Gestion des architectures évolutives dans ArchWare

Hervé Verjus* — Sorana Cîmpan* — Ilham Alloui* — Flavio Oquendo**

* *LISTIC*
Université de Savoie
B.P. 806, F - 74016 Annecy Cedex
{herve.verjus, sorana.cimpan, ilham.alloui}@univ-savoie.fr

** *VALORIA*
Université de Bretagne-Sud
B.P. 573, F - 56017 Vannes Cedex
flavio.oquendo@univ-ubs.fr

RÉSUMÉ. *La prise en compte de l'évolution très tôt dans le développement du logiciel est un phénomène assez récent et va de pair avec l'apparition de langages à fort pouvoir d'expression. C'est notamment dans les approches centrées architectures de ces dernières années que sont apparus des langages formels de description architecturale, certains intégrant des mécanismes permettant de prendre en compte l'évolution d'une architecture logicielle. Pour autant, plusieurs cas d'évolution peuvent être mis en évidence, selon que l'évolution est prévue ou non et qu'elle est traitée de façon dynamique ou non. L'article présente une approche centrée architecture, ArchWare, ainsi que ses mécanismes permettant de couvrir plusieurs types d'évolution qui seront illustrés au travers de scénarios.*

ABSTRACT. *Work on architecture-centred software development has been undertaken during the past years, leading to the proposition of architecture description languages with high expression power. The concerns that are focused on lately are related to the impact these new approach may have on the software evolution management, namely the possibility of extending the evolution issues to earlier development phases. Several evolution possibilities may be envisaged, depending on whether the evolution was or not foreseen during the design phase, and on whether the evolution takes place during the software execution or not. Existing propositions address some of these possibilities. In this paper we present how the ArchWare languages and environment allow us to cover all the identified evolution possibilities.*

MOTS-CLÉS : *évolution, architecture logicielle, langage formel, propriétés, scénarios d'évolution.*

KEYWORDS: *evolution, software architecture, formal language, properties, evolution scenarios.*

1. Introduction

Les activités économiques utilisent de plus en plus de logiciels pour atteindre des objectifs définis. Ceci a conduit à des systèmes centrés logiciel de plus en plus complexes et soumis aux exigences croissantes des utilisateurs en termes de qualité et de fonctionnalités. Ces systèmes doivent en plus 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. Ainsi, les systèmes logiciels doivent avoir une capacité d'adaptation importante.

Dans le domaine de la recherche, plusieurs approches sont proposées avec pour objectif l'amélioration du développement de ces systèmes comme les approches orientées aspects, celles fondées sur les patrons de conception et d'implémentation, les approches qui fournissent des langages et outils formels, etc. Plus récemment, les approches centrées architecture ont permis d'étudier de tels systèmes complexes en considérant leur architecture comme point central de leur cycle de vie. Des langages de description architecturale ont été proposés (Medvidovic et al., 2000), de même que des environnements centrés architecture (Schmerl et al., 2004, ArchStudio) et des outils d'analyse permettent d'étudier certaines propriétés des systèmes considérés (Corradini et al. 2002).

L'évolution d'un système est très importante, l'essentiel des budgets lui étant consacré (considérant que la maintenance est un cas d'évolution); elle devrait donc être considérée au cours de toutes les phases du cycle de vie des systèmes centrés logiciel. Les travaux que nous présentons dans cet article focalisent sur la prise en compte de l'évolution dans une approche de développement centré architecture, que nous avons expérimentée dans le cadre du projet ArchWare (ArchWare, 2001). Dans ce contexte, nous considérons des évolutions qui ont un impact sur l'architecture du système ; les activités de maintenance qui n'ont pas d'impact sur l'architecture pouvant être gérés de manière classique[1].

Nous distinguons quatre types d'évolution (Cîmpan et al., 2005), selon que l'évolution est *statique* (hors exécution du système) ou *dynamique* (pendant l'exécution du système), *prévue* (lors de la conception du système) ou *non-prévue*. Le cas de l'évolution dynamique n'a pas été souvent abordé, en particulier si l'évolution n'a pas été prévue avant le lancement de l'exécution du système.

