Dedal : un ADL à trois dimensions pour gérer l'évolution des architectures à base de composants

Huaxi Yulin Zhang, Christelle Urtado, Sylvain Vauttier

Dedal : un ADL à trois dimensions pour gérer l'évolution des architectures à base de composants

Huaxi (Yulin) Zhang, Christelle Urtado, Sylvain Vauttier

LGI2P / Ecole des Mines d'Alès,
Parc Scientifique G. Besse, F30035 Nîmes Cedex, France
{Huaxi.Zhang, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

Résumé. Une architecture logicielle peut être définie à différents niveaux d'abstraction, correspondants aux différentes étapes de son processus de développement : spécification, implémentation et déploiement. La cohérence entre les différentes définitions d'une architecture doit être maintenue : sa définition à un niveau d'abstraction doit être conforme à sa définition au niveau d'abstraction immédiatement supérieur. Ce principe permet de contrôler l'évolution d'une architecture, en validant les modifications réalisées à un certain niveau d'abstraction ou en motivant la création d'une nouvelle version pour propager les modifications entre niveaux d'abstraction. Malheureusement, aucun ADL ne propose un modèle de définition d'architectures séparant clairement les niveaux d'abstraction couvrant le cycle de vie d'une architecture. Cet article présente Dedal, un ADL permettant une définition séparée de la spécification, de la configuration et de l'assemblage d'une architecture afin de prévenir l'érosion ou la dérive qui surviennent lors des évolutions entre les différents niveaux de définitions des architectures.

1 Introduction

Le développement à base de composants est la principale réponse proposée à la problématique de l'accroissement de la complexité des logiciels, afin de diminuer les temps et les coûts de développement sans sacrifier la qualité des logiciels produits. Il propose une démarche de construction des logiciels à large maille, par assemblage de blocs de programmation préexistants – les composants – définis de manière suffisamment indépendante et découplée afin d'être réutilisables dans de multiples contextes (Sommerville, 2006; Crnkovic et al., 2006). La définition d'un logiciel ainsi produit est décrite sous la forme d'une architecture logicielle, listant composants utilisés et les connections devant les relier. Pour assurer la gestion d'un logiciel tout au long de son cycle de vie, depuis sa spécification jusqu'à son déploiement puis son exploitation, la définition d'une architecture logicielle doit rassembler des méta-informations correspondant à différents niveaux d'abstraction. La cohérence de la définition d'une architecture repose sur la conformance de sa description à un niveau d'abstraction donné avec sa description au niveau d'abstraction immédiatement supérieur. Ce principe permet de proposer des processus contrôlés de développement et d'évolution des architectures. Malheureusement,

aucun ADL ne propose un modèle de définition d'architectures couvrant le cycle de vie d'une architecture. La plupart des ADL, tels que Wright (Allen et Garlan, 1997), C2 (Medvidovic et al., 1999) et Darwin (Magee et Kramer, 1996), ne couvre que les étapes de spécification et d'implantation, négligeant la description du déploiement des architectures (assemblages d'instances des composants). Par ailleurs, les différents niveaux d'abstraction composant la définition d'une architecture ne sont pas explicitement identifiés et séparés. Il est alors difficile d'utiliser les définitions des architectures comme un support efficace aux processus de développement et d'évolution des applications.

Cet article présente Dedal, un ADL permettant de représenter explicitement trois niveaux d'abstraction dans la définition d'une architecture. Ces trois niveaux – la spécification, la configuration et l'assemblage d'une architecture – correspondent aux étapes de conception, d'implémentation et de déploiement d'une architecture. L'article décrit comment Dedal permet de conserver la trace des décisions des architectes dans un processus de développement (forward engineering). Il décrit également comment Dedal peut servir de support à un processus d'évolution contrôlée (reverse engineering), en identifiant clairement la portée d'une modification et les mécanismes de propagation de ses conséquences à l'ensemble de la définition d'une architecture. Ces mécanismes ont pour objectif d'éviter les problèmes d'érosion et de dérive affectant les définitions des architectures (Perry et Wolf, 1992).

La suite de cet article est organisée de la manière suivante. La section 2 compare l'expressivité des ADL existants. La section 3 décrit le modèle de définition d'architectures proposé dans Dedal. La section 4 définit le processus d'évolution contrôlée pouvant être mis en œuvre à l'aide de Dedal. La section 5 introduit des perspectives à ce travail.

