Un ADL pour les Architectures Distribuées à Composants Hétérogènes

Guillaume Dufrêne*, Lionel Seinturier*

*USTL-LIFL CNRS UMR 8022, équipe projet INRIA ADAM
Parc scientifique de la Haute-Borne
40, Avenue Halley
59650 Villeneuve d'Ascq Cedex - FRANCE
{guillaume.dufrene|lionel.seinturier}@lifl.fr

Résumé. Dans cet article, nous présentons un ADL pour les architectures à composants hétérogènes et distribuées, et son utilisation au moment du déploiement. Actuellement, il n'existe pas de solution générique pour déployer une architecture distribuée basée sur différents intergiciels. Pour répondre à cette problématique, nous proposons dans cet article une approche pour décrire une telle architecture et un support pour le déploiement. Notre solution s'appuie sur un langage de description d'architecture possédant des notions de dépendances verticales et horizontales. Un exemple simple est présenté pour illustrer notre langage et pour valider nos contributions.

1 Introduction

Problématique. Les composants logiciels sont un thème majeur aussi bien dans le monde académique que dans l'industrie informatique. Ils sont le sujet de nombreuses études dans différentes communautés telles que l'ingénierie logicielle ou les intergiciels (middleware). Ces dernières années, de nombreux Canevas de Composants (CC) ont été proposés comme ArchJava (Aldrich et al., 2002), PECOS (Genssler, 2002), K-Component (Dowling et Cahill, 2001), OpenCOM (Clarke et al., 2001), OSGi (OSGi Alliance, 2004) ou Fractal (Bruneton et al., 2006). Une étude plus détaillée montre qu'il existe en fait plus de 20 canevas différents. Certains de ces CC se veulent généralistes alors que d'autres visent des domaines d'application bien précis comme les systèmes temps réels ou les environnements dynamiques. Dans le domaine des intergiciels, la notion de composant logiciel prend une ampleur importante aussi bien dans le monde professionel avec des CC comme EJB (Sun Microsystems, 1997), .NET/COM+ (Lantim, 2003), CCM (Merle et al., 2002), SCA (OSOA, 2007) mais aussi dans le monde académique avec des CC tel que QuA (Eliassen et al., 2004). Les composants se présentent comme une solution bien plus adaptée que les objets, choisis il y a une dizaine d'années, pour implémenter des intergiciels à large échelle et adaptables. Les concepts de haut niveau de ces CC sont similaires. Ils partagent l'idée qu'*un composant logiciel est une unitée de composition possédant des interfaces définies contractuellement et des dépendances au contexte explicites. Un composant logiciel peut être déployé indépendamment et il est sujet à la composition par un*

tiers (Szyperski, 2002). Cependant, la définition détaillée des concepts clés comme les notions de composite, interface, port, service fourni/requis, propriété, gestion des services non fonctionnels diffèrent beaucoup entre les différents CC. Chaque CC implémente ces concepts avec ses propres spécificités pour satisfaire au mieux les besoins du domaine d'application visé ou pour promouvoir un style de programmation particulier. En conséquence, nous pensons qu'il est peu probable qu'un seul CC s'imposera dans les prochaines années comme un standard adapté à tous les usages. Nous pensons plutôt que de multiples CC vont coexister, nécessitant ainsi d'avoir des solutions pour gérer les architectures basées sur des CC différents. Le problème est donc de gérer cette hétérogénéité en terme de CC et de spécifier la composition et la configuration des applications à composants hétérogènes.

Contributions. Les composants sont largement utilisés dans tous les cycles de développement du logiciel depuis l'analyse des besoins, la conception, l'implémentation, le déploiement jusqu'à l'observation et l'administration des applications qui s'exécutent. Supporter des applications développées avec des CC hétérogènes a un impact sur chacune des étapes. Dans cet article, nous proposons les fondations pour un tel support au moment du déploiement, pour les intergiciels et les applications. Le déploiement concerne toutes les activités mises en œuvre lors de l'installation des instances de composant sur les nœuds du système.

La première contribution de notre travail est de résoudre le problème du déploiement de composants hétérogènes et distribués. Pour atteindre cet objectif nous proposons un ADL avec les concepts communs des CC et la notion de *Dépendance Horizontale*.