Les évolutions *non prévues* et *statiques* sont les plus couramment étudiées et mises en œuvre ; ces évolutions interviennent soit au niveau des spécifications du système et les modifications doivent être propagées dans les phases amont du processus de développement, soit directement au niveau du code (Demeyer et al., 2002). Cette dernière situation est la plus fréquente et les modifications sont effectuées sur le code indépendamment de l'architecture conceptuelle (abstraite) du système. D'une

[1] Dans le restant de ce papier nous employons le terme évolution pour l'évolution avec impact architectural. La prise en compte de cette évolution est considérée dans le cadre des approches centrées architecture.

part, il n'est généralement pas possible de raisonner (au sens formel) sur du code pour analyser des propriétés du système et d'autre part, se pose le problème de la cohérence entre l'architecture abstraite et le code modifié (Cîmpan et al., 2005) car les spécifications sont rarement mises à jour.
Dans le cas des évolutions *prévues*, que ce soit pour une prise en charge *statique* ou bien *dynamique*, il est possible de prévoir l'évolution de l'architecture d'un système logiciel selon divers scénarios que l'on peut anticiper ou que l'on connaît par avance. Par exemple, les systèmes à architecture ouverte doivent accepter de nouveaux composants logiciels en cours d'exécution. Dans ce contexte, deux aspects importants sont à prendre en considération : d'une part la connaissance des situations/scénarios nécessitant une évolution du système considéré, d'autre part des moyens permettant d'exprimer et de permettre à l'architecture du système logiciel en question d'évoluer selon les prévisions. Les évolutions *prévues* et *dynamiques* pourraient être effectuées au niveau de l'architecture sans intervention d'un tiers (humain ou autre) pourvu que l'architecture en question possède les mécanismes nécessaires pour le faire. Dans le cas des évolutions *non prévues* et qui sont à effectuer « à la volée » en temps d'exécution, l'architecture du système logiciel devrait être dotée d'un minimum de points d'entrée et de mécanismes permettant à un tiers (humain ou autre) de la faire évoluer.
Par ailleurs, quel que soit le type d'évolution effectué, il est important s'il n'est nécessaire de s'assurer de la cohérence du système dans sa nouvelle configuration. Une évolution qui pourrait rendre un système incohérent devrait être soit accompagnée par une analyse du système en temps de conception, soit être détectée pour être réparée par une analyse en temps d'exécution du système. Dans l'un ou l'autre de ces deux cas, on devrait pouvoir à la fois exprimer et analyser des propriétés du système en question. En termes d'architectures logicielles, les propriétés auxquelles on s'intéresse sont celles non fonctionnelles comme la complétude, la sûreté, etc. Ces propriétés peuvent porter aussi bien sur la structure que sur le comportement du système sujet à évolution.
Dans le reste de cet article, nous montrerons comment notre approche dans ArchWare permet de répondre à ces différents problèmes. La section 2 présente les fondements et la structuration des langages de description et d'analyse d'architecture proposés dans le cadre du projet. La section 3 présente 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 4), *non prévue* (section 5) couvrant les cas *statique* (section 5.1) et *dynamique* (section 5.2). La section 6 présente un positionnement de nos travaux et nous conclurons en section 7.

2. 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 de telles architectures à travers une machine virtuelle.
Cette section présente la famille de langages, notamment ses bases formelles et la structuration en couches, avec un noyau et un mécanisme d'extension pour la construction de langages de plus en plus spécifiques.

