProcessWeb* et fournit des informations en entrée, elles sont passées au support de processus *PWI* (ProcessWise Integrator system) qui est le moteur d'exécution des processus. Il construit la page Web, qui doit être retournée, et effectue les mises à jour nécessaires de manière dynamique sur les pages des autres utilisateurs. Le cœur du *PWI* est le Process Control Manager (PCM), ou le gestionnaire de commande des processus, qui joue le rôle de serveur central. PCM interprète les descriptions des processus qui sont écrites dans le langage appelé *PML* (Process Modelling Language*)*, et prend en charge la communication avec les utilisateurs. Les descriptions des processus et leurs états seront stockées dans une mémoire persistante. Ceci signifie que le *PCM* peut être arrêté pour effectuer des tâches particulières (maintenance), ou supporter des défaillances techniques (pannes), et être relancé à partir de son état courant. Chaque processus distinct dans *ProcessWeb* est associé à un modèle exécutable écrit dans le langage *PML* présenté dans le paragraphe suivant.

4.1.1. Le langage PML

Le langage PML permet de modéliser un processus comme un réseau de *rôles* et d'*interactions* (cf. Fig. V.6.).

- les *rôles* représentent les objets actifs. Ils reçoivent des messages passés par d'autres rôles ou venant de l'extérieur du système, et éventuellement renvoient une ou plusieurs réponses. Les rôles sont définis par les utilisateurs, ou certains, par la bibliothèque de *PML*,
- les *interactions* sont effectuées par des canaux unidirectionnels et asynchrones. Si un rôle doit interagir avec un autre rôle, il aura besoin de deux interactions : une pour envoyer le message, l'autre pour recevoir la réponse. Il est possible d'émettre des rôles et des interactions à travers une interaction. Une interaction est composée d'une file de données et d'un ensemble de ports (les points des interactions) qui donnent l'accès à la file de données (messages placés dans la file). Ces interactions sont typées.

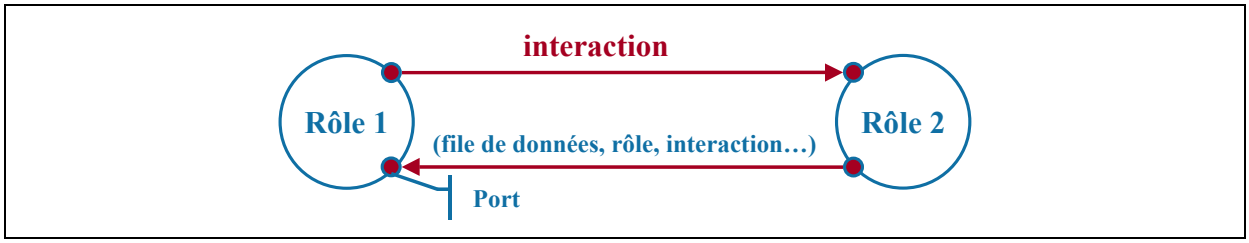


Figure V.6. : Modèle de processus en PML

PML est basé sur la notion de classes avec l'héritage simple. Il y a trois classes de base : *Entity*, *Role* et *Action*.

Les entités sont des données sur lesquelles les rôles et leurs actions opèrent. Il y a quatre classes d'entités primitives : booléen (*Bool*), entier (*Int*), réel (*Real*) et chaîne de caractères (*String*). La classe prédéfinie "*Entity*" est la superclasse de toutes les autres classes entités. Il y a deux constructeurs pouvant agréger les données dans PML : *collof* pour les collections et *tableof* pour les tableaux. Une *collection* est un ensemble de données ordonnées de même type, un *tableau* est indexé par des données de type *String*. Les propriétés des entités sont accessibles par l'utilisation de la notation *point* : (<entityname>.<propertyname>).

La définition de la classe *Role* consiste en un nombre de propriétés qui sont :

- les *ressources* définissant une donnée locale pour un rôle (il n'y a pas de donnée globale en *PML*). Certaines ressources utilisent les constructeurs *giveport* et *takeport* qui sont particulièrement intéressants, car c'est la seule façon pour un rôle de communiquer avec un autre : en donnant (envoyant) ou prenant (recevant) des messages,
- les *actions* (code) définissant ce que fait l'instance d'un rôle lorsqu'elle est exécutée. Il existe des actions prédéfinies permettant la manipulation des rôles comme *StartRole*, *FreezeRole*, *UnfreezeRole*, *ExtractRoleData*, *InsertRoleData*, etc.,

D'autres propriétés comme *initially*, *always* et *termconds* sont optionnelles.

La classe *Action* est définie pour représenter un calcul qui peut être appelé plusieurs fois avec des paramètres différents. Comme les rôles, les actions consistent en un nombre de propriétés qui sont : les paramètres *in* et *out*, les *ressources*, les *parts* (qui représentent les instructions à exécuter, comme les actions pour un rôle), les *preconds*, les *postconds*, et les *termconds*. La classe prédéfinie *Action* est la super classe de toutes les actions. Les actions prédéfinies de base qui manipulent les interactions sont les suivantes :

- *NewInteraction* (*giver = gp, take = tp*) crée une nouvelle interaction et lie les ports aux ressources *gp* et *tp*. Après l'appel de cette action, le rôle peut utiliser *gp* et *tp* pour envoyer des messages. Une fois qu'une interaction est créée, un port ou les deux ports à la fois, seront envoyés à un autre rôle qui envoie des messages aux autres rôles,
- *Give* (*interaction =gp, gram =message*) envoie un message (*gram* représente le message qui doit être envoyé) à travers *giveport gp,* .
- *Take* (*interaction = tp, gram = message*) reçoit le message (*gram* représente le message qui sera reçu). Dans le port *tp* cette action extrait les données de la file d'interactions et affecte la file par la variable *message*.

4.1.2. Architecture générale de ProcessWeb/PML

Nous donnons dans ce qui suit (cf. Fig. V.7.) une vision simplifiée de l'architecture de ProcessWeb. Celle-ci se compose de trois parties essentielles : le navigateur Web via lequel les utilisateurs se connectent, le serveur Web et le moteur de processus *PWI*.

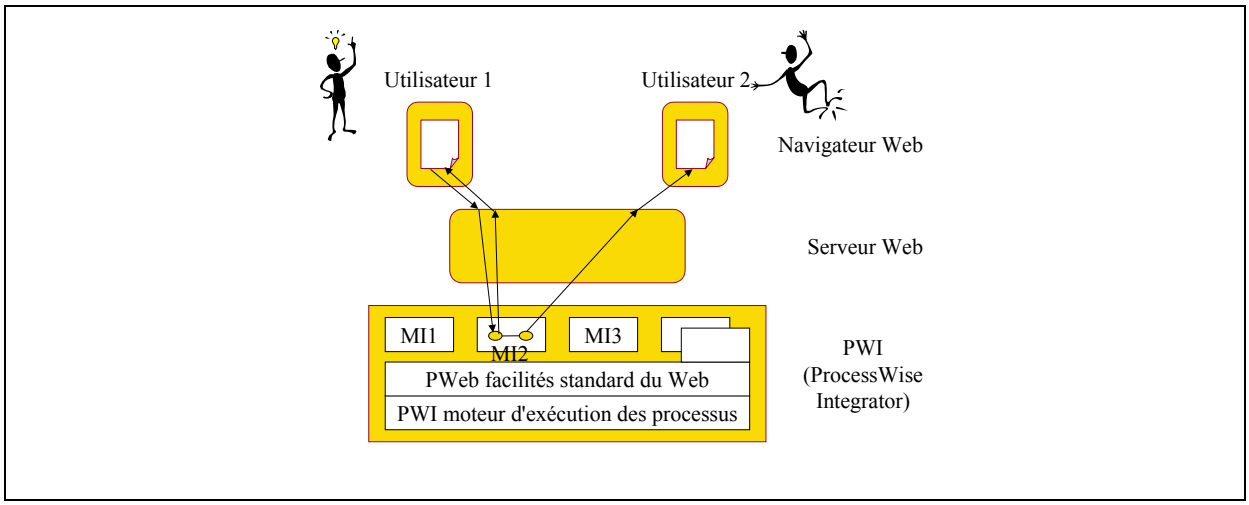


Figure V.7. : Architecture générale de ProcessWeb/PML

Cette figure est un instantané représentant deux utilisateurs reliés à un rôle du modèle de processus appelé MI2 (Model Instance 2), (c'est-à-dire reliés à un rôle d'une instance du modèle 2). Les deux utilisateurs visualisent chacun une page Web qui montre l'état actuel de l'instance du processus. Dans la situation illustrée, si *Utilisateur 1* fournit des données sous format HTML, elles seront transférées au rôle approprié. Ce dernier produira une nouvelle page Web reflétant le nouvel état et enverra celle-ci à nouveau au navigateur de l'utilisateur. De plus le rôle peut envoyer un message à un autre rôle, qui à son tour produira une nouvelle page Web qui sera envoyée à *Utilisateur 2*. Ainsi, après que *Utilisateur 1* ait envoyé des données sous format HTML, il peut se déconnecter du système, puis se reconnecter à une date ultérieure pour voir s'il y a de nouvelles activités à mettre en œuvre.

4.2. Génération de ProcessWeb/PML à partir de ArchWare ARL

La génération d'un système *ProcessWeb/PML* à partir d'une spécification en *ArchWare ARL* est vue comme une mise en correspondance d'un modèle concret avec une implémentation. Il s'agit d'une approche de synthèse où, de manière incrémentale et à partir de fragments en *ArchWare ARL*, est généré un modèle de processus en *PML*. Afin d'illustrer nos propos, nous utiliserons des fragments de l'étude de cas *DAS* (Data Acquisition System) spécifié en *ArchWare ARL* dans le chapitre 3. Nous présentons des fragments des patterns de synthèses définis pour les éléments architecturaux : *port*, *type*, *comportement*, *composant*, *connecteur* et *architecture*.

4.2.1. Synthèse de ports en *PML*

En général, dans *PML* les données échangées ainsi que les points des interactions sont définis comme des entités (classe *Entity*), ainsi un type de *port* dans *ArchWare ARL* correspond à un type *Entity* dans PML. Le pattern de synthèse que nous avons défini pour les ports est le suivant :

```
define grammar ARL_Port_Expression {
<portDeclaration> ::= archetype p1: <portName> is port {
                          incoming is { <inConnectionDeclarationSet> }
                          outgoing is { <outConnectionDeclarationSet> }
                      }
<inConnectionDeclarationSet> ::=
          <ConnectionDeclaration> . <inConnectionDeclarationSet> | deferred
<inConnectionDeclaration> ::= p2: connectionName is connection(p3 : <TypeList>)
```

```
<outConnectionDeclarationSet> ::=
        <ConnectionDeclaration> . <inConnectionDeclarationSet> |  deferred
<inConnectionDeclaration> ::=
        p4: <ConnectionName> is connection(p5:<TypeList>)
<ConnectionName> ::= identificateur
<portName>::= identificateur
}
define  grammar  PML_Entity_Expression {
<EntityDeclaration> ::=  p1: <portName>  e1: #isa Entity with { parts#
                         p2: <ConnectionName>  e2: ": giveport"  p3: <TypeList>
                         p4: <ConnectionName> e3: ": takeport"  P5: <TypeList>
                  e4:   #} end with#
define  mapping  Port_Expression
        from ARL_Port_Expression  to PML_Entity_Expression  {
    p1 → (p1, e1, e4) .    (p2,p3) → (p2, e2, p3) . (p4,p5) → (p4, e3, p5) .
    where  { (p1,(p2,p3)*, (p4,p5)*)  → (p1, e1, (p2, e2, p3)*, (p4, e3, p5)*,e4 ) } .
}
```

Dans ProcessWeb PML, un modèle est un ensemble de rôles concurrents (les composants), et un ensemble d'interactions (les connecteurs) qui sont liés entre eux. Ces interactions sont directionnelles. Un rôle qui envoie un message utilise une interaction via son *giveport.* Une connection d'entrée en *ArchWare ARL* correspondrait naturellement à un *giveport,* ceci est exprimé par le mapping *(p2,p3) → (p2, e2, p3)* alors qu'un rôle qui reçoit un message utilise un *takeport (*mapping : *(p4,p5) → (p4, e3, p5)).*

Nous pouvons également définir une interface plus complexe entre deux rôles en utilisant une entité qui collectionne ensemble un nombre de *giveport* et *takeport.* Ceci est exprimé par le mapping *where { (p1,(p2,p3)*, (p4,p5)*) → (p1, e1, (p2, e2, p3)*, (p4, e3, p5)*,e4) }.*

En nous référant à l'étude de cas *DAS* (cf. Chapitre 3), rappelons que la description suivante spécifie un port du composant *SensorDef* en *ArchWare ARL* :

```
archetype DasPort is port {
  incoming is { in is connection(Entry). key is connection(Key) }
  outgoing is { data is connection(Data) }
}
```

L'exécution du composant *Sigma* en utilisant le pattern de synthèse de port présenté précédemment génère le code PML suivant :

```
DasPort  isa Entity with
parts
    in  : giveport  Entry
    key  : giveport  Key
    data  : takeport  Data
 end with
```

où *in* et *key* sont créés par le constructeur *giveport* et permettent de fournir des messages de type *Entry* et *Key* respectivement, et, *data* est créée par le constructeur *takeport* et permet de prendre des messages de type *Data*.

4.2.2. Synthèse de types en PML

Le pattern de synthèse que nous avons défini pour la génération des types en *PML* est le suivant :

```
define grammar ARL_Types_Expression {
<ARLTypeDeclaration> ::= p1: types is { <TypeDeclarationSet> }
<TypeDeclarationSet> ::=
                    <TypesDeclaration> . <TypesDeclarationSet> } | p2: deferred
<TypeDeclaration> ::= p3 : <TypeName> is p4: <SimpleType>
<SimpleType> ::= String | Boolean | Any | Real
<ComplexType> ::= p5:tuple[p6: <TypeName> , p7: <TypeName> ] | bag[ ]...
<TypeName> ::= identificateur
}
define grammar PML_Entity_Expression {
<EntityDeclaration> ::= e0: ARLTypes e1: #isa Entity with { parts#
                              p3: <TypeName> e2: : p4: <SimpleType>
                             e3: #tuple (# p6:T<TypeName> e2: : p7: <TypeName>)
                    e3: #end with#
  }
define mapping Type_Expression
            from ARL_Type_Expression to PML_Entity_Expression {
      (p1,p2) → (e0, e1, e3) .
   where { (p1,(p3,p4)*, (p5,p6,p7)*) → (e0, e1, (p3, e2, p4)*, (e3, p6, e2, p7)*,e3 )
} .
  }
```

Remarque : pour simplifier nous n'avons utilisé que les types prédéfinis et compatibles des deux langages.

En nous référant de nouveau à l'étude de cas *DAS*, rappelons la spécification en ArchWare ARL des types du composant *SensorDef* :

```
types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
```

L'exécution du composant *Sigma* en utilisant le pattern de synthèse de types, défini précédemment, génère le code *PML* suivant :

```
TypesARL isa Entity with
 parts
    Key : Any
    Data : Any
    Entry : tuple(Key, Data)
 end with
```

4.2.3. Synthèse de comportements en PML

Le pattern de synthèse que nous avons défini pour la génération des comportements en *PML* est le suivant :

```
define grammar ARL_Behaviour_Expression {
<BehaviourDeclaration> ::= p1: #behaviour is# { <BehaviourBody> }

<BehaviourBody> ::= <action_prefix> | p2: deferred | ...

<action_prefix> ::= p3: via p4:<clause> p5:send [p6:<clause_list>]
                  | p7:via p8:<clause> p9:receive [p10:<labelled_type_list>]
                  | unobservable
                  | p11:choose { [<choice_list>] }
                    | p12: function ( [p14:<labelled_type_list>] ) := p15:<Type>
                    | ....
<choice_list> ::= p16:<clause> [or p17:<clause>]+

<clause_list> ::= <clause> [,<clause>]*

<labelled_type_list> ::= <TypeName> : <Type> [,<TypeName> : <Type>]*
<TypeName> ::= identificateur

define grammar PML_Action_Expression {
<ActionDeclaration>::= e1:actions <ActionGT> e0: #end with#
<ActionGT> ::= <ActionGive> | <ActionTake> | <ActionFunction> | <ActionChoose>
<ActionGive> ::= e2: #Give(interaction =# p4:<clause>
                 e3: #,gram = # p6:<clauseList>)
<ActionTake> ::= e4: #Take(interaction = # p8:<clause>
                 e5: #, gram = # p10:<labelled_type_list>)
<ActionFunction > ::= <NameAction> e6: #isa Action with#
                      e7:in <TypeName> : <Type>
                      e8:out <TypeName> : <Type>
                      e9: #end with#
<ActionChoose> ::= e10:{ p16:<clause> } e11: #when state := 100 #
                   e12:{ p17:<clause> } e13: #when state := 100 #
}
define mapping Behaviour_Expression
                from ARL_Behaviour_Expression to PML_Actions_Expression {
(p1,p2) → (e1, e0) . (p3,p4,p5,p6) → (e2,p4,e3,p6) . (p7,p8,p9,p10) → (e4,p8,e5,p10) .
(p11,p16,p17) → (e6,e7,e8) . (p12,p14,p15) → (e10,p16,e11,e12,p17,e13) .
  where { (p1,(p3,p4,p5,p6)*, (p7,p8,p9,p10)*,(p11,p16,p17)*, (p12,p14,p15)*)
          → (e1,(e2,p4,e3,p6)*, (e4,p8,e5,p10)*,
     (e6,e7,e8)*,(e10,p16,e11,e12,p17,e13)*,e0 ) }
}
```

Un comportement en *ArchWare ARL* correspond à plusieurs actions en *PML*. Pour traduire le *choix* spécifié en *ArchWare ARL*, une variable *state* est définie et utilisée, nous avons donné la valeur 100 à *state* dans deux actions une fois qu'on a un choix entre les deux. Pour l'enchaînement des actions, une fois que l'action est déclenchée, la variable *state* est incrémentée. Les actions prédéfinies *Take* et *Give* sont utilisées pour traduire respectivement *p5* et *p9*.

En nous référant à l'étude de cas *DAS*, la spécification suivante en *ArchWare ARL* du comportement du composant *SensorDef* :

```
behaviour is {
    process is function(d: Data) : Data { deferred }.
    replicate   via in receive entry.
                via toLink send tuple(entry::key, process(entry::data))
}
```

est traduite en PML par le composant *Sigma* (en utilisant le pattern de synthèse de comportements) de la manière suivante :

```
actions
  action1 : { process(d: Data) } state :=1 } when state := 0 ;
  action2 : { Take(interaction = in , gram = entry) state :=2} when state := 1 ;
  action3 : { Give(interaction = toLink, gram =(entry.key, process(entry.data)) ) state :=2}
            when state : = 1 ;
```

où dans la partie action est décrit le code qui sera exécuté. Quand la variable state a pour valeur :

- 0 : l'action est déclenchée puis la variable *state* est mise à 1,
- 1 : *Take* reçoit un message (*gram* représente le message qui sera reçu) à travers *giveport* puis la variable *state* est mise à 2. Cette action ajoute la valeur du *gram* dans la file d'interactions,
- 2 : *Give* envoie le message (*gram* représente le message qui doit être envoyé). Cette action extrait les données de la file d'interactions puis la variable *state* est mise à 0.

4.2.4. Synthèse des composant/connecteur/architecture en *PML*

Les trois patterns de synthèse de types, ports et comportements seront utilisés dans le pattern restant, à savoir celui qui englobe la synthèse de composants, connecteurs et architectures(appelés constituants) :

```
define grammar ARL_Constituent_Expression {
<ARLComponentDeclaration> ::=
      archetype p1: <ConstituentName> is <ARLConstituent>
   {  [<ARLTypeDeclaration>  <ARLPortDeclaration>  <ARLPortDeclaration>]      |
  p0:deferred}
<ARLConstituent>::= p2:component | p3:connecteur | p01:architecture
<ARLTypeDeclaration> ::= p4: #types is { <TypeDeclarationSet> }#
<ARLPortDeclaration> ::=  p5: #ports is { <portDeclarationSet> }#
<ARLPortDeclaration> ::=  p6: #behaviour is { < BehaviourDeclarationSet> }#
   }
   define grammar PML_Role_Expression {
   <ActionDeclaration> ::=  p1: <ConstituentName>  e1: #isa Role with #
                            e2: ressources  e3:@ Type_Expression
                            e4:@ Port_Expression <RessourcesDeclaration>
```

```
                                e5: actions  e6:@ Behaviour_Expression
<ActionsDeclaration>
                                e7:  #end with#
 }
 define  mapping  Constituent_Expression
          from ARL_Constituent_Expression  to PML_Role_Expression  {
     (p1,p0) → (p1, e1, e7) .   p4 → (e2, e3, e4) . p5 → (e2, e5, e6)
where { (p1,(p2 | p3 | p01)+, (p4,p5,p6)*)  → (p1,e1,e2,e3*,e4*,e5,e6*,e7 ) } .   }
```

Les composants, connecteurs et architectures en *ArchWare ARL* sont traduits en utilisant les rôles en PML. La partie *ressources* d'un rôle contient les paramètres prédéfinis *types* et *ports* d'où l'importation (désignée par @) respectivement des patterns *@Type_Expression et @Port_Expression,* et, la partie *actions* contient les clauses correspondant au comportement d'où l'importation du pattern *@ Behaviour_Expression*.

Dans l'étude de cas *DAS*, la spécification suivante du composant *SensorDef* en *ArchWare ARL* :

```
archetype SensorDef is component {
    types is { Key is Any. Data is Any. Entry is tuple[Key, Data] }
    incoming is { in is connection(Entry). key is connection(Key) }
    outgoing is { data is connection(Data) }
    behaviour is {
          process is function(d: Data) : Data { deferred }.
          replicate via in receive entry.
                    via toLink send tuple(entry::key, process(entry::data))
     }

 }
```

est traduite en PML par le composant *Sigma* (en utilisant le pattern de synthèse de constituants) de la manière suivante :

```
SensorDef isa  Role with
resources
    Key : Any
    Data : Any
    Entry : tuple(Key, Data)
    in  : giveport  Entry
    key  : giveport  Key
    data  : takeport  Data
actions
   action1 :  { process(d: Data) } state :=1 } when state := 0 ;
   action2 :  { Take(interaction = in , gram = entry) state :=2} when state := 1 ;
```

```
action3 : {
    Give(interaction = toLink, gram =(entry.key, process(entry.data)) ) state :=2 ;
        }   when state : = 1 ;

endwith
```

Notons que Sigma ne peut générer le code en totalité, par exemple pour les unifications de connections dans *PML* il n'y a pas d'action d'attachement explicite. Ainsi lorsque nous créons deux interactions pour connecter deux ressources, celles-ci sont passées aux rôles appropriés dans *PML* en utilisant une table de correspondances de paramètres (*binding* d'un nom de ressource avec sa valeur initiale). Par ailleurs, le code des interfaces utilisateur doit être écrit manuellement.