2 Expressivité des ADL existants

La définition d'une architecture synthétise les décisions prises lors du processus de conception d'un logiciel (Taylor et al., 2009). Elle est exprimée à l'aide d'un ADL qui permet de décrire la structure d'un logiciel en listant les composants qui le constituent et les connexions qui relient leurs interfaces. Des attributs de qualité sont parfois proposés pour décrire les aspects non fonctionnels du logiciel (*e.g.* xADL (Dashofy et al., 2002)). Le comportement dynamique est souvent également décrit (*e.g.* C2 (Medvidovic et al., 1999), Wright (Allen et Garlan, 1997), SOFA (Plásil et Visnovsky, 2002)) à l'aide de différents moyens (*e.g.* message-based communication, CSPs, expressions régulières). Les ADL décrivent en général de manière distincte la spécification et la configuration d'une architecture. La spécification définit l'ensemble des classes de composants et de connecteurs qui composent une architecture (un type de logiciel). Parfois, les types des connecteurs ne sont pas explicitement définis mais déduits des types des interfaces connectées (*e.g.* Darwin (Magee et Kramer, 1996)). Les types et les classes de composants utilisés sont définis dans une même définition (*e.g.* Wright (Allen et Garlan, 1997)) ou dans des définitions séparées (*e.g.* C2 (Medvidovic et al., 1999)), ce qui facilite leur réutilisation et leur évolution. La configuration d'une architecture (une instance exécutable d'un logiciel) définit les instances de composants et de connecteurs qui doivent être crées lors du déploiement du logiciel.

Quand un logiciel est trop complexe pour être facilement décrit, deux mécanismes sont utilisés pour diviser la définition de son architecture en modèles plus simples. La composition hiérarchique permet de représenter le logiciel à différents niveaux de granularité (*e.g.* Dar-

win (Magee et Kramer, 1996), SOFA (Plásil et Visnovsky, 2002) ou Fractal ADL (Bruneton et al., 2006)). L'architecture d'un logiciel est définie par ensemble de composants abstraits, de forte granularité, représentant chacun un sous-système du logiciel. La définition de chaque composant peut être détaillée sous la forme d'une architecture qui définit la structure du sous-système qu'il représente. Cette décomposition hiérarchique est itérée jusqu'à la définition de composants élémentaires. La définition de l'architecture du logiciel est obtenue par la composition des architectures de ses sous-systèmes. La décomposition en différents points de vue (*e.g.* vue statique et dynamique en UML (Booch et al., 2005)) fait coexister plusieurs définitions d'une architecture, afin d'isoler et détailler des descriptions spécifiques. La définition globale de l'architecture est obtenue par une synthèse de ces points de vue juxtaposés.

Les architectures devraient pouvoir être décrites à différents stades de leur conception. Certains travaux, UML (Booch et al., 2005) ou Taylor et al. (2009), décrivent des concepts proches. UML fournit une large palette de modèles (de stucture, de collaboration, de comportement, de déploiement, ...) couvrant l'ensemble du cycle de développement d'un logiciel mais ces modèles, orientés objets, n'ont pas encore été transposés au modèle orienté composants proposé dans UML2.0. Taylor et al. (2009) distinguent deux niveaux de description, utilisés lors de la conception et du codage, appelés respectivement l'architecture perspective (as-intended) et l'architecture descriptive (as-realized). Garlan et al. (2001) définissent une infrastructure à trois niveaux (tâche, modèle et exécution) et mettent en évidence l'utilité d'une telle infrastructure dans la gestion de l'évolution dynamique des logiciels. Cependant, aucun de ces travaux ne propose d'ADL permettant de décrire ces différents niveaux de description, adaptés au support des différentes étapes du cycle de vie d'une architecture. Dedal remédie à ce problème.

3 Dedal, un ADL à trois dimensions

Cette section présente la première partie de notre proposition : Dedal, un langage de description d'architecture qui permet la définition d'une architecture en trois modèles distincts ou dimensions. La figure 1 décrit l'architecture d'un système de location de vélos (Bike Rental System ou BRS) qui servira d'exemple dans cet article. Le composant *BikerGUI* gère l'interface utilisateur. Il collabore avec le composant *Session* qui gère l'exécution des commandes de l'utilisateur. Il collabore avec les composants *Account* et *BikeCourse* qui sont chargés d'identifier l'utilisateur, de vérifier le solde de son compte, de lui affecter un vélo disponible et de calculer le prix du trajet effectué lorsque le vélo est rendu. Les composants *BikeCourse* et *BikeCourseDB* serviront d'exemple, dans la section suivante, pour présenter les concepts et la syntaxe de Dedal.