Le langage noyau. Le langage formel noyau, appelé Archware π-ADL (Oquendo et al., 2002) est fondé sur le π-calcul d'ordre supérieur typé (Milner, 1999), la programmation persistante et les mécanismes de composition et décomposition. Archware π-ADL est une extension bien-formée du π-calcul 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) que des comportements internes d'éléments architecturaux. 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 de comportement* 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 basé sur la notion de style architectural. Un style architectural délimite une famille d'architectures logicielles, famille qui partage un certain nombre de propriétés (exprimées donc au niveau du style) et de types d'éléments architecturaux admis. Ainsi par exemple le paradigme composant-connecteur peut être vu comme un style architectural. Ce dernier indique que les éléments de l'architecture sont soit des composants, soit des connecteurs ce qui fournit de fait un nouveau vocabulaire (qui remplace la notion assez générique d'élément architectural ou abstraction de comportement). Ce style spécifie également que toute communication entre composants passe obligatoirement par un connecteur. Le langage ArchWare ASL permet la définition de styles architecturaux (Leymonerie, 2004) 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. La famille de langages ArchWare ADL propose entre autres un langage composant-connecteur, ArchWare C&C-ADL (Cîmpan et al., 2005b, Leymonerie, 2004) associé au style Composant-Connecteur et défini en utilisant ArchWare ASL à partir du langage noyau.
Les composants ainsi que les connecteurs sont des éléments de première classe, et 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. Les éléments architecturaux qu'ils soient atomiques ou bien composites peuvent avoir des attributs utilisés pour les paramétrer.
Le comportement d'un élément composite résulte de la composition parallèle des comportements des 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 description d'architectures dynamiques est possible de par la construction de cette couche composant-connecteur à partir du langage noyau (et donc du π-calcul) complété par des actions dédiées au comportement dynamique (création dynamique d'éléments et reconfiguration). 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 occurrences. 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.

3. Présentation du scénario

Pour illustrer nos propos et les différents cas d'évolution que nous avons présentés en introduction, nous utiliserons des scénarios en relation avec le même cas d'étude : celui d'un système client-serveur que nous avons choisi pour sa simplicité, l'approche ayant été par ailleurs validée dans plusieurs autres applications (Blanc dit Jolicoeur et al., 2003, Blanc dit Jolicoeur et al., 2004, Pourraz et al., 2006).
Dans un scénario normal, la demande du client (on ne discute pas actuellement de la nature de la requête) peut être satisfaite par le serveur qui lui retourne (si la requête l'exige) une réponse. Dans le scénario qui nous intéresse la demande du client ne peut être satisfaite par le serveur (plusieurs raisons possibles : surcharge du serveur, panne, etc.). Parmi les quatre types d'évolution présentés en introduction, trois d'entre eux seront illustrés dans la suite de cet article. Nous n'illustrerons pas le 4ème cas « évolution prévue et effectuée de manière statique », celui-ci étant de notre point de vue trivial.
Evolution 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 remplacer le serveur défaillant par un autre serveur (section 4).
Evolution non-prévue, statique. L'architecture du système ne peut évoluer elle-même puisqu'il s'agit d'une évolution non prévue. Dans ce cas, l'architecte modifie l'architecture abstraite, les propriétés architecturales attendues étant vérifiées sur cette nouvelle configuration qui est par la suite raffinée jusqu'à obtention d'une architecture concrète (section 5.1). Ceci nécessite l'arrêt de l'exécution du système.
Evolution non prévue, dynamique. L'évolution non prévue se fera alors que le système est en cours d'exécution. Nous montrerons comment l'architecte peut pallier

le dysfonctionnement du serveur en fournissant la partie de l'architecture qui devra remplacer la partie défaillante (section 5.2).

4. Architecture Client-Serveur 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, en utilisant les capacités du langage ArchWare C&C-ADL à représenter des architectures dynamiques. Dans notre scénario, l'architecture est formée par un client, un serveur et un connecteur. A chaque fois que le serveur tombe en panne, une nouvelle instance est créée dynamiquement et attachée au connecteur de façon complètement transparente pour le client qui utilise les services du serveur. Dans ce qui suit, cette architecture dynamique est présentée de manière incrémentale, à travers la définition du client, du serveur, du connecteur puis l'agencement global des différents éléments.

Le composant Client. Les composants atomiques comprennent trois parties. La partie dédiée à la déclaration des `méta-ports`, la partie `configuration` indiquant les ports prévus dans la configuration initiale et enfin la partie `computation` présentant le comportement du composant.

```
Client is component with{
   ports {
        access is port with {
             connections { call is connection(Any),
                           wait is connection(Any)}
             configuration { new call; new wait }
             protocol { via call send ;
                        via wait receive ;
                        recurse } } }
   configuration { new access }
   computation {
        via access~call send any();
        unobservable;
        via access~wait receive reply:Any;
        recurse  }
} – end méta-component Client
```

Description architecturale 1 Le composant atomique Client

La configuration initiale contient une instance du méta-port `access`. Le comportement récursif du client consiste en l'envoi (connexion `call`) d'une requête suivi par la réception (connexion `wait`) d'une réponse.

