Rapport interne

Année 2007

Gestion de l'évolution dans une approche d'ingénierie logicielle centrée architecture

Auteur(s) : Cîmpan S., Verjus H., Alloui I.

Rapport LISTIC n° 07/01

janvier 2007

LISTIC – **Polytech'Savoie**
Domaine Universitaire - B.P. 806 - 74016 Annecy cedex (France)
Tél. : +33 (0)4.50.09.65.80 - Fax : +33 (0)4.50.09.65.59
E-mail : dir-listic@univ-savoie.fr - http://www.listic.univ-savoie.fr

Introduction

Les activités économiques utilisent de plus en plus de logiciels pour atteindre des objectifs définis. Ceci a conduit à des systèmes logiciels (ou à forte composante logicielle) de plus en plus complexes et soumis aux exigences croissantes des utilisateurs en termes de qualité et de fonctionnalités. Ces systèmes doivent de surcroît pouvoir évoluer au cours de leur existence afin de répondre de manière cohérente aux changements dans leur environnement, que ces derniers soient liés à l'évolution du marché, aux exigences des utilisateurs ou à l'évolution des technologies. Les systèmes logiciels doivent en conséquence avoir une capacité d'adaptation importante (Lehman 1996).

Dans le domaine de la recherche, plusieurs approches sont proposées avec pour objectif l'amélioration du développement de tels systèmes : les approches orientées aspects, celles fondées sur les patrons de conception et d'implémentation, les approches qui fournissent des langages/outils formels, etc. Dans la même optique, les approches dites centrées architecture permettent de développer et de faire évoluer des systèmes complexes en considérant leur architecture comme point central de leur cycle de vie. Un processus de développement centré architecture inclut des activités telles que la définition de styles architecturaux (e.g. client-serveur, lecteur-rédacteur), la définition, l'analyse et le raffinement de l'architecture en architectures de plus en plus concrètes. La dernière étape du raffinement peut être la génération vers un langage/environnement de programmation cible. Des langages de description architecturale ont été proposés (Medvidovic et Taylor, 2000), de même que des environnements centrés architecture (Schmerl et Garlan, 2004, ArchStudio) et des outils d'analyse permettent d'étudier certaines propriétés de ces systèmes (Inveradi *et al.*, 2000, Monroe, 2001).

Le développement centré architecture s'inscrit dans les approches d'ingénierie dirigée par les modèles (Favre *et al.*, 2006) où tout processus se base sur des modèles lors de la conception et la réalisation de systèmes réels. Les modèles sont utilisés pour comprendre des aspects spécifiques du système, prédire les qualités du système, raisonner sur l'impact des changements, et communiquer avec les différents acteurs concernés par le système (développeurs, clients, utilisateurs, etc.). L'un des objectifs des approches dirigées par les modèles est de fournir suffisamment de détails pour permettre la génération de l'implémentation du système. Dans une approche centrée architecture, les modèles sont des descriptions d'architecture logicielle qui peuvent êtres exécutables ou non selon le support logiciel fourni.

Nous présentons dans cet article une approche centrée architecture permettant de prendre en compte l'évolution d'un système à travers son architecture à tous les niveaux (abstrait/conceptuel et concret) tout en permettant de raisonner (au sens formel par de la vérification de propriétés) sur les changements qui interviennent. Ainsi l'approche ArchWare présentée et nos récents travaux dans ce domaine (Verjus *et al.*, 2006) permettent de couvrir différents cas d'évolution qui se présentent aux architectes d'applications logicielles, en offrant un langage de description

architecturale dynamique, formel et de haut niveau ainsi qu'un ensemble de mécanismes et d'outils permettant de prendre en compte et de gérer l'évolution.

Nous illustrerons cette approche en traitant des exemples de cas d'évolution et nous verrons comment ces cas sont pris en compte en utilisant l'approche ArchWare (ArchWare, 2001). Dans ce contexte, nous considérons des évolutions qui ont un impact sur l'architecture logicielle du système ; nous ne nous intéressons pas aux activités de maintenance qui n'auraient pas d'impact sur l'architecture.

Prise en compte de l'évolution dans les systèmes logiciels

Nous distinguons quatre types d'évolution (Cîmpan et Verjus, 2005), selon que l'évolution est *statique* (hors exécution du système) ou *dynamique* (pendant l'exécution du système), *prévue* (dès la conception du système) ou *non prévue* (qui peut intervenir à tout moment du cycle de vie du système). Une évolution statique qu'elle soit prévue ou non représente le cas le plus couramment traité par les approches existantes. Une évolution dynamique quant à elle, est plus difficilement prise en charge, en particulier si elle n'a pas été prévue avant le lancement de l'exécution du système.

Les travaux qui ont été proposés ces dernières années pour supporter l'évolution des systèmes logiciels se sont focalisés sur le code indépendamment de l’architecture conceptuelle (abstraite) du système (Demeyer *et al.*, 2002, Mens *et al.*, 2005), le plus souvent de manière statique. Certains langages dynamiques permettent néanmoins de prendre en compte l'évolution du système en cours d'exécution (e.g. Smalltalk et Ruby).

Cependant, les modèles concrets (code) d'une architecture ne proposent souvent pas les bons artefacts ni les concepts architecturaux de haut niveau d’abstraction permettant de comprendre et de raisonner de façon globale sur l'architecture, en évitant de prendre en compte les détails liés à l'implémentation. En outre, peu de langages d'implémentation permettent une analyse approfondie des systèmes du point de vue des propriétés structurelles et comportementales attendues. Les langages formels qui permettent de décrire des spécifications exécutables tels que B ou VDM (Abrial, 1996, Plat et Gorm Larser, 1992) ne sont pas dotés de moyens permettant de gérer dynamiquement l'évolution, manipulent des abstractions peu intuitives et requièrent un niveau d'expertise élevé qui les rend peu utilisés en pratique.

Dans les approches centrées architecture l'architecture est étudiée souvent à deux niveaux d’abstraction (Tran et Holt, 1999, Riva 2004, Kazman et Carriere, 1999, Woods *et al.*, 1999, Medvidovic et Jakobac, 2006) entre lesquels les liens ne sont souvent pas maintenus, voir absents. On distingue ainsi le niveau abstrait, c´est-à-dire l´architecture telle qu´elle apparaît *intuitivement* à l´humain, et le niveau concret, c´est-à-dire celle qui est directement liée à l´implémentation. Dans ce contexte la cohérence ainsi que la synchronisation entre l'architecture abstraite et le code (Medvidovic *et al.*, 2003, Perry et Wolf, 1992, Riva 2004, Cîmpan et Verjus 2005) constitue un problème lors de l’évolution des systèmes. Dans le cas où les

changements sont effectués au niveau abstrait, aucun moyen ne garantit que le niveau concret intègrera fidèlement (au sens de la cohérence et de la complétude formelles) ces modifications. Dans le cas où les changements sont effectués au niveau du code, les spécifications sont rarement mises à jour. Ainsi, l'évolution qui serait prise en compte au niveau de la conception (en supposant qu'elle puisse être exprimée) devrait être accompagnée par une analyse du système permettant de vérifier si les propriétés sont préservées au niveau de l'exécution (en supposant que des mécanismes permettent des évolutions en cours d'exécution); il est alors nécessaire de disposer de moyens de détection et de réparation d'éventuelles incohérences. Dans le cas idéal, une approche ayant pour objectif de prendre en compte l'évolution durant l'ensemble du cycle de vie devrait combiner l'ensemble des possibilités d'analyse, de détection et de réparation.

L'évolution d'un système logiciel requiert pourtant un investissement important puisqu'on lui consacre l'essentiel des budgets ; elle devrait donc être considérée au cours de toutes les phases du cycle de vie des systèmes centrés logiciel de manière à réduire les coûts. Certaines évolutions peuvent déjà être prises en charge en temps de conception et ne nécessitent pas d'être retardées lors ou après la phase de codage.
Nous verrons que l'approche que nous utilisons se démarque des approches classiques : (i) elle peut être considérée comme une approche unifiant (si on le souhaite) les spécifications avec l'exécution, (ii) elle est accompagnée de moyens permettant d'exprimer des propriétés et de les analyser voire de les garantir à chaque étape du processus de développement.

Dans le reste de cet article, nous montrerons comment une approche centrée architecture (ArchWare) permet de répondre à ces différents problèmes. Le lecteur pourra trouver dans (Verjus *et al.*, 2006) d'autres exemples d'architectures évolutives (fondés sur des systèmes client-serveur) traités avec l'approche ArchWare.

La section 2 de cet article fait le point sur la prise en compte de l'évolution dans le cadre des approches centrées architecture alors que les fondements et la structuration des langages ArchWare de description et d'analyse d'architecture sont présentés en section 3. La section 4 introduit les scénarios que nous utilisons dans la suite de l'article pour illustrer la prise en compte des différents types d'évolution : *prévue dynamique* (section 5), *non prévue* (section 6) couvrant uniquement le cas *dynamique*, le cas statique étant traité par ailleurs. La section 7 présente un positionnement de nos travaux. Nos perspectives dans ce contexte sont présentées dans la section 8.

2. Le développement centré architecture et l'évolution

Plusieurs travaux dans la littérature traitent les différentes facettes du développent centré architecture. Ainsi, ont vu le jour plusieurs Langages de Description Architecturale (LDA) avec des pouvoirs d'expression très différents (Medvidovic et Taylor, 2000) : certains sont uniquement structurels, d'autres expriment le comportement, permettent l'expression de propriétés (Tibermacine *et al.*, 2005) ou

non, reposent sur une approche formelle ou non (certains étant basés sur UML par exemple), prennent en compte les styles architecturaux ou non, sont des langages extensibles ou non. D'une manière générale ces langages sont fondés sur les concepts de composant, connecteur et de configuration.

Notons que certains langages de description d'architecture sont accompagnés par des machines virtuelles, et donnent la possibilité d'obtenir des descriptions d'architectures exécutables (ceci est notamment le cas en ArchWare). Dans de tels cas, la représentation du système est exécutable à chaque niveau d'abstraction. Ainsi, l'implémentation du système peut être faite dans le langage de description de son architecture. Ceci implique certaines contraintes mais donne aussi la possibilité de faire évoluer le système plus facilement (Cîmpan et Verjus, 2005).

