Maintien de la cohérence entre les architectures pour une gestion autonomique de systèmes pervasifs

Approche basée sur les méta-modèles des architectures de conception et d'exécution

Stéphanie Chollet [1], **Philippe Lalanda** [2], **Catherine Hamon** [3]

1. *Laboratoire de Conception et d'Intégration des Systèmes*
50, rue Barthélémy de Laffemas, BP 54
26902 Valence Cedex 9, France
stephanie.chollet@lcis.grenoble-inp.fr

2. *Laboratoire d'Informatique de Grenoble*
220, rue de la chimie, BP 53
38041 Grenoble Cedex 9, France
philippe.lalanda@imag.fr

3. *Orange Labs*
28, chemin du vieux chêne
38243 Meylan, France
catherine.hamon@orange.com

RÉSUMÉ. *L'informatique orientée service est devenue très populaire ces dernières années, en particulier pour traiter des environnements dynamiques et imprévisibles. Cependant, cette nouvelle approche soulève de sérieux problèmes au niveau de l'administration et de la maintenance des systèmes. Dans cet article, nous développons une proposition où les architectures logicielles de conception et d'exécution sont utilisées pour gérer les systèmes orientés service et pour aider les administrateurs à suivre et comprendre les évolutions à l'exécution. Nous montrons comment des concepts liés à la conception et à l'exécution peuvent être liés et exploités par un gestionnaire autonomique ou par un administrateur humain. Cette approche est validée sur un cas d'utilisation issu du domaine de la santé pervasive et développé avec Orange Labs.*

ABSTRACT. *Service-oriented programming has become very popular in recent years, in particular to deal with dynamic, unpredictable environments. However, this novel approach raises*

major issues in term of administration. In this paper, we present an approach where design and runtime architectures are used to manage service-oriented systems and help administrators to follow and understand runtime evolutions. We show how concepts of design time and runtime can be linked and exploited by an autonomic manager or by a human administrator. This approach is validated on a real use case belonging to the pervasive health domain and built with the Orange Labs.

MOTS-CLÉS : *Informatique autonomique, Approche dirigée par les modèles, Approche orientée service*

KEYWORDS: *Autonomic Computing, Model-Driven Approach, Service-Oriented Computing*

---

1. Introduction

L'informatique orientée service (Papazoglou, 2003) est fortement utilisée aujourd'hui pour permettre des adaptations à l'exécution. Cette approche est fondée sur les principes de faible couplage, de liaison retardée et de substituabilité de modules logiciels appelés service. L'informatique orientée service permet ainsi d'aborder de nouveaux domaines d'application caractérisés notamment par de fortes contraintes de dynamisme et par une faible prédictibilité. De façon non surprenante au vu de ses propriétés, l'informatique orientée service rencontre un réel succès pour la mise en place d'applications ambiantes (ou pervasives), de plus en plus répandues aujourd'hui.

Par ailleurs, les adaptations sont souvent réalisées de façon autonomique (Lalanda *et al.*, 2013). Les propriétés autonomiques telles que l'auto-configuration, l'auto-optimisation, l'auto-réparation ou encore l'auto-protection peuvent permettre de faire face à la complexité croissante des besoins en adaptation. Ces propriétés peuvent permettre de diminuer significativement les coûts d'exploitation et de maintenance d'une part et palier la faible disponibilité des administrateurs d'autre part.

L'évolution autonomique des applications soulève néanmoins des problèmes de suivi et de compréhension de l'état des applications. Notre objectif est de mieux formaliser la connaissance des applications dans le cadre des systèmes orientés services et, notamment, pour les applications pervasives. Ceci dans le but de faciliter le travail des administrateurs ou la construction de gestionnaires autonomiques. Nous nous focalisons sur le niveau architectural qui nous semble être le bon niveau de granularité pour mener à bien des adaptations. Disposer d'une formalisation claire de l'architecture exécutée est très précieux pour guider les adaptations. Cependant, cette architecture seule n'est pas suffisante. En effet, il est nécessaire de disposer d'une autre connaissance architecturale qui donne l'ensemble de propriétés que l'on souhaite garantir à l'exécution et qui exprime ce qu'est une architecture valide afin de conduire les adaptations conformément à cette description. Cette seconde moitié de connaissance qui est issue de la phase de conception, sera nommée par la suite architecture de conception. Cette complémentarité entre l'architecture de l'exécution et l'architecture de conception avait déjà été perçue par Garlan et Schmerl (2002).