Le composant Serveur. Le méta-composant serveur est également un élément atomique qui possède un attribut booléen `down` qui stocke son état. Le traitement interne de la requête n'état pas important à ce niveau d'abstraction, il est représenté par une action `unobservable`. La valeur de l'attribut est accessible à travers la connexion `down` d'un port dédié nommé `attribute(`en lecture avec un `receive`,

en écriture avec un send). La valeur de l'attribut down est vérifiée en permanence. Si le serveur n'est pas en panne, il peut soit traiter une requête (réception, traitement, envoi de la réponse), soit s'arrêter. Tel qu'il est défini, le serveur ne peut pas s'arrêter pendant le traitement d'une requête. Après le choix (recurse), le serveur reprend avec la vérification de l'attribut down.

```
Server is component with{
   ports {
    sAccess is port with {
        connections { request is connection(Any),
                      reply is connection(Any)  }
        configuration { new request;   new reply }
        protocol { via request receive request:Any ;
                   via reply send any();
                   recurse }}
   }
   attributes {down: boolean default value is false }
   configuration { new sAccess }
   computation {
    via attribute~down receive v:Boolean;
    if not(v) then {
     choose
      { via sAccess~request receive request:Any;
        unobservable;
        via sAccess~reply send any() }
     or
      { via attribute~down send true}
        then recurse
   }    }
}– fin méta-composant Server
```

Description architecturale 2 Le composant Serveur avec prise en compte des pannes

Le connecteur Link. Le client et le serveur communiquent à travers un connecteur atomique nommé Link. Ce connecteur a deux ports, un pour la communication avec le client et l'autre pour la communication avec le serveur. Son comportement (représenté dans la partie routing de sa définition) consiste à faire passer la demande du client vers le serveur et l'envoi de la réponse du serveur vers le client.

```
Link is connector with{
   ports {  clientAccess is port with { … },
            serverAccess is port with { … }  }
   configuration { new clientAccess ;  new serverAccess }
   routing {
       via clientAccess~call receive request:Any;
       via serverAccess~request send request;
       via serverAccess~reply receive reply:Any;
       via clientAccess~wait send reply;
       recurse
   } }– fin méta-connecteur Link
```

Description architecturale 3 Le connecteur Link

Le composite DynamicClientServer. L'architecture globale est définie en tant que composant composite, nommé `DynamicClientServer`. La partie configuration comporte les occurrences initiales de ports et éléments constituants ainsi que les attachements entre les ports des constituants (éventuellement avec des ports du composite). La partie chorégraphe décrit le comportement dynamique du composite. La configuration initiale du `DynamicClientServer` indique l'existence d'un client, un serveur et un connecteur entre les deux. Le chorégraphe vérifie en permanence la valeur de l'attribut `down` pour le serveur courant. 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 serveur est accédée avec `Server#last`. Si le serveur est en panne, il est détaché du connecteur, une nouvelle instance est créée et attachée au connecteur. Le composite `DynamicClientServer` tel qu'il est défini n'a pas de ports, il représente donc une architecture fermée.

```
DynamicClientServer is component with{
   ports {
   }
   constituents {
    Client is component with{ … }
    Server is component with{ … }
    Link is connector with{ … }
   }– end constituents
   choreographer {
    via Server#last~attribute~down receive down:Boolean;
    if (down) then {
     detach Server#last~sAcess from Link~serverAccess ;
     new Server;
     attach  Server#last~sAcess to Link~serverAccess;
     recurse}
    else recurse
   }
   configuration {
    new Server; new Client ; new Link;
    attach Server~sAcess to Link~serverAccess;
    attach Client~acess to Link~clientAccess }
   } – fin méta-composant DynamicClientServer
```

Description architecturale 4 Architecture client serveur

Rappelons le fait que nous nous situons au niveau de la description de l'architecture. Comme nous venons de le voir, il est spécifié qu'un serveur peut tomber en panne entre deux traitements de requêtes. Ceci indique que dans le système prêt à l'exécution (après génération du code vers un langage cible) il faut prévoir des mécanismes permettant de gérer une panne lors du traitement d'une requête. La création d'une nouvelle instance spécifiée au niveau de l'architecture n'implique pas forcément, au niveau du système, la mise en place d'un nouveau serveur (peut correspondre à la réparation du serveur existant).