Dans ce cadre, l'idée de l'évolution est assez large et couvre différentes notions ou critères (Mens *et al.*, 2003, Mens *et al.*, 2005) qui sont plus ou moins bien pris en compte dans les travaux portant sur les architectures logicielles et notamment sur la gestion de la cohérence entre les spécifications et la représentation exécutable (Cîmpan et Verjus, 2005). On trouve également des travaux qui portent sur des transformations de modèles avec pour but de séparer la définition (et l'expression) des préoccupations de l'architecture fonctionnelle de l'application. Les préoccupations sont par la suite « intégrées » à l'architecture par un processus de tissage (Barais et Duchien, 2004, Manset *et al.*, 2006). Ces travaux mettent en évidence la nécessité de disposer de moyens (1) pour exprimer des préoccupations et (2) pour effectuer des opérations de transformation, essentiellement dès la phase de spécification, voire même jusqu'au déploiement (Manset *et al.*, 2006). D'autres travaux traitent du *refactoring* d'architectures (évolution, réorganisation de l'architecture) – en partant du code et en essayant d'inférer une architecture de plus haut niveau (Ding et Medvidovic, 2001) qui sera étudiée dans le but d'être réorganisée. Des travaux récents (Mens *et al.*, 2006) reposent sur l´utilisation de vues afin de faciliter la compréhension et la manipulation du logiciel (par l´identification de groupes d´entités logicielles et leurs relations) et en prenant en compte certaines contraintes qui doivent être préservées. Cependant, ces travaux se focalisent sur l´architecture concrète et non une architecture de plus haut niveau d'abstraction.

Dans la plupart des approches centrées architecture (Ding et Medvidovic, 2001, Barais et Duchien, 2004), les modifications d'architectures se font de façon déconnectée de l'exécution du système ; les modifications sur le système en exécution sont propagées ultérieurement, après modification du code (opérée de façon manuelle ou automatique, après re-génération du code – voir section 6). Dans deux des cas d'évolution que nous abordons dans cet article, nous souhaitons effectuer des modifications de façon *dynamique*, alors que le système est en cours d'exécution. Ces modifications n'entraînent pas l'arrêt du système. Avant de passer à la présentation des différents cas d'évolution, la section suivante introduit l'approche et les langages utilisés pour la description et l'analyse des logiciels, notamment dans le cadre du projet ArchWare. La présentation détaillée de l'approche couvrant la description ainsi que l'analyse d'une architecture logicielle quelconque ne fait pas

l'objet de cet article, et peut être trouvée dans d'autres publications (Cîmpan *et al.*, 2005, Oquendo, 2004).

3. Fondements et structuration des langages ArchWare de description d'architectures logicielles évolutives

Le but du projet ArchWare (ArchWare, 2001) est de fournir un environnement de développement centré architecture pour la construction de systèmes évolutifs (Oquendo *et al.* 2004). L'environnement fournit des langages et des outils pour la description des architectures et de leurs propriétés ainsi qu'un support pour l'exécution et l'analyse de telles architectures à travers une machine virtuelle. La cible privilégiée sont les systèmes évolutifs.
Cette section présente une partie de la famille de langages ArchWare, notamment ses bases formelles et sa structuration en couches, avec un noyau et un mécanisme d'extension pour la construction de langages spécifiques.

Le langage noyau. Le langage noyau Archware π-ADL (Oquendo *et al.*, 2002, Oquendo 2004) est une extension bien-formée du π-calcul d'ordre supérieur typé (Milner, 1999) ayant pour but de définir un calcul d'éléments architecturaux communicants et mobiles. Les éléments architecturaux sont définis en tant que *comportements* qui expriment à l'aide d'actions ordonnancées aussi bien des interactions d'éléments architecturaux (par échange de messages à travers des connexions : **`via`** `<connection>` **`send`** `<data>`, **`via`** `<connection>` **`receive`** `<data>`) que des comportements internes d'éléments architecturaux. (**`unobservable`**) Les différentes actions contenues dans les comportements sont ordonnancées, en utilisant des opérateurs du π-calcul pour représenter des séquences d'actions, le choix, la composition la réplication et le « matching ». La composition d'éléments architecturaux donne lieu à des éléments architecturaux composites. Archware π-ADL fournit le concept d'*abstraction* comme mécanisme pour la réutilisation des comportements paramétrés. L'application d'une abstraction revient à fournir un comportement.

Le mécanisme d'extension. Le mécanisme d'extension est le style architectural, qui représente une famille d'architectures ayant des caractéristiques communes et obéissant un certain nombre de contraintes. Des types d'éléments architecturaux peuvent être introduits par le style, et formeront le vocabulaire du style (Leymonerie, 2004). Il est ainsi possible de définir de styles architecturaux selon le principe suivant : si en utilisant la couche *n* de la famille des langages, un style architectural est défini alors son vocabulaire constitue un langage de description de couche *n+1*. Par construction, une architecture définie en utilisant le langage de couche *n* a son correspondant dans le langage de la couche *n-1*.

La couche composant-connecteur. En utilisant le mécanisme d'extension, un langage de niveau 1 a été construit à partir du langage noyau (niveau 0). Ce langage, nommé ArchWare C&C-ADL est associé donc à un style architectural, dans lequel

les éléments architecturaux sont soit des composants, soit des connecteurs (Cîmpan *et al.*, 2005, Leymonerie, 2004).

Les composants ainsi que les connecteurs peuvent être soit atomiques soit composés d'autres composants et connecteurs. L'interface d'éléments architecturaux, représentée par des connexions, est structurée en ports. Chaque port a un protocole qui est une projection du comportement de l'élément auquel il appartient (sur les connections du port). Les éléments architecturaux atomiques ainsi que les composites peuvent avoir des attributs utilisés pour les paramétrer.

Le comportement d'un élément composite résulte de la composition parallèle des comportements de composants et connecteurs qui le composent. L'élément composite a ses propres ports, auxquels les ports des éléments qui le composent peuvent être attachés.

La construction du langage sur une algèbre de processus permet la description du comportement des systèmes. Pour représenter les changements dynamiques d'une architecture, le langage C&C introduit des actions telles que la création dynamique d'éléments et la reconfiguration. Ces actions peuvent ensuite être utilisées dans la description du comportement des éléments architecturaux. De plus, toute entité architecturale est potentiellement dynamique, sa définition servant à la création dynamique de plusieurs occurrences. Ainsi, la définition est celle d'une *méta-entité*, une matrice contenant à la fois la définition ainsi que des informations permettant la création, la suppression et la gestion de plusieurs instances. L'évolution d'un composite est prise en compte par un élément architectural dédié, le *chorégraphe*. Ce dernier est en charge de changer la topologie en cas de besoin : changer les attachements entre éléments architecturaux, créer dynamiquement de nouvelles instances, exclure des éléments de l'architecture, en inclure d'autres, etc.

Le langage d'analyse d'architecture. Nous pouvons exprimer des propriétés architecturales en vue de leur analyse dans un langage dédié AAL (Architecture Analysis Language) (Alloui *et al.*, 2003, Alloui, 2005) qui fait partie de la famille Archware. AAL est un langage, formel, fondé à la fois sur la logique des prédicats de premier ordre et sur le μ-calcul (Bradfield et Stirling, 2001). La logique des prédicats représente un bon candidat pour l'expression des aspects structurels tandis que le μ-calcul fournit un grand pouvoir d'expression des aspects dynamiques et temporels d'un système évolutif. Une propriété peut-être exprimée dans AAL à l'aide d'une *formule de prédicat* (expression de structure), une *formule d'action* (expression de comportement), une *formule d'état* (patron d'état) ou une *formule régulière* (patron de comportement). AAL fournit le support pour l'analyse des propriétés à la fois par preuve de théorème (Azaiez et Oquendo, 2005) fondée sur la logique des prédicats du premier ordre et par la vérification de modèles (Bergamini *et al.*, 2004).

4. Présentation du cas d'étude et des scénarios d'évolution

Pour illustrer nos propos et les différents cas d'évolution que nous avons présentés en introduction, nous utiliserons des scénarios en relation avec une chaîne logistique regroupant une entreprise et ses fournisseurs (dont la gestion et les interactions sont prises en charge par une solution EAI incluant entre autre, un ERP) ainsi que des clients. L'architecture de cette chaîne logistique étendue est amenée à évoluer. D'autres architectures ont été présentées et validées dans (Leymonerie *et al.*, 2004, Ratcliffe *et al.*, 2005, Revillard *et al.*, 2005, Pourraz *et al.*, 2006, Verjus *et al.*, 2006).

Scénario original : un client demande un devis puis il passe une commande. Pour simplifier les scénarios et alléger le code, la décision de passer la commande à partir du devis n'est pas traitée. Le système de commande envoie au système de gestion de stock une demande de mise à jour du stock, fonction du produit et de la quantité commandée; il transmet ensuite les informations au système de facturation pour édition de la facture. Parallèlement, le système de gestion du stock, en fonction de l'état du stock, peut effectuer une demande de réapprovisionnement. Le système de réapprovisionnement peut alors contacter un fournisseur pour satisfaire cette dernière demande si le stock est insuffisant.

Scénarios d'évolution :

- Réorganisation du système d'information par remplacement du système de facturation : *évolution dynamique prévue*
- L'architecture initialement mono-client devient multi-clients : *évolution dynamique prévue*.
- Changement du fonctionnement du système de réapprovisionnement pour la prise en compte des fournisseurs multiples : *évolution dynamique non-prévue*.

Ainsi, parmi les quatre types d'évolution présentés en introduction, nous illustrerons dans la suite de l'article les évolution dynamiques. Les évolutions statiques impliquent l'arrêt du système afin de faire les modifications (qu'elles soient prévues ou non). Parmi les évolutions effectuées de manière *statique* celui-ci étant de notre point de vue trivial car il suffit de décrire au préalable toutes les possibilités d'évolution que nous envisageons au niveau de l'architecture du système. Concrètement dans notre approche, ces possibilités pourraient être exprimées au préalable à l'aide de comportements conditionnels. Le basculement d'une configuration architecturale à une autre au moment où l'évolution a lieu se fait de manière statique.

Évolution prévue, dynamique. L'architecture du système évolue elle-même sans intervention externe, l'évolution ayant été anticipée lors de la conception et faisant partie de la description de l'architecture. Dans ce cas, l'architecture se modifie dynamiquement pour, par exemple, ajouter de nouveaux clients ou modifier l'ERP en lui changeant le système de facturation (cf. section 5).