3.1 Spécification abstraite d'une architecture

La spécification (abstraite) d'une architecture est la première dimension de la description d'une architecture. Elle permet à un architecte de définir une architecture idéale capable de satisfaire les exigences du cahier des charges. Une spécification ne décrit pas les composants et les connexions de l'architecture en termes de types concrets, ni même de définition complètes de types abstraits. A l'instar du concept de rôle utilisé dans la définition des collaborations en UML (Booch et al., 2005), nous proposons de définir la spécification d'une architecture par

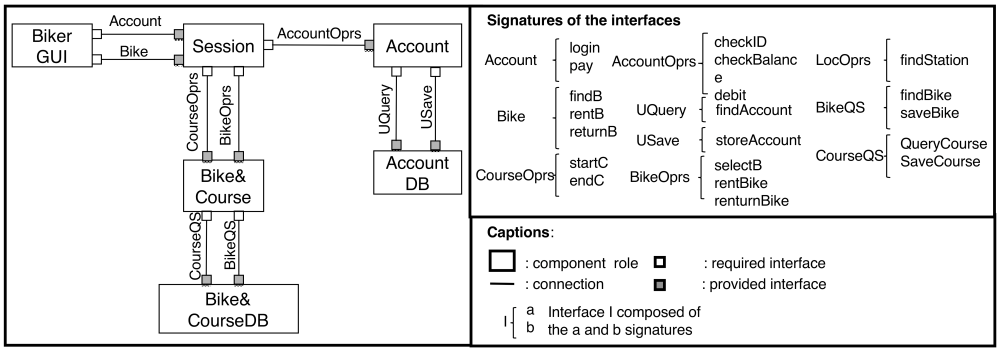

FIG. 1 – *Architecture du système B*RS

la description de l'ensemble des rôles joués par les composants qui participent à la réalisation de l'architecture. La définition concrète d'une architecture peut alors être dérivée de sa spécification en sélectionnant des classes de composants capables de remplir les rôles définis. La spécification d'une architecture peut ainsi être comparée aux architectures perspectives introduites par Taylor et al. (2009). La spécification d'une architecture contient la définition de sa structure (ensemble de rôles, de connections) et de son comportement (protocoles). Elle contient également des informations sur le versionnement des différents éléments qui la composent (voir la section 4). La figure 2 fournit un extrait d'une spécification d'architecture avec Dedal.

```
specification BRSSpec
 component_roles
  BikeCourse; BikeCourseDB
  ...
 connections
  connection connection1
   client BikeCourse.BikeQS
   server BikeCourseDB.BikeQS
  connection connection2
   client BikeCourse.CourseQS
   server BikeCourseDB.CourseQS
  ...
 architecture_behavior
  (?BikeCourse.BikeOprs.selectBike;
  !BikeCourse.BikeQS.findBike;
  ?BikeCourseDB.BikeQS.findBike;)
  +
  (?BikeCourse.CourseOprs.startC;
  !BikeCourse.CourseQS.findCourse;
  ?BikeCourseDB.CourseQS.saveCourse;)
  ...
 version 1.0
```

FIG. 2 : *Extrait de la spécification de l'architecture du système BRS*

Les rôles. Dans une spécification d'architecture, un rôle définit les responsabilités d'un composant, c'est-à-dire le comportement individuel qui est attendu du composant pour participer à une exécution correcte du comportement d'ensemble de l'architecture. Un rôle est décrit comme un type abstrait. Il définit la liste des interfaces et le protocole de comportement qu'un composant doit posséder pour pouvoir remplir le rôle attendu dans l'architecture. Dedal utilise la syntaxe proposée dans SOFA (Plásil et Visnovsky, 2002) pour décrire les comportements sous la forme d'expressions régulières[1]. Le protocole de comportement décrit les appels de méthodes que le composant doit recevoir et envoyer au travers de ses interfaces. Le type abstrait défini par un rôle n'est pas la spécification complète d'un composant mais décrit l'une de ses utilisations particulières (un ensemble de collaborations). Toutes les classes de composants conformes à la définition d'un rôle peuvent être utilisées dans l'implantation concrète de l'architecture. Un rôle peut correspondre à des fonctionnalités récurrentes (login, panier d'achats, etc.) et être utilisable dans la définition de différentes architectures. Les rôles sont ainsi définis indépendamment des spécifications d'architectures. Les figures 3 et 4 présentent respectivement les rôles *BikeCourse* et *BikeCourseDB*.