L'évolution dynamique prévue permet donc la prise en compte d'évolutions planifiées, pour faire face à certaines situations bien identifiées, avec des solutions

connues. Dans la section suivante nous allons voir comment faire face à des situations qui impliquent une évolution non prévue.

5. Architecture Client-Serveur avec évolution non prévue

Nous reprenons le scénario de l'exemple décrit en section 3. Cependant, nous y introduisons quelques spécificités pour illustrer et gérer ce cas d'évolution. Prenons l'exemple d'un système classique de sauvegarde de fichiers dont l'architecture est de type client-serveur. Dans le cas que nous traitons ici, nous partons du principe que l'architecture du système de sauvegarde n'intègre pas le fait qu'elle soit amenée à évoluer. Dans la pratique d'ailleurs, la gestion de la maintenance et de l'évolution de systèmes complexes (clients-serveurs 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 3, à savoir pour notre cas, un client demande l'enregistrement d'un fichier à un serveur de sauvegarde. Cette architecture permet de répondre à un problème posé sans pour autant couvrir l'ensemble des scénarios d'évolution possibles. En particulier, qu'advient-il lorsqu'un serveur tombe en panne ou bien lorsque la capacité maximale de sauvegarde est atteinte ? Imaginons que le système ait été déployé, qu'il fonctionne, qu'il donne même satisfaction depuis quelque temps déjà et que de tels aléas se produisent, l'architecture du système en place n'étant pas adaptée pour réagir à ce genre de situations, il faut intervenir depuis l'extérieur du système. Dans ce type d'évolution, nous écartons toute opération de maintenance corrective qui devrait se faire au niveau du code du système. Nous savons bien entendu que c'est ce qui se passe dans la très grande majorité des cas mais notre approche a justement pour objectif de présenter des alternatives. Dans la suite, nous verrons comment cette évolution *non prévue* peut-être gérée *statiquement* et *dynamiquement*.

5.1 Gestion statique

Ce type d'évolution est statique dans le sens où la description de l'architecture modifiée nécessite de générer de nouveau l'application (code source) avant de relancer son exécution. Dans notre cas, l'architecte partant d'une architecture reposant sur deux composants (*Client*, *Server*) et un connecteur (*Link*), peut la modifier manuellement en y ajoutant une nouvelle instance de serveur le moment où le premier est surchargé et en propageant lui-même le changement sur l'ensemble des autres éléments, en particulier, les attachements et détachements de composants/connecteurs.
A partir du moment où l'architecte a changé l'architecture, des 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 serveur suit toujours le même protocole (i.e. un processus répliqué composé d'une réception de requête suivi d'un envoi de réponse) ? etc.
Les deux propriétés suivantes (une structurelle et une comportementale) sont exprimées dans le langage d'analyse Archware AAL (Alloui et al., 2003), lequel langage est outillé pour la vérification de telles propriétés. Le langage AAL est formellement fondé sur la logique des prédicats et sur le μ-calcul modal, un calcul qui permet d'exprimer des propriétés sur des systèmes à transitions étiquetées. Les vérifications peuvent être faites par preuve de théorèmes, vérification de modèle ou bien des techniques spécifiques.
La première propriété exprime que tout composant doit être connecté à un connecteur. Ceci englobe le fait que le nouveau serveur *s'* est bien connecté à un connecteur et pas directement à un client.