5. Conclusion

L'environnement logiciel *Refiner* que nous avons proposé en tant que support informatique au développement de logiciel centré-architecture, est fondé sur deux composants : *RefinerTool* et *Sigma* qui à eux deux permettent de développer une grande classe d'applications logicielles. *RefinerTool* implémenté en *Maude* permet aux utilisateurs d'effectuer autant de raffinements qu'ils veulent sur une architecture. Dans *Sigma*, la méthode de *mapping générique* que nous proposons permet de relier des architectures quelconques par une correspondance d'interprétation syntaxique. Tout particulièrement nous nous sommes intéressés au passage d'une spécification d'architecture concrète exprimée en *ArchWare ARL* vers une implémentation codée en *ProcessWeb/PML* générée en grande partie automatiquement.

Chapitre 6 : Etude de cas e-Alliance

Chapitre 6 : Etude de cas e-Alliance

Ce chapitre est dédié à la validation de notre approche, à l'aide d'une étude de cas que nous appellerons *e-Alliance* du nom du projet dans le cadre duquel nous l'avons effectuée [Eall 01, CMO 02, AABCM02, MAO 03, AMO 03, CABBAM 03]. Plus précisément, nous expliquons comment une architecture logicielle abstraite spécifiée en *ArchWare ARL*, en utilisant l'environnement *Refiner* peut progressivement mener à la génération d'une application logicielle dans le langage persistant orienté objet ProcessWeb/PML (cf. chapitre 5).

1. Présentation de l'étude de cas e-Alliance

Dans le contexte du projet régional *e-Alliance*, projet visant à fournir une infrastructure logicielle pour regrouper des entreprises dans le but de leur permettre de coopérer et ainsi former une alliance, nous nous intéressons à la modélisation des différentes informations et des mécanismes liés à l'appartenance d'une entreprise à une alliance. Cette problématique est à considérer dans le cas de groupements ouverts, c'est-à-dire des groupements dans lesquels à tout moment des organisations peuvent entrer ou sortir.

Ces informations peuvent être abordées selon un point de vue global identifiant les informations communes à plusieurs participants, et, selon un point de vue local identifiant les informations nécessaires à chacun des participants pour pouvoir opérer au sein de l'alliance.

Au niveau global, nous sommes intéressés aux alliances et aux contrats. Ceux-ci sont deux expressions de la notion de groupement, contraignant le fonctionnement des participants concernés sur des durées et des degrés différents. La participation à une alliance (processus long) entre différents participants impose de respecter un ensemble de règles de fonctionnement qui peuvent être implicites ou formalisées ayant trait à différents aspects de la vie de l'alliance (adhésion, sortie, diffusion des informations, confidentialité, etc.) et notamment à l'élaboration de contrats entre ces participants. Ces contraintes sont actives à compter de l'adhésion de l'entreprise à l'alliance jusqu'à sa sortie, à moins de leur modification en cours d'exécution.

Les contrats établis (processus courts) définissent à leur tour des contraintes influant sur le fonctionnement local de chacun des participants (suivi d'exécution de la (des) tâche(s) concernée(s) par le contrat, conditions de remise en cause, …). Ces contraintes sont valides tant que le contrat n'est pas remis en cause ou terminé. Au niveau local, nous sommes intéressés aux informations (capacités, plans de charge, etc.) que chacun va collecter auprès des autres participants de l'alliance ou de chacun des participants aux contrats dont il est donneur d'ordre ou fournisseur, afin de gérer ses collaborations et ses engagements au sein de l'alliance.

2. L'infrastructure logicielle de *e-Alliance*

Afin de supporter les alliances décrites dans le paragraphe précédent, l'architecture conceptuelle de l'infrastructure logicielle proposée par les partenaires du projet *e-alliance* a été définie de la manière suivante (cf. Fig.VI.1) :

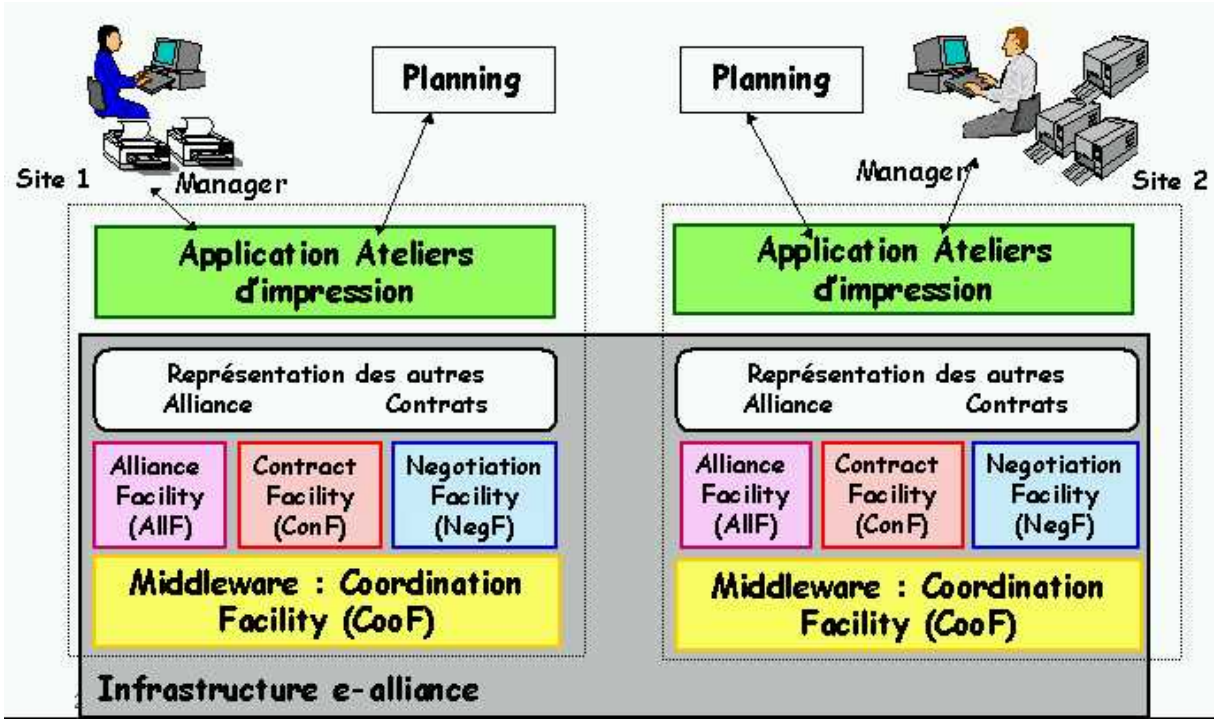


Figure VI.1. : Infrastructure logicielle d'e-Alliance

Chaque entreprise de l'alliance, une entreprise membre de l'alliance, dispose de trois différents composants :

- un *Alliance Facility* (*AllF*) permettant la gestion de la vie du membre,
- un *Contract Facility* (*ConF*) permettant la gestion des contrats avec les autres membres,
- un *Negociation Facility* (*NegF*) permettant la gestion des négociations en cours avec les autres membres.

Chaque membre dispose aussi d'un composant *middleware* (*CooF*) permettant de gérer les relations entre les services ainsi que la négociation entre les partenaires. Chaque membre de l'alliance dispose également d'une couche *représentation des autres* (*RDA*) permettant de se faire une idée personnelle des autres membres de l'alliance, information qui peut être modifiée selon les événements survenus lors des interactions (négociations, contrats, etc.) avec les autres membres.

Dans le reste de ce chapitre, nous détaillons d'abord les phases de cycles de vie auxquelles nous nous sommes intéressés (création, activation, évolution incluant l'adhésion et le retrait d'entreprises). Ensuite, nous traitons plus particulièrement le cas de l'adhésion d'une entreprise à l'alliance. Cette adhésion se traduit pour nous par le raffinement progressif d'une architecture abstraite de l'alliance vers une architecture concrète élargie au nouvel adhérent. Cette architecture englobera après plusieurs étapes de raffinement, les différents services (*AllF*, *NegF*, *ConF*, *CooF* et *RDA*). Dans un second temps, nous procédons à un raffinement vertical de l'architecture obtenue en rajoutant un composant de base de données appelé *BDLOG* qui permet le stockage d'informations portant sur l'historique de l'alliance. A partir l'architecture concrète obtenue, sera enfin générée l'application logicielle dans le langage cible *ProcessWeb/PML*.

3. Cycle de Vie de l'alliance – Modèle général

Une alliance étant un groupement d'entreprises, chaque entreprise dispose d'un contrat d'adhésion, celui-ci étant un ensemble de caractéristiques représentant entre autres les compétences, les protocoles du membre. La vie de chaque membre est soumise à des règles de fonctionnement de l'alliance. Le cycle de vie tel que devra le gérer l'alliance est défini par son existence depuis sa création jusqu'à sa disparition, l'évolution de l'alliance étant inhérente à cette existence. Concrètement, le cycle de vie d'une alliance comprend plusieurs phases parmi lesquelles : *la création de l'alliance, Vie de l'alliance* (l'adhésion de nouveaux membres, l'évolution du contrat d'adhésion et des préférences liés à un membre, le retrait définitif d'un membre de l'alliance, l'évolution des règles de fonctionnement de l'alliance) et enfin *la gestion et la prise en compte de l'historique*. La *dissolution* signe l'interruption de la vie de l'alliance. Pour décrire ces différentes phases, nous présentons pour chacune d'elle, l'objectif et les acteurs logiciels (services) qui y sont impliqués.

3.1. Création de l'alliance

Nous avons supposé dans notre travail que l'alliance a déjà été créée. L'ensemble initial des adhérents à une alliance, constituant le comité (*Comité*), configure le système à partir des objectifs, types de contrats, règles à respecter, comité de l'alliance, les variables qui pourront être négociées, les protocoles de négociation, les ontologies utilisées, etc.
Les règles de fonctionnement de l'alliance précisent entre autres les sanctions administrées aux membres qui refusent systématiquement de co-traiter des requêtes de clients. Pour ce faire, l'*AllF* garde trace (cf. Gestion et prise en compte de l'historique), pour les différentes requêtes, de l'identité des membres qui les ont prises en charge, et ceux qui ont refusé de le faire. L'idée sous-jacente à un tel historique n'est pas, bien entendu, l'espionnage des membres adhérents mais plutôt celle de préserver l'existence de l'alliance elle-même. Que se passerait-il, en effet, si tous les membres de l'alliance refusaient systématiquement de prendre en charge les requêtes de clients ? Cet ensemble initial d'adhérents fait connaître la nouvelle alliance (e. g. publicité, listes de diffusion, etc.) via, un site Web, présentant l'alliance et permettant de remplir un formulaire de demande d'adhésion à l'alliance.

3.2. Vie de l'alliance

3.2.1. Adhésion à l'Alliance

Objectif
Pouvoir mettre en commun des ressources et des compétences pour répondre à des requêtes.

Acteurs
Les acteurs intervenant dans cette phase de l'Adhésion sont : *Comité* et *AllF* du membre ayant demandé l'adhésion.

3.2.2. Evolution du contrat d'adhésion

Objectif
Un membre de l'alliance a besoin de modifier des caractéristiques du contrat d'adhésion (ajout/ suppression/ remplacement/ suspension).

Acteurs
Les acteurs intervenant dans cette phase sont : *Comité* et *AllF* du membre demandeur de l'évolution.

3.2.3. Retrait définitif d'un membre

Objectif
Un membre désirant se retirer de manière définitive de l'alliance.

Acteurs
Les acteurs intervenant dans cette phase sont : *Comité* et *AllF*, les autres membres.

3.2.4. Gestion et prise en compte de l'historique

Objectif
L'Alliance étant définie plus ou moins sur le long terme, la multitude et la complexité des processus mis en œuvre (négociations, contrats, règles de fonctionnement, etc.) requièrent une prise en compte et une gestion de l'historique des événements survenant au cours du cycle de vie. En particulier à chaque nouveau contrat ou nouvelle négociation, le *AllF* est notifié par le *ConF*, respectivement le *NegF*. De même à chaque étape d'avancement dans un contrat, le *AllF* met à jour le niveau d'activité correspondant (pourcentage effectué, terminée, etc.).

Acteurs
Les acteurs intervenant dans le cas d'un nouveau contrat de sous-traitance sont : le *Comité, AllF, ConF, NegF* et le *BDLOG* d'un membre de l'alliance.

3.3. Dissolution de l'alliance

Seul le comité peut décider à n'importe qu'elle moment la dissolution de l'alliance.

Dans ce qui suit, nous nous focalisons sur l'utilisation de *ArchWare ARL* pour la description et le raffinement de l'architecture de l'alliance en vue de prendre en compte l'adhésion de nouveaux membres.

4. Raffinement de l'architecture d'une alliance pour l'adhésion d'une entreprise

L'adhésion d'un nouveau membre à l'alliance se traduit par le raffinement progressif d'une architecture abstraite de l'alliance vers une architecture concrète élargie au nouvel adhérent. Ce raffinement se fera en trois phases : (i) obtention après plusieurs étapes de raffinement une architecture abstraite de l'alliance contenant les services pour le nouveau membre (*AllF*, *NegF*, *ConF*, *CooF* et *RDA*) ; (ii) l'architecture résultat sera raffinée en une architecture concrète, par l'ajout d'un composant de base de données appelé *BDLOG* qui permet le stockage d'informations portant sur l'historique des adhésions à l'alliance ; (iii) à partir l'architecture concrète obtenue, sera générée l'application logicielle dans le langage cible *ProcessWeb/PML*.

4.1. Architecture initiale

La demande d'adhésion à l'alliance se fait par l'intermédiaire d'un composant qui joue le rôle d'interface entre les demandeurs d'adhésion et l'alliance. Comme le montre la figure (cf. Fig. VI.2), ce composant peut être spécifié dans un premier temps de manière abstraite simplement par son nom : *AdhesionDef*.

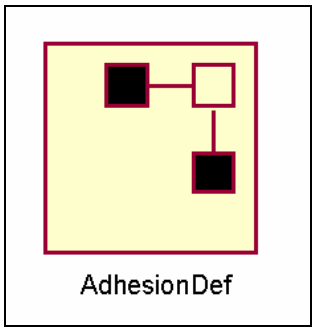


Figure VI.2. : Le modèle abstrait AdhesionDef

En utilisant *ArchWare ARL*, l'architecte pourrait spécifier *AdhesionDef* comme suit :

```
archetype AdhesionDef is architecture {
  types is { deferred }
  ports is { deferred }
  behaviour is { deferred }
}
```

Les types, les ports et le comportement sont non spécifiés (*deferred*).

4.2. Etape de raffinement pour obtenir une architecture abstraite

L'architecte pourrait raffiner le modèle précédent en le partitionnant en termes de composants et de connecteurs comme l'illustre la figure (Fig. VI.3).

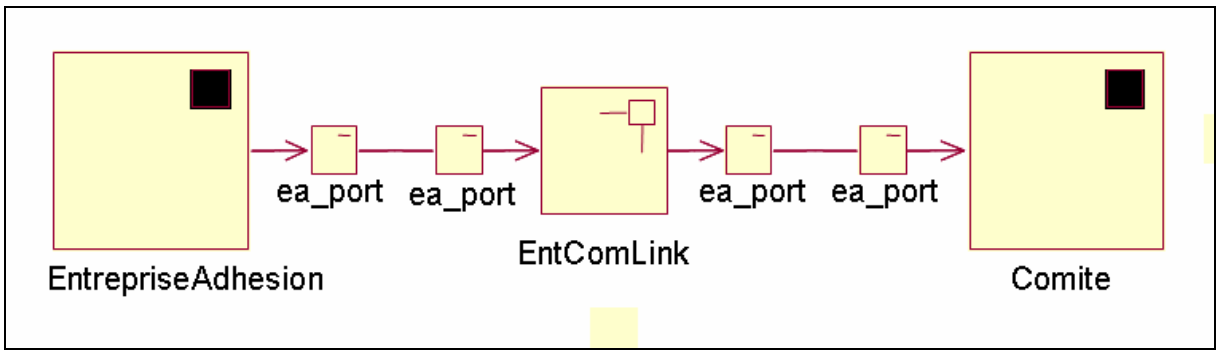


Figure VI.3. : Définition d'une architecture abstraite.

Après le partitionnement, l'architecture comporte deux composants : *EntrepriseAdhesion,* et *Comite*. Les données de l'environnement sont expédiées à distance via le connecteur *EntComLink* qui relie l'entreprise et le comité. Chaque composant est relié à un connecteur par un port. Tout au long de l'étude de cas, un seul type de port appelé *ea_port* est utilisé. Celui-ci est considéré comme générique et peut recevoir ou émettre des données à travers des connections.

On suppose qu'il n'y a qu'un seul membre de comité *Comite* représenté, cependant on peut imaginer qu'il y en ait plusieurs. *EntrepriseAdhesion* représente l'interface entre l'alliance et l'entreprise qui veut adhérer. Il englobe le site Web où se connectent les entreprises demandant l'adhésion.

Les deux composants *EntrepriseAdhesion* et *Comite* et le connecteur *EntComLink* sont décrits en *ArchWare ARL* comme suit. Seuls les aspects structurels sont modélisés. Les comportements restent non spécifiés, c'est-à-dire leur description est différée (*deferred*).

```
archetype EntrepriseAdhesion is component {
  types is { deferred }
  ports is {
       archetype   ea_port is port {
          incoming is { envoie_infos_in is connection(String) .
                        demande_infos_in is connection(String)  .
                        demande_action_in is connection(union(Any, String)) .
                        attente_resultat_in  is connection(Boolean) .
                        browser_in  is connection(String)      }
          outgoing is {  envoie_infos_out  is connection(String) .
                         demande_infos_out is connection(String) .
                         attente_resultat_out  is connection(Boolean) .
                         demande_action_out is connection(union(Any, String))    }
          }
   }
  behaviour is { deferred }
} .
archetype Comite is component {
  types is { deferred }
  ports is {
           archetype ea_port is port {  --- même type de port qu'avant  }
           }
          }
  behaviour is { deferred }
} .
archetype EntComLink is connector {
  types is { deferred }
  ports is {
           archetype ea_port is port {  --- même type de port qu'avant  }
           }
  }
   behaviour is { deferred }
}
```

Notons que le comportement des composants et connecteurs qui ont été rajoutés n'est pas du tout défini. Le comportement du système lui-même reste inchangé, puisque les nouvelles connections définies dans le port ne sont pas utilisées dans l'architecture.

4.3. Etapes de raffinement pour obtenir une architecture concrète englobant le nouvel adhérent

En utilisant les *EntrepriseAdhesion, EntComLink* et *Comite*, le modèle *AdhesionDef* précédent peut être à présent raffiné en définissant les aspects structurels et comportementaux de ses composants/connecteurs. L'architecture obtenue peut être décrite dans *ArchWare ARL* comme suit :

```
archetype AdhesionArchAbs refines AdhesionDef using {
  EntrepriseAdhesion::behaviour becomes abstraction() {
   replicate via browser_in receive Requete : String .
             via demande_infos_out send Requete .
             via envoie_infos_out receive Contrat_Adhesion : String, Doc : String .
             via demande_action_out send Contrat_Adhesion, Doc .
             via attente_resultat_in receive Reponse : Boolean .
             if ( Reponse ) then
                  {
                     via demande_action_in receive Application : Any
                  }
      } .
  Comite::behaviour becomes abstraction() {
   replicate via demande_infos_in receive Contrat_Adhesion : String .
             via envoie_infos_out send Contrat_Adhesion .
             via demande_action_in receive Contrat_Rempli : Any .
             choose
                {          via attente_resultat_out send true .
                  or       via attente_resultat_out send false
                }
  } .
```

```
EntComLink::behaviour becomes abstraction() {
 replicate via demande_infos_in receive Requete : String .
     via envoie_infos_out send Requête .
     via demande_action_in receive ContratRempli : Any .
     via demande_action_out send ContratRempli .
      via attente_resultat_in receive Reponse : Boolean .
      via envoie_infos_out send Reponse
} .
connections unifies EntrepriseAdhesion::ea_port with EntComLink::ea_port.
connections unifies EntComLink::ea_port with Comite::ea_port
}
```

Le scénario suivi pour la description des comportements de l'architecture *AdhesionArchAbs* peut être découpé en trois parties, chacune représentant le comportement d'un composant/connecteur :

- L'entreprise *(EntrepriseAdhesion)* désirant intégrer l'alliance demande l'adhésion au comité *(Comite)* de l'alliance en fournissant un certain nombre d'informations sur son identification, ses compétences, ses ressources, ses besoins, ses préférences. Ceci est fait via le site Web de l'alliance en remplissant un formulaire de demande d'adhésion.
- Le comité de l'alliance étudie la demande et donne éventuellement l'accord pour l'adhésion. Un contrat d'adhésion est alors défini en fonction des informations fournies.
- Le nouvel adhérent dispose dès lors d'une page de téléchargement de l'infrastructure *e-Alliance*. Le système peut ensuite être déployé sur son site et configuré pour lui, en fonction de son contrat d'adhésion.

Après ce raffinement comportemental, la structure du système demeure sans changement. Maintenant que nous avons défini les aspects structurels et comportementaux des éléments architecturaux de *AdhesionDef*, nous pouvons raffiner notre architecture par composition des composants et connecteurs à savoir *EntrepriseAdhesion, EntComLinK* et *Comite.* Ceci est exprimé dans *ArchWare ARL* par :

```
archetype AdhesionArchAbs1 refines AdhesionArchAbs using {
  compose { EntrepriseAdhesion and EntComLinK and Comite) }
  }
```

A ce stade, si la demande d'adhésion est acceptée, l'architecture logicielle devrait être raffinée pour contenir les différents services pour le nouvel arrivant. Rappelons que chaque entreprise de l'alliance dispose de 3 différents composants :

- un Alliance Facility (*AllF*) permettant la gestion de la vie du membre,
- un Contract Facility (*ConF*) permettant la gestion des contrats avec les autres membres,

- un Negociation Facility (*NegF*) permettant la gestion des négociations en cours avec les autres membres.

La figure suivante (cf. Fig.VI.4) présente la nouvelle architecture que nous obtenons par raffinement. Nous décrivons en détail, les étapes de raffinement dans cette section.