Dans cet article, nous montrons comment simplifier les tâches des administrateurs d'applications orientées services en maintenant des liens de traçabilité entre les architectures réalisées à différents moments du cycle de vie de l'application. L'article est structuré de la façon suivante. La section 2 détaille un cas d'utilisation et ses besoins. La section 3 présente notre approche qui est ensuite développée dans la section 4. La section 5 traite de l'implantation de notre approche et de sa validation. Certains travaux connexes sont discutés en section 6. Enfin, cet article se termine par une conclusion en section 7.

2. Motivation et cas d'utilisation

Notre cas d'utilisation est issu du projet FUI Medical [1]. C'est un cas d'utilisation fréquent dans le domaine de la santé ; il a pour objectif principal le maintien des personnes âgées à leur domicile. Plus précisément, nous avons travaillé sur l'application nommée actimétrie qui récupère un certain nombre de mesures et analyse l'activité physique d'une personne dans son environnement. L'idée est de tracer et enregistrer les habitudes d'un utilisateur pour détecter des changements d'habitudes imperceptibles par un médecin lors d'une consultation. Ces changements d'habitudes peuvent être un signe avant-coureur de sérieux problèmes de santé.

Pour mettre en place une telle application, il faut avoir des capteurs installés dans une maison permettant de récupérer des informations géolocalisées. Ces capteurs sont connectés par un réseau. Une de leurs principales caractéristiques est qu'ils sont hétérogènes et dynamiques. Pour ce type d'application (Figure 1), n'importe quel type de capteurs peut être utilisé : détecteurs de présence, tensiomètre et aussi des équipements comme une télévision, une machine à laver ou une machine à café connectées. Les données collectées sont régulièrement transmises à un serveur distant qui réalise une analyse. Cette analyse est utilisée pour détecter des modifications de comportements. Ces informations ne sont ensuite transmises qu'au médecin.

En pratique, la réalisation d'une telle application se base sur un ensemble de services logiciels. Les équipements et les applications sont exposés comme des services logiciels de différentes natures comme des services UPnP [2], des services Web [3], des services DPWS [4], etc. L'intégration de ces services distants ou locaux doit se faire de manière transparente (Lalanda *et al.*, 2010), (Escoffier *et al.*, 2014).

L'application d'actimétrie n'est pas contrainte au niveau du choix des équipements disponibles ; chaque maison peut être équipée d'un certain nombre de capteurs fournis par des fabricants différents. D'un point de vue logiciel, ceci est assez complexe à supporter. Il faut réaliser un code d'intégration conséquent et complexe. De plus,

1. Projet en collaboration avec Orange Labs, Université Joseph Fourier, Telecom ParisTech et ScalAgent : http://medical.imag.fr
2. http://www.upnp.org
3. www.w3c.org
4. http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01

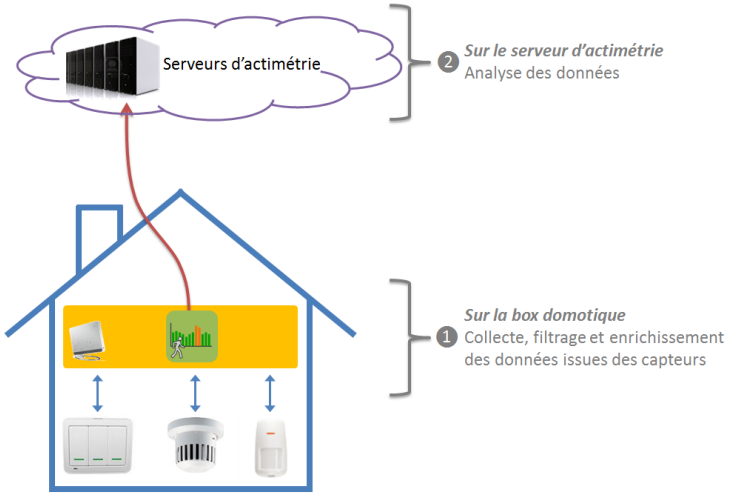

Figure 1. L'application Actimétrie.

il faut pouvoir gérer l'hétérogénéité et le dynamisme des capteurs. Toutes les situations ne peuvent pas être anticipées au moment de la conception de l'application et il faut apporter quelques informations supplémentaires au moment de l'exécution pour s'adapter à l'environnement changeant. C'est dans ce cadre que s'inscrit notre proposition.