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

Description architecturale 5 Propriété liée au style composant connecteur

La seconde propriété exprime le fait que la première action d'un serveur est toujours la réception d'une requête. Cette propriété doit être conservée après évolution de l'architecture (i.e. ajout du nouveau serveur).

```
firstServerActionOfClientServerArchitecture is property {
-- au début aucun reply ne peut avoir lieu avant un request
  on Server.instances apply
   forall {s | (on s.actions apply isNotEmpty) implies
    (on s.access~call.actionsIn apply
        exists {request | on c.access~wait.actionsOut apply
          forall {reply | every sequence {(not request)*. reply}
                          leads to state {false} }   } ) } }
```

Description architecturale 6 Propriété du protocole de communication client-serveur

Plus précisément, la propriété stipule que pour toute instance de serveur, si l'ensemble des actions de communication (envoi/réception) n'est pas vide, alors s'il existe une `request` reçue via `access~call`, toute réponse `reply` envoyée via `access~wait` ne doit pas avoir été envoyée avant la réception de `request`. Bien que nous ne présentions que ces deux exemples de propriétés, il est important de noter que le langage AAL permet d'exprimer des propriétés architecturales aussi bien génériques (indépendantes de toute application) que spécifiques et qui peuvent porter aussi bien sur la conception que sur l'exécution de l'architecture du système.

5.2 Gestion dynamique

Les travaux menés au sein du projet ArchWare (cf. section 2), incluent une machine virtuelle capable d'interpréter le code architectural. Dans sa version actuelle, la machine virtuelle interprète la couche noyau (cf. section 2) du langage, ArchWare π-ADL. Aussi, 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 4.
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 le modifie, les problèmes de persistance, de cohérence, bref, la gestion de l'évolution. L'environnement ArchWare est doté d'un certains nombre 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 et comment ils fonctionnent ensemble mais d'illustrer un cas d'évolution simple en utilisant les facilités du langage et de la machine virtuelle.

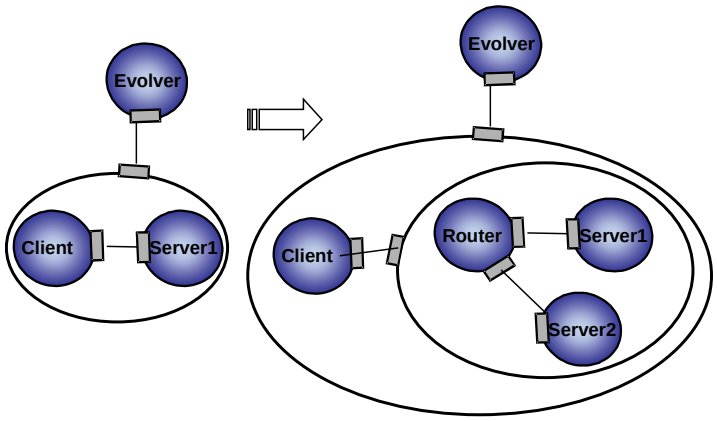
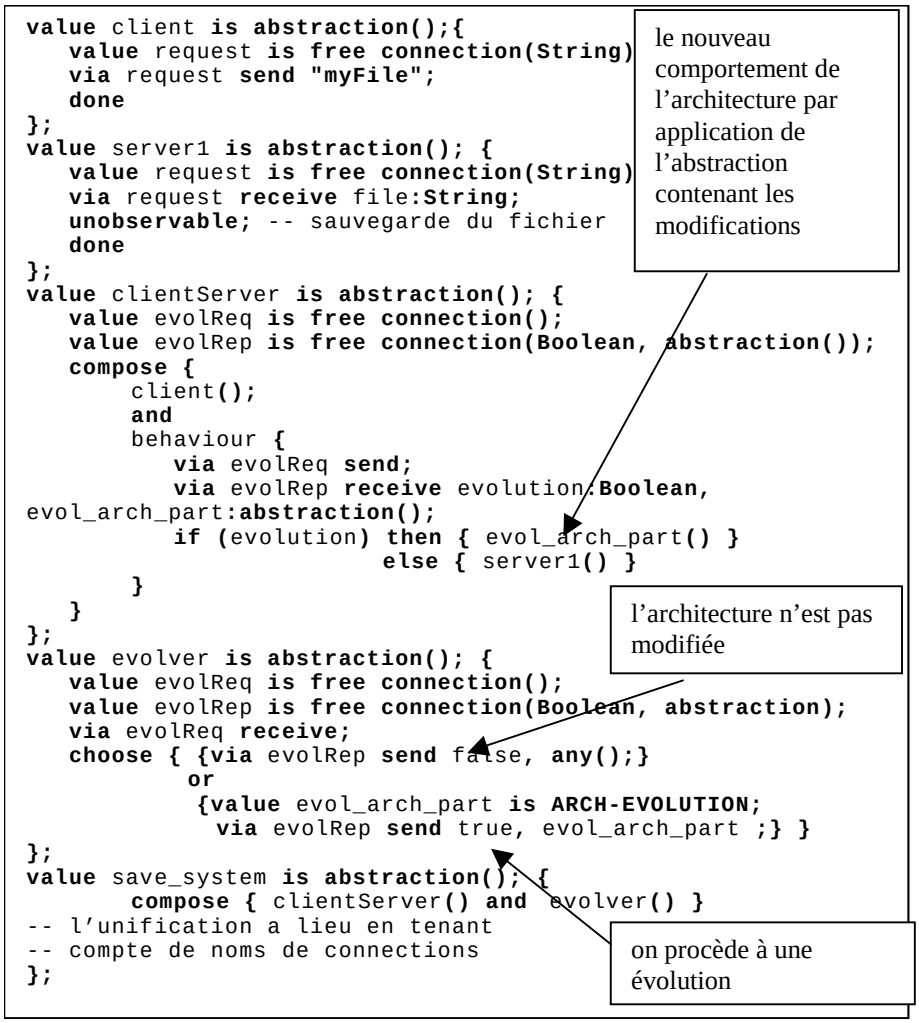