Nous proposons également une aide pour le déploiement des intergiciels et des librairies systèmes qui sont nécessaires pour lancer ces applications. Nous nous référons à cela sous le terme de *Dépendances Verticales*, décomposées en *Dépendances Descendantes* et *Dépendances Ascendantes*.

Par ces résultats, nous aboutissons à la description d'une achitecture en couches présentant de manière transparente un continuum de concepts et d'outils depuis l'application jusqu'à l'intergiciel.

Plan. Pour refléter nos objectifs, cet article est organisé de la manière suivante. La section 2 présente une application de planification de trajet à travers un réseau de transport public. Nous utilisons cet exemple pour illustrer l'ensemble de notre travail. La section 3 décrit notre ADL. La section 4 explique comment nous supportons un tel langage au moment du déploiement. La section 5 traite des travaux connexes existants. La section 6 conclut et introduit nos travaux à venir.

2 Cas d'exemple : Urban Trip Planner

Nous illustrons nos contributions avec l'application Urban Trip Planner[1] (UTP). Cette application est orientée service et nous avons choisi de la présenter ici avec une approche composant.

[1] Cet exemple est issu du projet RNTL FAROS (ANR, 2006) financé par l'ANR (Agence Nationale de la Recherche Française). Il est défini dans Baligand et al. (2007).

Les fonctionnalités d'UTP. UTP est une application distribuée qui permet à un utilisateur de trouver le parcours le plus rapide entre sa position actuelle et l'adresse de son choix en utilisant un réseau de transport urbain. La partie cliente de cette application est embarquée sur un appareil mobile (ex. SmartPhone, PDA) géo-localisable. La partie serveur renvoie un chemin exprimé sous forme d'une liste d'instructions à suivre qui guide l'utilisateur vers la station la plus proche, lui indique les changements de lignes et le trajet à pied vers sa destination finale. Une carte est également fournie couvrant le secteur entre la dernière station et sa destination.

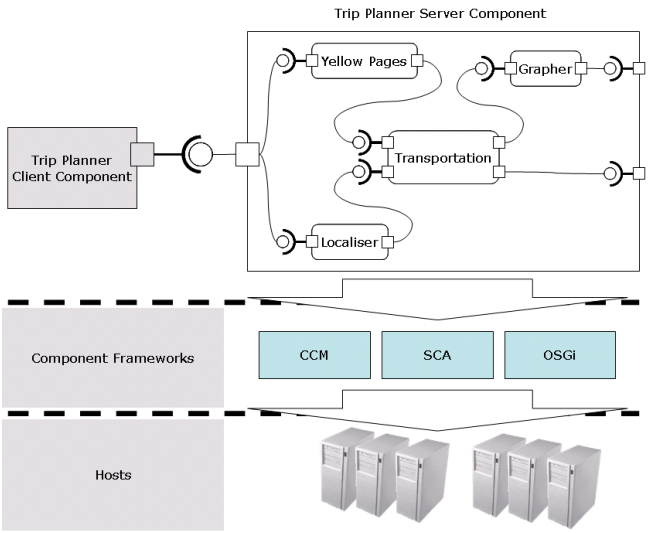

FIG. 1 – *L'architecture générale d'UTP*

Architecture globale. L'architecture globale d'UTP est illustrée sur la figure 1. La description de la partie cliente est en dehors du cadre de cet article. Nous avons représenté ici les différents composants et leurs interactions avec une notation proche de celle fournie par UML 2. Nos travaux souhaitent définir un language plus spécifique pour le domaine des composants.

Le composant de haut niveau `Trip Planner Server Component` est implémenté par quatre sous-composants. Les liens entre ces composants représentent les dépendances entre les données requises et fournies. Le composant `Localiser` fournit une position géodésique de l'appareil utilisé. Le composant `Yellow Pages` est un service de type pages jaunes qui fournit la position géodésique à partir d'une adresse postale. Avec ces deux positions, le composant `Transportation` calcule les stations les plus proches de l'utilisateur et de sa destination. Ensuite il retourne le meilleur chemin parmi les transports en commun. Enfin, le composant `Grapher` fournit une carte de la dernière partie du voyage.