3. Approche globale

3.1. Principes

Notre objectif est de simplifier les tâches des administrateurs d'application à base de composants orientés services en maintenant des liens pour la traçabilité entre les architectures réalisées à différent moment du cycle de vie de l'application. En effet, comme nous l'avons vu précédemment, l'architecture de l'exécution évolue suite à des modifications de l'environnement d'exécution (arrivée et/ou départ d'équipements). L'administrateur doit être capable de savoir si l'exécution qui est en cours est toujours valide par rapport à celle qui est souhaitée. Nous souhaitons automatiser la vérification de la validité de l'architecture à l'exécution. Pour ce faire, nous définissons trois types d'architectures qui sont utilisées à différents étapes du cycle de vie :

– **l'architecture de conception** qui est une composition de différents éléments liés les uns aux autres comprenant de la variabilité ; c'est-à-dire des contraintes lâches qui seront précisées dans les étapes suivantes du processus de développement.

– **l'architecture de déploiement** qui est issue de l'architecture précédente et qui contient en plus de nombreux éléments configurés. L'architecture de déploiement est celle qui doit être déployée sur la plate-forme cible. Les configurations sont dépendantes de cette plate-forme et du contexte. Cependant, dans cette architecture, il reste

encore un peu de variabilité qui sera résolue à l'exécution. Cette variabilité est restreinte et n'a de sens que pour une application basée sur une approche à composants orientés services.

– **l'architecture de l'exécution** qui est la représentation de l'application en exécution. Cette architecture est construite grâce à l'analyse de l'exécution avec des mécanismes de surveillance. Elle est une vue de l'exécution avec un certain niveau d'abstraction ; toutes les informations de l'exécution ne sont pas représentées. Dans cette architecture, il n'existe plus de variabilité ; ces dernières ont toutes été résolues par la machine d'exécution. La caractéristique principale de cette architecture est qu'elle évolue fréquemment, par opposition aux deux autres types d'architecture.

Pour maintenir le lien entre ces différentes architectures, nous proposons une approche basée sur les principes de l'informatique autonomique. La Figure 2 illustre notre approche globale. Nous ajoutons au-dessus de notre système une boucle autonomique MAPE-K (Kephart, Chess, 2003) et nous nous intéressons plus particulièrement à la structuration de la connaissance. C'est dans la base de connaissances que nous voulons maintenir le lien entre les architectures de conception et de l'exécution. Les phases d'analyse et de planification doivent permettre de vérifier la validité entre l'architecture de l'exécution et celle de la conception. Il faut noter que l'architecture de déploiement n'apparaît pas dans cette figure ; en effet, elle est utile uniquement pour le premier déploiement de l'application sur la plate-forme. Sitôt qu'elle est déployée, elle est réifiée sous la forme d'une architecture de l'exécution. Cette approche requiert de mettre en place des *touchpoints* (capteurs et actionneurs) sur le système permettant, à la phase de monitoring, de mettre à jour l'architecture de l'exécution et, à la phase d'adaptation, de modifier l'exécution suivant ce qui a été planifié (Morand *et al.*, 2011).

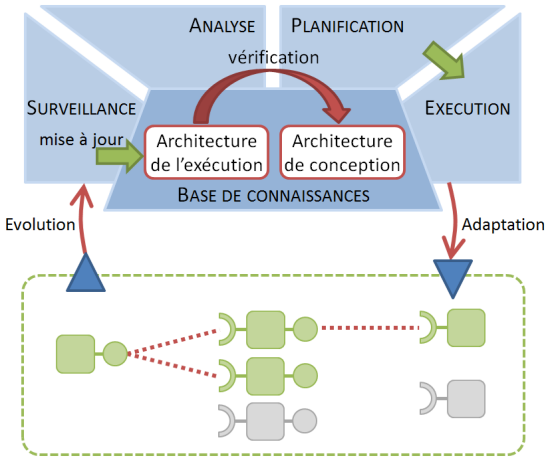

Figure 2. Principes de notre approche.