Évolution non prévue, dynamique. Dans ce cas, l'architecture du système ne peut évoluer d'elle-même. Cependant l'évolution se fera alors que le système est en cours d'exécution, c'est-à-dire, sans qu'il y ait interruption ni discontinuité dans l'exécution (cf. section 6.2). Nous montrerons comment l'architecte peut améliorer la gestion de réapprovisionnement du stock, par ajout de fournisseurs et par amélioration du processus de réapprovisionnement. Ce cas va d'une part compléter l'architecture initiale et d'autre part, modifier une partie de son fonctionnement initial interne. La particularité de ce cas réside dans le fait qu'il y a identité entre l'architecture abstraite et l'architecture concrète, grâce à l'existence d'une machine virtuelle permettant d'interpréter le code de l'architecture. Ainsi, une modification apportée à la description architecturale est immédiatement prise en compte au niveau de l'exécution, par la machine virtuelle et ses mécanismes de support à l'exécution (gestion de la persistance, de l'état du système, etc.).

Avec ces trois cas, nous montrerons qu'il est possible de modifier une architecture, non seulement au niveau de sa structure mais également en modifiant le comportement intrinsèque de certains de ses éléments. Nous montrerons également qu'il est possible de faire évoluer l'architecture initiale tout en en maintenant la cohérence, à travers la vérification et/ou la préservation des propriétés architecturales attendues du système. Nous focaliserons essentiellement sur comment dans ArchWare, une évolution architecturale peut se faire « *on-line* ».

5. Architecture d'une chaîne logistique étendue avec évolution *prévue* et gérée *dynamiquement*

Dans le cas que nous considérons ici, l'architecte a prévu l'évolution du système lors de sa conception. Deux scénarios d'évolution *prévue dynamique* nous permettront d'illustrer les possibilités offertes par le langage ArchWare C&C-ADL, notamment la gestion de l'évolution dynamique prévue (Cîmpan *et al.*, 2005) qui améliore certaines propositions existantes, telles que Dynamic Wright (Allen *et al.*, 1998) ou π-Space (Chaudet *et al.*, 2000).

En utilisant le langage ArchWare C&C-ADL, nous représentons le système à l'aide de composants et connecteurs. Ces éléments architecturaux sont regroupés à plusieurs niveaux sous forme de composants composites.
Une vue restreinte de l'architecture initiale, correspondant à l'ensemble de la chaîne logistique (Supply Chain), est présentée dans Figure 1. Représentée sous forme d'un composant composite, l'architecture de la chaîne logistique est formée initialement par un composant ERP, un fournisseur, un client ainsi qu'un connecteur qui lie ces deux composants. Le connecteur entre le fournisseur et l'ERP n'est pas représenté, pour des raisons de simplicité.

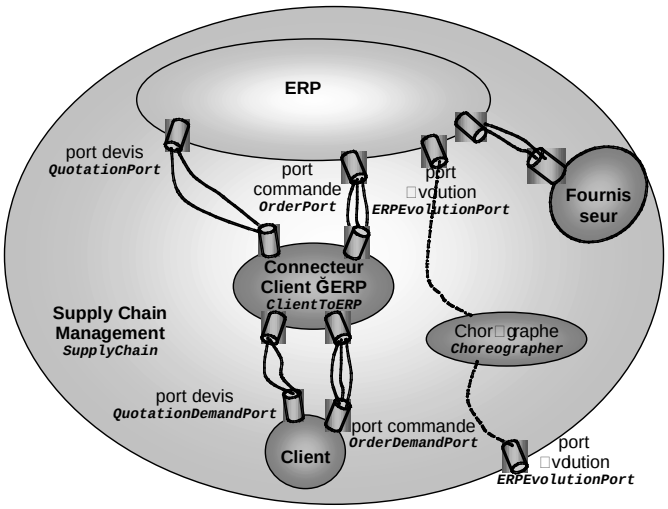


Figure 1 Le composite Supply Chain : vue restreinte avant l'évolution

Les clients communiquent avec le système ERP essentiellement pour demander des devis et pour passer des commandes. Des ports sont dédiés à cet effet à la fois au niveau du client, de l'ERP et du connecteur qui les relie. Un élément particulier est présent au niveau du composite, le chorégraphe, qui gère l'évolution dans ce composite. Comme mentionné précédemment, cet élément est implicitement connecté à tous les éléments du composite.

Deux scénarios d'évolution prévue sont pris en compte ici :

- l'intégration d'un nouveau système de facturation pour remplacer le système interne de l'ERP (ceci implique une évolution à la fois du composite chaîne logistique SupplyChain, et du composite ERP) ;
- l'arrivée dynamique de nouveaux clients qui sont connectés de façon transparente pour l'ERP et pour les clients existants.

Nous verrons par la suite la description de différents éléments architecturaux dans l'architecture initiale (section 5.1), les deux scénarios d'évolution (cf. respectivement section 5.2 et 5.3). Nous discuterons également des différentes propriétés que sont vérifiées afin d'assurer la cohérence du système après évolution (cf. section 5.4).

5.1 Architecture initiale avant évolution

Le composant atomique Client. Les composants atomiques comprennent trois parties : la partie dédiée à la déclaration des méta-ports et méta-attributs, la partie configuration indiquant les ports (instances de méta-ports) prévus dans la configuration initiale et enfin la partie calculatoire (computation) représentant le comportement du composant.

Le composant `Client` possède des ports pour faire des demandes de devis (`quotationP`) et pour passer des commandes (`orderP`) (cf. Description architecturale 1). Comme pour tout élément architectural décrit en utilisant ArchWare C&C-ADL, les déclarations correspondent à des déclarations de méta-éléments, ce qui signifie que plusieurs instances de chaque méta-élément peuvent co-exister. Ainsi `quotationP` et `orderP` sont des méta-ports. Une instance de chaque est créée dans la partie configuration. Des instances supplémentaires peuvent être créé dynamiquement (exemple dans la section 5.3). L'introduction des méta-éléments permet d'avoir un niveau supplémentaire de gestion entre les instances entre les types, dont nous tirons profit pour pouvoir gérer l'évolution dynamique.

```
Client is component with{
   ports {
        quotationP is QuotationDemandPort
        orderP is OrderDemandPort  }
   attributes { currentProduct:String ; currentQuantity:Integer}
   configuration { new quotationP; new orderP }
   computation {
     via attribute~currentProduct receive product:String;
     via attribute~currentQuantity receive quantity:Integer;
     choose {
        via quotationP~quotationReq send product, quantity;
        via quotationP~quotationRep receive quot:String}
        or { via orderP~orderReq send product, quantity;
             via orderP~orderRep receive ack:String;
             if (ack=="OK")  then {
                  via orderP~invoice receive invoice:String;
             }
        then recurse
   }
}
```

Description architecturale 1 Le composant atomique Client

Le comportement du client est restreint, c'est-à-dire qu'un seul devis ou une seule commande sont demandés ou passées à la fois. Les informations concernant le devis ou la commande en cours de traitement sont stockées à l'aide de deux attributs : `currentProduct` qui décrit le produit, et `currentQuantity` qui indique la quantité demandée.

Deux méta-ports sont déclarés, la configuration initiale contenant une instance de chaque. Le comportement récursif du client consiste en la récupération du produit et de la quantité à partir des attributs, suivie soit (1) d'une demande de devis par envoi sur la connexion `quotationRep` du même port, soit (2) d'une commande consistant en un envoi sur la connexion `orderReq` du port `orderP`, suivi par la réception d'un avis sur la connexion `orderRep` du même port. En cas d'avis favorable, une facture est également reçue. Notons que l'accès aux valeurs d'attributs se fait à travers un port spécifique, **`attribute`**, dans lequel il y a pour chaque attribut une connexion qui lui est dédiée. L'accès en lecture se fait avec un **`receive`**, celui qui est en

écriture avec un **`send`** sur cette connexion. Le port **`attribute`** est ouvert et les valeurs des attributs peuvent être changées à la demande d'un autre composant.

Dans ce qui suit nous allons présenter plus en détail le composite ERP, le composite SupplyChain et le connecteur entre le client et l'ERP.

Le composite ERP. La structuration du composant composite ERP est illustrée en Figure 2. Quatre composants se trouvent au sein de ce composite : un gestionnaire de stock, un gestionnaire de devis, un système de facturation et un système de commandes. . Un composant est dédié à la gestion de devis. Il est relié directement à un des ports du composite. Un deuxième composant également relié à un des ports du composite est le système de commande. Ce dernier est lié à un gestionnaire de stock et à un système de facturation qui ne sont pas liés directement à des ports du composite. Les éléments qui relient les différents composants sont des connecteurs basiques, définis implicitement. Ils ne sont donc représentés ni dans le schéma, ni dans le code.

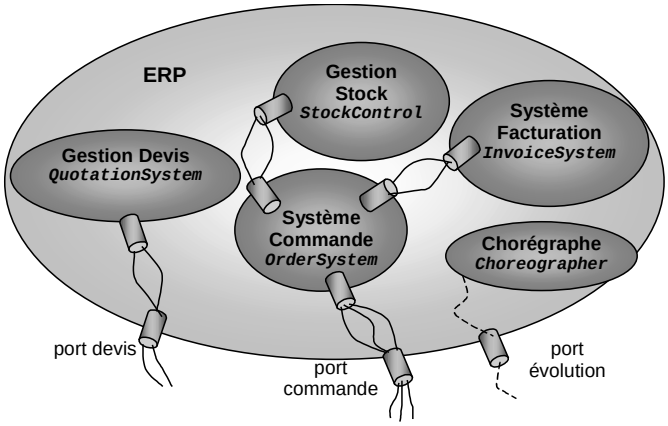


Figure 2 Le composant composite ERP avant évolution

Nous allons maintenant nous attarder sur les définitions ArchWare C&C-ADL du système de commande et du composite ERP. Le système de commande, représenté par le composant atomique `OrderSystem` possède trois ports (cf. Description architecturale 2). Le port `orderP` est connecté au composite, et permet de récupérer les commandes venues des clients, notamment l'identificateur du produit souhaité et sa quantité. Le port `stockP` permet d'effectuer l'opération de consultation de l'état du stock. Le gestionnaire de stock renvoie un message pour confirmer l'opération au niveau du stock ou pour indiquer l'indisponibilité du produit. Ce message est retransmis au client. Si le produit est disponible, il demande au système de facturation (auquel il se connecte en utilisant le port `invoiceP`) l'élaboration d'une facture qui, une fois reçue, est renvoyée au client. Ce comportement est répété de manière récursive.

```
OrderSystem is component with
   ports {
        stockP is StockDemandPort;
        invoiceP is InvoiceDemandPort;
        orderP is OrderPort;  }
   configuration { new stockP ; new stockP; new orderP }
   computation {
     via orderP~orderReq receive product:String, quantity:Integer;
     via stockP~invoiceReq send product, quantity ;
     via stockP~invoiceRep receive  ack: String;
     via orderP~orderRep send  ack;
     if (ack==“OK”) then {
       via invoiceP~invoiceReq send product, quantity ;
       via invoiceP~invoiceRep receive  invoice: String;
       via orderP~invoice send  invoice
     }
     recurse
   }
}
```

Description architecturale 2 Composant atomique Système de commande

5.2 Evolution dynamique prévue du système de facturation

La définition ArchWare C&C-ADL du composite `ERP` est présenté dans Description architecturale 3. La partie `constituents` contient la déclaration des différents méta-composants, une instance de chaque étant crée dans la partie `configuration`. Cette dernière contient également les différents attachements entre les composants.