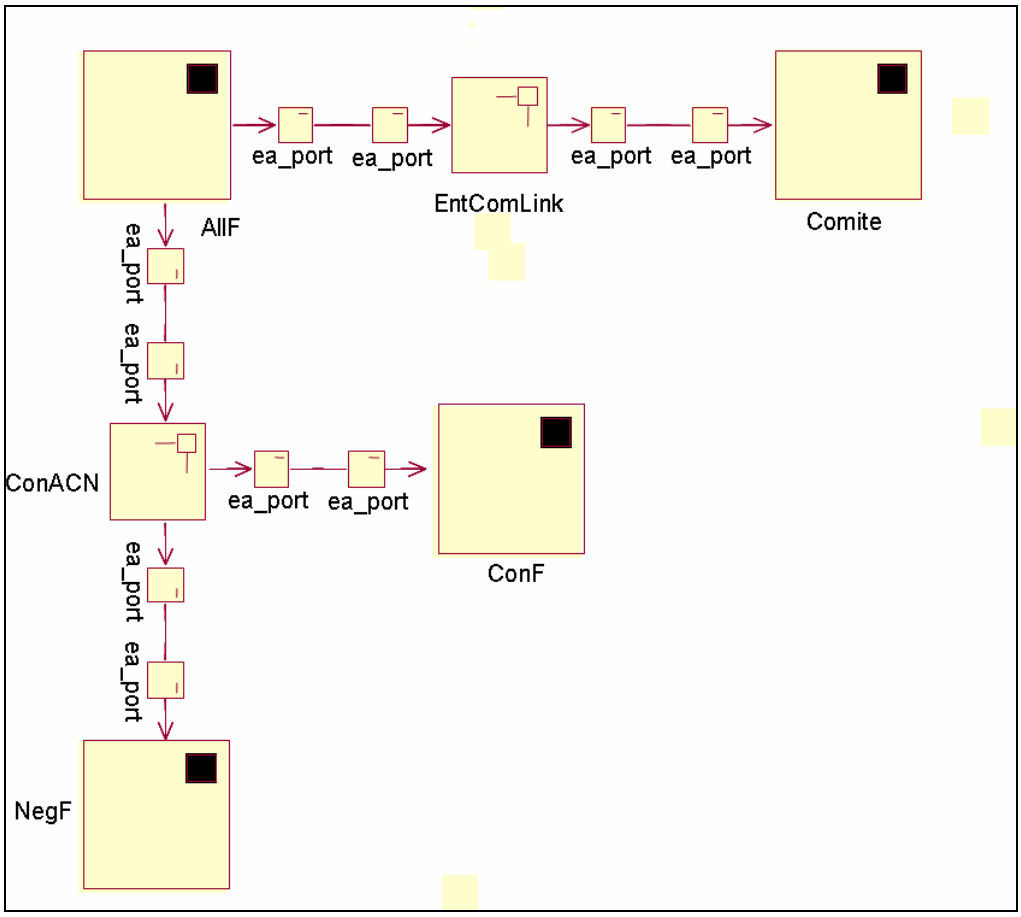


Figure VI.4. : Architecture concrète impliquant le nouvel adhérent.

En effet, après une adhésion et téléchargement de l'application par l'entreprise, un raffinement de l'architecture *AdhesionArchDef* est nécessaire. Ceci afin de prendre en compte le rajout des composants relatifs au nouvel adhérent de l'alliance. Pour aboutir à l'architecture présentée dans la Figure VI.4, trois actions de raffinement sont appliquées :

1. l'ajout des nouveaux sous-composants représentant les services *AllF*, *ConF* et *NegF* au composant *EntrepriseAdhesion* :

```
archetype AdhesionArchCon1 refines AdhesionArchAbs1 using {
  EntrepriseAdhesion::components
  includes  (AllF and ConF and NegF and ConACN)
  }
```

Dans *ArchWare ARL,* les composants peuvent être décrits de la manière suivante :

AllF :

```
archetype AllF is component {
  types is { deferred }
  ports is {  --- même type de port  ea_port qu'avant
          }
  behaviour is { replicate
    choose {
          {   --- entre le AllF et NegF

              via demande_action_in receive Negotiation : Boolean .
              if ( Negotiation )
               {
                    via attente_resultat_out send   true
               }
            else
               {
                    via attente_resultat_out send  false
               }
            }

        or  --- entre le AllF et le ConF
            {
              via demande_action_in receive  Evolution_Contrat : Boolean .
              if   ( Evolution_Contrat )  then
               {
                  via attente_resultat_out send  true
               }
          else
               {
                  via attente_resultat_out  send  false
               }
            }

        or  --- entre AllF et Comite
        {
            via  demande_action receive Retrait : Boolean .
             if ( Retrait )  then
              {
                  via attente_resultat_out send true
              }
            else
              {
                  via attente_resultat_out send  false
              }

          }
      } --- fin de choose
    }  --- fin de behaviour
  }
```

:

NegF :

```
archetype NegF is component {
  types is { deferred }
  ports is { archetype  ea_port is port {  --- même type de port qu'avant  }       }
  behaviour is {
                  --- entre le AllF et NegF
   replicate    via  demande_action_in receive   Action_NegF : Any  .
      choose {
                 via attente_resultat_out send  true
                or
                via  attente_resultat_out  send  false
              }
        }
}
```

ConF :

```
archetype ConF is component {
  types is { deferred }
  ports is { archetype  ea_port is port {  --- même type de port qu'avant  }       }
  behaviour is {

          replicate    via  demande_action_in receive   Action_ConF : Any  .

             --- entre le AllF et le ConF
       choose {
                  via attente_resultat_out   send   true
                  or
                  via attente_resultat_out   send  false

                }
       } }
```

2. Ajouter et connecter le connecteur *ConACN* qui relie *AllF*, *ConF* et *NegF* :

```
archetype AdhesionArchCon2 refines AdhesionArchCon1 using {
  connectors includes {
  archetype ConACN is connector {
    types is { deferred }
    ports is { archetype  ea_port is port {  --- même type de port qu'avant  } }
    behaviour is {
                via  demande_action_in  receive  Negociation : Boolean  .
                via  attente_resultat_out   send  Negociation .
                via  demande_action_in  receive Evolution_Contrat : Boolean .
                via  attente_resultat_out   send  Evolution_Contrat .
                via  demande_action_in  receive  Retrait : Boolean  .
                via  attente_resultat_out   send  Retrait
    }
  } .
  connections unifies AllF::ea_port with ConACN::ea_port .
  connections unifies AllF::ea_port with EntComLink::ea_port .
  connections unifies ConF::ea_port with ConACN::ea_port .
  connections unifies NegF::ea_port with ConACN::ea_port  .
  connections unifies Comite::ea_port with ConACN::ea_port

}
```

3. L'explosion du composant *EntrepriseAdhesion*, ce dernier disparaît de l'architecture et ce sont ces sous-composants qui le remplacent :

```
archetype AdhesionArchDef3 refines AdhesionArchDef2 using {
  behaviour is  { AdhesionArchDef::components explodes  EntrepriseAdhesion }
```

Du point de vue du raffinement le modèle de retrait englobe à peu près celui du modèle d'adhésion ; par exemple pour un retrait d'entreprise, il suffit de supprimer les composants de cette entreprise et séparer ses connections avec *EntComLink* en utilisant les actions de *ArchWare ARL* pour la séparation des

unifications et la suppression des éléments architecturaux. Mais il ne faut pas oublier de transformer le comportement de l'entreprise pour qu'elle puisse demander un retrait définitif. Inversement, pour que le comité accepte un retrait, il faut qu'il soit capable de le gérer. La dissolution est simple, il suffit de supprimer tous les éléments architecturaux de l'architecture.

4.4. Raffinement de l'architecture concrète par le rajout d'une base de données

Afin de gérer l'historique des adhésions, nous raffinons l'architecture par le rajout d'un composant base de données nommé *BDLOG* qui sera relié au comité de l'alliance (cf. Figure VI.5). *BDLOG* permettra de stocker les informations concernant la date d'adhésion, le contrat d'adhésion, etc.

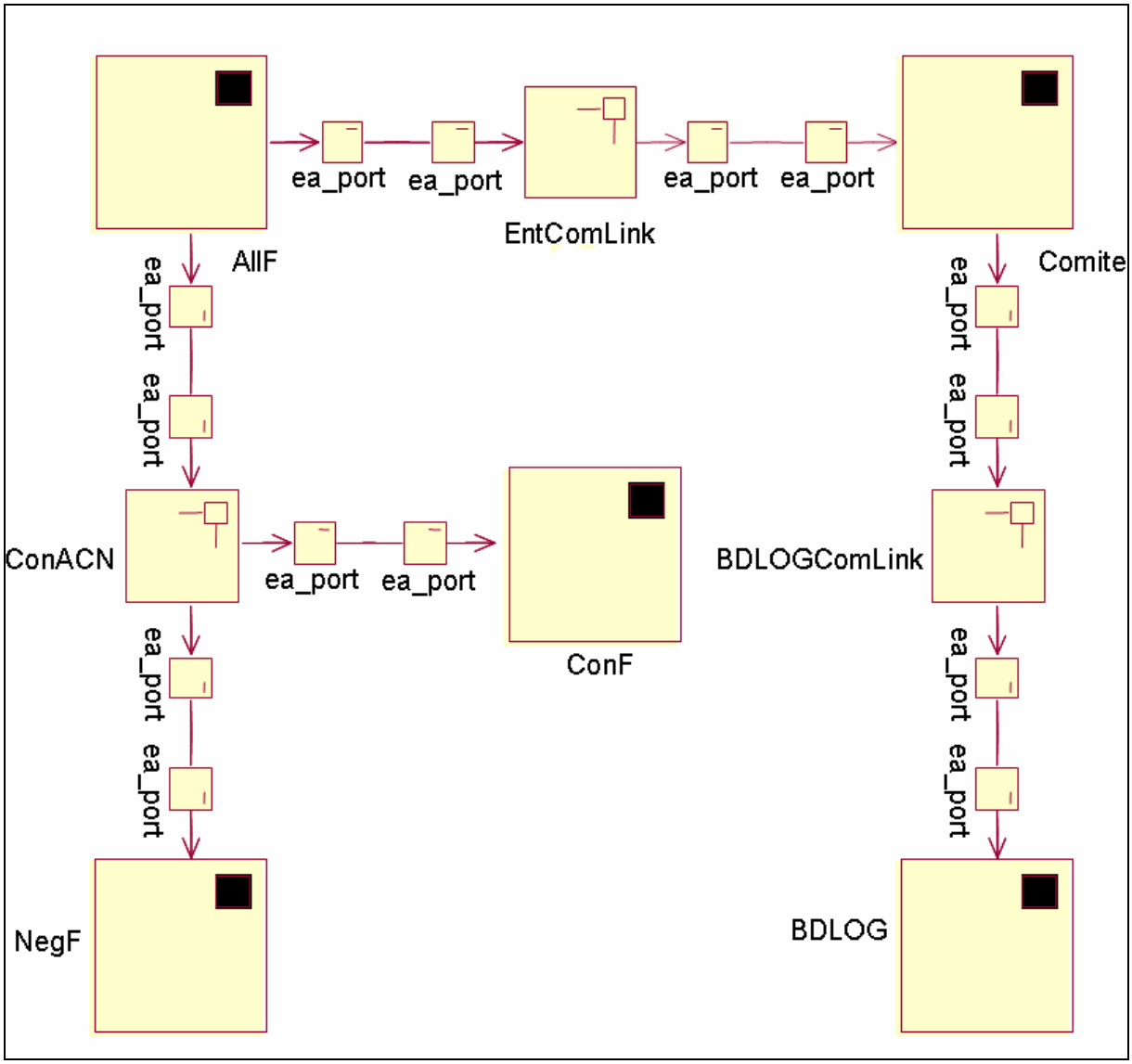


Figure VI.5. : Architecture concrète englobant un nouveau membre et une base de données

Le composant *BDLOG* peut être décrit en *ArchWare ARL* comme suit :

```
archetype is BDLOG  is component {
    types is { database is location(Set() : String)  }
    ports is { archetype  ea_port is port {  --- même type de port qu'avant  } }
    behaviour is {
          replicate choose {
          via demande_action_in receive data : String .
          database := database' including(data)
   or      via demande_action_in receive requete : Any .
           unobservable .
           via envoie_infos send  reponse : String
           }
}.
```

Pour connecter ce nouveau composant au comité, nous rajoutons un connecteur *BDLOGComLink* entre *Comite* et *BDLOG*. Ce connecteur permet le routage des informations (requêtes, réponses et données).

Le connecteur *BDLOGComLink* (avec les unifications) peut être décrit dans ArchWare ARL par :

```
archetype is BDLOGComLinK  is connector {
    ports is { archetype  ea_port is port {  --- même type de port qu'avant  } }
    behaviour is {
          replicate choose {
          via demande_action_in receive data : String .
          via demande_action_out   send  data
   or      via demande_action_in receive requete : Any .
           via demande_action_out send  requete
           }
    } .
  connections unifies Comite:: ea_port with BDLOGComLink:: ea_port  .
  connections unifies BDLOGComLink:: ea_port with BDLOG:: ea_port
}
```

5. Génération de l'application « e-Alliance » en ProcessWeb/PML

Pour la génération de l'application logicielle, nous avons appliqué les patterns de synthèse (cf. chapitre 5). Nous avons ensuite compilé le code généré dont l'exécutable se trouve sur le site du ProcessWeb (http://processweb.cs.man.ac.uk). La partie interface utilisateur (cf. Figure VI.6) est rajoutée manuellement au code généré.

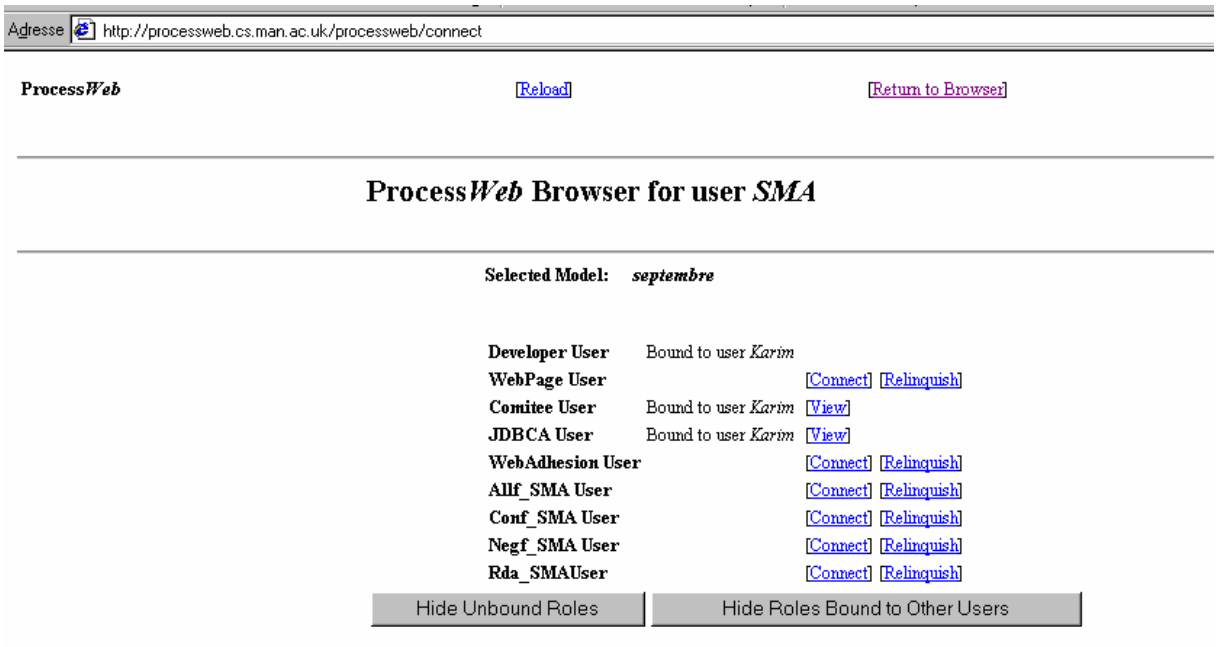


Figure VI.6. : Interface utilisateur de l'application dans ProcessWeb

6. Conclusion

A travers cette étude de cas *e-Alliance*, nous avons montré comment à partir d'une architecture abstraite spécifiée en *ArchWare ARL*, en utilisant l'environnement *Refiner*, nous avons pu générer une application logicielle dans *ProcessWeb/PML*. Nous nous sommes intéressés à la transformation d'une architecture d'alliance, engendrée par l'adhésion de nouveaux membres. Nous avons montré l'application de plusieurs actions de raffinement aboutissant au final à une application exécutable.

Dans l'environnement logiciel *Refiner,* le composant *RefinerTool* nous a permis d'effectuer autant de raffinements que nécessaire sur l'architecture. Le composant *Sigma* nous a permis quant à lui de facilement générer la plus grande partie du code source de l'application. Sans ce type de support, nous sommes convaincus que nous aurions passé énormément de temps avant que l'application ne soit générée, avec en plus un risque d'erreur plus grand sans les contrôles (de correction) associés aux actions de raffinement.

Chapitre 7 : Conclusion et perspectives

Chapitre 7 : Conclusion et perspectives

Nous nous sommes intéressés dans cette thèse aux problèmes liés à l'insuffisance de support informatique pour le raffinement d'architectures logicielles. En effet, le peu de ADL comme SADL et Rapide qui fournissent une solution à ces problèmes le font de manière très restreinte, bien qu'ils supportent des niveaux multiples d'abstraction. Le support de SADL exige des preuves manuelles pour les constructions de mise en correspondance entre un modèle abstrait et un modèle architectural plus concret et ne permet qu'un raffinement structurel. Rapide quant à lui, supporte bien les correspondances d'événements entre des architectures individuelles grâce à un sous-langage exécutable, mais par contre, seul le raffinement comportemental y est possible.

Plus précisément, notre objectif était de fournir un environnement logiciel, permettant de supporter :

- le raffinement correct d'architectures logicielles à travers multiples niveaux d'abstraction, avec prise en charge des différents éléments architecturaux (composant, connecteur, etc.),
- au moins les raffinements structurel et comportemental,
- la génération automatique (autant que possible) d'applications dans un langage de programmation cible.

L'environnement logiciel *Refiner* que nous proposons offre aux utilisateurs (architectes) la possibilité de mettre en œuvre des processus complets de raffinement selon une approche centrée-architecture du développement de logiciels. Nous l'avons construit autour du langage de raffinement d'architectures *ArchWare ARL*. *Refiner* permet à ses utilisateurs :

- de raffiner progressivement des éléments architecturaux (composant, connecteur, architecture, port, etc.) depuis des descriptions abstraites vers des descriptions concrètes, et ce à travers des niveaux d'abstraction multiples. A chaque étape de raffinement, peut être appliquée une action qui fournit par construction une transformation architecturale correcte. Ceci est possible de par le fait que les *pré-conditions* et *post-conditions* rattachées aux actions de raffinement dans *ArchWare ARL,* soient formalisées et implémentées dans le langage formel qu'est la logique de réécriture. Par conséquent, par construction, une action de raffinement transforme une architecture satisfaisant les pré-conditions en une architecture moins abstraite satisfaisant les post-conditions. Le modèle de transformation architecturale est correct si il satisfait les pré et les post-conditions.
- de raffiner plus que la structure et le comportement d'un élément architectural. En effet, l'approche *ArchWare ARL* permet au cours du processus de raffinement d'établir des relations de raffinement portant sur des comportements, des connections, des structures, et des données d'éléments architecturaux ;
- de construire au cours d'une étape de raffinement, aussi bien une description architecturale plus déterministe traduisant une architecture possible qu'une description simplifiée traduisant une architecture réduite. En effet, le premier type de raffinement est possible de par le principe de sous-spécification de *ArchWare ARL* (à un haut niveau d'abstraction, on spécifie un élément architectural tout en laissant certains aspects non spécifiés). La diminution de cette sous-spécification passe par l'établissement de relations de raffinement des comportements, des connections, des structures, et des données des éléments architecturaux. Le second type de raffinement (réduction ou simplification) est également rendu possible, grâce notamment aux actions de raffinement d'*explosion* et d'*implosion*.
- de supporter, au niveau le plus concret du raffinement architectural, la génération des applications dans des langages de programmation cibles. Ceci se fait par le biais du composant générateur d'applications *Sigma* fondé sur les concepts de *mapping* et de patterns de synthèse.

Perspectives

Ces travaux de thèse ont ouvert des perspectives à court, moyen et plus ou moins long terme. Elles portent sur : la poursuite du travail actuel, la validation de l'approche sur de nouvelles applications, l'investigation de nouveaux thèmes de recherche.

A court terme :

- ***Uniformisation de l'implémentation de Refiner*** : en effet, le composant *Sigma* du *Refiner* étant écrit dans le langage C et ne générant pas la totalité du code de l'application, nous envisageons de formaliser les patterns de synthèse en logique de réécriture pour les implémenter ensuite sur le logiciel Maude.
- ***Elaboration de l'interface utilisateur de Refiner :*** nous comptons développer une interface graphique sophistiquée pour le raffinement d'architectures, permettant aux utilisateurs de choisir graphiquement les modèles ainsi que les actions de raffinement. Le raffinement pourrait se faire de manière visuelle et la génération de code sur un clic de bouton par exemple.
- ***Nouvelles applications :*** notre proposition a été développée et validée dans le cadre du projet européen *ArchWare* avec une application portant sur les logiciels *EAI* (Enterprise Application Integration), et dans le cadre du projet régional e-Alliance avec une application portant sur la mise en place d'alliances inter-organisationnelles électroniques. D'autres applications envisagées pourraient concerner des domaines tels que les systèmes d'information distribués ou les systèmes logiciels embarqués distribués.

A moyen terme :

- ***Analyse de la correction du raffinement dans ArchWare ARL :*** jusqu'à présent dans ArchWare ARL, les pré et post-conditions des actions de raffinement sont vérifiées par Refiner. Cependant, tel n'est pas le cas pour les "hypothèses" rajoutées par l'utilisateur aux actions (*assuming*). Ces hypothèses étant des obligations de preuve, elles doivent être déchargées au moment de l'application des actions, et prises en compte lors de la vérification de la correction du raffinement. Jusqu'à maintenant, nous avons utilisé l'outil d'analyse *ArchWare AAL* développé également dans le cadre du projet *ArchWare* pour établir si un raffinement est correct ou non. Cet outil étant externe à *Refiner*, notre objectif est d'expérimenter et d'utiliser le formalisme LTL [EMS 02] de la logique de réécriture pour l'analyse de la correction du raffinement.
- ***Nouveaux aspects dans le raffinement d'architectures :*** deux aspects que nous voudrions traiter au niveau de *Refiner* sont le raffinement d'éléments architecturaux mobiles et le raffinement des styles architecturaux. La mobilité permet à un élément architectural de se déplacer à travers les connections d'une architecture vers une autre architecture, tout en respectant les contraintes de déplacement. Par conséquent, la prise en compte de la mobilité dans le raffinement rajoute des contraintes que *Refiner* devrait pouvoir supporter. Quant aux styles architecturaux, ils pourraient être pris supporté par le *Refiner* en rajoutant des actions de raffinement spécifiques aux styles architecturaux. Ce type de raffinement permettrait de définir progressivement des styles concrets de manière à ce qu'ils soient facilement exploitables par les architectes qui voudraient les instancier.