Les connections. Les connections définissent comment les interfaces des composants sont reliées pour assurer le fonctionnement de l'architecture. La figure 2 définit ainsi les connections nécessaires à l'utilisation des rôles *BikeCourse* et *BikeCourseDB* dans l'architecture du BRS.

Le comportement d'architecture. Le comportement d'architecture définit, sous la forme d'un protocole de comportement, comment sont combinés les comportements des composants qui la composent. Le protocole de comportement de l'architecture doit être conforme aux connections spécifiées et aux comportements individuels des composants définis dans les rôles utilisés. Cette conformité peut être vérifiée par un calcul d'inclusion de traces d'exécution comme proposé dans SOFA (Plásil et Visnovsky, 2002). La figure 2 propose une définition du comportement de l'architecture BRS. Il est intuitivement possible de vérifier sur cet exemple simple que le comportement défini pour l'architecture combine de manière cohérente le comportement défini dans les rôles *BikeCourse* et *BikeCourseDB*.

```
component_role BikeCourse
 required_interfaces BikeQS; CourseQS
 provided_interfaces BikeOprs; CourseOprs
 component_behavior
  (?BikeOprs.selectBike, !BikeQS.findBike;)
  +
  (?CourseOprs.startC, !CourseQS.findCourse;)
```

FIG. 3 : *Spécification du rôle de composant BikeCourse*

[1] `!i.m` et `?i.m` décrivent l'envoi et la réception d'un appel de méthode, respectivement. `A+B` signifie `A` ou `B` (alternative) et `A;B` signifie `A` suivi de `B` (séquence).

```
component_role BikeCourseDB
 provided_interfaces BikeQS; CourseQS
 component_behavior
  ?BikeCourseDB.BikeQS.findBike; + ?BikeCourseDB.BikeOprs.findCourse;
```

FIG. 4 : *Spécification du rôle de composant BikeCourseDB*

3.2 Configuration concrète d'une architecture

La configuration (concrète) d'une architecture est la deuxième dimension de sa définition. Elle a pour objectif de définir les classes de composants et de connecteurs utilisées pour implanter l'architecture. Elle correspond aux architectures descriptives proposées par Taylor et al. (2009). La configuration d'une architecture est dérivée de sa spécification en utilisant la description des rôles comme critère de recherche et de sélection dans une bibliothèque de composants (Arévalo et al., 2009). Les types des classes de composants sélectionnées doivent correspondre aux types de leur(s) rôle(s) dans l'architecture. Cette correspondance n'est pas exacte ou stricte mais repose sur une notion de compatibilité entre types de composants : le type d'une classe de composants peut se substituer au type de son rôle dans la spécification (Desnos et al., 2008). La configuration d'une architecture, par la sélection d'un ensemble de classes de composants, définit une implantation particulière de l'architecture. Différentes configurations, correspondant à différentes implantations d'une architecture, peuvent être définies pour une même spécification, afin d'obtenir des qualités non fonctionnelles spécifiques (ligne de produit) ou pour tracer les évolutions des classes de composants (versionnement). La définition d'une configuration contient des informations permettant de tracer la définition et les évolutions des différentes versions d'une configuration. La figure 5 donne un exemple de configuration correspondant à la spécification présentée figure 2. Elle définit l'utilisation des classes de composants *BikeTrip* et *BikeTripDB* dans les rôles *BikeCourse* et *BikeCourseDB*.

```
configuration BRSConfig
 implements BRSSpec (1.0)
 component_classes
  BikeTrip (1.0) as BikeCourse;
  BikeTripDBClass (1.0) as BikeCourseDB
 version 1.0
```