Dans les deux parties suivantes, nous allons présenter d'abord la formalisation de la connaissance, puis la méthode de vérification de la validité entre les architectures.

3.2. Formalisation de la connaissance à base de modèles et méta-modèles

La formalisation des architectures a pour objectif d'identifier clairement les différents éléments qui les composent. Pour ce faire, notre proposition est basée sur la définition d'un ensemble de méta-modèles (OMG, 2002 ; Favre, 2004). Un méta-modèle permet de spécifier de manière claire, précise et non-ambiguë le langage utilisé pour modéliser les architectures. Plus précisément, un méta-modèle sert à définir une grammaire et un vocabulaire afin de réaliser des modèles cohérents et conformes à celui-ci. Dans notre cas, nous proposons de définir un méta-modèle pour chaque type d'architecture afin de pouvoir construire des applications conformes et cohérentes.

Cependant, la définition des trois méta-modèles (architecture de conception, de déploiement et de l'exécution) pour exprimer les différentes architectures met en œuvre un ensemble de concepts communs dû au fait que se sont toutes les trois des architectures. Ces dernières comprennent certes des concepts de sens commun mais avec des particularités propres au domaine d'application de l'architecture. Nous proposons de concevoir un méta-modèle commun pour les trois architectures et qui sera hérité pour chacun des méta-modèle des architectures.

Un des principaux avantages de cette organisation est d'avoir une continuité entre les concepts. Il est clair qu'un composant dans l'architecture de conception doit être le même composant dans l'architecture de déploiement avec certes des caractéristiques supplémentaires pour la phase de déploiement. Un composant exécuté est, quant à lui, réifié dans l'architecture de l'exécution mais il reste intrinséquement un composant comme défini dans les autres architectures.

Comme nous l'avons présenté précédemment, les différentes architectures sont définies avec des niveaux de variabilité qui diffèrent comme présenté dans la Figure 3. L'architecture de conception est la plus variable. Le méta-modèle doit donc supporter l'expression de cette variabilité. L'architecture de déploiement est un raffinement d'une architecture de conception ; elle contient moins de variabilité mais respecte les contraintes de l'architecture de conception dont elle est issue. Les seuls éléments de variabilité qui restent sont ceux qui pourront être résolus par la machine d'exécution. L'architecture de l'exécution est une vue de ce qui s'exécute et, par conséquent, il n'y a plus aucune variabilité dans ce modèle.

3.3. Vérification de la conformité entre modèles

La conformité entre les modèles est assurée en deux étapes :

– premièrement, par construction, l'architecture de déploiement est issue de l'architecture de conception. La variabilité exprimée dans l'architecture de conception est spécifiée et restreinte au niveau du déploiement. L'architecture de l'exécution est issue de l'architecture de déploiement au démarrage. Elle est uniquement une représentation de l'architecture de déploiement au moment du premier déploiement. Cependant, pendant l'exécution cette architecture de l'exécution évolue. Le lien de conformité est naturellement perdu.

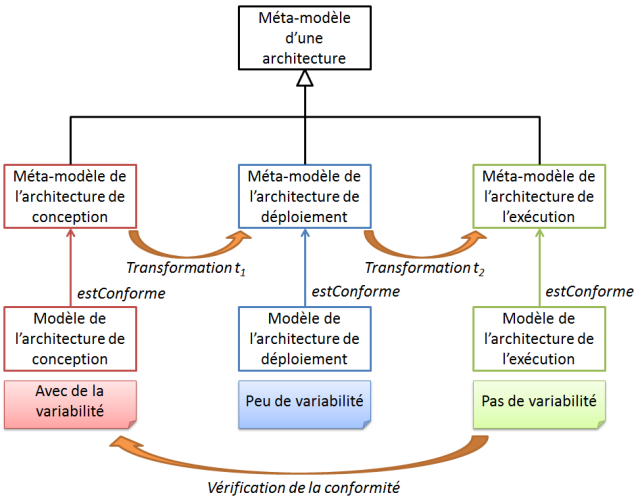

Figure 3. Modélisation des différentes architectures avec de la variabilité.

– deuxièmement, nous proposons un algorithme de vérification de la conformité entre l'architecture de l'exécution et l'architecture de conception. Cet algorithme permet de vérifier si l'architecture est toujours valide et sinon en informer soit l'administrateur soit le gestionnaire autonomique de niveau supérieur pour qu'il soit alerté du problème.