Figure 1 Principe d'évolution de l'architecture du système de sauvegarde

Etudions cette évolution. Suite à un aléa qui s'est produit (panne par exemple), le serveur de sauvegarde se trouve en incapacité de fonctionner correctement. Le système de sauvegarde est ainsi bloqué. Nous allons faire évoluer ce système (donc son architecture) par l'ajout d'un routeur et mettre en parallèle deux serveurs de sauvegarde (faire de la duplication de sauvegarde) et que cette évolution se fasse de façon transparente pour le client. La requête de demande de sauvegarde du client doit être désormais traitée par le routeur (cf. description ci-dessous), qui envoie simultanément et en parallèle deux requêtes de sauvegarde pour les deux serveurs.



Description architecturale 7 Représentation d'une architecture client serveur dans le langage noyau

La description architecturale ci-dessus comprend un élément architectural appelé ici *evolver*. Cet élément va être sollicité par le système client-serveur (sur la connexion nommée *evolReq*), dès lors qu'une évolution est nécessaire. Certes les cas nécessitant qu'une évolution intervienne sont prévus (ils font partie de la description), mais en revanche, aucune hypothèse n'a été faite sur le type d'évolution ou la stratégie qui interviendra. Dans le cas qui nous intéresse, lorsque l'élément *evolver* est sollicité, soit (*choose* dans le code) l'utilisateur décide de ne rien changer à l'architecture (le paramètre *evolution* reçu sur la connexion est alors égal à *false*),

soit on procède à une évolution (le paramètre *evolution* reçu sur la connexion est alors égal à *true*).
Dans ce cas, la machine virtuelle dispose d'une abstraction particulière ARCH-EVOLUTION. Un fichier ARCH-EVOLUTION.adl est alors chargé dynamiquement par la machine virtuelle et l'abstraction *ARCH-EVOLUTION* (une abstraction dans la version noyau du langage est un concept très proche du concept de processus en π-calcul (Milner, 1999) est alors envoyée sur la connexion *evolRep* qui est ensuite appliquée à l'architecture globale.
Cette nouvelle abstraction met en œuvre désormais un routeur et deux serveurs de sauvegarde. Le nouveau comportement de l'architecture du système prend désormais en compte la substitution du serveur *server1* par l'application de l'élément *evol_arch_part*.

```
value server1 is abstraction(); {
        value request1 is free connection(String);
        via request1 receive file:String;
        unobservable; -- saves the file
        done
};
value server2 is abstraction(); {
        value request2 is free connection(String);
        via request2 receive file:String;
        unobservable; -- saves the file
        done
};
value router is abstraction(); {
        value request is free connection(String);
        via request receive file:String;
        value request1 is free connection(String);
        value request2 is free connection(String);
        compose {
                via request1 send file;
        and
                via request2 send file;
        }
        done
};
value ARCH-EVOLUTION is abstraction();
{
        compose
        { router() and server1() and  server2() };
};
```

l'abstraction qui sera envoyée à l'architecture, puis appliquée par cette dernière

Description architecturale 8 Représentation de la partie qui a évolué

Ainsi, nous avons vu (1) que le système initial était en capacité d'évoluer par l'application d'éléments architecturaux inconnus à l'origine, (2) que l'évolution arrive dynamiquement, en cours d'exécution, et son contenu n'a pas été décidé au préalable.