A long terme :

- ***Nouveaux thèmes de recherche :*** nous avons proposé *Refiner* en tant que support informatique au développement centré-architecture de logiciels ; nous n'avons cependant pas abordé dans cette thèse les aspects maintenance et évolution des applications générées. Un de nos objectifs est alors de compléter notre approche par ces aspects.

Conclusion

La contribution principale de cette thèse a été de fournir un support informatique pour le raffinement d'architectures logicielles. Le choix de construire un tel support autour de *ArchWare ARL* permet aux utilisateurs de bénéficier de la puissance de ce dernier en termes de pouvoir d'expression et de correction par construction des transformations effectuées sur les éléments architecturaux. En effet, l'approche *ArchWare ARL* pour le raffinement des architectures étant fondée d'une part sur des transformations formelles de descriptions d'architectures, et d'autre part sur une sémantique compositionnelle, l'environnement *Refiner* proposé permet par conséquent aux architectes de raffiner correctement des éléments architecturaux, puis de recomposer les éléments raffinés pour la construction d'architectures concrètes. Le choix de la logique de réécriture comme fondement à la fois de la sémantique formelle et de la sémantique opérationnelle de *Refiner* permet de naturellement décrire des architectures logicielles, leur raffinement par le mécanisme de réécriture ainsi que leur exécution grâce aux outils proposés pour cette logique.

Références bibliographiques

Références bibliographiques

[AABCM 02] I. Alloui, J.M. Andreoli, O. Boissier, Stefania Castellani et K. Megzari, *"E-Alliance : a software infrastructure for inter-organisational alliances"*, 9th ISPE International Conference on Concurrent Engineering: Research and Applications (CE2002) - Cranfield University (UK), 27- 31 juillet 2002.

[AAR 03] ArchWare Annual Report, « *Annual Report n°1 – Project Achievements in Year 1 (2002)* ». ArchWare European RTD Project IST-2001-32360, Deliverable D0.4.1, février 2003.

[Abr 03] J.-R. Abrial, *« The B-Book: Assigning programs to meanings »*, Cambridge University Press, 1996.

[ADG 97] R. Allen, R. Douence, D. Garlan, "Specifying Dynamism in Software Architectures", *Proceedings of the workshop on foundations of component-based systems*, pp. 11-22, Zurich, Switzerland, septembre 1997.

[AG 97] R. Allen, D. Garlan, "A formal basis for architectural connection", *ACM Transactions on Software Engineering and Methodology*, vol. 6, n°3, pp. 213-249, juillet 1997.

[AG 94] R. Allen, D. Garlan, "Formalizing Architectural Connection", *16th International conference on software engineering,* Sorrento, Italy, mai 1994.

[AGM 02] I. Alloui, H. Garavel, R. Mateescu et F. Oquendo, « *The ArchWare Architecture Analysis Language* ». ARCHWARE European RTD Project IST-2001-32360. Deliverable D3.1b, décembre 2002.

[AlG 94] R. Allen et D. Garlan, « *Formalizing Architectural Connection* ». 16th International Conference Software Engineering, IEEE Computer Soc. Press, Los Alamitos, Californie, pp. 71-80, 1994.

[AlG 97] R. Allen et D. Garlan, « *A Formal Basis for Architectural Connection* ». ACM Transactions on Software Engineering and Methodology, 6(3), juillet 1997.

[All 97] R. Allen, *A formal approach to software architecture*, Ph. D. Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, Mai 1997.

[AMO 03] I.Alloui,, K. Megzari , F. Oquendo, *" Modelling and Generating Business-To-Business Applications using an Architecture Description Language – Based Approach – "*, 5th International Conference on Entreprise Information Systems (ICEIS 2003) – Angers (France), 23-26 avril 2003

[Arc 02] Projet « *ArchWare – Architecting Evolvable Software* ». ARCHWARE European RTD Project IST-2001-32360. www.arch-ware.org, 2002-2005.

[Eall 01] Projet « *e-Alliances – ALLIANCES INTER-ORGANISATIONNELLES»*. Programmes Thématiques prioritaires de la région Rhône-Alpes, 2001-2003.

[BCD 00] M. Bernardo, P. Ciancarini, and L. Donatiello. «*On the formalization of architectural*

types with process algebras». In D. S. Rosenblum, editor, Proc. of the 8th ACM Int.Symp. on the Foundations of Software Engineering (FSE-8), pages 140-148. ACM Press, novembre 2000.

[BCK 99] L. Bass, P. Clements et R. Kazman, « *Software Architecture in Practice* ». Addison Wesley, ISBN 0-201-19930-0, 1999.

[BEJ 96] P. Binns, M. Engelhart, M. Jackson et S. Vestal, « *Domain-Specific Software Architectures for Guidance, Navigation, and Control* ». International Journal of Software Engineering and Knowledge Engineering, 1996.

[BGRSW 94] F. Bruynooghe, M. Greenwood, I. Robertson, J. SA, R.A. Snowdon, B. Warboys, "PADM: towards a total process modelling system", in *FINKELSTEIN, A., KRAMER, J., and NUSEIBEH, B. (Eds.), Software process modelling and technology* (Research Studies Press), pp 293–334, England, 1994.

[BiV 93] P. Binns et S. Vestal, « *Formal real-time architecture specification and analysis* ». 10th IEEE Workshop on Real-Time Operating Systems and Software, mai 1993.

[BMR 96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad et M. Stal, « *Pattern Oriented Software Architecture: A System of Patterns* ». John Wiley & Sons, 1996.

[Boa 95] M. Boasson, « *The Artistry of Software Architecture* ». Guest editor's introduction, IEEE Software, 1995.

[Bol 04] T. Bolusset, « *β-SPACE : Raffinement de descriptions architecturales en machines abstraites de la méthode formelle B* ». Thèse de doctorat de l'université de Savoie, septembre 2004.

[CABBAM 03] S. Castellani, J.M. Andreoli, M. Bratu, O. Boissier, I. Alloui, K. Megzari *"E-Alliance: A Negotiation Infrastructure for Virtual Alliances"*, Kluwer Academic Publishers Journals Editorial Office, Group Decision and Negotiation, Issue 2, March 2003

[CDELMMQ 99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. «*Maude: Specification and Programming in Rewriting Logic* ». Manuel de Maude 99.

[CDELMMQ 00] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. *«A Maude Tutorial* ». Manuscript, Mars 2000.

[CDELM 98] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. «*Metalevel Computation in Maude* ». In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98).
Electronic Notes in Theoretical Computer Science, Vol. 15. 1998

[CDEMO 99] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr.
«Maude as a Formal Meta-Tool. In World Congress on Formal Methods » (FM'99). Lecture Notes in Computer Science, Vol. 1709, pp. 1684-1703. 1999

[CPT 97] C.Canal, E. Pimentel, J.M.Troya, *«LEDA: A Specification Language for Software Architecture »*. Jornadas en Ingeniería del Software, San Sebastián (2001) 207-219.

[ChO 01] Christiano de Oliveira Braga. *«Rewriting Logic as a Semantic Framework for Modular*

Structural Operational Semantics». PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro. September, 2001.

[Cle 96] C. Clements, «*A Survey of Architecture Description Languages* ». Eight International Workshop on Software Specification and Design, Allemagne, mars 1996.

[CLO 03] S. Cîmpan, F. Leymonerie et F. Oquendo, « *ArchWare ADL Foundation Style* ». ARCHWARE European RTD Project IST-2001-32360. Internal Report, R1.3-1. Juin 2003.

[COB 02] S. Cimpan, F. Oquendo, D. Balasubramaniam, G. Kirby et R. Morrison, « *The ArchWare ADL: Definition of the Textual Concrete Syntax* ». ArchWare European RTD Project IST-2001-32360, Deliverable D1.2b, décembre 2002.

[CMO 00] C.Chaudet, K.Megzari, F.Oquendo, "*A Formal Architecture-Driven Approach for Designing and Generating Component-Based Software Process Models*", In Proceedings of the 4th World Multiconference on Systemics, Cybernetics And Informatics (SCI 2000), Track on Process Support for Distributed Team-based Software Development (PDTSD), Orlando, Florida, U.S.A., July 23-26, 2000.

[CSPJ 96] M.Clavel, S. Eker, P.Lincoln, and J.Meseguer.
«Principles of Maude». In 1st International Workshop on Rewriting Logic and its Applications (WRLA'96).Electronic Notes in Theoretical Computer Science, Vol. 4. 1996

[DAO 96] J. Davies et J. Woodcock, *«Using Z: Specification, Refinement and Proof, Prentice Hall International Series in Computer Science ».* 1996.

[Eall 01] Projet « *e-Alliances – ALLIANCES INTER-ORGANISATIONNELLES»*. Programmes Thématiques prioritaires de la région Rhône-Alpes. 2001-2003.

[EMS 02] Steven Eker, José Meseguer and Ambarish Sridharanarayanan. *«The Maude LTL Model Checker ».*
In 4th International Workshop on Rewriting Logic and its Applications (WRLA'02).

[EnV 04] V. Englebert, F. Vermau, *«Attribute-Based Refinement of Software Architectures - Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04) June 2004».* Oslo, Norway. p. 301-305

[Eke 03] Steven Eker *«Associative-Commutative Rewriting on Large Terms».*
In Proc. Rewriting Techniques and Applications, 2003 , Springer-Verlag LNCS 2706, 14-29, June 2003

[EP 93] Susan Eisenbach, Ross Paterson, "π-Calculs Semantics for the concurrent configuration language Darwin*", Hawaii International conference on system sciences*, Koloa, Hawaii, January 1993.

[ICL 96] ICL "*ProcessWise Integrator, PML Reference Manual*" ICL/PW/635/0, April 1996.

[Jon 93] C. B. Jones. *«Process-algebraic foundations for an object-based design notation».*. Technical Report UMCS-93-10-1, Department of Computer Science, University of Manchester,
1993.

[Jon 90,93] C. B. Jones, *« Systematic Software Development using VDM*». Prentice-Hall, second

edition,1990. Out of print.

[GAO 94] D. Garlan, R. Allen et J. Ockerbloom, « *Exploiting Style in Architectural Design Environments* ». SIGSOFT'94, ACM Press, New York, pp. 179-185, décembre 1994.

[Gar 95] D. Garlan, «*A introduction to the Aesop System*». School of Computer Science, Carnegie Mellon University, Juillet 1995.

[Gar 00] D. Garlan, « *Software Architecture: a Road Map* ». Proceedings of the conference on The future of Software engineering, mai 2000.

[GaS 93] D. Garlan et M. Shaw, « *Introduction to Software Architecture* ». Advances in Software Engineering and Knowledge Engineering. World Scientific Publishing Company, 1993.

[GHJ 95] E. Gamma, R. Helm, R. Johnson et J. Vlissides, « *Design Patterns, Elements of reusable Object-Oriented Software* », Addison-Wesley Publishing Company, 1995.

[GMW 97] D. Garlan, R. Monroe, D. Wile, "ACME: an architecture description interchange language", In *Proceedings of CASCON'97*, November 1997.

[LV 95] D.C. Luckham, J. Vera, "An event-based architecture definition language", *IEEE Transactions on software engineering*, vol. 21, n°9, pages 717-734, September 1995.

[LKA 95] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan et W. Mann, « *Specification and analysis of system architecture using Rapide* ». IEEE Transaction on Software Engineering, vol. 21, n°4, pp. 336-355, April 1995.

[Hoa 85] C. A. R. Hoare. «*Communicating Sequential Processes*». Prentice-Hall, 1985.

[MDE 95] J. Magee, N. Dulay, S. Eisenbach et J. Kramer, « *Specifying Distributed Software Architectures* ». Proceedings of the Fifth European Engineering Conference (ESEC'95), Barcelona, septembre 1995.

[MAO 03] K. Megzari, I. Alloui, F. Oquendo, *"Maquette Alliance Facility"*, E-Alliance ; Inter-organisationnel alliances, Projet Régional Rhône Alpes, Livrable LAC2.2, Septembre 2003.

[MaM 96] Narciso Martí-Oliet and José Meseguer. *«Rewriting Logic as a Logical and Semantic Framework»*. In 1st International Workshop on Rewriting Logic and its Applications (WRLA'96). Electronic Notes in Theoretical Computer Science, Vol. 4. 1996

[Med 96] N. Medvidovic, "ADLs and dynamic architecture changes", In *A.L. Wolf, ed., Proceedings of the second international software architecture workshop (ISAW-2)*, pp. 24-27, San Fransisco, CA, October 1996.

[MeG 92] E. Mettala et M. H. Graham, « *The domain-specific software architecture program* ». Technical report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, juin 1992.

[Mei 00] J.P. Meinadier, « *Ingénierie et intégration des systèmes* », Édition Hermes, 543 pages, octobre 2000.

[Mes 02] J. Meseguer, *« Rewriting Logic Revisited»*
Slides of tutorial presented at WRLA 2002, Pisa, Italy, September 2002.

[Mes 98] J. Meseguer, « *Research Directions in Rewriting Logic*», Computational Logic, NATO Advanced Study Institute, 1998.

[MeT 97] N. Medvidovic et R. Taylor, « *A Classification and Comparison Framework for Software Architecture Description Languages* ». Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, février 1997.

[MFSPNJC 03] M. Clavel, F. Durán, S.Eker, P. Lincoln, N.Martí-Oliet, José Meseguer and C.Talcott. «*The Maude 2.0 System*».
In Proc. Rewriting Techniques and Applications, 2003 , Springer-Verlag LNCS 2706, 76-87, June 2003.

[MK 96] J. Magee, J. Kramer, "Dynamic structure in software architectures", *Proceedings of ACM SIGSOFT'96: Fourth symposium on the foundations of software engineering (FSE4)*, pp. 3-14, San Francisco, CA, October 1996.

[MOM 93] N. Martí-Oliet et J. Meseguer, « *Rewriting logic as a Logical and Semantic Framework, rapport technique, SRI International* ». Computer Science Laboratory, Menlo Park, août 1993.

[MOR 96] N.Medvidovic, P. Oreizy, D.S Rosemblum et R. Taylor, « *Using Object-Oriented Typing to Support Architectural Design in the C2 style* ». Proceedings of ACM SIGSOFT'96: Fourth symposium on the foundations of software engineering (FSE4), pp. 24-32, San Francisco, CA, octobre 1996.

[MoR 97] M. Moriconi et R. A. Riemenschneider, «*Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies*», Rapport Technique SRI-CSL-97-01, Computer Science Laboratory, SRI International, mars 1997.

[MQR 95] M. Moriconi, X. Qian et R. Riemenschneider, « *Correct Architecture Refinement* ». IEEE Transactions on Software Engineering, Volume 21, Number 4, pp.356-372, 1995.

[MRT 99] N. Medvidovic, D.S. Rosenblum and R.N. Taylor, "A language and environment for architecture-based software development and evolution", *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999.

[MTW 96] N. Medvidovic, R.N. Taylor, E.J. Whitehead, Jr. "Formal modeling of software architectures at multiple levels of abstraction", *Proceedings of the California software symposium 1996*, pp. 28-40, Los Angeles, CA, April 1996.

[OAC 02] F. Oquendo, I. Alloui, S. Cimpan et H. Verjus, « *The ArchWare ADL: Definition of the*

Abstract Syntax and Formal Semantics ». ArchWare European RTD Project IST-2001-32360, Deliverable D1.1b, décembre 2002.

[Oqu 03] F. Oquendo, « *Final Definition of the ArchWare. Architecture Refinement Language* ». ArchWare European RTD Project IST-2001-32360, Deliverable D6.1b, décembre 2003.

[Poo 01] J. D. Poole. «*Model-Driven Architecture : Vision, Standards And Emerging Technologies*». ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models, April 2001.

[Rie 98] R. A. Riemenschneider, « *Correct Transformation Rules for Incremental Development of Architecture Hierarchies*», Working Paper DSA-98-01, SRI CSL Dependable System Architecture Group, février 1998.

[RoM 03] R.Bruni ans J. Meseguer, *« Generalized Rewrite Theories»*.
In Proc. Thirtieth International Colloquium on Automata, Languages and Programming, Springer-Verlag LNCS, 2003.

[RDT 97] Rapide Design Team, « *Rapide 1.0. Pattern Language* ». Reference Manual, July 1997.

[SDK 95] M.Shaw, R.DeLine, D.Klein, T.Ross, D.Young et G.Zelesnik, « *Abstractions for Software Architecture and Tools to Support Them* ». IEEE Transactions on Software Engineering: Special Issue on Software Architecture. Avril 1995.

[Spi 92] J. M. Spivey, The Z Notation: A Reference Manual, Second Edition, by Prentice Hall International (UK) Ltd, première publication en 1992.

[Ste 98] Steria, Manuel de référence du langage B, version 1.8.1, STERIA support Atelier B, Aix-en-Provence, 27 novembre 1998.

[Tea 94] Rapide Design Team, "Rapide 1.0. Pattern Language", *Reference Manual*, July 1997.

[TrS 00] H. Treharne et S. Schneider, « *How to Drive a B Machine*», ZB'2000: Formal Specification and Development in Z and B, LNCS 1878, 2000.

[War 89] B.Warboys, "The IPSE 2.5. project: process modelling as the basis for a support environment", *1st International Conference on Software Development, Environments and Factories* , pp 59–74, Berlin, Germany 1989.

[WoD 98] J.Woodcock and J. Davies. «*Using Z: Specification, Refinement, and Proof*». Prentice-Hall, 1998.

[Zel 96] G. Zelesnik, «*The UniCon Language Reference Manual*». School of Computer Science Carnegie Mellon Universtity, Pittsburgh , Mai 1996.

G. Zelesnik, «*The UniCon Language User Manual*». School of Computer Science Carnegie Mellon Universtity, Pittsburgh, Juin 1996.