Dans notre approche, nous proposons de définir deux transformations entre les méta-modèles pour assurer la conformité par construction, comme illustré dans la Figure 3. Ces deux transformations ne sont pas du même type. La première transformation entre le méta-modèle de l'architecture de conception et celui de l'architecture de déploiement est l'expression de la spécialisation/configuration d'un certain nombre d'éléments. La deuxième transformation entre le méta-modèle de l'architecture de déploiement et celui de l'architecture de l'exécution est la projection sur une plate-forme d'exécution. Evidemment, ces transformations exprimées au niveau du méta-modèle doivent être valides au niveau du modèle.

Pendant l'exécution, l'architecture de l'exécution évolue à cause de changements dans l'environnement d'exécution. Nous proposons de mettre en œuvre un algorithme de validation de la conformité qui s'assure que les éléments modifiés à l'exécution respecte les contraintes définies dans l'architecture de conception. La difficulté principale pour cette vérification est qu'il faut savoir à quel moment elle doit être opérée. Il faut que la vérification se fasse à un moment où l'état de l'exécution est valide ; c'est-à-dire que ce soit un état « stable » et non dans un état où des modifications « en cascade » ne sont pas terminées. De plus, l'algorithme de vérification surcharge le fonctionnement dit normal de l'application, il ne faut pas l'appeler trop fréquemment comme cela peut être le cas lorsqu'un équipement est « instable » ; par exemple, il est en limite de portée pour être détecté en continu.

4. Méta-modèles et modèles

4.1. Méta-modèle commun aux architectures

Le méta-modèle commun aux trois architectures est présenté à la Figure 4.

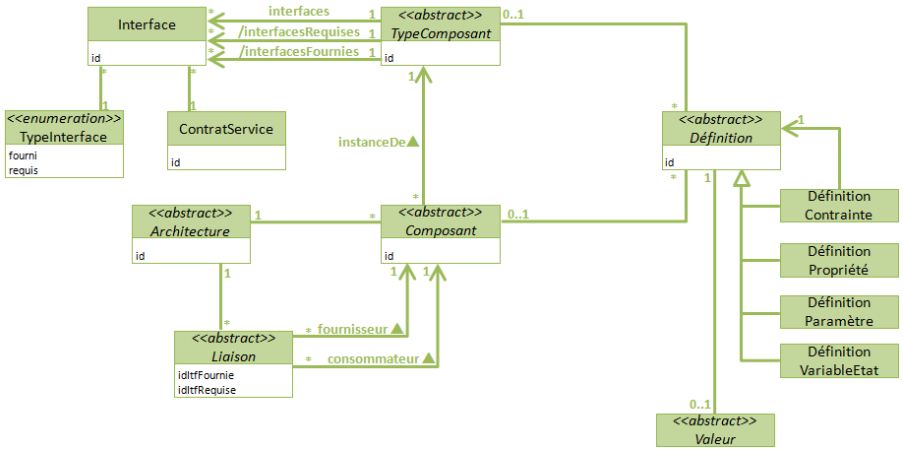

Figure 4. Méta-modèle commun définissant une architecture.

Une architecture est représentée par la classe abstraite *Architecture*. Chaque architecture est définie par un identifiant (*id*) unique, des composants et des liaisons. Chaque liaison et chaque composant a nécessairement une unique architecture d'appartenance. Chaque composant est défini par un identifiant (*id*) tel que tous les composants, qui font partie d'une même architecture, ont tous des identifiants différents. Chaque composant est une instance d'un type de composant, défini par un identifiant (*id*) unique. Chaque type de composant dispose d'un ensemble d'interfaces, qui ne peuvent pas être partagées entre composants. Chaque interface est identifiée avec un *id*, de sorte qu'un type de composant ne puisse pas disposer de deux interfaces avec le même identifiant.

Une interface peut être soit fournie, soit requise, comme l'indique son *TypeInterface*. Une interface fournie permet à un composant d'offrir un service, alors qu'une interface requise créé au contraire une dépendance de service. Les différentes interfaces d'un type de composant peuvent donc être réparties entre les interfaces fournies et les interfaces requises. C'est la raison pour laquelle nous avons les relations dérivées *interfacesRequises* et *interfacesFournies* (contrainte OCL 1).