Composants distribués hétérogènes. Les sous-composants d'UTP peuvent être distribués sur différents hôtes et peuvent être implémentés avec des CC différents. Par exemple, considérons que les composants `Localiser` et le `Grapher` sont implémentés avec CCM alors

que les composants `Transportation` et le `Yellow Pages` sont implémentés avec SCA. Considérons en plus que ces composants sont déployés sur des hôtes différents.

Le scénario présenté ici introduit de réelles difficultés. Comme établi dans nos objectifs, nous souhaitons faire face à cette complexité en fournissant un ADL pour décrire des architectures distribuées (voir section 3). En plus, pour alléger le travail de déploiement sur des environnement hétérogènes, nous proposons un outil de déploiement basé sur notre ADL (voir section 4).

3 Description de l'ADL

Pour concevoir un ADL pour les architectures à composants hétérogènes et distribués, il est nécessaire de comprendre les notions générales des CC et les différentes dépendances rencontrées. Nous présentons ensuite une syntaxe textuelle qui définit notre ADL, supportant ces notions.

3.1 Notions générales

Bien que de nombreux ADL existent avec leurs notions spécifiques, des notions de haut niveau peuvent être extraites pour bâtir un ADL plus général. L'idée étant d'obtenir une vue abstraite de l'architecture et de générer la description spécifique qui cible un CC particulier. Notre description d'architecture se veut applicable au déploiement pour les architectures à composants. Donc, nous devons fournir les notions générales des CC et du déploiement.

Les notions principales de CC que nous avons retenue sont : Composant, Interface, Port et Propriété. Un `Composant` est une entité qui fournit des fonctionnalités exposées par des `Ports` et définies par une `Interface`. Un composant peut aussi requérir des fonctionnalités fournies par d'autres composants. Les composants sont configurables avec des propriétés de type "clé-valeur". Nous distinguons deux sortes de composants : les primitifs et les composites. Le composant primitif est généralement associé à l'idée de boite noire : il encapsule un comportement qui est caché de l'extérieur. Au contraire, pour les composites, on souhaite pouvoir décrire leur contenu sous la forme d'un assemblage de composants.

Afin de décrire des configurations qui sont déployables, nous introduisons également les concepts liés au déploiement. Un hôte est un appareil qui exécute le code des fonctionnalités fournies des composants. Cet hôte contient une adresse réseau unique (ex. une adresse IP ou un nom complet).

Dépendances Horizontales. Comme expliqué auparavant, les composants sont liés en connectant les ports des interfaces requises et fournies. Les ports de sortie sont connectés avec des ports d'entrée compatibles. A côté de ces dépendances métiers, des contraintes d'ordre technique sont rencontrées durant le déploiement. Ces dépendances peuvent être considérées comme 'verticales' car elles sont appliquées entre des couches superposées de l'application, de l'intergiciel et des librairies sous-jacentes.

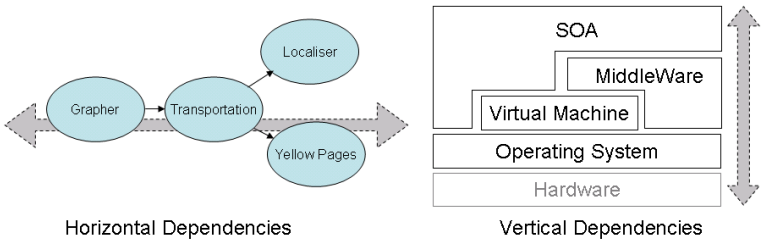

FIG. 2 – *Dépendances Horizontales et Verticales*

Dépendances Verticales. Nous distinguons deux types de dépendances verticales :

- *Dépendances Descendantes*. Chaque CC possède son propre environnement d'exécution et son propre modèle de programmation, plus ou moins spécifique au domaine ciblé. Il existe généralement une forte dépendance entre l'environnement et le CC.
  Déployer un CC est une opération technique principalement réalisée manuellement. Les étapes nécessaires sont souvent décrites dans une documentation d'installation dont l'interprétation peut être source d'erreurs. Dans le meilleur des cas un programme d'installation est fourni mais aucune solution générale n'est proposée pour déployer un CC sur différents hôtes.
- *Dépendances Ascendantes*. Les applications déployées sur un CC ou sur un serveur d'application nécessitent parfois des fonctionnalités particulières. Dans ce cas, une configuration supplémentaire doit être réalisée sur le CC. Par exemple, imaginons que UTP soit déployé sur un système possédant une librairie graphique utilisée par le composant `Grapher`. Si cette librairie est limitée à produire des images de taille 320x240, notre composant doit prendre en compte cette contrainte. Une telle caractéristique est spécifiée comme une propriété associée au composant.