FIG. 5 : *Exemple de configuration d'architecture pour le système BRS*

Les classes de composants. Dedal permet de décrire les classes de composants qui sont utilisées dans les configurations d'architecture. La description d'une classe de composants est définie comme un type abstrait (méta-modèle) qui documente ses fonctionnalités et son comportement. On distingue les classes de composants primitifs et composites. Une classe de composants primitifs (*e.g. BikeTrip* présentée dans la figure 6) est définie par l'ensemble de ses interfaces, son protocole de comportement, sa version et la classe qui contient son implémentation (code exécutable stocké dans la bibliothèque de composants). Les classes de com-

posants composites se distinguent des classes de composants primitifs en étant implantées par un ensemble de composants internes. L'organisation des composants internes est définie par une configuration d'architecture décrivant la structure interne du composant composite. Il est possible de définir, tant pour les classes de composants primitifs que composites, des attributs permettant de paramétrer et de gérer leur instanciation lors du déploiement des architectures. De même, la version d'une classe peut être définie par son numéro de révision et la motivation de sa création (évolution corrective ou perfective). Les motivations sont utilisées pour gérer l'évolution graduelle des architectures par substitution de composants versionnés (Zhang et al., 2009). Ainsi, contrairement aux ADL existants, il est possible de représenter, sous la forme de versions multiples, les différentes implantations existantes d'une classe de composants.

```
component_class BikeTrip
 implements BikeCourse
 using fr.ema.BikeTripImpl
 required_interfaces BikeQS; CourseQS; LocOprs
 provided_interfaces BikeOprs; CourseOprs
 component_behavior
  (?BikeOprs.selectBike;
!LocOprs.findStation;
!BikeQS.findBike;)
  +
  (?CourseOprs.startC,
!CourseQS.findCourse;)
 version 1.0
 attributes company; currency
```

FIG. 6 : *Définition de la classe du composant primitif BikeTrip*

Les classes de connecteurs. Il est possible de définir explicitement le type de connecteur utilisé pour réaliser une connexion dans l'architecture. Par défaut, les connecteurs sont considérés comme des entités génériques administrées et fournis par les conteneurs de composants (environnements d'exécution) dans lesquels les architectures sont déployées. La définition explicite d'une classe de composants permet à l'architecte de préconiser l'utilisation de mécanismes de communication, de coordination et de médiation spécifiques, pour répondre par exemple à des besoins d'adaptation (de types ou de protocoles).

3.3 Assemblage d'une architecture

L'assemblage d'une architecture décrit l'ensemble des instances, composants et connecteurs, qui la composent. C'est la troisième dimension de la description d'une architecture, correspondant à son déploiement et à son exécution. Elle permet de décrire des paramétrages spécifiques des états des composants (attributs), correspondant à différentes utilisations de l'architecture. Ces paramétrages sont exprimés par des affectations de valeurs ou des contraintes. Différents assemblages peuvent être définis pour une même configuration, permettant de répertorier les décisions de conception correspondant à ces différents contextes d'utilisation (Shaw et Garlan, 1996). Des informations de versionnement permettent de représenter les multiples

versions d'un assemblage et de tracer leur évolution. La figure 7 propose la définition d'un assemblage qui instancie la configuration présentée sur la figure 5.

```
assembly BRSAss
 instance_of BRSConfig (1.0)
 component_instances
  BikeTripC1 as BikeCourse;
  BikeTripDBC1 as BikeCourseDB;
 assembly_constraints
  BikeCourse.currency="Euro";
  BikeCourseDB.company=BikeCourse.company
 version 1.0
  component_instance BikeTripC1
   instance_of BikeTrip (1.0)
  component_instance BikeTripDBC1
   instance_of BikeTripDBClass (1.0)
```

FIG. 7 : *Assemblage de composants pour le BRS*

Les composants instances. La description d'un composant instance définit son nom, sa classe et les valeurs spécifiques affectées à ses attributs. Les noms des composants instances sont utilisés dans la définition des assemblages pour identifier leurs rôles. La structure d'un assemblage n'est pas nécessairement isomorphe à celle de la configuration associée, un composant instance pouvant y jouer plusieurs rôles (décision de conception). La définition des composants instances est extérieure et indépendante de la définition des assemblages. Il est ainsi possible de définir des bibliothèques d'instances de composants réutilisables, correspondant par exemple à des préférences ou à des collections de données métier.