Quel que soit le type de l'interface, le service dont il est question, est décrit grâce au *ContratService* de l'interface. Chaque *ContratService* dispose lui aussi d'un identifiant unique.

Une liaison permet de lier deux composants par l'intermédiaire de leurs interfaces, définies au niveau de leur type. C'est la raison pour laquelle une liaison est définie par

```
context TypeComposant
    inv: self.interfaces → select(i | i.TypeInterface = requis) →
        forAll(i | self.interfacesRequises → exists(ir | ir=i))
    inv: self. interfacesRequises →
        forAll(i | self.interfaces → select(i | i.TypeInterface = requis) → exists(ir | ir=i))
    inv: self.interfaces → select(i | i.TypeInterface = fourni) →
        forAll(i | self.interfacesFournies → exists(if | if=i))
    inv: self.interfacesFournies →
        forAll(i | self.interfaces → select(i | i.TypeInterface = fourni) → exists(if | if=i))
```

Contrainte OCL 1: Relations entre *interfacesRequises* et *interfacesFournies*.

les deux composants ainsi que par les identifiants des interfaces (présents dans leur type) à lier (contrainte OCL 2).

```
context Liaison
    inv: self.fournisseur.instanceDe.interfacesFournies → one(ili.id=self.idItfFournie)
    inv: self.consommateur.instanceDe.interfacesRequises → one(ili.id=self.idItfRequise)
```

Contrainte OCL 2: Types des interfaces pour les liaisons.

De plus, une liaison ne peut lier que des interfaces qui exposent le même contrat de service. Il est impossible d'établir une liaison entre deux interfaces d'un même composant. Enfin, il n'est pas possible de trouver deux liaisons reliant exactement les mêmes interfaces au sein d'une même architecture (contrainte OCL 3).

```
context Liaison
    inv: self.fournisseur.instanceDe.interfacesFournies →
        select(i | i.id = self.idItfFournie) → asSequence() →
            first().ContratService.id = self.consommateur.instanceDe.interfacesRequises →
                select(i | i.id = self.idItfRequise) → asSequence() → first().ContratService.id
    inv: self.fournisseur <> self.consommateur
context Architecture
    inv: self.Liaison → forAll(l1, l2 | l1 <>l2
    implies (l1.fournisseur <> l2.fournisseur)
        or (l1.consommateur <> l2.consommateur)
        or (l1.idItfFournie <> l2.idItfFournie)
        or (l1.idItfRequise <> l2.idItfRequise))
```

Contrainte OCL 3: Contraintes sur les liaisons.

Les composants peuvent disposer de propriétés, de paramètres de variables d'état et de contraintes, éventuellement valués selon le type de composant considéré. Ces éléments permettent de gérer une partie de la variabilité supportée. Par exemple, on peut définir une variable d'état *tempsTraitement* que l'on souhaite d'un niveau moyen à la phase de conception et, au moment de l'exécution, le temps de traitement est valué. Nous avons séparé l'aspect définition de la valeur pour ces quatre concepts. Une définition est en relation soit avec un *TypeComposant*, soit avec un *Composant* (de manière exclusive). De plus, toutes les définitions d'un composant ou d'un type de

composant ont des identifiants uniques au sein de celui-ci. Une définition de contrainte est en relation avec une définition, qui ne peut pas être une contrainte (contrainte OCL 4).

context DéfinitionContrainte
inv: not(self.Définition.oclIsTypeOf(DéfinitionContrainte))

Contrainte OCL 4: Non récursivité des définitions de contraintes.

4.2. Méta-modèle de l'architecture de conception

L'architecture de conception est formalisée avec le méta-modèle présenté dans la Figure 5. Ce méta-modèle étend le méta-modèle commun présenté précédemment.