3.2 Syntaxe

La définition de notre ADL est illustrée par une syntaxe du type BNF dans la figure 3. Ce langage est un ensemble de définitions (ligne 1-3) reprenant les notions de Composant (Component), Composite, Cavenas (Framework) et Hôte (Host). Les composants sont typés par le nom des spécifications du CC qu'ils visent. Naturellement, si un composant est implémenté en suivant les spécifications générales d'un CC, il peut être déployé sur n'importe quel CC respectant ces spécifications. Les composites peuvent être hétérogènes au sens où ils peuvent être composés de plusieurs sous-composants implémentés avec différents CC, ils ne sont donc pas typés. Le modèle proposé est hiérarchique, un composite peut lui-même contenir différents composites. Primitifs et composites déclarent tous deux les ports requis et fournis et un ensemble de propriétés sous une forme arborescente. Ces propriétés permettent la configuration automatique des fonctionnalités nécessaires à d'autres composants. Les composites connectent leurs sous-composants en liant les ports requis aux ports fournis par d'autres composants. Certains ports de sous-composants sont promus comme étant ceux du composite. Tous ces composants sont déployés sur un CC compatible, c'est-à-dire avec les mêmes spécifications. Les CC sont déployés avec leurs librairies sur un Hôte choisi.

```
ADL ::= <Definition>*
Definition ::= <PrimitifDef> | <CompositeDef>
             | <FrameworkDef> | <HostDef>
PrimitifDef ::= component { <TypeDef> <ProvidedItfs>?
                 <RequiredItfs>? <Properties>* }
TypeDef ::= type <name>
ProvidedPorts ::= provide { <PortDef> (, <PortDef>)* }
RequiredPorts ::= require { <PortDef> (, <PortDef>)* }
PortDef ::= <Port_Name> : <Interface_Type>
Properties ::= properties { TreeStruct+ }
TreeStruct ::= <nodeName> <nodeValue>
nodeValue  ::= "<CHAR>*" | { TreeStruct* }
CompositeDef ::= composite { <ProvidedItfs>? <RequiredItfs>?
                  <Connection>* <Properties>* }
Connection ::= <PathDef> -> <PathDef>
PathDef ::= <AbsPath> | <RelPath>
AbsPath ::= /<RelPath>
RelPath ::= (../)*<itf_Name><AbsPath>*
FrameworkDef ::= ComponentFramework <name> {
                  <TypeDef> <NeededLibraries>* }
NeededLibraries ::= <name>*
HostDef ::= Host <name> {
             address <HOSTNAME>
             shell <name>
             transfert <name> }
Port_Name, Interface_Type, nodeName, name ::= <ALPHA><ALPHANUM>+
```

FIG. 3 – *Définition de notre ADL sous forme d'une syntaxe de type BNF*

```
Component Transportation {
  type sca
  provide {
    instructions: Details, lastStation:
        Address
  }
  require {
    start: Station, destination: String
  }
  properties {
    dbconf {
      host sql.mydbserver.com
      user lambda
      password xxxxxxx
      dbname mydb1
    }
  }
}

Component Grapher {
  type ccm
  provide {
    map: Map
  }
  require {
    from: Address, to: Address
  }
}

Composite UTP {
  provide {
    instructions: Details, tripMap: Map
  }
  require {
    deviceID: Integer, destination:
        String
  }
  ( . . . )
  /Transportation/instruction ->
      instruction;
  /Transportation/lastStation -> /Grapher
      /from;
  /Grapher/map -> tripMap;
}

ComponentFramework openCCM {
  type ccm
  libraries openccm_api
}

Host gentse {
  address gentse.lifl.fr
  shell ssh
  transfert scp
}
```