Annexe A

```
fmod TRANSLATOR-REVIEW is

pr QID .
pr STRING .
--- pr BOOL .


sort Description  DescriptionSet  DescriptionRec LabelledType LabelledTypeSet  ValueDeclaration
ValueDeclarationSet .

--- subsort Description DescriptionRec < DescriptionSet .
--- op deferredDescription : -> DescriptionSet .
--- op _;_ : DescriptionSet DescriptionSet -> DescriptionSet [prec 50 assoc comm id:
deferredDescription]  .
op value_is abstraction();{_} : Qid  ValueDeclarationSet  -> Description [ ctor prec 48 ] .
op value_is abstraction(_);{_} : Qid  LabelledTypeSet ValueDeclarationSet  -> DescriptionRec [
prec 48 ] .

--- subsort Qid < ValueDeclarationSet .

--- op translator : Description -> ValueDeclarationSet .
--- var ID : Qid .
--- var VS : ValueDeclarationSet .
--- eq translator(value ID is abstraction();{VS}) = VS .


op archetype_is component { types is{deferredTypeDL} ports is{deferredAPL} incoming
is{deferredCDL} outgoing is{deferredCDL} behaviour is{_}} :
                                Qid  ValueDeclarationSet -> ArchetypeConstituent
                               [format (n d d d d d d d d d d d d d d d d d d d d d d d d d d d d
d d d n) ]  .

subsort  ArchetypeConstituent < Description .
var ID : Qid .
var VS : ValueDeclarationSet .

sort ArchetypeConstituent .
op translator : Description -> ArchetypeConstituent .
--- op translator : DescriptionSet -> ArchetypeConstituent .

op translator : DescriptionRec -> ArchetypeConstituent .

eq  translator(value ID is abstraction();{VS}) =
archetype ID is component { types is{deferredTypeDL} ports is{deferredAPL} incoming
is{deferredCDL} outgoing is{deferredCDL} behaviour is{VS}} .

var LTS :  LabelledTypeSet  .
eq  translator(value ID is abstraction(LTS);{VS}) =
archetype ID is component { types is{deferredTypeDL} ports is{deferredAPL} incoming
is{deferredCDL} outgoing is{deferredCDL} behaviour is{VS}} .

--- sort  ArchetypeConstituentsSet .
--- op deferredCons : -> ArchetypeConstituentsSet .
--- op _._ : ArchetypeConstituentsSet ArchetypeConstituentsSet -> ArchetypeConstituentsSet [assoc
comm id: deferredCons ] .

--- op translator : DescriptionSet -> ArchetypeConstituentsSet .
--- var DS : DescriptionSet .
--- var D : Description .
--- eq translator(D ; DS) = translator(D) . translator(DS) .
--- eq translator(deferredDescription) = deferredDescription .

subsort ValueDeclaration < ValueDeclarationSet .
op defferedV : -> ValueDeclarationSet .
op _;_ : ValueDeclarationSet ValueDeclarationSet -> ValueDeclarationSet [prec 49 assoc  id:
defferedV ] .
op value_is_ : IdentifierList Literal -> ValueDeclaration [prec 46 ] .

sort Prefix .
```

```
subsort Prefix < ValueDeclaration .
op via_send_ :  Qid IdentifierList -> Prefix [prec 46] .
op via_send_ : Qid String -> Prefix [prec 46] .
op via_receive_ : Qid LabelledTypeSet -> Prefix [prec 46] .
op unobservable : -> Prefix .
op done : -> Prefix .
op recurse : -> Prefix .


sort TrueFalse .
--- op via_send_ : Qid TrueFalse -> Prefix [prec 46] .
op true : -> TrueFalse .
op false : -> TrueFalse .


sorts  PrefixOr PrefixOrList .
op choose{_} : PrefixOrList -> Prefix  .

sort Compose ComposeAndList .
subsort Compose < ComposeAndList .
op compose{_} : ComposeAndList -> ValueDeclaration .

op defferedAC : -> ComposeAndList .
op _and_ : ComposeAndList ComposeAndList  -> ComposeAndList  [ prec 46 comm assoc id: defferedAC]
.

op _(_) : Qid String -> Compose .
op _() : Qid -> Compose .


subsort PrefixOr < PrefixOrList .
op defferedOP : -> PrefixOrList .
op _or_ : PrefixOrList PrefixOrList -> PrefixOrList [comm assoc id: defferedOP] .
op {_} : ValueDeclarationSet -> PrefixOr .

op {_} : Prefix -> Prefix .

op if(_)do{_} : Condition ValueDeclarationSet ->  Prefix [prec 46] .
op if(_)then{_}else{_} : Condition ValueDeclarationSet ValueDeclarationSet  ->  Prefix [prec 46]
.



subsort LabelledType < LabelledTypeSet .
op deferredLabel : -> LabelledTypeSet .
op _`,_ : LabelledTypeSet LabelledTypeSet -> LabelledTypeSet [prec 43 comm assoc id:
deferredLabel] .
op _:_ : Qid Type  -> LabelledType [ctor] .



sort IdentifierList .
subsort Qid < IdentifierList .
subsort TrueFalse < IdentifierList .
op deferredId : -> IdentifierList .
op _,_ : IdentifierList IdentifierList -> IdentifierList [comm assoc id: deferredId] .


sort Literal BooleanLiteral StringLiteral ConnectionLiteral .
subsorts  BooleanLiteral StringLiteral ConnectionLiteral <  Literal  .



sort Condition .
subsort BooleanLiteral  Qid < Condition .

op _==_ : Qid Qid  -> Condition .
op _<>_ : Qid Qid  -> Condition .
op _<>_ : Qid String  -> Condition .

--- op true : -> BoolLiteral .
--- op false : -> BoolLiteral .
```

```
subsort String < StringLiteral  .


op connection(_) : TypeList -> ConnectionLiteral .
op free connection(_) : TypeList -> ConnectionLiteral .

sort Type  TypeList .
subsort Type < TypeList .
op deferredType : -> TypeList .
op _,_ : TypeList TypeList -> TypeList [id: deferredType] .
op Boolean : -> Type .
op String : -> Type .
op Any : -> Type .
op connection[_] : TypeList -> Type .
op abstraction[_] : TypeList -> Type .

endfm


    ----------


mod SemanticsRefinerArch is

extending ARLPARSER-LAST .



--- projectAPorts(a)
 --- op archetype_is architecture{_____} : ArchitectureDef TypingDeclaration PortDeclaration
IncomingDeclaration
 ---                                    OutgoingDeclaration BehaviourDeclaration
 ---                                    ->  ArchetypeDefinition .

    op projectAPorts : ArchetypeDefinition -> ArchetypeDefinitionPortList .
    op projectAConstituents : ArchetypeDefinition -> ArchetypeDefinitionConstituentsList .
    op projectAComponents : ArchetypeDefinition -> ArchetypeDefinitionConstituentsList .
    op projectAConnectors : ArchetypeDefinition -> ArchetypeDefinitionConstituentsList .


    var PDD PDD'  : PortDef .
    var AD : ArchitectureDef .
    var CD : ComponentDef .
    var CCD : ConnectorDef .
    var TD : TypingDeclaration .
    var PD : PortDeclaration .
    var ADPL ADPL' : ArchetypeDefinitionPortList .
    vars ADCL  ADCL' ADCL'' ADCL1  ADCL2  : ArchetypeDefinitionConstituentsList .

    var ADC : ArchetypeDefinitionComp .
    var ADCC : ArchetypeDefinitionConn .
    var ID : IncomingDeclaration .
    var OD : OutgoingDeclaration .
    var BD : BehaviourDeclaration .
    vars CL CL' : ConnectionDeclarationADLList  .
    var APL : ArchetypeDefinitionPortList .
    var ACC ACC' ACC2 : ArchetypeDefinitionConstituents .
    var ADPP : ArchetypeDefinitionPortList .
    var ADP : ArchetypeDefinitionPort .


    eq projectAPorts(archetype AD is architecture{ TD ports is {ADPL} ID OD BD }) = ADPL .
    eq projectAConstituents(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} }) =
ADCL .

    op projectCConstituents : ArchetypeDefinitionComp -> ArchetypeDefinitionConstituentsList .
    eq projectCConstituents(archetype CD is component{ TD PD ID OD behaviour is {ADCL} }) = ADCL
.

    eq projectCConstituents(archetype CD is component{ behaviour is {ADCL} }) = ADCL .


    op projectCConstituents : ArchetypeDefinitionConn -> ArchetypeDefinitionConstituentsList .
```

```
    eq projectCConstituents(archetype CD is connector{ TD PD ID OD behaviour is {ADCL} }) = ADCL
.

    eq projectCConstituents(archetype CD is connector{ behaviour is {ADCL} }) = ADCL .




    op comp : ArchetypeDefinitionConstituentsList -> ArchetypeDefinitionConstituentsList .
    eq comp(ADCC . ADCL) = comp(ADCL)  .
    eq comp(ADC . ADCL) = (ADC . comp(ADCL)) .
    eq comp(deferredCC2) = deferredCC2 .
    eq comp(ADCL) = ADCL .


    op extractsACom :  Qid  ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionConstituentsList .
    ceq extractsACom( Q, (archetype CD is component{ TD PD ID OD BD } . ADCL))
                              = archetype CD is component{ TD PD ID OD BD }  if (CD == Q)  .
    ceq extractsACom( Q, (archetype CD is component{ TD PD ID OD BD } . ADCL))
                             =  deferredBeh  if (CD =/= Q)  .


    var  TAL1 TDL TDL1 : TypeDeclarationADLList .
    var  ADPL1 ADPL2 : ArchetypeDefinitionPortList .
    var  ACON1 ACON2 : ConnectionDeclarationADLList .

    op extractsTypesCons :  ArchetypeDefinitionConstituents ->  TypeDeclarationADLList  .
    eq extractsTypesCons(archetype CD is component{ types is {TAL1} PD ID OD behaviour is {ADCL}
}) =  TAL1 .
    eq extractsTypesCons(archetype CD77 is connector{ types is {TAL1} PD ID OD behaviour is
{ADCL} }) =  TAL1 .


    op changeTypesCons : TypeDeclarationADLList TypeDeclarationADLList
ArchetypeDefinitionConstituents
                                       -> ArchetypeDefinitionConstituents .
    eq changeTypesCons(TDL,TDL1,(archetype CD is component{ types is {TDL} PD ID OD BD })) =
archetype CD is component{ types is {TDL1} PD ID OD BD } .
    eq changeTypesCons(TDL,TDL1,(archetype CD77 is connector{ types is {TDL} PD ID OD BD })) =
archetype CD77 is connector{ types is {TDL1} PD ID OD BD } .


    op changePortsCons : ArchetypeDefinitionPortList ArchetypeDefinitionPortList
ArchetypeDefinitionConstituents
                                       -> ArchetypeDefinitionConstituents .
    eq changePortsCons(ADPL1,ADPL2,(archetype CD is component{ types is {TDL} ports is {ADPL1} ID
OD BD })) =
                                archetype CD is component{ types is {TDL} ports is {ADPL2} ID
OD BD } .

    eq changePortsCons(ADPL1,ADPL2,(archetype CD77 is connector{ types is {TDL} ports is {ADPL1}
ID OD BD })) =
                                archetype CD77 is connector{ types is {TDL} ports is {ADPL2} ID
OD BD } .


    var  ACCN ACCN1 ACCN2 :  ArchetypeDefinitionConstituentsList .


    op changeConsCons : ArchetypeDefinitionConstituentsList ArchetypeDefinitionConstituentsList
ArchetypeDefinitionConstituents
                                       -> ArchetypeDefinitionConstituents .
    eq changeConsCons(ACCN1,ACCN2,(archetype CD is component{ types is {TDL} ports is {ADPL1} ID
OD behaviour is {ACCN1} })) =
                                archetype CD is component{ types is {TDL} ports is {ADPL1} ID
OD behaviour is {ACCN2} } .

    eq changeConsCons(ACCN1,ACCN2,(archetype CD77 is connector{ types is {TDL} ports is {ADPL1}
ID OD behaviour is {ACCN1} })) =
                                archetype CD77 is connector{ types is {TDL} ports is {ADPL1} ID
OD behaviour is {ACCN2} } .
```

```
    var OCN OCN1 : ConnectionDeclarationADLList .
    op changeOCCons : ConnectionDeclarationADLList ConnectionDeclarationADLList
ArchetypeDefinitionConstituents
                                      -> ArchetypeDefinitionConstituents .
    eq changeOCCons(OCN,OCN1,(archetype CD is component{ types is {TDL} PD ID outgoing is {OCN}
BD })) =
                                        archetype CD is component{ types is {TDL} PD ID
outgoing is {OCN1}  BD } .
    eq changeOCCons(OCN,OCN1,(archetype CD77 is connector{ types is {TDL} PD ID outgoing is {OCN}
BD })) =
                                        archetype CD77 is connector{ types is {TDL} PD ID
outgoing is {OCN1}  BD } .


     var ICN ICN1 : ConnectionDeclarationADLList .
    op changeICCons : ConnectionDeclarationADLList ConnectionDeclarationADLList
ArchetypeDefinitionConstituents
                                      -> ArchetypeDefinitionConstituents .
    eq changeICCons(ICN,ICN1,(archetype CD is component{ types is {TDL} PD incoming is {ICN}
outgoing is {OCN} BD })) =
                                        archetype CD is component{ types is {TDL} PD incoming
is {ICN1} outgoing is {OCN}  BD } .

    eq changeICCons(ICN,ICN1,(archetype CD77 is connector{ types is {TDL} PD incoming is {ICN}
outgoing is {OCN} BD })) =
                                        archetype CD77 is connector{ types is {TDL} PD incoming
is {ICN1} outgoing is {OCN}  BD } .


    op  extractsATypes :     ArchetypeDefinition -> TypeDeclarationADLList .
    eq extractsATypes(archetype AD is architecture{ types is {TAL1} ports is {ADPL} ID OD BD }) =
TAL1 .

    op extractsICArch :  ArchetypeDefinition  ->  ConnectionDeclarationADLList .
    eq extractsICArch(archetype AD is architecture{ types is {TAL1} ports is {ADPL} incoming is
{ACON1} OD BD }) = ACON1 .

    op extractsOCArch :  ArchetypeDefinition  ->  ConnectionDeclarationADLList .
    eq extractsOCArch(archetype AD is architecture{ types is {TAL1} ports is {ADPL} ID outgoing
is {ACON2} BD }) = ACON2 .


    op extractsPortsCons :  ArchetypeDefinitionConstituents  ->  ArchetypeDefinitionPortList .
    eq extractsPortsCons(archetype CD is component{ TD  ports is {ADPL1} ID OD behaviour is
{ADCL} }) = ADPL1 .
    eq extractsPortsCons(archetype CD77 is connector{ TD  ports is {ADPL1} ID OD behaviour is
{ADCL} }) = ADPL1 .



    op extractsICCons :  ArchetypeDefinitionConstituents  ->  ConnectionDeclarationADLList .
    eq extractsICCons(archetype CD is component{ types is {TAL1} PD incoming is {ACON1} OD
behaviour is {ADCL} }) =  ACON1 .
    eq extractsICCons(archetype CD77 is connector{ types is {TAL1} PD incoming is {ACON1}  OD
behaviour is {ADCL} }) =  ACON1 .

    op extractsOCCons :  ArchetypeDefinitionConstituents  ->  ConnectionDeclarationADLList .
    eq extractsOCCons(archetype CD is component{ types is {TAL1} PD   ID outgoing is {ACON2}
behaviour is {ADCL} }) =  ACON2 .
    eq extractsOCCons(archetype CD77 is connector{ types is {TAL1} PD ID outgoing is {ACON2}
behaviour is {ADCL} }) =  ACON2 .


    op extractsConsCons :  ArchetypeDefinitionConstituents   ->
ArchetypeDefinitionConstituentsList  .
    eq extractsConsCons(archetype CD is component{ TD PD ID OD behaviour is {ADCL} }) =  ADCL .
    eq extractsConsCons(archetype CD77 is connector{ TD PD ID OD behaviour is {ADCL} }) =  ADCL .
```

```
    op extractsAConn :  Qid  ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionConstituentsList .
    ceq extractsAConn( Q, (archetype CD77 is connector{ TD PD ID OD BD } . ADCL))
                               = archetype CD77 is connector{ TD PD ID OD BD }  if (CD77 == Q)
.
    ceq extractsAConn( Q, (archetype CD77 is connector{ TD PD ID OD BD } . ADCL))
                            =  deferredBeh  if (CD77 =/= Q)  .

    var ADCL55  ADCL66  :  ArchetypeDefinitionConstituentsList .

     op transformBehComp : ArchetypeDefinitionComp  ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionComp .
     eq transformBehComp(archetype CD is component{ TD PD ID OD behaviour is {ADCL} } , ADCL55) =
                                 archetype CD is component{ TD PD ID OD behaviour is {ADCL55} }
.

   --- 10/06/2004
    --- By Karim

     var ADCL36 :  BehaviourExpressionADL  .

     op transformBehComp : ArchetypeDefinitionComp  ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionComp .
     eq transformBehComp(archetype CD is component{ TD PD ID OD behaviour is {ADCL36} } , ADCL55)
=
                                 archetype CD is component{ TD PD ID OD behaviour is {ADCL55} }
.




     op transformBehComp : ArchetypeDefinitionComp  ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionComp .
     eq transformBehComp(archetype CD is component{ TD PD ID OD behaviour is {deferredBeh} } ,
ADCL55) =
                                 archetype CD is component{ TD PD ID OD behaviour is
abstraction() {ADCL55} } .
    --- 10/06/2004
    --- By Karim



     var  CD55 : ComponentDef  .
     var  AR55 : ArchetypeDefinition .
     var AD55 : ArchitectureDef .

     op transfBehComArch :  ComponentDef  ArchetypeDefinition ArchitectureDef
ArchetypeDefinitionConstituentsList -> ArchetypeDefinition .
     eq transfBehComArch(CD55,archetype AD is architecture{ TD PD ID OD behaviour is {ADCL66}
},AD55,ADCL55) =
                         archetype AD55 is architecture{ TD PD ID OD behaviour is

{replaceco(extractsACom(CD55,ADCL66),transformBehComp(extractsACom(CD55,ADCL66),ADCL55), ADCL66)}
} .

   --- 10/06/2004
    --- By Karim




     var  CD77 : ConnectorDef  .
     var  AR77 : ArchetypeDefinition .
     var  AD77 : ArchitectureDef .
```

```
     op transformBehConn : ArchetypeDefinitionConn  ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionConn .
     eq transformBehConn(archetype CD77 is connector{ TD PD ID OD behaviour is {ADCL} } , ADCL55)
=
                                    archetype CD77 is connector{ TD PD ID OD behaviour is
abstraction() {ADCL55} } .

     op transformBehConn : ArchetypeDefinitionConn  ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionConn .
     eq transformBehConn(archetype CD77 is connector{ TD PD ID OD behaviour is {deferredBeh} } ,
ADCL55) =
                                    archetype CD77 is connector{ TD PD ID OD behaviour is
abstraction() {ADCL55} } .




       op transfBehConnArch :  ConnectorDef  ArchetypeDefinition ArchitectureDef
ArchetypeDefinitionConstituentsList -> ArchetypeDefinition .
       eq transfBehConnArch(CD77,archetype AD is architecture{ TD PD ID OD behaviour is {ADCL66}
},AD55,ADCL55) =
                           archetype AD55 is architecture{ TD PD ID OD behaviour is

{replaceco(extractsAConn(CD77,ADCL66),transformBehConn(extractsAConn(CD77,ADCL66),ADCL55),
ADCL66)} } .



    eq projectAComponents(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} })
          = comp(ADCL) .

    op conn : ArchetypeDefinitionConstituentsList -> ArchetypeDefinitionConstituentsList .
    eq conn(ADC . ADCL) = conn(ADCL)  .
    eq conn(ADCC . ADCL) = (ADCC . conn(ADCL)) .
    eq conn(deferredCC2) = deferredCC2 .
    eq conn(ADCL) = ADCL .

    eq projectAConnectors(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} })
          = conn(ADCL) .

    op projectAPorts : ArchetypeDefinition -> ArchetypeDefinitionPortList .

    eq projectAPorts(archetype AD is architecture{ TD ports is{ APL}  ID OD BD })
          = APL .

    op projectCPorts : ArchetypeDefinitionConstituents -> ArchetypeDefinitionPortList  .
    op projectCCPorts : ArchetypeDefinitionConstituentsList -> ArchetypeDefinitionPortList  .

    eq projectCPorts(archetype CD is component { TD ports is{APL} ID OD BD }  )  = APL  .
    eq projectCPorts(archetype CCD is connector { TD ports is{APL} ID OD BD } ) = APL .
    eq projectCPorts(deferredCC2) = deferredAPL .

    eq projectCCPorts(ACC' . ADCL') = (projectCCPorts(ACC') . projectCCPorts(ADCL'))  .
    eq projectCCPorts(ACC') = projectCPorts(ACC') .
    eq projectCCPorts(deferredCC2) = deferredAPL .


    op projectAllCPorts : ArchetypeDefinitionConstituents -> ArchetypeDefinitionPortList  .
    eq projectAllCPorts(archetype CD is component { TD ports is{APL} ID OD behaviour is {ADCL1} }
)  =
                      (APL  . projectCCPorts(ADCL1)) .
    eq projectAllCPorts(archetype CD is connector { TD ports is{APL} ID OD behaviour is {ADCL1} }
)  =
                      (APL  . projectCCPorts(ADCL1)) .
    eq projectAllCPorts(archetype CD is connector { TD ports is{APL} ID OD behaviour is
{deferredBeh} }  )  =
                       APL   .
    eq projectAllCPorts(archetype CD is component { TD ports is{APL} ID OD behaviour is
{deferredBeh} }  )  =
                       APL   .
```

```
    op projectAllCCPorts : ArchetypeDefinitionConstituentsList -> ArchetypeDefinitionPortList  .
    eq projectAllCCPorts(ACC2 . ADCL2) = (projectAllCCPorts(ACC2) . projectAllCCPorts(ADCL2))  .
    eq projectAllCCPorts(ACC2) = projectAllCPorts(ACC2) .
    eq projectAllCCPorts(deferredCC2) = deferredAPL  .


   op projectAllPorts : ArchetypeDefinition -> ArchetypeDefinitionPortList .
   eq projectAllPorts(archetype AD is architecture{ TD ports is {APL} ID OD behaviour is { ADCL''
} }) =
                  (APL . projectCCPorts(ADCL'')) .

   op projectAllAPorts : ArchetypeDefinition -> ArchetypeDefinitionPortList .
   eq projectAllAPorts(archetype AD is architecture{ TD ports is {APL} ID OD behaviour is {
ADCL'' } }) =
                  (APL . projectAllCCPorts(ADCL'')) .


  op ExtractsPortInAPorts : PortDef ArchetypeDefinitionPortList -> ArchetypeDefinitionPort .
  ceq ExtractsPortInAPorts (PDD, (archetype PDD' is port {ID OD} . ADPL') )
                  = archetype PDD' is port {ID OD} if (PDD == PDD') .

   eq ExtractsPortInAPorts (PDD, (archetype PDD' is port {ID OD} . ADPL') )
                  = deferredAPL [owise] .

   op ExtractsICPort : ArchetypeDefinitionPort -> ConnectionDeclarationADLList .
   eq ExtractsICPort(archetype PDD is port { incoming is {CL} OD }) = CL .

   op ExtractsOCPort : ArchetypeDefinitionPort -> ConnectionDeclarationADLList .
   eq ExtractsOCPort(archetype PDD is port { ID outgoing is {CL'} }) = CL' .

   op ExtractsCPort : ArchetypeDefinitionPort -> ConnectionDeclarationADLList .
   eq ExtractsCPort(archetype PDD is port {  incoming is {CL} outgoing is {CL'} } ) = (CL . CL')
.
   eq ExtractsCPort(deferredAPL) = deferredCDL .

   op ExtractsICPorts : ArchetypeDefinitionPortList -> ConnectionDeclarationADLList .
   eq ExtractsICPorts(ADP . ADPL) = (ExtractsICPort(ADP) . ExtractsICPorts(ADPL)) .
   eq ExtractsICPorts(deferredAPL) = deferredCDL .

   op ExtractsOCPorts : ArchetypeDefinitionPortList -> ConnectionDeclarationADLList .
   eq ExtractsOCPorts(ADP . ADPL) = (ExtractsOCPort(ADP) . ExtractsOCPorts(ADPL)) .
   eq ExtractsOCPorts(deferredAPL) = deferredCDL .

   op ExtractsOCPorts : ArchetypeDefinitionPortList -> ConnectionDeclarationADLList .
   eq ExtractsOCPorts(ADP . ADPL) = (ExtractsOCPort(ADP) . ExtractsOCPorts(ADPL)) .
   eq ExtractsOCPorts(deferredAPL) = deferredCDL .

   op ExtractsCPorts : ArchetypeDefinitionPortList -> ConnectionDeclarationADLList .
   eq ExtractsCPorts(ADP . ADPL) = (ExtractsCPort(ADP) . ExtractsCPorts(ADPL)) .
   eq ExtractsCPorts(deferredAPL) = deferredCDL .

   op ExtractsAPIC : ArchetypeDefinition PortDef -> ConnectionDeclarationADLList .
   eq ExtractsAPIC(archetype AD is architecture{ TD ports is{APL} incoming is {CL} outgoing is
{CL'} BD }, PDD) =
               (ExtractsCPort(ExtractsPortInAPorts(PDD,APL)) ) .

   op ExtractsAPOC : ArchetypeDefinition PortDef -> ConnectionDeclarationADLList .
   eq ExtractsAPOC(archetype AD is architecture{ TD ports is{APL} incoming is {CL} outgoing is
{CL'} BD }, PDD) =
               (ExtractsOCPort(ExtractsPortInAPorts(PDD,APL)) ) .

   op projectAConnections : ArchetypeDefinition PortDef -> ConnectionDeclarationADLList .
   eq projectAConnections(archetype AD is architecture{ TD ports is{APL} incoming is {CL}
outgoing is {CL'} BD }, PDD) =
               (ExtractsCPorts(APL) . CL . CL' ) .




--- projectAPorts (architecture)
--- projectABehPorts (architecture)
--- projectAConstituents(architecture)
```

```
--- projectAComponents(architecture)
--- projectAConnectors(architecture)
--- projectAConnections(architecture)


var TP TP' TP1 TP2 : TypeDeclarationADL .
var TPS TPS' TPS'' : TypeDeclarationADLList .

op  includesTypes? :  TypeDeclarationADL TypeDeclarationADLList   -> Bool .
eq  includesTypes?(TP' , deferredTypeDL) = false .
eq  includesTypes?( TP',(TP . TPS)) =  (TP' == TP) or includesTypes?(TP', TPS) .

op  includesTypes? : TypeDeclarationADLList TypeDeclarationADLList -> Bool .
eq  includesTypes?( deferredTypeDL, TPS) = false .
eq  includesTypes?((TP' . TPS'),TPS) =
      includesTypes?(TP',TPS) or includesTypes?(TPS', TPS) .

op  add : TypeDeclarationADL TypeDeclarationADLList -> TypeDeclarationADLList .
ceq add(TP, TPS) = TPS
    if includesTypes?(TP, TPS) .
ceq add(TP,TPS) = (TPS . TP)
    if not(includesTypes?(TP, TPS)) .

op  add : TypeDeclarationADLList TypeDeclarationADLList -> TypeDeclarationADLList .
eq  add(deferredTypeDL, TPS) = TPS .
eq  add((TP' . TPS'), TPS) =
    add(TPS',add(TP', TPS)) .

--- les deux listes doivent Ãªtre disjoint pour rajouter un Ã©lÃ©ment !
--- ajouter la premiÃ¨re dans la deuxiÃ¨me

--- op  add : TypeDeclarationADLList TypeDeclarationADLList -> TypeDeclarationADLList .
--- ceq add(TPS', TPS) = TPS
---    if includesTypes?(TPS', TPS) .
--- ceq add(TPS',TPS) = (TPS . TPS')
---    if not(includesTypes?(TPS', TPS)) .

op  remove : TypeDeclarationADL TypeDeclarationADLList -> TypeDeclarationADLList .
eq  remove(TP,deferredTypeDL) = deferredTypeDL .
ceq remove(TP', (TP . TPS)) = remove(TP',TPS)
    if (TP' == TP) .
ceq remove(TP', (TP . TPS)) = (TP . remove(TP',TPS))
    if (TP' =/= TP) .

op  remove : TypeDeclarationADLList TypeDeclarationADLList -> TypeDeclarationADLList .
eq  remove(deferredTypeDL, TPS) = TPS .
eq  remove((TP' . TPS'), TPS) =
    remove(TPS', remove(TP', TPS)) .


--- Tester si le deuxiÃ¨me param (ensemble) est un sous-ensemble du premier param (ensemble)
strictement .

op subTypes? : TypeDeclarationADLList TypeDeclarationADLList -> Bool .
eq subTypes?(TPS', deferredTypeDL) = true .
eq subTypes?(TPS',(TP . TPS) ) = includesTypes?(TP, TPS') and subTypes?(TPS',TPS) .


op  replace : TypeDeclarationADL TypeDeclarationADL TypeDeclarationADLList ->
TypeDeclarationADLList .
eq  replace(TP1,deferredTypeDL,(TP1 . TPS)) = TPS .
ceq  replace(TP1, TP2,(TP1 . TPS)) = (TP2 . TPS)
   if (TP2 =/= TP1) and not(includesTypes?(TP1,TPS))  .
ceq  replace(TP1, TP2, TPS) = TPS
   if (TP2 =/= TP1) and not(includesTypes?(TP1,TPS))  .
ceq replace(TP1, TP2, (TP1 . TPS)) = (TP1 . TPS)
   if (TP2 == TP1) .

op becomes : TypeDeclarationADLList TypeDeclarationADLList TypeDeclarationADLList ->
TypeDeclarationADLList .
eq becomes(TPS,TPS',TPS'') = remove(TPS, add(TPS',TPS'')) .
```

```
vars   CL1 CL2 : ConnectionDeclarationADL  .
vars CLS1 CLS2 CLS3  : ConnectionDeclarationADLList  .

op  includesConnections? :  ConnectionDeclarationADL ConnectionDeclarationADLList  ->  Bool .
eq  includesConnections?(CL1 , deferredCDL) = false .
eq  includesConnections?(CL1,(CL2 . CLS1)) =  (CL1 == CL2) or includesConnections?(CL1, CLS1) .

op  includesConnections? : ConnectionDeclarationADLList ConnectionDeclarationADLList  -> Bool .
eq  includesConnections?(deferredCDL, CLS1) = false .
eq  includesConnections?((CL2 . CLS2),CLS1) =
       includesConnections?(CL2,CLS1) or includesConnections?(CLS2, CLS1) .

op  addcc : ConnectionDeclarationADL ConnectionDeclarationADLList  ->
ConnectionDeclarationADLList  .

ceq addcc(CL1, CLS1) = CLS1
    if includesConnections?(CL1, CLS1) .
ceq addcc(CL1,CLS1) = (CLS1 . CL1)
    if not(includesConnections?(CL1, CLS1)) .

op  addcc : ConnectionDeclarationADLList ConnectionDeclarationADLList ->
ConnectionDeclarationADLList .
eq  addcc(deferredCDL, CLS1) = CLS1 .
eq  addcc((CL2 . CLS2), CLS1) =
    addcc(CLS2,addcc(CL2, CLS1)) .

op  removecc : ConnectionDeclarationADL ConnectionDeclarationADLList ->
ConnectionDeclarationADLList .
eq  removecc(CL1,deferredCDL) = deferredCDL .
ceq removecc(CL2, (CL1 . CLS1)) = removecc(CL2,CLS1)
    if (CL2 == CL1) .
ceq removecc(CL2, (CL1 . CLS1)) = (CL1 . removecc(CL2,CLS1))
    if (CL2 =/= CL1) .

op  removecc : ConnectionDeclarationADLList ConnectionDeclarationADLList ->
ConnectionDeclarationADLList .
eq  removecc(deferredCDL, CLS1) = CLS1 .
eq  removecc((CL2 . CLS2), CLS1) =
    removecc(CL2, removecc(CLS2, CLS1)) .

op subConnections? : ConnectionDeclarationADLList ConnectionDeclarationADLList -> Bool .
eq subConnections?(CLS2, deferredCDL) = true .
eq subConnections?(CLS2,(CL1 . CLS1) ) = includesConnections?(CL1, CLS2) and
subConnections?(CLS2,CLS1) .

op  replacecc : ConnectionDeclarationADL ConnectionDeclarationADL
                                        ConnectionDeclarationADLList ->
ConnectionDeclarationADLList .

eq  replacecc(CL1,deferredCDL,(CL1 . CLS1)) = CLS1 .
ceq  replacecc(CL1, CL2,(CL1 . CLS1)) = (CL2 . CLS1)
   if (CL2 =/= CL1) and not(includesConnections?(CL1,CLS1))  .
ceq  replacecc(CL1, CL2, CLS1) = CLS1