Un des ports du composite est dédié à la réception de messages d'évolution. Comme nous l'avons mentionné, l'évolution d'un composite est gérée par son chorégraphe. Ainsi le composant composite ERP est prêt à faire évoluer son système de facturation. Un nouveau système de facturation est reçu à travers le port `erpEvolveP` (les langages ArchWare étant basés sur le π-calcul, ils supportent la mobilité des éléments architecturaux qui peuvent être envoyés et reçus sur des connexions). Lors de cette réception le chorégraphe détache les composants `orderComponent` et `invoiceComponent`. Le nouvau composant de facturation est ensuite inséré dans le méta-composant `invoiceComposant`. Nous rappelons que tout composant est dynamique puisque sa déclaration correspond à celle d'un méta-composant. La dernière instance d'un méta-composant peut être accédée en utilisant le nom du méta-composant suivi de **`#last`**. Ainsi l'instance courante du système de facturation est accédée avec `invoiceSystem`**`#last`**. En utilisant cette référence, le chorégraphe attache le nouveau système de facturation au système de commande.

```
ERP is component with{
   ports {
        erpOrderP is OrderPort;
        erpQuotationP is QuotationPort;
        erpEvolveP is ERPEvolutionPort;}
   constituents {
    orderComponent is OrderSystem;
    invoiceComponent is InvoiceSystem;
    stockComponent is StockControl;
    quotationComponent is QuotationSystem; }
   configuration {
    new orderComponent; new invoiceComponent;
    new stockComponent; new quotationComponent;
    attach orderComponent~orderP to erpOrderP;
    attach quotationComponent~quotationP to erpQuotationP;
    attach orderComponent~stockP to stockComponent~stockP;
    attach orderComponent~invoiceP to invoiceComponent~invoiceP;}
   choreographer {
    via erpEvolveP~newInvoice
        receive newInvoiceComponent:InvoiceSystem;
    detach orderComponent from invoiceComponent ;
    insert component newInvoiceComponent in invoiceComponent;
    attach orderComponent~invoiceP
        to invoiceComponent#last~invoiceP;
    via erpEvolveP~ackP send "ok";
    recurse;
   }
}
```

Description architecturale 3 Composant composite ERP

Nous venons d'illustrer le premier cas d'évolution dynamique prévue. Le système de facturation est dynamiquement remplacé par un nouveau qui est intégré dans l'architecture de l'ERP.

5.3 Evolution dynamique prévue des clients

Revenons à présent au composite global, `SupplyChain`, qui représente la chaîne logistique étendue. La Figure 3 présente une vue complète du système. Attardons-nous sur notre deuxième scénario d'évolution dynamique prévue qui concerne la gestion de l'arrivée de nouveaux clients. Cette gestion est faite par le chorégraphe du composite `SupplyChain et` également par le connecteur qui relie le client à l'ERP.

Le connecteur `ClientToERP` est un connecteur dynamique. Il possède des ports pour la communication avec les clients, et avec l'ERP (cf. Description architecturale 4) ainsi qu'un port dédié à l'évolution. Dans la configuration initiale il y a une instance de chaque méta-port. Récursivement, le connecteur a le choix soit de faire passer une demande de devis, soit de faire passer une commande, soit de gérer une demande d'évolution qui correspond à l'arrivée d'un nouveau client. Ainsi, à chaque demande d'évolution, le connecteur crée deux nouvelles instances des méta-ports

`clientOrderP` et `clientQuotationP` afin de permettre la communication avec le nouveau client qui rejoint l'architecture du composite.

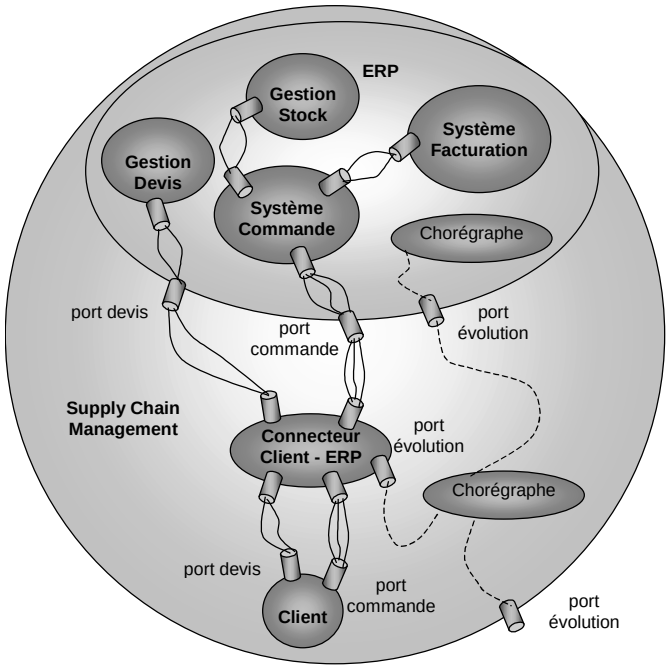


Figure 3 L'architecture globale de la chaîne logistique étendue

Au niveau du comportement permettant la communication des demandes de devis ou des commandes, il est nécessaire de tenir compte du fait que le nombre de clients est inconnu à l'avance, tout comme le nombre d'instances de méta-ports `clientOrderP` et `clientQuotationP`. Chaque méta-entité contient une liste avec toutes ses instances. Le nom de la méta-entité suivi par **`#i`** fait référence à la i-ème instance créée tandis que le nom suivi par **`#any`** fait référence à une instance quelconque. Dans le comportement du connecteur « `clientQuotationP#any = i~quotationReq` » décrit une référence vers la connexion `quotationReq` appartenant à une instance quelconque du méta-port `clientQuotationP` avec sauvegarde de cette référence dans la variable **`i`**. Le fait de garder la référence vers l'instance de la connexion concernée lors de la réception de la requête, permet au connecteur de renvoyer une réponse à l'émetteur de la requête.

```
ClientToErpConnector is connector with {
   ports {
        clientOrderP is OrderPort;
        clientQuotationP is QuotationPort;
        erpOrderP is OrderDemandPort;
        erpQuotationP is QuotationDemandPort;
        newClientP is port with {
             connections {
                  createI is connection(Any),
                  createO is connection(Any) }
             configuration { new createI; new createO       }
             protocol {
                  via createI receive;
                  via createO send ;
                  recurse
             }
        }}
   configuration {
        new clientOrderP; new clientQuotationP ;
        new erpOrderP; new erpQuotationP ; new newClientP}
   routing {
    choose{
        via clientQuotationP#any=i~quotationReq
             receive product:String, quantity:Integer;
        via erpQuotationP~quotationReq send product, quantity;
        via erpQuotationP~quotationRep receive price:Float;
        via clientQuotationP#i~quotationRep send price;
        }
   or   {
        via clientOrderP#any=i~orderReq
             receive product:String, quantity:Integer;
        via erpOrderP~orderReq send product, quantity;
        via erpOrderP~orderRep receive ack:String;
        via clientOrderP#i~orderRep send ack;
        if (ack==“OK”) then {
             via erpOrderP~invoice receive  invoice: String;
             via clientOrderP#i~invoice send  invoice;
        }
   or {
        via newClientP~createI receive ;
        new clientOrderP; new clientQuotationP;
        via newClient~createO send
        }
    then recurse
    }
}
```

Description architecturale 4 Connecteur liant les clients à l’ERP

Le chorégraphe du composite `SupplyChain` (cf. Description architecturale 5) gère les deux scénarios d’évolution : la réception d’un nouveau système de facturation, qu’il fait passer au composite `ERP`, et l’arrivée d’un nouveau client. Dans ce dernier cas, le client est inséré dans le méta-composant `Client`, et un message est envoyé au connecteur `ClientToERP` pour lui demander d’évoluer , comme nous venons de le montrer. Il fait ensuite les attachements entre les derniers ports créés du

connecteur, et la dernière instance créée de client. L'accès à la dernière instance du méta-composant se fait avec le suffixe **#last** aussi bien pour le méta-composant client que pour les méta ports du connecteur.

```
SupplyChain is component with{
   ports {
        erpEvolveP is ERPEvolutionPort;
        newClientP is ClientPort;  }
   constituents {
    clientComponent is Client;
    erpComponent is ERP;
    clientToErp is ClientToErpConnector; }
   configuration {
    new clientComponent; new erpComponent; new clientToErp;
    attach clientComponent~orderP to clientToErp~clientOrderP;
    attach clientToErp~erpOrderP to erpComponent~erpOrderP;
    attach clientComponent~quotationP
        to clientToErp~clientQuotationP;
    attach clientToErp~erpquotationP
        to erpComponent~erpQuotationP;}
   choreographer {
        choose {
             via erpEvolveP~newInvoice
                  receive newInvoiceComponent:InvoiceSystem;
             via erpComponent~erpEvolveP~newInvoice
                  send newInvoiceComponent;
             via erpComponent~erpEvolveP~ack receive ack:String; }
        or
        {    via newClientP~createOut receive c : Client;
             insert component c in Client ;
             via clientToErp~newClientP~createIn send ;
             via clientToErp~newClient~createOut receive ;
             attach clientComponent#last~orderP
                  to clientToErp~clientOrderP#last;
             attach clientComponent#last~quotationP
                  to clientToErp~clientQuotationP#last;}
     then recurse
   }
```

Description architecturale 5 Composant composite SupplyChain

Nous venons de présenter notre deuxième scénario d'évolution dynamique prévue, de nouveaux clients intègrent dynamiquement l'architecture, le connecteur s'adapte en créant de nouveaux ports pour leur permettre de communiquer avec l'ERP.

À travers ces deux exemples d'évolution nous avons montré comment ArchWare C&C-ADL permet de définir des architectures pour des systèmes qui évoluent dynamiquement. Les évolutions présentées incluent des formes de mobilité, puisque à la fois le nouveau système de facturation et les nouveaux clients arrivent dans l'architecture depuis l'extérieur du système. Ces évolutions sont possibles du fait du fondement du langage, notamment l'utilisation du π-calcul d'ordre supérieur typé.