FIG. 4 – *Architecture d'UTP décrite avec notre ADL*

Toutes les notions introduites précédemment sont illustrées sur la figure 4 avec une description partielle de l'architecture de l'UTP. Cet extrait de code commence avec deux définitions de composant primitif (ligne 1-17, 19-27). Sont définis ici : le type général (ligne 2), les ports requis et fournis (ligne 3) désignés par leur nom et leur interface. Des propriétés extra-fonctionnelles utilisées pour garantir les dépendances ascendantes peuvent être ajoutées. La ligne 29 définit un composite qui assemble d'autres composants en connectant leurs ports fournis et requis (ligne 37-39). Puis, les CC sont déclarés (ligne 42). Enfin, les Hôtes sur lesquels les CC sont installés sont définis (ligne 47-51).

4 Aide au déploiement

Dans cette section, nous présentons les concepts de haut niveau liés au processus de déploiement en illustrant le déploiement de l'application UTP. D'abord, nous expliquons quelles sont les contraintes d'un déploiement d'une architecture à composants. Ensuite, nous montrons comment mettre en oeuvre un tel déploiement avec un outil existant.

4.1 Le processus de déploiement

Nous avons vu dans la section précédente que chaque partie de l'architecture doit être déployée dans un ordre précis. En effet, les composants sont dépendants les uns des autres et des entités de plus bas niveau. Certaines règles peuvent être identifiées selon chaque type de dépendance.

Pour les dépendances horizontales, un composant qui fournit des fonctionnalités requises par un autre composant doit être installé lorsque ce dernier démarre. Pour les dépendances verticales, les entités de bas niveau doivent être installées avant celles du dessus.

Nous souhaitons déployer des entités de l'architecture quelle que soit leur granularité. Depuis les intergiciels jusqu'aux applications, services ou même objets, chaque élément doit pouvoir être déployé sur une cible en fonction de la description donnée par l'utilisateur.

En environnement distribué, un processus de déploiement se base au moins sur une manière de transférer des données ou du code, et un méchanisme d'exécution distante de code. Par exemple, FTP, HTTP ou SCP peuvent être utilisés comme protocole de transfert, alors que bash, csh ou des scripts windows seraient utilisés comme interpréteur distant. Nous avons donc besoin d'un canevas générique de déploiement facilement extensible qui sera présenté dans la prochaine section.

Dans l'application UTP, une dépendance horizontale impose que le composant `Grapher` soit installé lorsque les composants `Transportation` et `Yellow Pages` sont lancés. Ces composants étant prévu pour les spécifications CCM, un canevas particulier tel que OpenCCM doit être installé avant que les composants ne s'exécutent. Il en sera de même pour les composants `Localiser` et `Grapher` qui nécessitent l'installation d'un canevas implémentant SCA. Toute la configuration de l'application doit être adaptée selon l'ensemble des propriétés contenues au sein de notre description d'architecture.

4.2 Outil pour le déploiement

FDF (Fractal Deployment Framework) présenté dans Flissi et Merle (2006), aussi connu sous le nom de Deployware, est un canevas de déploiement générique. Ce travail de recherche récent abstrait chaque notion du processus de déploiement sous la forme d'un composant et fournit une manière d'exécuter le déploiement. FDF est basé sur le modèle à composant Fractal (Bruneton et al., 2006). Le processus de déploiement peut être spécialisé en étendant ces composants abstraits nommés `Personnalités`. Une personnalité implémente toute les fonctions nécessaire à l'installation, le démarrage, l'arrêt et la désinstallation de l'application. Cette solution permet d'aboutir à une abstraction de haut niveau cachant au développeur toute la complexité de l'orchestration du déploiement prise en charge par le canevas.

```
1  Port(itf_type, content) = { ... }
2  ComponentFramework( cftype, runtime ) = software.Hosting( cftype ) {
3    dependsOn {
4      # add libraries dependencies here
5    }
6  }
7  Component( cftype ) = software.Hosted( cftype ) {
8      provide {  }
9      require {  }
10     properties {  }
11     # ...
12 }
```

FIG. 5 – *Déclaration partielle de personnalités FDF*

Nous proposons un ensemble de personnalités FDF permettant de déployer les architectures décrites avec notre ADL. La Figure 5 présente par exemple la définition partielle des personnalités liés aux notions de Port, de Composant et de Canevas.