Les contraintes d'assemblage. Les contraintes d'assemblage définissent les conditions qui doivent être vérifiées par les attributs des composants pour respecter la cohérence d'une architecture. Les contraintes sont exprimées sur les rôles et sont appliquées sur les composants associés à ces rôles. Les contraintes définissent ainsi de manière explicite des conditions génériques qui sont respectées dans toutes les instanciations d'une architecture. Pour l'instant, les contraintes proposées en Dedal définissent les valeurs imposées à certains attributs, des conditions entre attributs et des contraintes de cardinalités pour les connexions sur les interfaces. Des exemples de contraintes sont proposées sur la figure 7. La valeur de l'attribut *currency* du composant jouant le rôle *BikeCourse* est fixée à la valeur *Euro*. De même, les valeurs de l'attribut *company* des composants jouant les rôles *BikeCourse* et *BikeCourseDB* doivent être égales. Cette dernière contrainte impose que le composant *BikeCourse*, qui implante la logique métier, utilise un composant d'accès aux données métier *BikeCourseDB* de la même société. Les types de contraintes proposés sont très simples et ne permettent pas l'expression de schéma complexes tels que des alternatives ou des négations, ni la propagation de contraintes ou la résolution d'éventuels conflits. La définition de nouveaux types de contraintes, en s'inspirant de l'expression de styles architecturaux à l'aide de contraintes (Allen et Garlan, 1997; Cheng et al., 2002), est l'une des perspectives de ce travail.

Cohérence entre la configuration et l'assemblage d'une architecture. La vérification de la cohérence entre la définition d'une architecture au niveau configuration et au niveau assemblage est directe. Un composant désigné dans l'assemblage pour jouer un rôle dans l'architecture doit être une instance de la classe de composants désignée dans la configuration pour implanter ce rôle. Les valeurs des attributs des composants doivent vérifier les contraintes d'assemblage définies.

4 Processus d'évolution

4.1 Support de l'évolution dans les principaux ADL existants

Dans les ADL existants, les évolutions sont généralement initiées à l'étape de configuration des architectures (*e.g.* C2 (Oreizy et al., 1998; Medvidovic, 1996), Darwin (Magee et Kramer, 1996), Wright (Allen et al., 1997)). Ces ADL ne représentent pas les assemblages et ne gèrent donc pas l'évolution pendant l'exécution. La cohérence entre configurations et spécifications est assurée (*e.g.* C2 (Oreizy et al., 1998; Medvidovic, 1996), Wright (Allen et al., 1997)) mais la propagation des changements est limitée. A notre connaissance, les changements sont uniquement propagés du haut vers le bas (*e.g.* C2 (Oreizy et al., 1998; Medvidovic, 1996), Darwin (Magee et Kramer, 1996)), c'est-à-dire adaptés au cadre d'une ingénierie directe ou d'une re-conception. Ce mécanisme s'applique aux évolutions réalisées par un architecte sur les niveaux conceptuels puis propagées pour réutiliser, par versionnement, les implantations existantes. Ainsi, les évolutions de composants ou d'architectures qui interviennent à l'exécution (logiciels autonomiques, environnements ouverts, etc.) sont une exception. MAE (Roshandel et al., 2004) propose une gestion de configuration contrôlant les versions des composants à l'exécution, mais ne fournit aucun support pour conserver une trace de ces nouvelles architectures par le versionnement des configurations ou des spécifications d'architectures correspondantes. Les trois niveaux de représentation de Dedal, associés à un outillage adapté, en particulier à l'exécution (Zhang et al., 2009), permettent de gèrer les évolutions nécessitant des propagations de haut en bas (ingénierie directe ou re-conception) et de bas en haut (retro-ingénierie), afin d'éviter l'apparition d'incohérences (érosion ou dérive).

4.2 Propagation des changements

La cohérence est une propriété interne des modèles architecturaux qui assure que les différentes vues (dans notre cas, les différentes dimensions) d'une même architecture ne se contredisent pas entre elles (Taylor et al., 2009). L'objectif de la vérification de cohérence est de prévenir que des changements induisent des incohérences intra ou inter-dimensionnelles. Lorsque des changements préservent la cohérence d'une architecture, ils sont autorisés et appliqués. Dans le cas contraire, soit les changements sont interdits, soit ils déclenchent le versionnement de l'architecture afin de préserver la cohérence de la version existante. Le versionnnement peut être limité à l'une des représentations de l'architecture ou nécessiter, par propagation de l'évolution, un versionnement de ses différentes dimensions.