6. Positionnement

Dans la plupart des approches centrées architectures (Ding et al., 2001, Barais et al., 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 5). Dans deux des cas qui nous ont intéressés, nous avons souhaité effectuer des modifications de façon dynamique, alors que le système est en cours d'exécution. Ces modifications n'entraînent pas l'arrêt du système. La gestion de l'évolution dynamique prévue s'inspire et améliore certaines proposition existantes (Cîmpan et al., 2005b), telles que Dynamic Wright (Allen et al., 1998) ou π-Space (Chaudet et al., 2000)
L'idée même 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 la gestion de la cohérence entre les spécifications et l'exécutable (Cîmpan et al., 2005). On trouve des travaux qui portent sur des transformations de modèles (séparant les préoccupations à intégrer de l'architecture fonctionnelle dans laquelle seront intégrées les préoccupations) par tissage (Barais et al., 2004, Manset et al., 2006), ce qui déjà, témoigne de la possibilité (1) d'exprimer des préoccupations et (2) d'effectuer des opérations de transformation, essentiellement au niveau de la spécification voire même jusqu'au déploiement (Manset et al., 2006) on trouve aussi des travaux traitant du *refactoring* d'architectures – en partant du code et en essayant d'inférer une architecture de plus haut niveau (Ding et al., 2001) enfin beaucoup de propositions de Langages de Description Architecturale (LDA) avec des pouvoirs d'expression très différents entre les LDA proposés (Medvidovic et al., 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 (basés sur UML par exemple), sont des langages extensibles ou non.

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 ArchWare 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) et les outils qui sont sollicités dans ce cadre d'autre part et notamment un support à l'exécution (très peu de LDA possèdent un support à l'exécution, Cîmpan et al., 2005).
En particulier, l'approche ArchWare et son environnement permettent de gérer le découplage et le raffinement du niveau abstrait vers le niveau concret d'une architecture tout autant que de considérer uniquement une description architecturale

exécutable (par interprétation de la description par une machine virtuelle). Bien entendu, des inconvénients (gestion des transformations pour le cas du raffinement, nécessité d'utiliser la machine virtuelle ArchWare pour l'exécution) accompagnent ces deux approches tout autant qu'elles possèdent 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 avec l'utilisation d'une machine virtuelle).
Des scénarios plus complexes (Pourraz et al., 2006) illustrant l'évolution dynamique prévue ou non ont été utilisés et ont permis de valider notre approche.

8. Bibliographie

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 Project IST-2001-32360 Deliverable D3.1b*, 2003.

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

Andrade L.F., Fiadeiro J.L., Architecture Based Evolution of Software Systems. *LNCS 2804*: 148-181 (2003)

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

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.

Blanc dit Jolicoeur L., Dindeleux R., Montaud A., Leymonerie F., Gaspard S., Braesch C., Definition of Architecture Styles and Process Model for Business Case1. *ArchWare Project IST-2001-32360 Deliverable D7.2b*, 2003.

Blanc dit Jolicoeur L., Braesch C., Dindeleux R., *Customised ArchWare Engineering Environment for Business Case 1. ArchWare Project IST-2001-32360 Deliverable D7.3*, 2004.

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.

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

Corradini F., Inverardi P., Wolf A., On relating Functional Specifications to Architectural Specifications : A case Study. *Technical Report CU-CS-933-02, University of Colorado,* juin 2002.

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, *In Proceedings of the 2001 Working IEEE/IFIP Conference on Software Architectures (WICSA-2)*, pages 191-200, Amsterdam, the Netherlands, août 2001.

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.

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 (accepté pour publication).

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.

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.

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

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.

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 (accepté pour publication).

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*, Accepté pour publication.

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.