Nous utilisons le méchanisme fourni par FDF pour définir toutes les dépendances techniques entre chaque couche applicative. Comme présenté dans la Figure 5, une dépendance entre le CC et le composant est exprimée par le paramètre `cftype` qui est le nom de la spécification du CC.

L'étape de configuration d'un CC doit prendre en compte les propriétés contenues à l'intérieur de la définition du composant pour qu'il soit adapté à son exécution. Il n'y a bien sûr pas de magie, les informations sont présentes mais des mécanismes précis d'adaptation du CC doivent être mis en œuvre. Les liaisons entre les composants distribués doivent être configurées selon les protocoles de communication offerts par chacun des CC. Si un support donné n'existe pas pour un CC particulier, alors un médiateur ou un adaptateur peut être généré au moment du déploiement. Celà peut se faire à l'aide de la programmation générative à partir du code compilé ou du code source de l'application à déployer. Dans le monde Java, différents outils tels que spoon (Pawlak, 2006) ou ASM (Bruneton et al., 2002) permettent ce genre de manipulations. Une autre alternative serait une approche avec de la Programmation par aspect appliquée à une achitecture à composant afin de greffer le code d'adaptation dans l'architecture d'origine. Ce type d'approche a par exemple été étudiée dans FAC (Pessemier et al., 2006).

Les dépendances horizontales sont réalisées par les définitions de `provide` et `require`. Elles contiennent la liste des ports requis ou fournis associés à une `Interface` contractualisant les opérations et la structure des données échangées.

Nous avons donc vu comment il était possible de construire un langage d'architecture prennant en considération les problématiques de distribution et d'hétérogénéité des technologies dans le domaine des applications à base de composants. Nous avons également vu pourquoi il était important de savoir décrire à la fois les dépendances horizontales d'ordre métier, et les dépendances verticales d'ordre technique. Cette dernière section a montré qu'avec notre langage, il est maintenant possible de décrire simplement une telle architecture et de la déployer facilement.

5 Travaux similaires

Nous pouvons regrouper les travaux similaires en deux domaines. D'abord, les propositions d'ADL extensible ou les description d'architectures hétérogènes. Ensuite, les solutions proposées pour déployer des architectures distribuées.

ADL

- Une chaine estensible est présentée dans Leclercq et al. (2007) pour faciliter le déploiement des architectures à base de composants. Elle n'est pas liée à un ADL spécifique, elle utilise un méchanisme de description de grammaires pour accepter des langages variés en entrée. Le cœur de cette chaîne est un assemblage de composants à faible granularité qui implémentent un patron de conception flexible et facilement extensible pour répondre à de nouveaux besoins. Notre approche est plus spécifique car elle est basé sur un ADL particulier mais elle prend en compte la description des intergiciels et des contraintes systèmes rencontrées lors du déploiement.
- Certains ADL comme Wright (Douence et al., 1997) sont plus ou moins spécifiques à la conception ou à l'analyse d'architectures. D'autres comme Darwin (Magee et al., 1995), Olan (Balter et al., 1998) ou ArchJava (Aldrich et al., 2002) sont liés à la description de l'implémentation d'une architecture. ArchJava est une approche interressante qui étend le langage de programmation JAVA avec les notions de description d'architecture. Notre ADL ne se veut pas spécifique à une étape du développement logiciel et n'est pas intégré à un quelconque langage de programmation.

Déploiement

- JSR88 (JSR 88, 2002) définit une librairie de programmation standard pour permettre un déploiement des composants J2EE (Java 2 Platform, Enterprise Edition) ainsi que le développement d'outils de déploiement indépendant de la plateforme d'exécution. Le serveur applicatif de Sun en version 9 (Sun Microsystems, 2006) implémente cette librairie. Les applications sont déployées sous la forme d'une archive contenant tout le code exécutable, les autres ressources, les fichiers de configuration de l'application et un fichier `manifest` utilisé pour le déploiement. Le processus de déploiement est assez monolithique et les adaptations sont assez limitées et restreintes à l'environnement JAVA. De plus, il n'y a pas la possibilité de déployer le serveur d'application lui-même. Notre proposition est extensible, n'est pas liée à JAVA, et permet de déployer le canevas lui-même.
- OSGi (OSGi Alliance, 2004) est un sytème de modules dynamiques pour JAVA, basé sur des composants réutilisables qui peuvent être déployés et changés dynamiquement, sans