5.4 Vérification des propriétés lors de l'évolution

Les vérifications qui sont faites concernent essentiellement le typage. Ainsi, par exemple, par la connexion `newClient` du port `newClientP` ne peuvent transiter que des composants de type `Client`. Cependant nous mentionnons que nous n'abordons pas d'autres aspects liés à la mobilité, et nous ne faisons qu'une gestion primaire de son état. Plus précisément, dans l'exemple précédent, un composant de type Client doit posséder deux ports de type `QuotationDemandPort` et `OrderDemandPort`, i.e. des ports qui respectent le même protocole. Ceci nous assure que le composant que nous intégrons dans l'architecture est capable d'interagir selon le même protocole que celui de n'importe quel autre client.

En plus de vérifier le typage des composants, d'autres vérifications s'imposent quant à la cohérence globale du système. Parmi les propriétés qui pourraient être vérifiées il y a celle structurelle de la connectivité (i.e. tout élément est attaché à au moins un autre élément) : est-ce qu'une fois modifiée, l'architecture reste toujours correctement connectée ? Il y a également des propriétés portant sur le comportement : est-ce que le nouveau système de facturation suit toujours le même protocole ? etc. Les propriétés peuvent être combinées c'est-à-dire porter à la fois sur la structure et sur le comportement des éléments architecturaux.

Les propriétés suivantes (structurelles et/ou comportementales) sont exprimées dans le langage d'analyse Archware AAL (Alloui *et al.*, 2003), lequel rappelons-le est un langage outillé pour la vérification des propriétés.

La propriété `connectivityOfERPArchitecture` exprime que tout composant doit être connecté à un connecteur : dans notre cas d'étude, l'architecte doit vérifier qu'après remplacement du système de facturation, chaque composant est bien connecté à un connecteurt et non directement à un autre composant de l'ERP.

```
connectivityOfERPArchitecture is property {
-- chaque composant est lié à un connecteur
on self.components.ports apply
            forall { port1 | on self.connectors.ports apply
                  exists { port2 | attached(port1, port2) }
            }
}
```

La propriété `requestBeforeReplyOfOrderSystem` exprime le fait qu'un système de commande ne peut envoyer une réponse avant d'avoir reçu une requête. Cette propriété devrait être conservée après évolution de l'architecture (c'est-à-dire après le remplacement du système de facturation).

```
requestBeforeReplyOfOrderSystem is property {
-- au début aucun reply ne peut avoir lieu avant un request
  on OrderSystem.instances apply
   forall {os | (on os.actions apply isNotEmpty) implies
    (on os.orderP~orderReq.actionsIn apply
        exists {request | on os.orderP~orderRep.actionsOut apply
          forall {reply | every sequence {(not request)*. reply}
                          leads to state {false} }   } ) } }
```

Les évolutions présentées dans cette section doivent être prévues à l'avance, lors de la conception de l'architecture. Nous allons dans les sections qui suivent montrer comment il est possible dans ArchWare de gérer des évolutions *non prévues*.

6. Architecture d'une chaîne logistique étendue avec évolution non prévue

6.1 Considérations sur des cas d'évolution imprévue, gérée de manière dynamique

Nous reprenons l'architecture originale de la chaîne logistique étendue. Dans le cas que nous traitons ici, nous partons du principe que l'architecture de la chaîne logistique étendue que nous avons présentée n'intègre pas les évolutions qui sont censées se produire ni quels sont les éléments architecturaux sur lesquels vont porter les changements provoqués par une évolution. Dans la pratique d'ailleurs, la gestion de la maintenance et de l'évolution de systèmes complexes (client-serveur, distribués à grande échelle ou autres) est gérée de façon très pragmatique, au cas par cas, sans véritablement reposer sur une démarche méthodique (Demeyer *et al.*, 2002).

Soit l'architecture basée sur le scénario décrit en section 4. Cette architecture permet de répondre à un problème posé sans pour autant couvrir l'ensemble des scénarios d'évolution possible. En particulier, qu'advient-il lorsqu'une commande ne peut être satisfaite en raison d'un défaut de stock alors que cela n'avait pas été prévu à l'avance ? Ou alors, que se passe-t-il si le système de facturation doit subitement être externalisé, ou bien si la relation donneur d'ordres fournisseur change ? L'architecture initiale du système en place n'est pas adaptée pour réagir à ce genre d'aléas. Il faut donc intervenir.

Dans ce cadre, nous écartons toute opération de maintenance corrective qui devrait se faire au niveau du code du système; c'est ce qui se passe dans la très grande majorité des cas mais cela s'accompagne de nombreux problèmes d'érosion (Perry et Wolf, 1992) et de non conformité entre le niveau conceptuel et le niveau implémentation (Roshandel *et al.*, 2004). Notre approche a justement pour objectif de présenter une solution lorsque des changements impactent l'architecture: nous pensons que de tels changements doivent justement être pris en compte au niveau architectural et ne devraient pas faire l'objet de modification directe du code. Dans les sections suivantes nous verrons comment cette évolution *non prévue* peut-être gérée *dynamiquement*.

Les travaux menés au sein du projet ArchWare (cf. section 3), incluent une machine virtuelle capable d'interpréter le code architectural. Dans sa version actuelle, la machine virtuelle interprète la couche noyau (cf. section 3) du langage ArchWare π-ADL.

L'approche consistant à pouvoir modifier dynamiquement l'architecture en cours d'interprétation (par la machine virtuelle) est séduisante mais pose un certain nombre de questions parmi lesquelles, la difficulté de gérer l'état du système alors qu'on modifie ce dernier, les problèmes de persistance, de cohérence, en résumé, la gestion de l'évolution en cours d'exécution. L'environnement ArchWare est doté d'outils permettant justement de prendre en compte ces problèmes et de gérer l'évolution (Oquendo *et al.*, 2004). Notre objectif ici n'est pas de présenter ces outils en détail et comment ils fonctionnent ensemble mais d'illustrer un cas d'évolution en nous focalisant sur les mécanismes du langage et de la machine virtuelle permettant de prendre en charge dynamiquement une évolution imprévue. Le scénario d'évolution illustrant le cas de l'évolution *dynamique et non prévue* est différent de celui présenté en section 5, même si tous deux sont basés sur le scénario original (section 4). En particulier, nous introduisons ici, un système de réapprovisionnement au sein de l'ERP.

6.2 Présentation de l'architecture initiale avant évolution

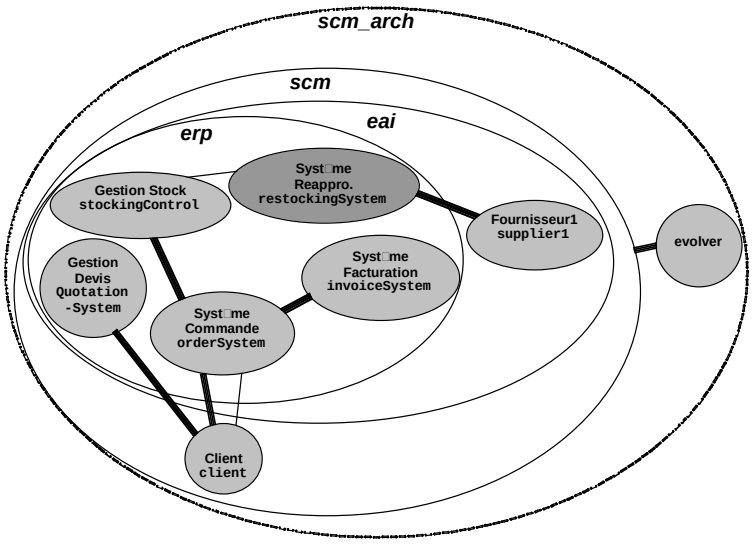


Figure 4 : Architecture initiale

Dans l'exemple que nous allons utiliser pour traiter ce cas, le code architectural est exprimé en utilisant la version noyau du langage et non une surcouche telle que celle utilisée dans la section 5. De ce fait, la structure même de l'architecture diffère un

peu de celle présentée dans les sections précédentes puisque les concepts de *composant* et *composite* sont absents de cette version noyau du langage qui ne comprend que le concept générique d'*abstraction* (une *abstraction* peut composer d'autres abstractions: on parlera parfois pour ces dernières de sous-abstractions). Pour ces mêmes raisons, nous utiliserons par la suite une notation graphique différente pour représenter les architectures (cf. Figure 4).

```
value client is abstraction(String: quotationRequest, Integer: qty);{
   value quotationReq is free connection(String);
   value quotationRep is free connection(Float);
   value orderReq is free connection(String,Integer);
   value orderRep is free connection(String);
   value invoiceToClient is free connection(String);
   value quotationBeh is behaviour {
      via quotationReq send quotationRequest;
      via quotationRep receive amount:Float;
      unobservable; }
   quotationBeh();
   replicate {
       choose {
          quotationBeh();
        or
          behaviour {
             via orderReq send quotationRequest, qty;
             unobservable;
             via orderRep receive ack:String;
              if (ack == "OK) then {
                 via invoiceToClient receive invoice:String; } } } };
   done };
value supplier1 is abstraction(); {
   value restockingOrder1Req is free connection(String, Integer);
   value restockingOrder1Rep is free connection(String);
   replicate {
      via restockingOrder1Req receive wares:String, quantity:Integer;
      unobservable;
      via restockingOrder1Rep send "OK" };
   done };
```

Description architecturale 9 : Le client et le (premier) fournisseur

La Description architecturale 9 écrite en utilisant le langage noyau ArchWare π-ADL, présente le code du client et du premier fournisseur (`supplier1`), qui sont des abstractions relativement simples. Le comportement du client est d'envoyer une demande de devis, d'attendre d'obtenir la réponse à la demande de devis pour soit refaire une demande de devis (dans le cas où il sa demande précédente aurait été infructueuse), soit pour procéder à une commande et attendre la facture. Concernant le fournisseur (`supplier1`), il adopte un comportement très simple: il reçoit une demande de réapprovisionnement et la satisfait; dans le scénario initial, nous avons choisi une relation donneur d'ordres fournisseur simplifiée pour laquelle toute demande de réapprovisionnement est satisfaite (c'est – contractuellement - à la charge du fournisseur de satisfaire la demande de réapprovisionnement peu importe les solutions qu'il doit envisager – sous-traitance, etc.) et d'envoyer un accusé de réception lorsqu'il est prêt à réapprovisionner. Nous verrons par la suite que cette relation va évoluer.