   if (CL2 =/= CL1) and not(includesConnections?(CL1,CLS1))  .
ceq replacecc(CL1, CL2, (CL1 . CLS1)) = (CL1 . CLS1)
   if (CL2 == CL1) .


op becomes : ConnectionDeclarationADLList ConnectionDeclarationADLList
ConnectionDeclarationADLList -> ConnectionDeclarationADLList .
eq becomes(CLS1,CLS2,CLS3) = removecc(CLS1,addcc(CLS2,CLS3)) .


vars   P1 P2 : ArchetypeDefinitionPort .
vars PS1 PS2 PS3  : ArchetypeDefinitionPortList  .

op  includesPorts? :  ArchetypeDefinitionPort ArchetypeDefinitionPortList  ->  Bool .
eq  includesPorts?(P1 , deferredAPL) = false .
eq  includesPorts?(P1,(P2 . PS1)) =  (P1 == P2) or includesPorts?(P1, PS1) .

op  includesPorts? : ArchetypeDefinitionPortList ArchetypeDefinitionPortList  -> Bool .
eq  includesPorts?(deferredAPL, PS1) = false .
eq  includesPorts?((P2 . PS2),PS1) =
```

```
       includesPorts?(P2,PS1) or includesPorts?(PS2, PS1) .

op  addpp : ArchetypeDefinitionPort ArchetypeDefinitionPortList  ->  ArchetypeDefinitionPortList
.

ceq addpp(P1, PS1) = PS1
    if includesPorts?(P1, PS1) .
ceq addpp(P1,PS1) = (PS1 . P1)
    if not(includesPorts?(P1, PS1)) .

op  addpp : ArchetypeDefinitionPortList ArchetypeDefinitionPortList ->
ArchetypeDefinitionPortList .
eq  addpp(deferredAPL, PS1) = PS1 .
eq  addpp((P2 . PS2), PS1) =
    addpp(PS2,addpp(P2, PS1)) .

op  removepp : ArchetypeDefinitionPort ArchetypeDefinitionPortList -> ArchetypeDefinitionPortList
.
eq  removepp(P1,deferredAPL) = deferredAPL .
ceq removepp(P2, (P1 . PS1)) = removepp(P2,PS1)
    if (P2 == P1) .
ceq removepp(P2, (P1 . PS1)) = (P1 . removepp(P2,PS1))
    if (P2 =/= P1) .

op  removepp : ArchetypeDefinitionPortList ArchetypeDefinitionPortList ->
ArchetypeDefinitionPortList .
eq  removepp(deferredAPL, PS1) = PS1 .
eq  removepp((P2 . PS2), PS1) =
    removepp(P2, removepp(PS2, PS1)) .

op subPorts? : ArchetypeDefinitionPortList ArchetypeDefinitionPortList -> Bool .
eq subPorts?(PS2, deferredAPL) = true .
eq subPorts?(PS2,(P1 . PS1) ) = includesPorts?(P1, PS2) and subPorts?(PS2,PS1) .

op  replacepp : ArchetypeDefinitionPort ArchetypeDefinitionPort
                                        ArchetypeDefinitionPortList ->
ArchetypeDefinitionPortList .

eq  replacepp(P1,deferredAPL,(P1 . PS1)) = PS1 .
ceq  replacepp(P1, P2,(P1 . PS1)) = (P2 . PS1)
   if (P2 =/= P1) and not(includesPorts?(P1,PS1))  .
ceq  replacepp(P1, P2, PS1) = PS1
   if (P2 =/= P1) and not(includesPorts?(P1,PS1))  .
ceq replacepp(P1, P2, (P1 . PS1)) = (P1 . PS1)
   if (P2 == P1) .


op becomes : ArchetypeDefinitionPortList ArchetypeDefinitionPortList ArchetypeDefinitionPortList
-> ArchetypeDefinitionPortList .
eq becomes(PS1,PS2,PS3) = removepp(PS1,addpp(PS2,PS3)) .



op includesInputConnectionPort : ConnectionDeclarationADLList PortDef ArchetypeDefinitionPortList
->  ConnectionDeclarationADLList  .
eq includesInputConnectionPort(CL,PDD,PS1) = addcc(CL,
ExtractsICPort(ExtractsPortInAPorts(PDD,PS1))) .

op changeInputPortIncluding : PortDef  ConnectionDeclarationADLList  ArchetypeDefinitionPortList
                                        -> ArchetypeDefinitionPortList .

eq changeInputPortIncluding(PDD,CL,PS1) =
              replacePort(PDD,archetype PDD is port { incoming is {
includesInputConnectionPort(CL,PDD,PS1)}
                                      outgoing is {ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1))
}},PS1) .




var Q' Q'' : Qid .
var C11 : ConnectionDeclarationADL .
```

```
op isConnection : Qid ConnectionDeclarationADL -> ConnectionDeclarationADL .

ceq isConnection(Q,Q' is connection(Q'')) = Q' is connection(Q'') if (Q == Q') .

ceq isConnection(Q,Q' is connection(Q'')) = deferredCDL   if (Q =/= Q') .

eq isConnection(Q,deferredCDL) =  deferredCDL   .

op isConnectionList : Qid ConnectionDeclarationADLList -> ConnectionDeclarationADL .
eq isConnectionList(Q,(C11 . CL)) = (isConnection(Q,C11) . isConnectionList(Q,CL)) .

ceq isConnectionList(Q,Q' is connection(Q'')) = Q' is connection(Q'') if (Q == Q') .

ceq isConnectionList(Q,Q' is connection(Q'')) = deferredCDL   if (Q =/= Q') .

eq isConnectionList(Q,deferredCDL) =  deferredCDL   .

op isAConnectionsList : Contenu ConnectionDeclarationADLList -> ConnectionDeclarationADLList .
eq isAConnectionsList(({Q1 ; QL}),CL) = (isConnectionList(Q1,CL) . isAConnectionsList(({QL}),CL))
.
eq isAConnectionsList({nonecons},CL) = deferredCDL .



var C22 : ConnectionDeclarationADL .


op includesConnection? :  ConnectionDeclarationADL  ConnectionDeclarationADLList  -> Bool .
eq  includesConnection?(C11, deferredCDL) = false .
eq includesConnection?(C11,(C22 . CL1)) = (C11 == C22) or includesConnection?(C11,CL1) .

---  A completer
op isConnectionList? :   ConnectionDeclarationADLList ConnectionDeclarationADLList  -> Bool .

--- A completer

op subConnections? : ConnectionDeclarationADLList  ConnectionDeclarationADLList  -> Bool .


--- A completer


op includesInputConnectionPort : ConnectionDeclarationADLList PortDef ArchetypeDefinitionPortList
->  ConnectionDeclarationADLList  .
eq includesInputConnectionPort(CL,PDD,PS1) = addcc(CL,
ExtractsICPort(ExtractsPortInAPorts(PDD,PS1))) .

op changeInputPortIncluding : PortDef  ConnectionDeclarationADLList  ArchetypeDefinitionPortList
                                    -> ArchetypeDefinitionPortList .

eq changeInputPortIncluding(PDD,CL,PS1) =
               replacePort(PDD,archetype PDD is port { incoming is {
includesInputConnectionPort(CL,PDD,PS1)}
                                   outgoing is {ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1))
}},PS1) .




op includesOutputConnectionPort : ConnectionDeclarationADLList PortDef
ArchetypeDefinitionPortList ->  ConnectionDeclarationADLList  .
eq includesOutputConnectionPort(CL,PDD,PS1) = addcc(CL,
ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1))) .

op changeOutputPortIncluding : PortDef  ConnectionDeclarationADLList  ArchetypeDefinitionPortList
                                    -> ArchetypeDefinitionPortList .

eq changeOutputPortIncluding(PDD,CL,PS1) =
               replacePort(PDD,archetype PDD is port { incoming is
{ExtractsICPort(ExtractsPortInAPorts(PDD,PS1)) }
                                   outgoing is { includesOutputConnectionPort(CL,PDD,PS1)}}
,PS1) .
```

```
op removeConnection : Qid  ConnectionDeclarationADLList  -> ConnectionDeclarationADLList  .
eq removeConnection(Q,CL) = removecc(isConnectionList(Q,CL),CL) .

op removeConnection : Contenu ConnectionDeclarationADLList -> ConnectionDeclarationADLList .
eq removeConnection(C,CL) = removecc(isAConnectionsList(C,CL),CL) .

op replaceConnection : Qid  ConnectionDeclarationADL ConnectionDeclarationADLList  ->
ConnectionDeclarationADLList  .
eq replaceConnection(Q,C11,CL) = replacecc(isConnectionList(Q,CL),C11,CL) .



op excludesInputConnectionPort : Contenu  PortDef ArchetypeDefinitionPortList ->
ConnectionDeclarationADLList  .
eq excludesInputConnectionPort(C,PDD,PS1) = removeConnection(C,
ExtractsICPort(ExtractsPortInAPorts(PDD,PS1))) .

op changeInputPortexcluding : PortDef  Contenu  ArchetypeDefinitionPortList
                                      -> ArchetypeDefinitionPortList .

eq changeInputPortexcluding(PDD,C,PS1) =
              replacePort(PDD,archetype PDD is port { incoming is {
excludesInputConnectionPort(C,PDD,PS1)}
                                     outgoing is {ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1))
}},PS1) .



op excludesOutputConnectionPort : Contenu  PortDef ArchetypeDefinitionPortList ->
ConnectionDeclarationADLList  .
eq excludesOutputConnectionPort(C,PDD,PS1) = removeConnection(C,
ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1))) .

op changeOutputPortexcluding : PortDef  Contenu  ArchetypeDefinitionPortList
                                      -> ArchetypeDefinitionPortList .

eq changeOutputPortexcluding(PDD,C,PS1) =
              replacePort(PDD,archetype PDD is port { incoming is
{ExtractsICPort(ExtractsPortInAPorts(PDD,PS1)) }
                                      outgoing is { excludesOutputConnectionPort(C,PDD,PS1)}
},PS1) .




var QC :  ConnectionDeclarationADL .

op replacesInputConnectionPort : Qid ConnectionDeclarationADL  PortDef
ArchetypeDefinitionPortList ->  ConnectionDeclarationADLList  .
eq replacesInputConnectionPort(Q,QC,PDD,PS1) =
replaceConnection(Q,QC,ExtractsICPort(ExtractsPortInAPorts(PDD,PS1))) .


op changeInputPortreplacing : Qid ConnectionDeclarationADL  PortDef  ArchetypeDefinitionPortList
                                      -> ArchetypeDefinitionPortList .

eq changeInputPortreplacing(Q,QC,PDD,PS1) =
              replacePort(PDD,archetype PDD is port { incoming is {
replacesInputConnectionPort(Q,QC,PDD,PS1)}
                                     outgoing is {ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1))
}},PS1) .



op replacesOutputConnectionPort : Qid ConnectionDeclarationADL  PortDef
ArchetypeDefinitionPortList ->  ConnectionDeclarationADLList  .
```

```
eq replacesOutputConnectionPort(Q,QC,PDD,PS1) =
replaceConnection(Q,QC,ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1))) .


op changeOutputPortreplacing : Qid ConnectionDeclarationADL  PortDef  ArchetypeDefinitionPortList
                                      -> ArchetypeDefinitionPortList .

eq changeOutputPortreplacing(Q,QC,PDD,PS1) =
               replacePort(PDD,archetype PDD is port {incoming is
{ExtractsICPort(ExtractsPortInAPorts(PDD,PS1)) }
                                      outgoing is { replacesOutputConnectionPort(Q,QC,PDD,PS1)}
},PS1) .

var CL11 CL22 CL33 : ConnectionDeclarationADLList .

op becomes : ConnectionDeclarationADLList ConnectionDeclarationADLList
ConnectionDeclarationADLList  -> ConnectionDeclarationADLList  .
eq becomes(CL11,CL22,CL33) = removecc(CL11,addcc(CL22,CL33)) .

op changeOutputPortbecomes : Contenu ConnectionDeclarationADLList PortDef
ArchetypeDefinitionPortList ->  ArchetypeDefinitionPortList .
eq changeOutputPortbecomes(C,CL11,PDD,PS1) =
               replacePort(PDD,archetype PDD is port {incoming is
{ExtractsICPort(ExtractsPortInAPorts(PDD,PS1)) }
                                      outgoing is {
becomes(isAConnectionsList(C,ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1))), CL11,
                                          ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1))) }
},PS1) .


op changeInputPortbecomes : Contenu ConnectionDeclarationADLList PortDef
ArchetypeDefinitionPortList ->  ArchetypeDefinitionPortList .
eq changeInputPortbecomes(C,CL11,PDD,PS1) =
               replacePort(PDD,archetype PDD is port { incoming is {
becomes(isAConnectionsList(C,ExtractsICPort(ExtractsPortInAPorts(PDD,PS1))), CL11,
                                          ExtractsICPort(ExtractsPortInAPorts(PDD,PS1))) }
                                         outgoing is
{ExtractsOCPort(ExtractsPortInAPorts(PDD,PS1)) } },PS1) .



--- excludesTypes?
--- replacesTypes?

--- includesPorts?
--- excludesPorts?
--- replacesPorts?

--- includesConnections?
--- excludesConnections?
--- replacesConnections?


op deferredCC2 : -> ArchetypeDefinitionConstituentsList .
op _._ : ArchetypeDefinitionConstituentsList ArchetypeDefinitionConstituentsList
                        -> ArchetypeDefinitionConstituentsList
                                             [prec 93 ctor assoc comm id: deferredCC2 format (d
d d d) ] .

vars   C1 C2 : ArchetypeDefinitionConstituents .
vars CS1 CS2 CS3  : ArchetypeDefinitionConstituentsList  .

op  includesConstituents? :  ArchetypeDefinitionConstituents ArchetypeDefinitionConstituentsList
->  Bool .
eq  includesConstituents?(C1 , deferredCC2 ) = false .
eq  includesConstituents?(C1,(C2 . CS1)) =  (C1 == C2) or includesConstituents?(C1, CS1) .

op  includesConstituents? : ArchetypeDefinitionConstituentsList
ArchetypeDefinitionConstituentsList  -> Bool .
eq  includesConstituents?(deferredCC2 , CS1) = false .
eq  includesConstituents?((C2 . CS2),CS1) =
       includesConstituents?(C2,CS1) or includesConstituents?(CS2, CS1) .
```

```
op  addco : ArchetypeDefinitionConstituents ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionConstituentsList  .

ceq addco(C1, CS1) = CS1
    if includesConstituents?(C1, CS1) .
ceq addco(C1,CS1) = (CS1 . C1)
    if not(includesConstituents?(C1, CS1)) .

op  addco : ArchetypeDefinitionConstituentsList ArchetypeDefinitionConstituentsList ->
ArchetypeDefinitionConstituentsList .
eq  addco(deferredCC2 , CS1) = CS1 .
eq  addco((C2 . CS2), CS1) =
    addco(CS2,addco(C2, CS1)) .

op  removeco : ArchetypeDefinitionConstituents ArchetypeDefinitionConstituentsList ->
ArchetypeDefinitionConstituentsList .
eq  removeco(C1,deferredCC2 ) = deferredCC2  .
ceq removeco(C2, (C1 . CS1)) = removeco(C2,CS1)
    if (C2 == C1) .
ceq removeco(C2, (C1 . CS1)) = (C1 . removeco(C2,CS1))
    if (C2 =/= C1) .

op  removeco : ArchetypeDefinitionConstituentsList ArchetypeDefinitionConstituentsList ->
ArchetypeDefinitionConstituentsList .
eq  removeco(deferredCC2,CS1) = CS1 .
eq  removeco((C2 . CS2), CS1) =
    removeco(C2,removeco(CS2, CS1)) .

op subConstituents? : ArchetypeDefinitionConstituentsList ArchetypeDefinitionConstituentsList ->
Bool .
eq subConstituents?(CS2, deferredCC2 ) = true .
eq subConstituents?(CS2,(C1 . CS1) ) = includesConstituents?(C1, CS2) and
subConstituents?(CS2,CS1) .

op  replaceco : ArchetypeDefinitionConstituents ArchetypeDefinitionConstituents
                                        ArchetypeDefinitionConstituentsList ->
ArchetypeDefinitionConstituentsList .

eq  replaceco(C1,deferredCC2 ,(C1 . CS1)) = CS1 .
ceq  replaceco(C1, C2,(C1 . CS1)) = (C2 . CS1)
   if (C2 =/= C1) and not(includesConstituents?(C1,CS1))  .
ceq  replaceco(C1, C2, CS1) = CS1
   if (C2 =/= C1) and not(includesConstituents?(C1,CS1))  .
ceq replaceco(C1, C2, (C1 . CS1)) = (C1 . CS1)
   if (C2 == C1) .

eq replaceco(deferredCC2,C2,CS1) = CS1 .


op becomes : ArchetypeDefinitionConstituentsList ArchetypeDefinitionConstituentsList
ArchetypeDefinitionConstituentsList -> ArchetypeDefinitionConstituentsList .
eq becomes(CS1,CS2,CS3) = removeco(CS1,addco(CS2,CS3)) .


op isTypeComponent?  : ArchetypeDefinitionConstituents  -> Bool .
eq isTypeComponent?(archetype CD1 is component{ TD PD ID OD BD }) = true .
eq isTypeComponent?(archetype CD2 is connector{ TD PD ID OD BD }) = false .
eq isTypeComponent?(deferredCC2) = false .

op isTypeConnector?  : ArchetypeDefinitionConstituents  -> Bool .
eq isTypeConnector?(archetype CD1 is component{ TD PD ID OD BD }) = false .
eq isTypeConnector?(archetype CD2 is connector{ TD PD ID OD BD }) = true .
eq isTypeConnector?(deferredCC2) = false .

--- project(a,p,connections_incomming)
--- project(a,p,connections_outgoing)
--- project(a,behaviour_connections)

var Q : Qid .
var ADC1 : ArchetypeDefinitionConstituents .
var CD1 : ComponentDef .
var CD2 : ConnectorDef .
```

```
op isComponent : Qid  ArchetypeDefinitionConstituents -> ArchetypeDefinitionComp .

ceq isComponent(Q,archetype CD1 is component{ TD PD ID OD BD }) =
                        archetype CD1 is component{ TD PD ID OD BD } if (Q == CD1) .
ceq isComponent(Q,archetype CD1 is component{ TD PD ID OD BD }) =
                        deferredCC2  if (Q =/= CD1) .
ceq isComponent(Q,archetype CD2 is connector{ TD PD ID OD BD }) =
                        deferredCC2  if (Q =/= CD2) .
ceq isComponent(Q,archetype CD2 is connector{ TD PD ID OD BD }) =
                        deferredCC2  if (Q == CD2) .

eq isComponent(Q,deferredCC2) = deferredCC2 .


op isComponentList : Qid  ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionConstituentsList .
eq isComponentList(Q,(ADC1 . ADCL)) = (isComponent(Q,ADC1) . isComponentList(Q,ADCL)) .
eq isComponentList(Q,archetype CD2 is connector{ TD PD ID OD BD }) = deferredCC2 .
ceq isComponentList(Q,archetype CD1 is component{ TD PD ID OD BD }) =
                        deferredCC2  if (Q =/= CD1) .
eq isComponentList(Q,deferredCC2) = deferredCC2 .

ceq isComponentList(Q,archetype CD1 is component{ TD PD ID OD BD }) =
                        archetype CD1 is component{ TD PD ID OD BD } if (Q == CD1) .


---------------

op isConnector : Qid  ArchetypeDefinitionConstituents -> ArchetypeDefinitionConn .

ceq isConnector(Q,archetype CD2 is connector{ TD PD ID OD BD }) =
                        archetype CD2 is connector{ TD PD ID OD BD } if (Q == CD2) .
ceq isConnector(Q,archetype CD2 is connector{ TD PD ID OD BD }) =
                        deferredCC2  if (Q =/= CD2) .
ceq isConnector(Q,archetype CD1 is component{ TD PD ID OD BD }) =
                        deferredCC2  if (Q =/= CD1) .
ceq isConnector(Q,archetype CD1 is component{ TD PD ID OD BD }) =
                        deferredCC2  if (Q == CD1) .

eq isConnector(Q,deferredCC2) = deferredCC2 .


op isConnectorList : Qid  ArchetypeDefinitionConstituentsList  ->
ArchetypeDefinitionConstituentsList .
eq isConnectorList(Q,(ADC1 . ADCL)) = (isConnector(Q,ADC1) . isConnectorList(Q,ADCL)) .
eq isConnectorList(Q,archetype CD1 is component{ TD PD ID OD BD }) = deferredCC2 .
ceq isConnectorList(Q,archetype CD2 is connector{ TD PD ID OD BD }) =
                        deferredCC2  if (Q =/= CD2) .
eq isConnectorList(Q,deferredCC2) = deferredCC2 .


ceq isConnectorList(Q,archetype CD2 is connector{ TD PD ID OD BD }) =
                        archetype CD2 is connector{ TD PD ID OD BD } if (Q == CD2) .

var AR : ArchetypeDefinition .

op AComponentsIncludes? : ArchetypeDefinition Qid -> Bool .

ceq AComponentsIncludes?(AR,Q) = true  if  (isComponentList(Q,projectAComponents(AR)) =/=
deferredCC2)  .
ceq AComponentsIncludes?(AR,Q) = false  if  (isComponentList(Q,projectAComponents(AR)) ==
deferredCC2)  .

op AConnectorsIncludes? : ArchetypeDefinition Qid -> Bool .

ceq AConnectorsIncludes?(AR,Q) = true  if  (isConnectorList(Q,projectAConnectors(AR)) =/=
deferredCC2)  .
ceq AConnectorsIncludes?(AR,Q) = false  if  (isComponentList(Q,projectAConnectors(AR)) ==
deferredCC2)  .

--- op AComponetsExplodes : ArchetypeDefinition ArchetypeDefinitionComp ->
ArchetypeDefinitionConstituentsList .
--- eq AComponentsExplodes(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} }, Q) =
```

```
--- var Q : Qid .

op AComponentsExplodes : ArchetypeDefinition Qid  -> ArchetypeDefinitionConstituentsList .
ceq AComponentsExplodes(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q) =

removeco(isComponentList(Q,(ADCL)),addco(projectCConstituents(isComponentList(Q,ADCL)),(ADCL)))
      if  AComponentsIncludes?(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q)
.

ceq AComponentsExplodes(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q) =
        ADCL
      if not(AComponentsIncludes?(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL}
},Q)) .

op AConnectorsExplodes : ArchetypeDefinition Qid  -> ArchetypeDefinitionConstituentsList .
ceq AConnectorsExplodes(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q) =

removeco(isConnectorList(Q,(ADCL)),addco(projectCConstituents(isConnectorList(Q,ADCL)),(ADCL)))
      if  AConnectorsIncludes?(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q)
           and (ADCL =/= deferredCC2)  .

--- and add test : the behaviour of the connector must be not empty ?!!

ceq AConnectorsExplodes(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q) =
        ADCL
   if not(AConnectorsIncludes?(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL}
},Q))
         or (ADCL =/= deferredCC2) .


--- eq AConnectorsExplodes(archetype AD is architecture{ TD PD ID OD behaviour is {ADCL} },Q) =
ADCL [owise] .


sort  ConsName  ConsNameList  Contenu .
subsorts  ComponentDef ConnectorDef < ConsName .
subsort ConsName < ConsNameList .

var Q1 : ConsName .
vars  QL   : ConsNameList .
var C : Contenu .

op nonecons : -> ConsNameList .
op _;_ : ConsNameList ConsNameList -> ConsNameList [ assoc comm id: nonecons format (d d d d)] .
op {_} : ConsNameList -> Contenu .


op isAComponentsList : Contenu ArchetypeDefinitionConstituentsList ->
ArchetypeDefinitionConstituentsList .
eq isAComponentsList(({Q1 ; QL}),ADCL) = (isComponentList(Q1,ADCL) .
isAComponentsList(({QL}),ADCL)) .
eq isAComponentsList({nonecons},ADCL) = deferredCC2 .

op isAConnectorsList : Contenu ArchetypeDefinitionConstituentsList ->
ArchetypeDefinitionConstituentsList .
eq isAConnectorsList(({Q1 ; QL}),ADCL) = (isConnectorList(Q1,ADCL) .
isAConnectorsList(({QL}),ADCL)) .
eq isAConnectorsList({nonecons},ADCL) = deferredCC2 .

op isAConstituents : Contenu ArchetypeDefinitionConstituentsList ->
ArchetypeDefinitionConstituentsList .
eq isAConstituents(C,ADCL) = (isAComponentsList(C,ADCL) . isAConnectorsList(C,ADCL))  .


--- op isConstituents? : Contenu
op isAConstituents? : Contenu ArchetypeDefinitionConstituentsList -> Bool .
ceq isAConstituents?(C,ADCL) = true   if  isAConstituents(C,ADCL) =/= deferredCC2  .
ceq isAConstituents?(C,ADCL) = false  if  isAConstituents(C,ADCL) == deferredCC2  .

op AComponentsImplodes : Contenu  ArchetypeDefinitionConstituentsList  ComponentDef ->
ArchetypeDefinitionConstituents .
```

```
eq AComponentsImplodes(C,ADCL,CD) = archetype CD is component { behaviour is {
(isAComponentsList(C,ADCL) . isAConnectorsList(C,ADCL)) } } .


op AImplodes : Contenu  ArchetypeDefinitionConstituentsList  ComponentDef ->
ArchetypeDefinitionConstituentsList .
ceq AImplodes(C,ADCL,CD) =
addco((removeco(projectCConstituents(AComponentsImplodes(C,ADCL,CD)),
ADCL)),AComponentsImplodes(C,ADCL,CD))
   if isAConstituents?(C,ADCL) .

ceq AImplodes(C,ADCL,CD) = ADCL
   if not(isAConstituents?(C,ADCL)) .

var CD5 : ConnectorDef .

op AConnectorsImplodes : Contenu  ArchetypeDefinitionConstituentsList  ConnectorDef ->
ArchetypeDefinitionConstituents .
eq AConnectorsImplodes(C,ADCL,CD5) = archetype CD5 is connector { behaviour is {
(isAComponentsList(C,ADCL) . isAConnectorsList(C,ADCL)) } } .


op AImplodesC : Contenu  ArchetypeDefinitionConstituentsList  ConnectorDef ->
ArchetypeDefinitionConstituentsList .
ceq AImplodesC(C,ADCL,CD5) =
addco((removeco(projectCConstituents(AConnectorsImplodes(C,ADCL,CD5)),
ADCL)),AConnectorsImplodes(C,ADCL,CD5))
   if isAConstituents?(C,ADCL) .

ceq AImplodesC(C,ADCL,CD5) = ADCL
   if not(isAConstituents?(C,ADCL)) .


--- op archetype_is port{__} : PortDef   IncomingDeclaration  OutgoingDeclaration
---                                    ->  ArchetypeDefinitionPort
---                     [prec 84 ctor gather (& & &) format ( nb! o g! g! b! n n ni b!) ] .

--- subsort ArchetypeDefinitionPort < ArchetypeDefinitionPortList .
--- op deferredAPL : ->  ArchetypeDefinitionPortList  .
--- op _._ : ArchetypeDefinitionPortList ArchetypeDefinitionPortList ->
ArchetypeDefinitionPortList
---                       [assoc comm prec 85 id: deferredAPL format (d d d d)] .
--- op archetype_is port{__} : PortDef   IncomingDeclaration  OutgoingDeclaration
----                                     ->  ArchetypeDefinitionPort .

var PDD1 : PortDef .
var IPD : IncomingDeclaration .
var OPD : OutgoingDeclaration  .
var APORT : ArchetypeDefinitionPort .
var APORTL : ArchetypeDefinitionPortList .


op isPort : Qid  ArchetypeDefinitionPort -> ArchetypeDefinitionPort .

ceq isPort(Q,archetype PDD1 is port{ IPD OPD}) = archetype PDD1 is port{ IPD OPD}
                      if (Q == PDD1) .

ceq isPort(Q,archetype PDD1 is port{ IPD OPD}) = deferredAPL
                      if (Q =/= PDD1) .
eq isPort(Q,deferredAPL) = deferredAPL .


op isPortList : Qid  ArchetypeDefinitionPortList -> ArchetypeDefinitionPort .
eq isPortList(Q,(APORT . APORTL)) = (isPort(Q,APORT) . isPortList(Q,APORTL)) .
ceq isPortList(Q, archetype PDD1 is port{ IPD OPD}) = archetype PDD1 is port{ IPD OPD}
                      if (Q == PDD1) .
ceq isPortList(Q,archetype PDD1 is port{ IPD OPD}) = deferredAPL
                      if (Q =/= PDD1) .

eq isPortList(Q,deferredAPL) = deferredAPL .

op isAPortsList : Contenu ArchetypeDefinitionPortList -> ArchetypeDefinitionPortList .
eq isAPortsList(({Q1 ; QL}),APORTL) = (isPortList(Q1,APORTL) . isAPortsList(({QL}),APORTL)) .
```

```
eq isAPortsList({nonecons},APORTL) = deferredAPL .


op removePort : Qid  ArchetypeDefinitionPortList -> ArchetypeDefinitionPortList .
eq removePort(Q,APORTL) = removepp(isPortList(Q,APORTL),APORTL) .

op removePort : Contenu ArchetypeDefinitionPortList -> ArchetypeDefinitionPortList .
eq removePort(C,APORTL) = removepp(isAPortsList(C,APORTL),APORTL) .

op replacePort : Qid  ArchetypeDefinitionPort ArchetypeDefinitionPortList ->
ArchetypeDefinitionPortList .
eq replacePort(Q,APORT,APORTL) = replacepp(isPortList(Q,APORTL),APORT,APORTL) .


vars   U1 U2 :  UnifiesDeclaration  .
vars US1 US2 US3  : UnifiesDeclarationList  .

op  includesUnifies? : UnifiesDeclaration UnifiesDeclarationList -> Bool .
eq  includesUnifies?(U1 , deferredU ) = false .
eq  includesUnifies?(U1,(U2 . US1)) =  (U1 == U2) or includesUnifies?(U1, US1) .

op  includesUnifies? : UnifiesDeclarationList  UnifiesDeclarationList -> Bool .
eq  includesUnifies?(deferredU , US1) = false .
eq  includesUnifies?((U2 . US2),US1) =
       includesUnifies?(U2,US1) or includesUnifies?(US2, US1) .

op  addun : UnifiesDeclaration UnifiesDeclarationList   -> UnifiesDeclarationList  .

ceq addun(U1, US1) = US1
    if includesUnifies?(U1, US1) .
ceq addun(U1,US1) = (US1 . U1)
    if not(includesUnifies?(U1, US1)) .

op  addun :  UnifiesDeclarationList  UnifiesDeclarationList  -> UnifiesDeclarationList  .
eq  addun(deferredU , US1) = US1 .
eq  addun((U2 . US2), US1) =
    addun(US2,addun(U2, US1)) .


op  removeun : UnifiesDeclaration UnifiesDeclarationList   -> UnifiesDeclarationList  .
eq  removeun(U1,deferredU ) = deferredU  .
ceq removeun(U2, (U1 . US1)) = removeun(U2,US1)
    if (U2 == U1) .
ceq removeun(U2, (U1 . US1)) = (U1 . removeun(U2,US1))
    if (U2 =/= U1) .

op  removeun :  UnifiesDeclarationList  UnifiesDeclarationList  -> UnifiesDeclarationList  .
eq  removeun(deferredU,US1) = US1 .
eq  removeun((U2 . US2), US1) =
    removeun(U2,removeun(US2, US1)) .


var PU PU1  : PortDef .
var AU   : ArchitectureDef .
var CS CDU  : ConstituentsDef .
var COU COU1 : Qid .
var A11 : ArchetypeDefinition .

var ATU : TypingDeclaration .
var  APDU : PortDeclaration .
var AIDU : IncomingDeclaration .
var AODU : OutgoingDeclaration  .
var ABDU : BehaviourDeclaration .
var AADCLU : ArchetypeDefinitionConstituentsList .

op _ :: connections? _in_ : ArchitectureDef UnifiesDeclaration ArchetypeDefinition ->  Bool .

               ceq   AU :: connections? unifies CS :: PU :: COU with CDU :: PU1 :: COU1
                     in (archetype AU is architecture {ATU APDU AIDU AODU behaviour is {AADCLU}
}) = true
               if  (isAConstituents?({CS ; CDU},AADCLU)) and

                   (includesConnections?(isConnectionList(COU1,
```

```
                        ExtractsICPort(
                            ExtractsPortInAPorts(PU1,
                                        projectCPorts(
                                              isAConstituents({CDU},AADCLU))))),
                            ExtractsICPort(
                            ExtractsPortInAPorts(PU1,
                                        projectCPorts(
                                              isAConstituents({CDU},AADCLU) ))))) and

                    (includesConnections?(isConnectionList(COU,
                           ExtractsOCPort(
                            ExtractsPortInAPorts(PU,
                                        projectCPorts(
                                              isAConstituents({CS},AADCLU))))),