les redémarrer. Les composants sont conditionnés dans des `bundles` qui sont connectés les uns aux autres par les paquetages déclarés comme étant exportés ou importés. Les bundles peuvent enregistrer des services dans l'environnement qui seront utilisés par d'autres. Ce type de déploiement reste toutefois homogène puisqu'il repose sur la seule technologie JAVA. D'autre part, les bundles sont définis de manière individuel par des fichiers manifest et OSGi ne propose pas d'ADL pour décrire d'architecture globale d'une application. Notre approche est plus structurée car elle repose sur un ADL, et nous offrons la possibilité de manipuler des entités hétérogènes.

– Le projet Apache Tuscany (Tuscany team, 2007) propose une implémentation des spécifications SCA (OSOA, 2007). Un élément intéressant de Tuscany est le fait que les composants peuvent être implémentés dans différents langages et qu'ils supportent différents protocoles de communication. SCDL est le language d'architecture dédié à SCA pour définir tous les composants et leurs interactions. Cette description est limitée au contexte d'un nœud de déploiement et une fois encore, il n'est pas possible de décrire une architecture distribuée dans son intégralité. Tuscany ne propose pas de système de déploiement distant ni d'outils pour se déployer lui-même à large échelle contrairement à ce que nous proposons.
– L'OMG a défini D&C (OMG, 2004) qui sont des spécifications pour le déploiement et la configuration d'applications à base de composants sur les environnement hétérogènes et distribués. Ces spécifications se concentrent sur le processus de déploiement et sont fortement liées au modèle CORBA. Notre travail n'est pas lié à une technologie particulière. Et bien que notre langage soit utilisé pour l'instant au moment du déploiement, cette description d'architecture couvre une problématique plus générale qui pourra couvrir l'ensemble du cycle de vie du logiciel.

6 Conclusion et travail à venir

Dans cet article, nous avons présenté un ADL pour les architectures hétérogènes et distribuées. Nous avons introduit les notions de dépendances horizontales et verticales et montré en quoi cela peut être utile pour le déploiement. L'application orientée service UTP a permis d'illustrer nos contributions.

En guise de perspective, bien que nous nous sommes interressés ici au déploiement, notre travail peut apporter des contributions sur toutes les étapes du développement logiciel :

– Pour la conception, notre langage peut devenir un pivot pour la description des architectures et une manière d'exprimer à haut niveau les dépendances extra-fonctionnelles
– Lors de l'implémentation, nous pouvons fournir des outils générant du code à partir de la description.
– Pour les tests unitaires et d'intégration, nous pouvons vérifier différentes propriétés structurelles de l'architecture.
– Pour le déploiement, nous organisons le processus d'orchestration pour garantir l'ordre de déploiement des différents éléments et la configuration de l'application.

Par ailleurs, nous pensons aussi proposer un système de déploiement global multi-couches. L'objectif est d'obtenir une manière de déployer tous les éléments depuis le système d'exploitation jusqu'aux applications et d'offrir un mécanisme d'adaptation au contexte. Nous étudions également la possibilité de déployer des aspects durant l'exécution des applications.

Remerciements. Nous souhaitons remercier Fabien Baligand, auteur du scénario UTP, ainsi que Franck Chauvel pour nous avoir fourni un diagramme de collaboration pour ce scénario.

Références

Aldrich, J., C. Chambers, et D. Notkin (2002). ArchJava : Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pp. 187–197. ACM Press.

ANR (2006). *FAROS : Fiabilité d'ARchitectures Orientés Services*. ANR. EDF, France Telecom, IRISA, I3S, LIFL. http ://www.lifl.fr/faros/.

Baligand, F., T. Ledoux, et P. Combes (2007). Une approche pour garantir la qualité de service dans les orchestrations de services web. In *NOTERE 2007*, Marrakech, Maroc.

Balter, R., F. Boyer, M. Riveill, et J.-Y. Vion-Dury (1998). Architecturing and Configuring Distributed Applications with Olan. In *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lake District, pp. 241–256. IFIP.

Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma, et J.-B. Stefani (2006). The fractal component model and its support in java. *Software Practice and Experiences (SPE) 36*(11-12), 1257–1284.

Bruneton, E., R. Lenglet, et T. Coupaye (2002). Asm : a code manipulation tool to implement adaptable systems.

Clarke, M., G. Blair, G. Coulson, et N. Parlavantzas (2001). An efficient component model for the construction of adaptive middleware. In *Proceedings of Middleware'01*, Heidelberg, Germany, pp. 160–178. Springer-Verlag.

Douence, R., R. Allen, et D. Garlan (1997). Specifying dynamism in software architectures. In *Proceedings of the First Workshop on the Foundations of Component-Based Systems*, pp. 11–22.

Dowling, J. et V. Cahill (2001). The K-Component architecture meta-model for self-adaptative software'. In *Proceedings of Reflection'01*, Volume 2192 of *Lecture Notes in Computer Science*, pp. 81–88. Springer-Verlag.

Eliassen, F., R. Staehli, G. Blair, et J. Øyvind Aagedal (2004). Qua : Building with reusable qos-aware components. In *OOPSLA'04*, Vancouver, British Columbia, Canada, pp. 154–155. ACM.

Flissi, A. et P. Merle (2006). A generic deployment framework for grid computing and distributed applications. In *Proceedings of the 2nd International OTM Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06)*, Volume 4279 of *Lecture Notes in Computer Science*, Montpellier, France, pp. 1402–1411. Springer-Verlag.

Genssler, T. (2002). Pecos in a nutshell. http ://www.pecos-project.org/.

JSR 88 (2002). *Java EE Application Deployment.* JSR 88. jcp.org/en/jsr/detail ?id=88.

Lantim, D. (2003). *.NET*. Eyrolles.

Leclercq, M., A. E. Ozcan, V. Quéma, et J.-B. Stefani (2007). Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE'07 research papers*, Minneapolis, USA, pp. 209–219. IEEE Computer Society.

Magee, J., N. Dulay, S. Eisenbach, et J. Kramer (1995). Specifying Distributed Software Architectures. In *Proc. 5th European Software Engineering Conf. (ESEC 95)*, Volume 989, pp. 137–153. Springer-Verlag.

Merle, P., D. Sevilla Ruiz, H. Böhme, S. Leblanc, M. Vadet, T. Ritter, et J. Scott Evans (2002). *CORBA Component Model Tutorial*. OMG. Document ccm/02-06-01. `www.omg.org/cgi-bin/doc?ccm/2002-06-01`.

OMG (2004). *Deployment and Configuration of Component-based Distributed Applications Specification*. OMG. Document ptc/04-08-02. `www.omg.org`.

OSGi Alliance (2004). *OSGi Technical Whitepaper*. OSGi Alliance. Revision 3.0. www.osgi.org.

OSOA (2007). *SCA Service Component Architecture - Assembly Model Specification*. OSOA. Version 1.00.

Pawlak, R. (2006). Spoon : Compile-time annotation processing for middleware. In *Distributed Systems Online*, Volume 7. IEEE.

Pessemier, N., L. Seinturier, T. Coupaye, et L. Duchien (2006). A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, Volume 4089 of *Lecture Notes in Computer Science*, Vienna, Austria, pp. 259–273. Springer-Verlag.

Sun Microsystems (1997). *Enterprise Java Beans*. Sun Microsystems. www.javasoft.com/products/ejb.

Sun Microsystems (2006). *JAVA System Application Server*. Sun Microsystems.

Szyperski, C. (2002). *Component Software - Beyond Object-Oriented Programming* (2nd ed.). Addison-Wesley.

Tuscany team (2007). *Apache Tuscany SCA v0.9*. Tuscany team. http ://incubator.apache.org/tuscany.

Summary

In this paper, we present an ADL for heterogeneous and distributed component-based applications and its application at deployment time. There is no general solution to deploy a distributed hererogeneous application which use different middleware. This kind of task can be difficult because each part of the application is defined in its own middleware context and no general view is provided. To solve this problem, we propose in this paper an approach to describe such an architecture and a support at deployment-time. Our solution is based on an Architecture Description Language with notions of vertical and horizontal dependencies. A simple example is presented to illustrate our ADL and to show how its features allow to validate our contribution.