Le modèle de versions de Dedale permet de garder une trace des changements intervenus entre versions d'un même arbre de dérivation. Au sein de chaque dimension, les versions enregistrent un identifiant de version, l'identifiant de la version précédente et la liste des chan-

gements dans la version courante (deltas). Peu d' ADLs permettent d'exprimer les changements entre versions. C2 (Oreizy et al., 1998; Medvidovic, 1996) et Darwin (Magee et Kramer, 1996) gèrent des changements de configuration transactionnels sur le système en cours d'exécution mais ne conservent pas la trace de ces changements dans les représentations des architectures. Dedale représente les changements entre versions (deltas) comme des listes d'opérations de changement (ajouts, suppressions ou modifications). La figure 9 présente la version *2.0* de la spécification du BRS. Cette architecture diffère de la précédente par l'addition du composant *GIS*.

4.3 Scénario d'évolution pour le BRS

Ce scénario d'évolution étudie l'ajout de la classe de composant (*StationData*) à la configuration du BRS(*cf.* figure 9). Il illustre les mécanismes de propagation évoqués précédemment. Ce nouveau composant se connecte à *BikeTrip* par l'interface requise *LocOprs* (*cf.* figure 8). La compatibilité syntaxique et sémantique de l'interface *LocOprs* du composant *BikeTrip* avec l'interface *LocOprs* du composant *StationData* est vérifiée. Le protocole de comportement du composant *BikeTrip* fait appel à la méthode *findStation* fournie par l'interface *LocOprs* du composant *StationData* (*cf.* figure6). La classe de composants *StationData_1.0* (*cf.* figure8) n'impose aucune contrainte supplémentaire. Cette nouvelle configuration est cohérente, mais sa définition n'est plus compatible avec sa spécification. La propagation des modifications déclenche la création d'une nouvelle version de la spécification par ajout du rôle *GIS*. De même, la propagation des modifications vers l'assemblage déclenche la création d'une nouvelle version, décrivant l'instanciation du composant *StationData* (*cf.* figure 9).

```
component_class StationData
  implements GIS
  using fr.ema.StationDataImpl
  provided_interfaces LocOprs
  version 1.0
```

FIG. 8 : *Description de la classe de composants StationData*

5 Conclusion

Dedale permet la représentation explicite et distincte des spécifications, des configurations et des assemblages qui constituent les trois dimensions des architectures. Les décisions de conception des architectures peuvent ainsi être énoncées et leur impact suivi tout au long du cycle de développement. Les évolutions peuvent intervenir dans n'importe laquelle des trois dimensions. Leur cohérence est vérifiée et les changements nécessaires propagées dans les autres dimensions. Dedale propose ainsi un support de l'évolution original car il permet la propagation de haut en bas mais aussi de bas en haut et une solution pour prévenir la dérive et l'érosion des architectures. L'ADL Dedale ainsi que le processus d'évolution présentés dans ce papier ont été implémentés comme une extension de Julia, l'implémentation open-source

```
specification BRSSpec
 component_roles
  BikeCourse; BikeCourseDB; GIS
 connections
  connection connection1
  ...
  connection connection3
   client BikeCourse.LocOprs
   server GIS.LocOprs
 architecture_behavior
  (!BikeCourse.BikeOprs.selectBike;
  ({?BikeCourse.LocOprs.findLoc;
  !GIS.LocOprs.findLoc;
  ?BikeCourse.BQuery.findBike;}
  +
  {?BikeCourse.BQuery.findBike;}))
  +
  ...
 version 2.0;
 pre_version 1.0;
 by addition of GIS;
```

```
configuration BRSConfig
 implements BRSSpec (2.0)
 component_classes
  BikeTrip (1.0) implements BikeCourse;
  BikeTripDBClass (1.0) implements
   BikeCourseDB;
  StationData (1.0) implements GIS
 version 2.0;
 pre_version 1.0;
 by addition of StationData;
```

```
assembly BRSAss
 instance_of BRSConfig (2.0)
 component_instances
  BikeTripC1 as BikeCourse;
  BikeTripDBC1 as BikeCourseDB
  StationDataC1 as GIS
 assembly_constraints ...
 version 2.0
 pre_version 1.0;
 by addition of StationDataC1;
 component_instance StationData
  instance_of StationData (1.0)
```

FIG. 9 : *Description de l'architecture après évolution*

de référence du modèle de composants Fractal, et de ses outils associés[2]. La gestion de la compatibilité entre protocoles est réalisée grâce à SOFA, qui propose une implémentation de ces mécansimes dans le cadre de Fractal. Des expérimentations préliminaires ont été menées sur les architectures logicielles d'instruments de musique électronique configurables. Une des perspectives envisagée pour ces travaux est leur expérimentation dans le cadre de la gestion de lignes de produits logiciels à base de composants.