                            ExtractsOCPort(
                            ExtractsPortInAPorts(PU,
                                        projectCPorts(
                                              isAConstituents({CS},AADCLU) ))))  .


                         ceq   AU :: connections? unifies CS :: PU :: COU with CDU :: PU1 :: COU1
                      in (archetype AU is architecture {ATU APDU AIDU AODU behaviour is {AADCLU}
}) = false
                if  not((isAConstituents?({CS ; CDU},AADCLU)) and

                    (includesConnections?(isConnectionList(COU1,
                           ExtractsICPort(
                            ExtractsPortInAPorts(PU1,
                                        projectCPorts(
                                              isAConstituents({CDU},AADCLU))))),
                            ExtractsICPort(
                            ExtractsPortInAPorts(PU1,
                                        projectCPorts(
                                              isAConstituents({CDU},AADCLU) ))))) and

                    (includesConnections?(isConnectionList(COU,
                           ExtractsOCPort(
                            ExtractsPortInAPorts(PU,
                                        projectCPorts(
                                              isAConstituents({CS},AADCLU))))),
                            ExtractsOCPort(
                            ExtractsPortInAPorts(PU,
                                        projectCPorts(
                                              isAConstituents({CS},AADCLU) ))))  .


                        --- and
                       --- (
includesConnections?(COU1,ExtractsICPort(ExtractsPortInAPorts(PU1,projectCPorts(isAConstituents({
CDU},AADCLU))))))


endm
```

```
mod ExecutableRefinerTools is

extending SemanticsRefinerArch .

--- Actions for Adding Type Declarations of an Architecture

--- op archetype_is architecture{_____} : ArchitectureDef TypingDeclaration PortDeclaration
IncomingDeclaration
---                                   OutgoingDeclaration BehaviourDeclaration
---                                   ->  ArchetypeDefinition
--- op  types is {_} : TypeDeclarationADLList  -> TypingDeclaration .
--- op  ports is {_} : ArchetypeDefinitionPortList -> PortDeclaration .
--- op  incoming is {_} : ConnectionDeclarationADLList  -> IncomingDeclaration  .
--- op  outgoing is {_} : ConnectionDeclarationADLList  -> OutgoingDeclaration  .
--- op  behaviour is {_} : BehaviourExpressionADL ->  BehaviourDeclaration  .
```

```
sort RefAppli .
subsort ArchetypeDefinition ArchetypeDefinition2 < RefAppli .

vars A1 A2 :  ArchitectureDef .
vars AT1 AT2 :  TypeDeclarationADL .
vars  ATS ATS1 ATS2 :  TypeDeclarationADLList .
var AT : TypingDeclaration .
var  APD : PortDeclaration .
var AID : IncomingDeclaration .
var AOD : OutgoingDeclaration  .
var ABD : BehaviourDeclaration .


op archetype_refines_using{types includes(_)} in_ : ArchitectureDef ArchitectureDef
TypeDeclarationADLList
                                        ArchetypeDefinition  -> RefAppli .

rl[addingTypeDeclarations] :   archetype A2 refines A1 using{types includes(ATS1) }
                in (archetype A1 is architecture {types is {ATS2} APD AID AOD ABD})
        => archetype A2 is architecture { types is {add(ATS2,ATS1)} APD AID AOD ABD} .


op archetype_refines_using{types excludes(_)} in_ : ArchitectureDef ArchitectureDef
TypeDeclarationADLList
                                        ArchetypeDefinition  -> RefAppli .

rl[removingTypeDeclarations] :   archetype A2 refines A1 using{types excludes(ATS1) }
                in (archetype A1 is architecture {types is {ATS2} APD AID AOD ABD})
        => archetype A2 is architecture { types is {remove(ATS1,ATS2)} APD AID AOD ABD} .

op archetype_refines_using{types replaces_by_} in_ : ArchitectureDef ArchitectureDef
TypeDeclarationADL
                                      TypeDeclarationADL  ArchetypeDefinition  -> RefAppli  .

rl[replacingTypeDeclarations] :   archetype A2 refines A1 using{types replaces AT1 by AT2 }
                in (archetype A1 is architecture {types is {ATS1} APD AID AOD ABD})
        => archetype A2 is architecture { types is {replace(AT1,AT2,ATS1)} APD AID AOD ABD} .


op archetype_refines_using{types where{_}becomes{_}} in_ : ArchitectureDef ArchitectureDef
TypeDeclarationADLList
                                      TypeDeclarationADLList  ArchetypeDefinition  -> RefAppli
.

rl[becomesTypeDeclarations] :   archetype A2 refines A1 using{types where {ATS1} becomes {ATS2}}
                in (archetype A1 is architecture {types is {ATS} APD AID AOD ABD})
        => archetype A2 is architecture { types is {becomes(ATS1,ATS2,ATS)} APD AID AOD ABD} .


--- Ports
var APS1 APS2 : ArchetypeDefinitionPortList .
var AP2 : ArchetypeDefinitionPort .
var A3 : ArchitectureDef .

op archetype_refines_using{ports includes(_)} in_ : ArchitectureDef ArchitectureDef
ArchetypeDefinitionPortList
                                        ArchetypeDefinition  -> RefAppli .

rl[addingPortDeclarations] :   archetype A2 refines A1 using{ports includes(APS2) }
                in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
        => archetype A2 is architecture { AT ports is {addpp(APS2,APS1)} AID AOD ABD} .

rl[addingPortDeclarations] :   archetype A2 refines A1 using{ports includes(APS2) }
                in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
        => archetype A2 is architecture { AT ports is {addpp(APS2,APS1)} AID AOD ABD}  .

--- rl[addingPortDeclarations] :   archetype A2 refines A1 using{ports includes(APS2) }
---                 in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
---         => archetype A3 is architecture {AT ports is {APS1} AID AOD ABD}  .


var CP : Contenu .
```

```
op archetype_refines_using{ports excludes(_)} in_ : ArchitectureDef ArchitectureDef  Contenu
                                        ArchetypeDefinition  -> RefAppli .

rl[removingPortDeclarations] :   archetype A2 refines A1 using{ports excludes(CP) }
                in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
        => archetype A2 is architecture { AT ports is {removePort(CP,APS1)} AID AOD ABD} .

var PDD : PortDef .
op archetype_refines_using{ports excludes(_)} in_ : ArchitectureDef ArchitectureDef  PortDef
                                        ArchetypeDefinition  -> RefAppli .

rl[removingPortDeclarations] :   archetype A2 refines A1 using{ports excludes(PDD) }
                in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
        => archetype A2 is architecture { AT ports is {removePort(PDD,APS1)} AID AOD ABD} .

var   ADDP : ArchetypeDefinitionPort .
op archetype_refines_using{ports replaces_by_} in_ : ArchitectureDef ArchitectureDef  PortDef
                                      ArchetypeDefinitionPort ArchetypeDefinition  -> RefAppli .

rl[removingPortDeclarations] :   archetype A2 refines A1 using{ports replaces PDD by ADDP  }
               in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
         => archetype A2 is architecture { AT ports is {replacePort(PDD,ADDP,APS1)} AID AOD ABD}
.

op archetype_refines_using{ports where{_} becomes {_}} in_ : ArchitectureDef ArchitectureDef
Contenu
                                      ArchetypeDefinitionPortList ArchetypeDefinition  -> RefAppli
.

var C5 : Contenu .
var PS5 PS6 : ArchetypeDefinitionPortList .
rl[becomesPortsDeclarations] :   archetype A2 refines A1 using{ports where{C5} becomes {PS5}}
                in (archetype A1 is architecture {AT ports is {PS6} AID AOD ABD})
        =>  archetype A2 is architecture { AT ports is {becomes(isAPortsList(C5,PS6),PS5,PS6)}
AID AOD ABD} .


var PDD5 : PortDef .
var CL5 : ConnectionDeclarationADLList .
var C55 : Contenu .
var CC55 :  ConnectionDeclarationADL .
var Q55 : Qid .

op archetype_refines_using{_ :: incoming includes(_)} in_ : ArchitectureDef ArchitectureDef
PortDef
                                  ConnectionDeclarationADLList  ArchetypeDefinition  -> RefAppli
.

rl[AddingInputConnections] :    archetype A2 refines A1 using{ PDD :: incoming includes(CL5)  }
               in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
                =>  archetype A2 is architecture { AT ports is
{changeInputPortIncluding(PDD,CL5,APS1)} AID AOD ABD } .

op archetype_refines_using{_ :: incoming excludes(_)} in_ : ArchitectureDef ArchitectureDef
PortDef
                                  Contenu  ArchetypeDefinition  -> RefAppli .

rl[RemovingInputConnections] :    archetype A2 refines A1 using{ PDD :: incoming excludes(C55)  }
               in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
                =>  archetype A2 is architecture { AT ports is
{changeInputPortexcluding(PDD,C55,APS1)} AID AOD ABD } .

op archetype_refines_using{_ :: incoming replaces_by_} in_ : ArchitectureDef ArchitectureDef
PortDef
                                  Qid  ConnectionDeclarationADL  ArchetypeDefinition  -> RefAppli
.


rl[ReplacingInputConnections] :    archetype A2 refines A1 using{ PDD :: incoming replaces Q55 by
CC55  }
               in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
                =>  archetype A2 is architecture { AT ports is
{changeInputPortreplacing(Q55,CC55,PDD,APS1)} AID AOD ABD } .
```

```
var CC : Contenu .
var CL23 : ConnectionDeclarationADLList .

op archetype_refines_using{_ :: outgoing where{_}becomes{_}} in_ : ArchitectureDef
ArchitectureDef PortDef
                                    Contenu ConnectionDeclarationADLList  ArchetypeDefinition
-> RefAppli   .

 rl[becomesConnectionDeclarations] :   archetype A2 refines A1 using { PDD :: outgoing  where
{CC} becomes {CL23}}
                                in (archetype A1 is architecture {AT ports is {APS1} AID AOD
ABD})
                                =>  archetype A2 is architecture { AT ports is
{changeOutputPortbecomes(CC,CL23,PDD,APS1)}  AID AOD ABD} .


op archetype_refines_using{_ :: incoming where{_}becomes{_}} in_ : ArchitectureDef
ArchitectureDef PortDef
                                    Contenu ConnectionDeclarationADLList  ArchetypeDefinition
-> RefAppli   .

 rl[becomesConnectionDeclarations] :   archetype A2 refines A1 using { PDD :: incoming  where
{CC} becomes {CL23}}
                                in (archetype A1 is architecture {AT ports is {APS1} AID AOD
ABD})
                                =>  archetype A2 is architecture { AT ports is
{changeInputPortbecomes(CC,CL23,PDD,APS1)}  AID AOD ABD} .



op archetype_refines_using{_ :: outgoing includes(_)} in_ : ArchitectureDef ArchitectureDef
PortDef
                                ConnectionDeclarationADLList  ArchetypeDefinition  -> RefAppli
.

rl[AddingOutputConnections] :    archetype A2 refines A1 using{ PDD :: outgoing includes(CL5)  }
               in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
                =>  archetype A2 is architecture { AT ports is
{changeOutputPortIncluding(PDD,CL5,APS1)} AID AOD ABD } .



op archetype_refines_using{_ :: outgoing excludes(_)} in_ : ArchitectureDef ArchitectureDef
PortDef
                                Contenu  ArchetypeDefinition  -> RefAppli .

rl[RemovingOutputConnections] :    archetype A2 refines A1 using{ PDD :: outgoing excludes(C55)
}
               in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
                =>  archetype A2 is architecture { AT ports is
{changeOutputPortexcluding(PDD,C55,APS1)} AID AOD ABD } .


op archetype_refines_using{_ :: outgoing replaces_by_} in_ : ArchitectureDef ArchitectureDef
PortDef
                                Qid  ConnectionDeclarationADL  ArchetypeDefinition  -> RefAppli
.


rl[ReplacingOutputConnections] :    archetype A2 refines A1 using{ PDD :: outgoing replaces Q55
by CC55  }
               in (archetype A1 is architecture {AT ports is {APS1} AID AOD ABD})
                =>  archetype A2 is architecture { AT ports is
{changeOutputPortreplacing(Q55,CC55,PDD,APS1)} AID AOD ABD } .



var PU11 PU111  : PortDef .
var CS1 CDU1  : ConstituentsDef .
var COU11 COU111 : Qid .
```

```
var ATU1 : TypingDeclaration .
var  APDU1 : PortDeclaration .
var AIDU1 : IncomingDeclaration .
var AODU1 : OutgoingDeclaration  .
var ABDU1 : BehaviourDeclaration .
var AADCLU1 : ArchetypeDefinitionConstituentsList .
var UDL : UnifiesDeclarationList  .


op  archetype_refines_using{_ :: connections?_}in_ : ArchitectureDef ArchitectureDef
ArchitectureDef
                                                        UnifiesDeclaration ArchetypeDefinition
-> RefAppli .

rl[UnifyingConnectionsArchitecture] :    archetype A2 refines A1 using{ A1 :: connections?
unifies CS1 :: PU11 :: COU11 with CDU1 :: PU111 :: COU111}
                                              in (archetype A1 is architecture {ATU1 APDU1 AIDU1
AODU1 behaviour is {AADCLU1} {UDL} } )
                                            => (archetype A2 is architecture {ATU1 APDU1 AIDU1
AODU1 behaviour is {AADCLU1}
                                              {addun(unifies CS1 :: PU11 :: COU11 with CDU1 ::
PU111 :: COU111,UDL)} }) .




--- vars A1 A2 :  ArchitectureDef .
--- vars AT1 AT2 :  TypeDeclarationADL .
--- vars  ATS ATS1 ATS2 :  TypeDeclarationADLList .
--- var AT : TypingDeclaration .
--- var  APD : PortDeclaration .
--- var AID : IncomingDeclaration .
--- var AOD : OutgoingDeclaration  .
--- var ABD : BehaviourDeclaration .

var AC : Contenu .
var AQ : ComponentDef .
var AADCL : ArchetypeDefinitionConstituentsList .

op archetype_refines_using{components implodes_as_} in_ : ArchitectureDef ArchitectureDef Contenu
                                            ComponentDef ArchetypeDefinition  -> RefAppli  .
rl[implodingComponents] : archetype A2 refines A1 using{components implodes AC as AQ}
                               in (archetype A1 is architecture {AT APD AID AOD behaviour is
{AADCL} })
              => archetype A2 is architecture {AT APD AID AOD behaviour is
{AImplodes(AC,AADCL,AQ)} } .

op archetype_refines_using{components explodes_} in_ : ArchitectureDef ArchitectureDef
                                            ComponentDef ArchetypeDefinition  -> RefAppli  .
rl[explodingComponents] : archetype A2 refines A1 using{components explodes AQ}
                            in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
=>
--- 10/06 A1 par A2
archetype A2 is architecture {AT APD AID AOD behaviour is {
 (AComponentsExplodes(archetype A2 is architecture {AT APD AID AOD behaviour is {AADCL} }
                                    ,AQ))} } .


op archetype_refines_using{connectors implodes_as_} in_ : ArchitectureDef ArchitectureDef Contenu
                                            ConnectorDef ArchetypeDefinition  -> RefAppli  .
rl[implodingConnectors] : archetype A2 refines A1 using{connectors implodes AC as AQ}
                               in (archetype A1 is architecture {AT APD AID AOD behaviour is
{AADCL} })
              => archetype A2 is architecture {AT APD AID AOD behaviour is
{AImplodesC(AC,AADCL,AQ)} } .

var CD7 : ConnectorDef .

op archetype_refines_using{connectors explodes_} in_ : ArchitectureDef ArchitectureDef
                                            ConnectorDef ArchetypeDefinition  -> RefAppli  .
```

```
rl[explodingConnectors] : archetype A2 refines A1 using{connectors explodes AQ}
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
=>
archetype A2 is architecture {AT APD AID AOD behaviour is {
 (AConnectorsExplodes(archetype A2 is architecture {AT APD AID AOD behaviour is {AADCL} }
                                    ,AQ))} } .


var C1 : Contenu .

op  archetype_refines_using{behaviour is compose_}with{_}in_ : ArchitectureDef ArchitectureDef
                                                               Contenu
ArchetypeDefinitionConstituentsList
                                                                  ArchetypeDefinition  -> RefAppli  .

rl[ComposeConstituents] :  archetype A2 refines A1 using{behaviour is compose C1 } with {AADCL}
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is
{deferredBeh} })
=>
archetype A2 is architecture {AT APD AID AOD behaviour is {isAConstituents(C1,AADCL)} } .


var BA ADCL1 : ArchetypeDefinitionConstituentsList .



op  archetype_refines_using{behaviour becomes{_}} in_ : ArchitectureDef ArchitectureDef
                                                        ArchetypeDefinitionConstituentsList
                                                          ArchetypeDefinition  -> RefAppli  .


rl[ABehBecomesAbstraction] :  archetype A2 refines A1 using { behaviour  becomes {BA} }
                                in (archetype A1 is architecture{ AT APD AID AOD  behaviour is
{ADCL1} })
            =>   archetype A2 is architecture{ AT APD AID AOD  behaviour is abstraction() {BA} }
.


rl[ABehBecomesAbstraction] :  archetype A2 refines A1 using { behaviour becomes {BA} }
                                in (archetype A1 is architecture{ AT APD AID AOD  behaviour is
{deferredBeh} })
            =>   archetype A2 is architecture{ AT APD AID AOD  behaviour is abstraction() {BA} }
.




var  AR56 : ArchetypeDefinition .
var  CD56 : ComponentDef .

op  archetype_refines_using{_ :: behaviour component  becomes{_}} in_ : ArchitectureDef
ArchitectureDef  ComponentDef
                                                      ArchetypeDefinitionConstituentsList
                                                        ArchetypeDefinition  -> RefAppli  .

rl[ArchBehBecomesAbstraction] :  archetype A2 refines A1 using { CD56 :: behaviour component
becomes {BA} }
                              in  AR56
            =>    transfBehComArch(CD56,AR56,A2,BA) .



var CD57 : ConnectorDef .
var  AR57 : ArchetypeDefinition .

op  archetype_refines_using{_ :: behaviour connector becomes{_}} in_ : ArchitectureDef
ArchitectureDef  ConnectorDef
                                                    ArchetypeDefinitionConstituentsList
                                                      ArchetypeDefinition  -> RefAppli  .

rl[ArchBehConnBecomesAbstraction] :  archetype A2 refines A1 using { CD57 :: behaviour connector
becomes {BA} }
```

```
                          in  AR57
            =>    transfBehConnArch(CD57,AR57,A2,BA) .


var AADCL1 : ArchetypeDefinitionConstituentsList .

op  archetype_refines_using{components includes(_)} in_ : ArchitectureDef ArchitectureDef
                                                          ArchetypeDefinitionConstituentsList
                                                            ArchetypeDefinition  -> RefAppli  .

rl[AComponentsIncludes] :  archetype A2 refines A1 using{components includes(AADCL1)}
                            in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>   archetype A2 is architecture {AT APD AID AOD behaviour is
{addco(comp(AADCL1),AADCL)} } .



op  archetype_refines_using{connectors includes(_)} in_ : ArchitectureDef ArchitectureDef
                                                          ArchetypeDefinitionConstituentsList
                                                            ArchetypeDefinition  -> RefAppli  .

rl[AConnectorsIncludes] :  archetype A2 refines A1 using{connectors includes(AADCL1)}
                            in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>   archetype A2 is architecture {AT APD AID AOD behaviour is
{addco(conn(AADCL1),AADCL)} } .


op  archetype_refines_using{constituents includes(_)} in_ : ArchitectureDef ArchitectureDef
                                                          ArchetypeDefinitionConstituentsList
                                                            ArchetypeDefinition  -> RefAppli  .

rl[AConstituentsIncludes] :  archetype A2 refines A1 using{constituents includes(AADCL1)}
                            in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>   archetype A2 is architecture {AT APD AID AOD behaviour is
{addco(AADCL1,AADCL)} } .



var C111 : Contenu .

op  archetype_refines_using{components excludes(_)} in_ : ArchitectureDef ArchitectureDef
                                                          Contenu
                                                            ArchetypeDefinition  -> RefAppli  .

rl[AComponentsExcludes] :  archetype A2 refines A1 using{components excludes(C111)}
                            in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>   archetype A2 is architecture {AT APD AID AOD behaviour is
{removeco(comp(isAConstituents(C111,AADCL)),AADCL)} } .


op  archetype_refines_using{connectors excludes(_)} in_ : ArchitectureDef ArchitectureDef
                                                          Contenu
                                                            ArchetypeDefinition  -> RefAppli  .

rl[AConnectorsExcludes] :  archetype A2 refines A1 using{connectors excludes(C111)}
                            in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>   archetype A2 is architecture {AT APD AID AOD behaviour is
{removeco(conn(isAConstituents(C111,AADCL)),AADCL)} } .



op  archetype_refines_using{constituents excludes(_)} in_ : ArchitectureDef ArchitectureDef
                                                          Contenu
                                                            ArchetypeDefinition  -> RefAppli  .

rl[AConstituentsExcludes] :  archetype A2 refines A1 using{constituents excludes(C111)}
```

```
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>   archetype A2 is architecture {AT APD AID AOD behaviour is
{removeco(isAConstituents(C111,AADCL),AADCL)} } .


var CD111 : ComponentDef .
var CD1111 : ConnectorDef .
var ADC11 : ArchetypeDefinitionConstituents .

op  archetype_refines_using{components replaces_by_} in_ : ArchitectureDef ArchitectureDef
ComponentDef
                                                          ArchetypeDefinitionConstituents
                                                            ArchetypeDefinition  -> RefAppli  .

crl[AComponentsReplaces] :  archetype A2 refines A1 using{components replaces CD111 by ADC11}
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>   archetype A2 is architecture {AT APD AID AOD behaviour is
{replaceco(comp(isAConstituents({CD111},AADCL)),ADC11,AADCL)} }
                                              if isTypeComponent?(ADC11) .

crl[AComponentsReplaces] :  archetype A2 refines A1 using{components replaces CD111 by ADC11}
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>   archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL} }
                                              if not(isTypeComponent?(ADC11)) .



op  archetype_refines_using{connectors replaces_by_} in_ : ArchitectureDef ArchitectureDef
ConnectorDef
                                                          ArchetypeDefinitionConstituents
                                                            ArchetypeDefinition  -> RefAppli  .

crl[AConnectorsReplaces] :  archetype A2 refines A1 using{connectors replaces CD1111 by ADC11}
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>   archetype A2 is architecture {AT APD AID AOD behaviour is
{replaceco(conn(isAConstituents({CD1111},AADCL)),ADC11,AADCL)} }
                                           if isTypeConnector?(ADC11) .

crl[AConnectorsReplaces] :  archetype A2 refines A1 using{connectors replaces CD1111 by ADC11}
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                 =>  archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL} }
                                           if not(isTypeConnector?(ADC11)) .



------------------------------------------------------------


var CDN : ConstituentsDef  .
var TDLN : TypeDeclarationADLList .

op archetype_refines_using{_ :: types includes(_)} in_ : ArchitectureDef ArchitectureDef
ConstituentsDef
                                  TypeDeclarationADLList  ArchetypeDefinition  -> RefAppli .

rl[AddTypesCompInArch] :  archetype A2 refines A1 using{ CDN :: types includes(TDLN)}
                           in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                              => archetype A2 is architecture {AT APD AID AOD behaviour is {
                                                replaceco(  isAConstituents({CDN},AADCL) ,

changeTypesCons(extractsTypesCons(isAConstituents({CDN},AADCL)),

add(TDLN,extractsTypesCons(isAConstituents({CDN},AADCL))) ,

isAConstituents({CDN},AADCL) ) , AADCL) }} .
```

```
var  APLN : ArchetypeDefinitionPortList .

op archetype_refines_using{_ :: ports includes(_)} in_ : ArchitectureDef ArchitectureDef
ConstituentsDef
                                  ArchetypeDefinitionPortList  ArchetypeDefinition  -> RefAppli .

rl[AddPortsCompInArch] :  archetype A2 refines A1 using{ CDN :: ports includes(APLN)}
                           in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                              => archetype A2 is architecture {AT APD AID AOD behaviour is {
                                                replaceco(  isAConstituents({CDN},AADCL) ,

changePortsCons(extractsPortsCons(isAConstituents({CDN},AADCL)),

addpp(APLN,extractsPortsCons(isAConstituents({CDN},AADCL))) ,

isAConstituents({CDN},AADCL) ) , AADCL) }} .


var ACCNN : ArchetypeDefinitionConstituentsList .
op archetype_refines_using{_ :: constituents includes(_)} in_ : ArchitectureDef ArchitectureDef
ConstituentsDef
                                  ArchetypeDefinitionConstituentsList  ArchetypeDefinition  ->
RefAppli .

 rl[AddConsConsInArch] :  archetype A2 refines A1 using{ CDN :: constituents includes(ACCNN)}
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                             => archetype A2 is architecture {AT APD AID AOD behaviour is {
                                                replaceco(  isAConstituents({CDN},AADCL) ,

changeConsCons(extractsConsCons(isAConstituents({CDN},AADCL)),

addco(ACCNN,extractsConsCons(isAConstituents({CDN},AADCL))) ,

isAConstituents({CDN},AADCL) ) , AADCL) }} .


var CDLN : ConnectionDeclarationADLList .

op archetype_refines_using{_ :: outgoing including(_)} in_ : ArchitectureDef ArchitectureDef
ConstituentsDef
                                  ConnectionDeclarationADLList  ArchetypeDefinition  -> RefAppli
.

rl[AddOCCompInARch] :  archetype A2 refines A1 using{ CDN :: outgoing including(CDLN)}
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                              => archetype A2 is architecture {AT APD AID AOD behaviour is {
                                                replaceco(  isAConstituents({CDN},AADCL) ,

changeOCCons(extractsOCCons(isAConstituents({CDN},AADCL)),

addcc(CDLN,extractsOCCons(isAConstituents({CDN},AADCL))) ,

isAConstituents({CDN},AADCL) ) , AADCL) }} .

op archetype_refines_using{_ :: incoming including(_)} in_ : ArchitectureDef ArchitectureDef
ConstituentsDef
                                  ConnectionDeclarationADLList  ArchetypeDefinition  -> RefAppli
.

rl[AddICCompInARch] :  archetype A2 refines A1 using{ CDN :: incoming including(CDLN)}
                          in (archetype A1 is architecture {AT APD AID AOD behaviour is {AADCL}
})
                              => archetype A2 is architecture {AT APD AID AOD behaviour is {
                                                replaceco(  isAConstituents({CDN},AADCL) ,

changeICCons(extractsICCons(isAConstituents({CDN},AADCL)),

addcc(CDLN,extractsICCons(isAConstituents({CDN},AADCL))) ,
```

```
isAConstituents({CDN},AADCL) ) , AADCL) }} .


endm
```