La Description architecturale 10 présente l'abstraction `erp`, qui est une abstraction composée par d'autres abstractions. Rappelons qu'un comportement dans la version noyau du langage est un concept très proche du concept de processus en π-calcul (Milner, 1999). L'ERP est ici composé de système de gestion des devis (`quotationSystem`), du système de gestion des commandes (`orderSystem`), du système de gestion de la facturation (`invoiceSystem`), du système de contrôle du stock (`stockingControl`) et du système de réapprovisionnement (`restockingSystem`).

```
value quotationSystem is abstraction(Float: price); { É }
value orderSystem is abstraction();{ É }
value stockingControl is abstraction(Integer: stock); { É }
value restockingSystem is abstraction();{ É }
value invoiceSystem is abstraction();{ É }
value erp is abstraction(Float: price, Integer: stock); {
       compose {      quotationSystem(price)
              and     orderSystem()
              and     invoiceSystem()
              and     stockingControl(stock)
              and     restockingSystem() } };
value eai is abstraction(Float: price, Integer: stock); {
       compose {      supplier1(20)
              and
                      erp(price, stock) } } };
```

Description architecturale 10 : L'ERP intégré à la solution EAI

La structuration d'une architecture en ArchWare π-ADL repose sur le mécanisme de composition d'abstractions. Ainsi l'abstraction `scm_arch` représente l'architecture complète de la Figure 4 (l'abstraction `scm` (Supply Chain Management) sera détaillée plus loin).

```
value scm_arch is abstraction(); {
       compose {
              scm()
       and
              evolver()} };
```

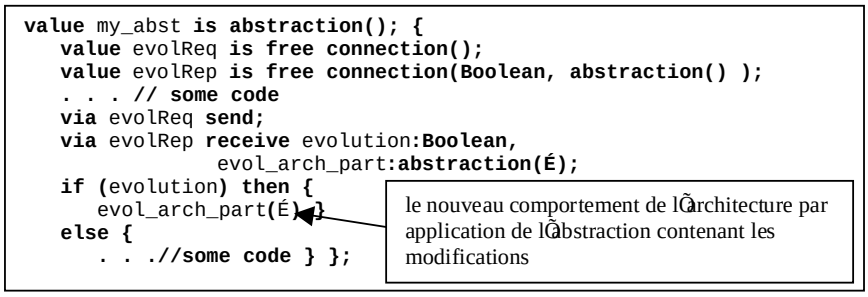
Description architecturale 11 : L'abstraction (globale) `scm_arch`

6.3 Principe de gestion dynamique d'une évolution non prévue

Les mécanismes permettant de traiter des cas d'évolution dynamique non prévue repose sur les caractéristiques du langage ArchWare π-ADL ainsi que la présence d'une machine virtuelle interprétant ce langage et intégrant des mécanismes de support à l'évolution.

Concernant les caractéristiques du langage permettant de gérer l'évolution dynamique, nous retiendrons tout particulièrement la mobilité: en ArchWare π-ADL, une abstraction (un comportement) peut être envoyée par une abstraction sur l'une de

ses connexions, reçue par une autre abstraction puis peut être appliquée par cette dernière (Oquendo *et al.*, 2002). Dans la mesure où l'abstraction communiquée peut être un comportement quelconque, il est possible de modifier dynamiquement le comportement de l'abstraction qui reçoit l'abstraction transmise pour qu'elle adopte un nouveau comportement (celui de l'abstraction transmise). Ce changement de comportement est bien dynamique puisque d'une part le nouveau comportement est reçu en cours d'exécution et que d'autre part sa définition peut également être obtenue au dernier moment et "au vol" pendant l'exécution.



Description architecturale 12 Evolution dynamique d'une abstraction

Dans l'exemple de Description architecturale 12, `my_abst` reçoit sur sa connexion `evolRep` un booléen (`evolution`) et une abstraction (`evol_arch_part`). Le booléen indique, par sa valeur, si `my_abst` va se comporter comme l'abstraction reçue en appliquant cette dernière (si le booléen a pour valeur *true*) ou non. Ainsi, `my_abst` est susceptible d'avoir son comportement modifié sans savoir par avance quelle est la nature des modifications (on constate que le contenu de `evol_arch_part` – donc le nouveau comportement - est inconnu à ce niveau).

Pour recevoir cette abstraction `evol_arch_part` dont la définition doit pouvoir être dynamiquement obtenue, nous avons besoin d'une abstraction toute particulière nommé `evolver`. Cette abstraction (`evolver`) doit être connectée à l'abstraction qui est censée évoluer (ici `my_abst`).

```
value evolver is abstraction(); {
       value evolReq is free connection();
       value evolRep is free connection(Boolean, abstraction);
       via evolReq receive;
       choose {
              {via evolRep send false, any();}
       or
              {value evol_arch_part is ARCH-EVOLUTION;
               via evolRep send true, evol_arch_part ;} } };
```

Description architecturale 13 Description de l'abstraction `evolver`

Le fonctionnement de l'abstraction `evolver` est particulier puisqu'il est lié à la machine virtuelle et décide d'une évolution ou non (`choose` dans Description architecturale 13). L'abstraction `evol_arch_part` est de type `ARCH-EVOLUTION`.

Dans le cas où on décide de procéder à une évolution, la machine virtuelle va rechercher la présence d'un fichier nommé `ARCH-EVOLUTION.adl` contenant la définition d'une abstraction `ARCH-EVOLUTION` (voir ci-après). L'abstraction `evol_arch_part` ayant été déclarée comme de type `ARCH-EVOLUTION` va donc avoir la définition de l'abstraction `ARCH-EVOLUTION` (contenue dans le fichier `ARCH-EVOLUTION.adl`). Ce mécanisme est défini dans l'abstraction `evolver`.

```
value ARCH-EVOLUTION is abstraction(É); {
        // some code };
```

Le code architectural qui se trouve dans un fichier nommé `ARCH-EVOLUTION.adl` peut être exprimé/fourni en cours d'exécution par l'architecte. Cette description architecturale dynamiquement obtenue, et désormais identifiée par `evol_arch_part` au niveau du code, peut ainsi être envoyée à l'abstraction sujette à une évolution (dans l'exemple il s'agit de `my_abst`), pour être appliquée par la suite (reportez-vous à l'exemple traité en section 6.4).

Dans la mesure où la composition est un des mécanismes important du langage ArchWare π-ADL, la portée d'une évolution est directement liée à l'unification[1] d'une abstraction `evolver` à une autre abstraction; dit autrement, le contenu architectural dynamiquement obtenu dans le fichier `ARCH-EVOLUTION.adl` ne pourra impacter que l'abstraction (et donc éventuellement les autres (sous)abstractions qu'elle compose) unifiée avec l'abstraction `evolver`. Il serait toujours possible d'attacher une abstraction `evolver` à toute autre abstraction de l'architecture (permettant ainsi à l'architecture d'évoluer à quel que niveau que ce soit); ceci est cependant contraignant au niveau architectural introduisant des abstractions particulières (que l'on pourrait qualifier de non fonctionnelles). Une alternative consiste à cibler l'abstraction de plus haut niveau (rappelons que les abstractions peuvent en composer d'autres) décrivant la partie de l'architecture susceptible d'évoluer; cette stratégie s'accompagne également de quelques inconvénients: dans un cas il faut pouvoir anticiper la partie de l'architecture susceptible d'évoluer; dans un autre cas, si on choisit l'abstraction qui compose l'ensemble du système (`scm_arch` dans notre cas), une évolution qui ne concernerait qu'une petite partie de l'architecture oblige à reprendre quasiment tout le code architectural dans la définition de l'abstraction contenant l'évolution.

En terme de vérification de propriétés, il est possible de vérifier la préservation ou non des propriétés de l'architecture en cours d'exécution; dans ce cas, on procède à une analyse de l'architecture modifiée avec le contenu du fichier `ARCH-EVOLUTION.adl` avant de charger de manière effective son contenu dans l'architecture en cours d'évolution. La machine virtuelle sollicitera l'utilisateur qui décidera de poursuivre l'évolution (le booléen recevra la valeur *true*) ou de l'annuler (le booléen recevra la valeur *false*).

[1] cf. (Milner, 1999; Oquendo *et al.*, 2002)

6.4 Illustration du principe de gestion dynamique d'une évolution architecturale non prévue

Le cas d’évolution que nous traitons dans cette partie est le suivant : dans l’architecture initiale une hypothèse a été faite selon laquelle il n’y a qu’un seul fournisseur qui peut satisfaire toute demande de réapprovisionnement dès lors qu’elle est déclenchée par le système de réapprovisionnement (on suppose qu'il s'agit d'un engagement contractuel du fournisseur).

Imaginons maintenant que le fournisseur soit en incapacité de satisfaire la demande de réapprovisionnement (pour de multiples raisons) ou/et que l’on veuille diversifier nos réapprovisionnements…que l'on veuille modifier le contrat liant le donneur d'ordres au fournisseur. Nous souhaitons désormais traiter avec un nouveau fournisseur (`supplier2`). Le fournisseur (`supplier1`) de l'architecture initiale reste mais n'intervient qu'en second ressort, dans le cas où le nouveau fournisseur ne peut satisfaire la demande de réapprovisionnement. Ainsi, en fonction d’une commande passée par un client, si l’état du stock est insuffisant, une demande de réapprovisionnement est effectuée. Si la quantité de réapprovisionnement excède la capacité du nouveau fournisseur, la différence entre la quantité souhaitée et celle fournie par le nouveau fournisseur (`supplier2`) fait l’objet d’une seconde demande de réapprovisionnement effectuée auprès du fournisseur (`supplier1`). Ce cas d’évolution est intéressant; d’une part il exige de modifier la structure et la topologie de l’architecture initiale (par apparition d’un nouveau fournisseur non prévu initialement) et, d’autre part, il nécessite de changer le processus de réapprovisionnement du système de réapprovisionnement de l’ERP. Ce processus de réapprovisionnement doit évoluer pour prendre en compte le fait qu’une demande de réapprovisionnement peut ne pas être satisfaite par un seul fournisseur et qu’une autre demande doit être adressée à un autre fournisseur avec la quantité manquante nécessaire à la satisfaction de la commande du client. Ainsi le comportement du système de réapprovisionnement doit être dynamiquement modifié.

L'architecture conforme à ces changements est celle présentée par la Figure 5.