Références

Allen, R. et D. Garlan (1997). A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol. 6*(3), 213–249.

Allen, R., D. Garlan, et R. Douence (1997). Specifying dynamism in software architectures. In *Proc. of the Workshop on Foundations of Component-Based Software Engineering*, Zurich, Switzerland.

Arévalo, G., N. Desnos, M. Huchard, C. Urtado, et S. Vauttier (2009). Formal concept analysis-based service classification to dynamically build efficient software component directories. *Int. J. Gen. Syst. 38*(4), 427–453.

Booch, G., J. Rumbaugh, et I. Jacobson (2005). *Unified Modeling Language User Guide, (The 2nd Edition)*. Addison-Wesley.

Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma, et J.-B. Stefani (2006). The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper. 36*(11-12), 1257–1284.

Cheng, S. W., D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, et P. Steenkiste (2002). Using architectural style as a basis for system self-repair. In *3rd IEEE/IFIP Working Conf. on Software Architecture*, Montreal, Canada, pp. 45–59.

Crnkovic, I., M. Chaudron, et S. Larsson (2006). Component-based development process and component lifecycle. In *Proc. of the Intl. Conf. on Software Engineering Advances*, Papeete, French Polynesia, pp. 44.

Dashofy, E. M., A. van der Hoek, et R. N. Taylor (2002). An infrastructure for the rapid development of XML-based architecture description languages. In *Proc. of ICSE '02*, Orlando, FA, USA, pp. 266–276.

Desnos, N., M. Huchard, G. Tremblay, C. Urtado, et S. Vauttier (2008). Search-based many-to-one component substitution. *J. Softw. Maint : Res. Pract. 20*(5), 321–344.

Garlan, D., B. Schmerl, et J. Chang (2001). Using gauges for architecture-based monitoring and adaptation. In *Proc. Working Conf. on Complex and Dynamic Systems Architecture*, Brisbane, Australia.

Magee, J. et J. Kramer (1996). Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes 21*(6), 3–14.

Medvidovic, N. (1996). ADLs and dynamic architecture changes. In *Joint proc. of ISAW-2 and Viewpoints '96 on SIGSOFT '96 workshops*, San Francisco, CA, USA, pp. 24–27.

[2] `http ://www.objectweb.org`

Medvidovic, N., D. S. Rosenblum, et R. N. Taylor (1999). A language and environment for architecture-based software development and evolution. In *Proc. of ICSE'99*, Los Angeles, CA, pp. 44–53.

Oreizy, P., N. Medvidovic, et R. N. Taylor (1998). Architecture-based runtime software evolution. In *Proc. of ICSE '98*, Kyoto, Japan, pp. 177–186.

Perry, D. E. et A. L. Wolf (1992). Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes 17*(4), 40–52.

Plásil, F. et S. Visnovsky (2002). Behavior protocols for software components. *IEEE Trans. Softw. Eng. 28*(11), 1056–1076.

Roshandel, R., A. V. D. Hoek, M. Mikic-Rakic, et N. Medvidovic (2004). MAE–a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol. 13*(2), 240–276.

Shaw, M. et D. Garlan (1996). *Software architecture : perspectives on an emerging discipline*. Prentice-Hall, Inc.

Sommerville, I. (2006). *Software Engineering* (8th ed.). Addison-Wesley.

Taylor, R. N., N. Medvidovic, et E. M. Dashofy (2009). *Software Architecture : Foundations, Theory, and Practice*. John Wiley & Sons.

Zhang, H. Y., C. Urtado, et S. Vauttier (2009). Connector-driven process for the gradual evolution of component-based software. In *Proc. of the 20th Aust. Softw. Eng. Conf.*, Gold Coast, Australia.

Summary

Component-based development promotes a software development process that focuses on reuse and composition. Software architectures can be described at three development stages: architecture specification, architecture configuration and instantiated component assembly. Conformance must be guaranteed top-down from an abstract description level to the next, more concrete one. But such descriptions could also be used bottom-up to control software evolution. Surprisingly, no architecture description language proposes such a detailed description for architectures that covers the whole component-based development cycle. This paper proposes Dedale, a three-dimensional ADL that enables the description of architecture specification, configuration and assembly. It then shows how these descriptions can be used during a controlled software architecture evolution mechanism that helps build, test and record versions of component-based software. This mechanism tackles the well-known issues of architecture erosion and architecture drift that denote mismatches between the different architecture definitions.