RESUME

Nous nous sommes intéressés dans cette thèse aux problèmes liés à l'insuffisance de support informatique pour le raffinement d'architectures logicielles. Le peu de langages de description d'architectures (ADL) comme SADL et Rapide qui abordent ces problèmes le font de manière très restreinte. En effet, le support de SADL ne permet qu'un raffinement structurel et exige des preuves manuelles pour les constructions de mise en correspondance entre un modèle abstrait et un modèle architectural plus concret. Dans Rapide, la mise en correspondance d'événements entre des architectures individuelles est bien supportée grâce à un sous-langage exécutable, mais par contre, seul le raffinement comportemental y est possible. Notre contribution consiste en la fourniture d'un environnement logiciel que nous avons nommé *Refiner*, fondé sur la logique de réécriture. Le choix de cette dernière est motivé par le fait qu'elle soit bien adaptée à la description de systèmes concurrents, leur raffinement par le mécanisme de réécriture et également par le fait qu'elle soit bien supportée par des outils logiciels. Nous avons construit *Refiner* autour du langage de raffinement d'architectures *ArchWare ARL* de manière à ce qu'il permette à ses utilisateurs :

- de raffiner progressivement des éléments architecturaux (composants, connecteurs, ports, etc.) depuis des descriptions abstraites vers des descriptions concrètes, et ce à travers des niveaux d'abstraction multiples. A chaque étape de raffinement, peut être appliquée une action qui fournit par construction, sous certaines obligations de preuves, une transformation architecturale correcte. Ceci est possible de par le fait que les *pré-conditions* et *post-conditions* rattachées aux actions de raffinement dans *ArchWare ARL,* soient formalisées et implémentées dans le langage formel qu'est la logique de réécriture. Par conséquent, par construction, une action de raffinement transforme une architecture satisfaisant les pré-conditions en une architecture moins abstraite satisfaisant les post-conditions. Le modèle de transformation architecturale est correct s'il satisfait les pré et les post-conditions ainsi que les obligations de preuve.
- de raffiner plus que la structure et le comportement d'un élément architectural. En effet, l'approche *ArchWare ARL* permet au cours du processus de raffinement d'établir des relations de raffinement portant sur des comportements, des ports, des structures, et des données d'éléments architecturaux.
- de construire au cours d'une étape de raffinement, aussi bien une description architecturale plus concrète traduisant une architecture possible qu'une description simplifiée traduisant une architecture réduite. En effet, le premier type de raffinement est possible de par le principe de sous-spécification de *ArchWare ARL* (à un haut niveau d'abstraction, on spécifie un élément architectural tout en laissant certains aspects non spécifiés). La diminution de cette sous-spécification passe par l'établissement de relations de raffinement des comportements, des ports, des structures, et des données des éléments architecturaux. Le second type de raffinement (réduction ou simplification) est également rendu possible, grâce notamment aux actions de raffinement d'*explosion* et d'*implosion*.
- de supporter, au niveau le plus concret du raffinement architectural, la génération des applications dans des langages de programmation cibles. Ceci se fait par le biais du générateur d'applications *Sigma* fondé sur les concepts de *mapping* et de *patterns de synthèse*.

L'environnement *Refiner* proposé permet aux architectes de raffiner correctement des éléments architecturaux, puis de recomposer les éléments raffinés pour la construction d'architectures concrètes. Le choix de la logique de réécriture comme fondement à la fois de la sémantique formelle et de la sémantique opérationnelle de l'environnement permet de naturellement décrire des architectures logicielles, leur raffinement par le mécanisme de réécriture ainsi que leur "exécution" grâce aux outils proposés pour cette logique.