Les changements entre l'architecture initiale (voir Figure 4) et l'architecture modifiée se font au sein de l'abstraction `scm` (voir Figure 5): c'est elle qui va/doit être modifiée; cette abstraction compose les abstractions `eai` et `client` avant l'évolution, puis les abstractions `evol_arch_part` et `client` après évolution (l'abstraction `evol_arch_part` ayant remplacé l'abstraction `eai`). L'abstraction `evolver` est quant à elle unifiée (Oquendo *et al.*, 2002) avec l'abstraction sujette à évolution (`scm`) comme nous l'avons vu lors de la section précédente. L'abstraction `evolver` va être sollicitée par le système `scm` sur la connexion nommée `evolReq`, dès lors qu’une évolution est nécessaire. Notons qu'aucune hypothèse n’a été faite ni sur la nature de l’évolution ni sur la stratégie d’évolution qui interviendront. Dans le cas qui nous intéresse, après que l'abstraction `evolver` ait été sollicitée et si le paramètre `evolution` est égal à *true*, on procède à une évolution: l'abstraction `evol_arch_part` reçue (de l'abstraction `evolver`), est appliquée.

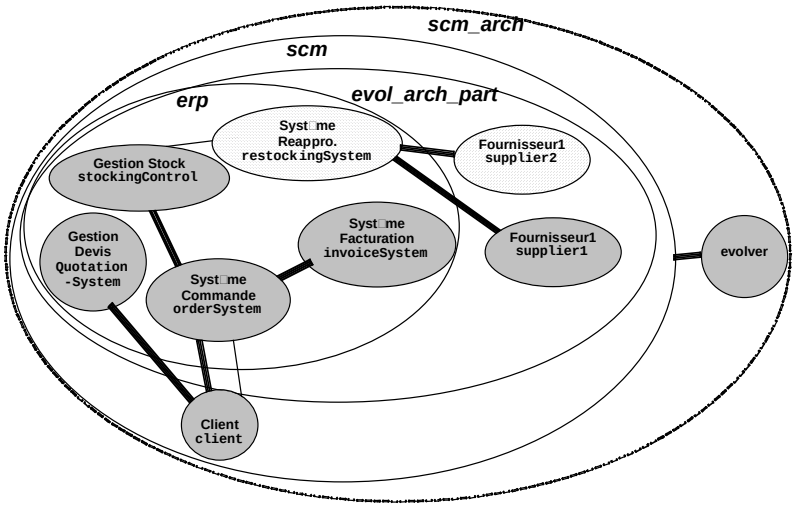


Figure 5 : Architecture avec évolution

Ainsi (cf. Description architecturale 14), l'abstraction `scm` adopte un nouveau comportement qui est celui de l'abstraction `evol_arch_part`. On voit bien dans sa description que l'abstraction `scm` peut se comporter soit comme l'abstraction `eai` (cas de l'architecture initiale) soit comme une abstraction (`evol_arch_part`) dont le comportement est inconnu lors de la définition (puisqu'il est obtenu dynamiquement).

```
value scm is abstraction(); {
   value evolReq is free connection();
   value evolRep is free connection(Boolean, abstraction() );
   compose {
       behaviour {
          via evolReq send;
          via evolRep receive evolution:Boolean,
                 evol_arch_part:abstraction(Float,Integer);
          if (evolution) then {
                evol_arch_part(100.00, 32) }
          else {
                eai(100.00, 32) }}
     and
       client("rollings", 12)       }      };

value scm_arch is abstraction(); {
   compose {   scm()
       and     evolver()} };
```

Application de l'abstraction re□ue

Description architecturale 14 : Evolution de l'abstraction scm

Comme cela a été expliqué dans la section précédente, la machine virtuelle dispose d'une abstraction `ARCH-EVOLUTION`. La définition de cette abstraction est fournie par un fichier `ARCH-EVOLUTION.adl` qui est chargé dynamiquement par la machine virtuelle; l'abstraction `ARCH-EVOLUTION` est alors envoyée sur la connexion `evolRep` puis est ensuite appliquée à l'architecture globale. La définition de cette abstraction

intégrant les changements est donnée par la Description architecturale 15 (ne figurent pas les abstractions n'ayant pas subi de modification afin de pas alourdir le code).

```
value supplier2 is abstraction(Integer capacity); {
  value restockingOrder2Req is free connection(String, Integer);
  value restockingOrder2Rep is free connection(String, Integer);
  via restockingOrder2Req receive wares:String, quantity:Integer;
  unobservable;
  if (quantity > capacity) then {
       via restockingOrder2Rep send "NOK",capacity; }
  else {
       via restockingOrder2Rep send "OK",capacity; }
  done };
value restockingSystem is abstraction(); {
  value restockingReq is free connection(String, Integer);
  value restockingOrder2Req is free connection(String, Integer);
  value restockingOrder2Rep is free connection(String, Integer);
  value restockingOrder1Req is free connection(String, Integer);
  value restockingOrder1Rep is free connection(String);
  via restockingReq receive wares:String, quantity:Integer;
  via restockingOrder2Req send wares, quantity;
  via restockingOrder2Rep receive ack:String, qtyReceived:Integer;
  if (ack == "NOK") then {
       via restockingOrder1Req send wares, (quantity-qtyReceived);
       unobservable;
       via restockingOrder1Rep receives ack2:String; }
  unobservable;
  done };
value erp is abstraction(Float: price, Integer: stock); {
       compose {      quotationSystem(price)
               and    orderSystem()
               and    invoiceSystem()
               and    stockingControl(stock)
               and    restockingSystem() } };
value ARCH-EVOLUTION is abstraction(Float:price, In
       compose {
                      erp(price, stock)
               and    supplier1()
               and    supplier2(20)  } };
```

lÕabstraction qui sera envoyée à lÕarchitecture `scm`, puis appliquée par cette dernière

Description architecturale 15 Définition de l'abstraction `ARCH-EVOLUTION`

Cette nouvelle abstraction diffère de l'abstraction `scm` de l'architecture initiale puisqu'elle intègre désormais un second fournisseur (`supplier2`) impliquant un changement dans le comportement du système de réapprovisionnement (`restockingSystem`) de l'ERP. Le fournisseur nouvellement ajouté (`supplier2`) n'adopte pas le même comportement que le fournisseur déjà présent dans l'architecture initiale (`supplier1`) : en effet, le réapprovisionnement au niveau du nouveau fournisseur (`supplier2`) ne sera assuré que dans le cas où la quantité demandée est inférieure ou égale à la capacité de réapprovisionnement du fournisseur en question. Il est intéressant de souligner qu'en plus d'ajouter dynamiquement un fournisseur, le comportement de ce dernier est différent du fournisseur présent dans l'architecture initiale. Quant au comportement du système de réapprovisionnement, il tient désormais compte des deux fournisseurs : dans le

cas où le premier fournisseur sollicité par la demande de réapprovisionnement (`supplier2` dans l'architecture modifiée) ne peut répondre à la quantité demandée pour le réapprovisionnement, la demande est transmise à un second fournisseur (`supplier1` dans l'architecture modifiée) celui-là même de l'architecture initiale).

Le nouveau comportement de l'architecture (globale) du système prend désormais en compte la substitution de l'élément architectural *eai* (abstraction nommée `eai` dans le code –Description architecturale 10) par l'application de l'abstraction `evol_arch_part` (qui intègre toutes les modifications architecturales). Le passage de l'architecture initiale à l'architecture modifiée (intégrant l'évolution) se fait au niveau de l'abstraction `scm` (voir Description architecturale 14), en utilisant les connexions `evolReq` et `evolRep` comme nous l'avons expliqué précédemment.

Les abstractions étant typées, il serait possible de définir plusieurs stratégies d'évolution, chacune représentée et décrite par un type particulier d'abstraction et précisé au niveau de l'élément architectural `evolver`.

Les changements non initialement prévus effectués en cours d'exécution du système requièrent plus que jamais l'analyse des propriétés architecturales souhaitées. Comme dans les deux autres types d'évolution, les propriétés à analyser peuvent être structurelles, comportementales ou combinées. Cependant les expressions de propriétés AAL porteront sur les concepts du noyau d'ADL comme le montre l'expression suivante de la propriété de connectivité de l'abstraction `scm_arch` :

```
connectivityOfscm_arch is property {
-- chaque abstraction est structurellement connectée à au moins une autre
on self.abstractions apply
    forall { abst1 | on self.abstractions apply
        exists { abst2 | (abst2 <> abst1) and ((abst2.connections
          apply intersection(abst1.connections)) apply isNotEmpty) }  } }
```

Ainsi, nous avons vu (1) que le système initial était en capacité d'évoluer par l'application d'éléments architecturaux inconnus à l'origine et qui peuvent être spécifiés/formalisés en cours d'exécution (à la volée), (2) que l'évolution arrive dynamiquement, en cours d'exécution, et son contenu n'a pas été décidé au préalable, (3) qu'il est toujours possible, avant d'appliquer une évolution (l'abstraction `evol_arch_part` dans l'exemple), de procéder à la vérification des propriétés de la même façon que cela a été montré dans le cas d'évolution prévue dynamique (section 5) de manière à ce que les changements introduits n'entraînent pas une violation des propriétés du système. D'autre part, avec les mécanismes d'évolution que nous avons présentés dans cet article, il est possible de procéder à des évolutions en cascade qui peuvent transformer considérablement une architecture initialement décrite, tout en vérifiant les propriétés à chaque pas d'évolution.

7. Bilan et remarques

A notre connaissance, il n'existe pas d'approche permettant de couvrir l'évolution selon les quatre dimensions présentées en introduction (*prévue* ou *non prévue*, *statique* ou *dynamique*). Or, il est important de fournir un support pour chacun de ces cas. L'approche présentée dans le cadre de cet article permet de couvrir ces quatre cas d'évolution, d'une part par la richesse et le pouvoir d'expression du langage ArchWare ADL, par différentes « formes » de processus centrés architecture (illustrés dans les scénarios que nous avons présentés), d'autre part par les outils de l'environnement et notamment un support à l'exécution (très peu de LDA possèdent un support à l'exécution, Cîmpan et Verjus, 2005).

En particulier, cette approche et son environnement permettent d'envisager deux manières (qui peuvent être combinées) de concevoir et de mettre en œuvre des descriptions architecturales: (1) le découplage et le raffinement du niveau abstrait vers le niveau concret d'une architecture, avec éventuellement la génération du code, (2) de considérer uniquement une description architecturale exécutable (par interprétation de la description par une machine virtuelle). Bien entendu, des avantages et des inconvénients (ou difficultés) sont inhérents à chacune de ces deux manières de concevoir et de mettre en œuvre; on notera des avantages indéniables (processus en étapes, séparation du niveau abstrait et du niveau concret pour l'approche orientée raffinement ; identité entre la description abstraite et l'exécution de cette dernière, même sémantique d'interprétation entre la spécification et l'exécution pour l'utilisation d'une machine virtuelle), ainsi que des difficultés (la gestion des transformations pour le cas du raffinement, et la nécessité d'utiliser la machine virtuelle pour l'exécution). On pense qu'il serait intéressant de disposer de mécanismes permettant de gérer finement l'évolution dynamique non prévue, c'est-à-dire, être doté de moyens permettant d'analyser la portée, l'impact d'un changement, de décider si les changements sont acceptables, tolérés, permis, etc. Des travaux (Alloui, 2005) se consacrent à la gestion de l'évolution en termes d'analyse dans ce contexte ; ces travaux sont complémentaires à ceux présentés dans le cadre de cet article.

Des scénarios similaires mais relevant de cas industriels très concrets (Pourraz *et al.*, 2006) illustrant l'évolution dynamique prévue ou non ont été utilisés et ont permis de valider notre approche.

Des travaux récents (Huang *et al.*, 2006) font de l´extraction et de l´évolution dynamique d´architectures en capturant l'état et le comportement d'un système à base de composants en cours d´exécution, en en déduisant une représentation de l´architecture et en effectuant des modifications sur l'architecture déduite toujours en temps d´exécution du système logiciel. La représentation de l´architecture déduite est cependant sommaire et ne concerne que les systèmes basés sur l´utilisation de composants.

Nos travaux se poursuivent sur l'étude des systèmes à forte composante logicielle en nous focalisant sur leurs architectures. En particulier, nous étudions comment mieux

gérer les abstractions d'évolution (evolver) ou, du moins, déterminer l'impact architectural de telles abstractions. Nos travaux portent également sur l'extraction d'architecture de plus haut niveau (abstrait) à partir de l'architecture du niveau implémentation (concret) et ce, afin de nous focaliser sur la réachitecturisation (dans le cadre du projet Cook (Ducasse et al., 2005).

8. Bibliographie

Abrial J.-B., The B-Book: Assigning programs to meaning, *Cambridge University Press*, 1996

Allen R., Douence R. , Garlan D., Specifying and Analyzing Dynamic Software Architectures, *Proceedings on Fundamental Approaches to Software Engineering*, Lisbonne, Portugal, mars 1998.

Alloui I., Garavel H., Mateescu R., Oquendo F., The ArchWare Architecture Analysis Language. *ArchWare Deliverable D3.1b*, 2003.

Alloui I., Property verification and change impact analysis for model evolution, *1ères journées sur l'Ingénierie Dirigée par les Modèles (IDM'05)*, Paris, 2005, pp. 169-174.

ArchStudio http://www.isr.uci.edu/projects/archstudio.

ArchWare Consortium, 2001. The EU funded ArchWare – Architecting Evolvable Software - project : http://www.arch-ware.org

Azaiez S., Oquendo F., Final ArchWare Architecture Analysis Tool by Theorem-Proving: The ArchWare Analyser, *ArchWare European RTD Project IST-2001-32360, Delivrable D3.5b*, mai 2005.

Barais O., Duchien L., Maîtriser l'evolution d'une architecture logicielle. *In Langages, Modèles, Objets - Journées Composants (LMO-JC'04)*, volume 10 of L'objet, pages 103-116, Lille, France, Mars 2004. Hermès Sciences.

Bergamini D., Champelovier D., Descoubes N., Garavel H., Mateescu R., Serwe W., Final ArchWare Architecture Analysis Tool by Model-Checking, *ArchWare European RTD Project IST-2001-32360, Delivrable D3.6c*, décembre 2004.

Bolusset T., ß-SPACE : Raffinement de descriptions architecturales en machines abstraites de la méthode formelle B, *Thèse de doctorat*, Université de Savoie, septembre 2004

Bradfield J. C., Stirling C., Modal logics and mu-calculi: an introduction. *In Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier, 2001.

Chaudet C., Oquendo F., π-SPACE: A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems**,** *Proceedings of 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, Grenoble - France, septembre 2000.

Cîmpan S., Verjus H., Challenges in Architecture Centred Software Evolution, *CHASE: Challenges in Software Evolution*, Berne, Suisse, avril 2005, pp. 1-4 (a).

Cîmpan S., Leymonerie F., Oquendo F., Handeling Dynamic Behaviour in Software Architectures, *European Workshop on Software Architectures*, Pise, Italie, juin 2005.

Demeyer S., Ducasse S., Nierstrasz O., Object-Oriented Reengineering Patterns, *Morgan Kaufmann, 1-55860-639-4*, 2002.

Ding L., Medvidovic N., Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution, *Proceedings of the 2001 Working IEEE/IFIP Conference on Software Architectures (WICSA-2)*, pages 191-200, Amsterdam, the Netherlands, août 2001.

Ducasse S., Alloui I., Cimpan S., Verjus H. et Couturier V., COOK: Réarchitecturisation des applications industrielles objets, projet GIP-ARN (JC05 42872), 2005.

Favre J.-M., Estublier J., Blay M., L'Ingénierie Dirigée par les Modèles : au-délà du MDA, *Edition Hermes-Lavoisier*, 240 pages, ISBN 2-7462-1213-7, février 2006.

Huang G., Mei H., Yang F.-Q., Runtime recovery and manipulation of software architecture of component-based systems, *Journal of Automated Software Engineering*, Volume 13, Issue 2, Pages: 257 – 281, 2006.

Inverardi P., Wolf A., Yankelevich D., Static Checking of System Behaviors Using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239-272, juillet 2000.

Kazman R., Carriere S., Playing detective: Reconstructing software architecture from available evidence, *Journal of Automated Software Engineering*, 6(2):107-138, 1999.

Lehman M. M., Laws of Software Evolution Revisited, *In Proceedings of the European Workshop on Software Process Technology*, pages 108-124, 1996.

Leymonerie F., ASL language et outils pour les styles architecturaux. Contribution a la description d'architectures dynamiques, *Thèse de doctorat, Université de Savoie*, décembre 2004.

Leymonerie F., Blanc dit Jolicoeur L., Cimpan S., Braesch C., Oquendo F., Towards a business process formalisation based on an architecture centered approach, *6th Int. Conf. on Enterprise Information Systems (ICEIS 2004)*, Vol. 3, Porto, Portugal, April 2004, pp. 513-518

Manset D., Verjus H., McClatchey R., Oquendo F., A Formal Architecture-Centric Model-Driven Approach For The Automatic Generation Of Grid Applications, *In proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS'06)*, mai 2006, Paphos – Chypre.

Medvidovic N., Taylor R.N., A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 70-93*, janvier 2000.

Medvidovic N., Egyed A., Gruenbacher P, Stemming architectural erosion by architectural discovery and recovery, *in STRAW'03: Second International SofTware Requirements to Architectures Workshop at ICSE 2003*, Portland, Oregon, 2003.

Medvidovic N., Jakobac V., Using software evolution to focus architectural recovery, *Automated Software Engineering*, Volume 13, Issue 2, Pages: 225 – 256, 2006.

Mens T., Buckley J., Rashid A., Zenger M., Towards a taxonomy of software evolution, *In Workshop on Unanticipated Software Evolution*, Varsovie, Poland, 2003.

Mens T., Wermelinger M., Ducasse S., Demeyer S., Hirschfeld R., Challenges in software evolution, *In Proceedings IWPSE'05 (8th International Workshop on Principles of Software Evolution)*, pages 123-131. IEEE Press, 2005.

Mens K., Kellens A., Pluquet F., Wuyts R., Co-evolving code and design with intensional views – a case study, *Journal of Computer Languages, Systems and Structures* 32, 2, pages 140-156., 2006.

Milner R., Communicating and Mobile Systems: the pi-calculus. *Cambridge University Press*, 1999.

Monroe R., Capturing Software Architecture Design Expertise with Armani. Technical Report CMU-CS-98-163, January 2001

Oquendo F., π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures, *ACM Software Engineering Notes*, Vol. 29, No. 3, pp. 15-28, mai 2004.

Oquendo F., Alloui I., Cîmpan S., Verjus H., The ArchWare ADL: Definition of the Abstract Syntax and Formal Semantics. ArchWare European RTD Project IST-2001-32360, Deliverable D1.1b, décembre, 2002.

Oquendo F., Warboys B., Morrison R., Dindeleux R., Gallo F., Garavel H., Occhipinti C., 2004. ArchWare: Architecting Evolvable Software. In proceedings of the first European Workshop on Software Architecture (EWSA 2004), pages 257-271, St Andrews - UK, mai 2004.

Perry D.E., Wolf A.L., Foundations for the study of software architecture, *ACM SIGSOFT Software Engineering Notes* 17, 4, pages 40-52, 1992.

Plat N., Gorm Larser P., An Overview of the ISO_VDM-SL Standard, *ACM SIGPLAN Notices*, 1992.

Pourraz F., Verjus H., Oquendo F., An Architecture-Centric Approach For Managing The Evolution Of EAI Service-Oriented Architecture *In proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS'06)*, mai 2006, Paphos – Chypre .

Ratcliffe O., Cimpan S., Oquendo F.**,** Case study on architecture-centered design for monitoring views at CERN, *5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, Pittsburgh, Pennsylvania, EU, November 2005

Revillard J., Cimpan S., Benoit E., Oquendo F., Intelligent Instrument Design With ArchWare ADL, *5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, Pittsburgh, Pennsylvania, EU, November 2005

Riva C., View-based software architecture reconstruction, *Ph.D. thesis*, Technical University of Vienna, 2004.

Roshandel R., van der Hoek A., Mikic-Rakic M., Medvidovic N.,Mae: A System Model and Environment for Managing Architectural Evolution, *ACM Transactions on Software Engineering and Methodology*, Vol. 13, Issue 2, pages 240-276, April 2004.

Schmerl B., Garlan D, Acme Studio: Supporting Style Centered Architectural Development, *Proceedings of the 2004 Internation Conference on Software Engineering*, Edimbourg, Ecosse, mai 2004.

Tibermacine C., Fleurquin R., Sadou S., Preserving Architectural Choices throughout the Component-Based Software Development Process. In Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05), Pittsburgh, Pennsylvania, USA, novembre 2005

Tran J., Holt R., Forward and reverse repair of software architecture, *In Proceedings of CASCON 2003*, Toronto, Canada, 1999.

Verjus H., Cîmpan S., Alloui I., Oquendo F., Gestion des architectures évolutives dans ArchWare, 1ère Conférence francophone sur les Architectures Logicielles (CAL 2006), Nantes, France, septembre 2006, pp. 41-57.

Woods S., Carriere S., Kazman R., The perils and joys of reconstructing architectures, *SEI Interactive, Software Engineering Institute*, September 1999.