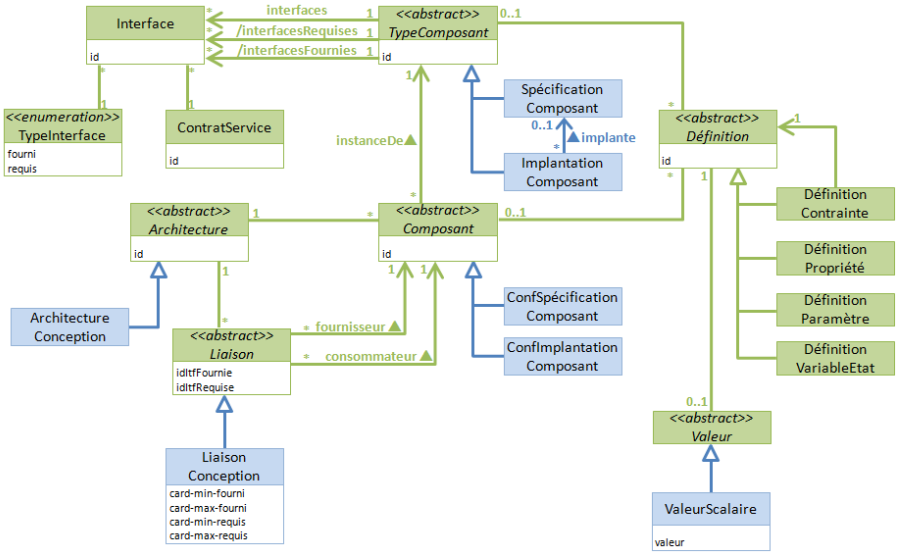

Figure 5. Méta-modèle de l'architecture de conception.

Ce méta-modèle représente les concepts nécessaires pour définir des modèles de l'architecture de conception. Nous devons retrouver dans ce méta-modèle les éléments qui sont particuliers à l'architecture de conception : les composants sont soit des spécifications, soit des implantations. Cette architecture peut disposer de configurations de spécifications et de configurations d'implantations de composant. Comme nous l'avons vu, une configuration de spécification (respectivement d'implantation) de composant instancie forcément une spécification (respectivement une implantation) de composant. Cela est exprimé avec la contrainte OCL 5.

De plus, cette architecture supporte une variabilité exprimée au niveau des liaisons avec des cardinalités. Ces dernières sont présentes sur chacune des extrémités des liaisons : côté interface fournie (*card-min-fourni* et *card-max-fourni*) et côté interface requise (*card-min-requis* et *card-max-requis*).

```
context ArchitectureConception
    inv: self.Composant → forAll(c.oclIsTypeOf(ConfSpécificationComposant)
        or c.oclIsTypeOf(ConfImplantationComposant))
context ConfSpécificationComposant
    inv: self.référence.oclIsTypeOf(SpécificationComposant)
context ConfImplantationComposant
    inv: self.référence.oclIsTypeOf(ImplantationComposant)
```

Contrainte OCL 5: Contraintes sur les types de composants des architectures de conception.

Une architecture de conception peut disposer de configurations de spécifications et d'implantations de composants. Une configuration de spécification (resp. d'implantation) de composant instancie forcément une spécification (resp. d'implantation) de composant. De plus, une liaison permet d'exprimer qu'une implantation de composant peut ou non implanter une unique spécification de composant. Cette liaison est orientée, car la spécification n'a pas besoin de savoir quelles sont les implantations qui l'implantent.

4.3. Méta-modèle de l'architecture de l'exécution

Le méta-modèle de l'architecture de l'exécution (Figure 6) vient, elle-aussi, étendre le méta-modèle commun. Il existe plusieurs architectures de l'exécution qui peuvent être issues d'une architecture de conception. Les architectures de l'exécution référencent une architecture de conception via une indirection. Cette dernière n'est navigable que dans un sens, car une architecture de conception n'a pas à savoir par quelles autres architectures elle est utilisée.

Une architecture de l'exécution ne peut disposer que d'instances de composants (contrainte OCL 6), chacune instanciant une implantation de composant.

```
context ArchitectureExécution
    inv: self.Composant → forAll(c.oclIsTypeOf(InstanceComposant))
context InstanceComposant
    inv: self.instanceDe.oclIsTypeOf(ImplantationComposant)
```

Contrainte OCL 6: Contraintes sur les types de composants des architectures de l'exécution.

Notons enfin que la hiérarchie de classes issue de la classe *Valeur* est différente par rapport aux autres architectures. En effet, dans le cas des variables d'état, nous souhaitons disposer d'un historique horodaté des valeurs successives prises par celle-ci. La classe *Collection* permet donc d'agréger un ensemble de valeurs scalaires, par l'intermédiaire de la classe horodatage.

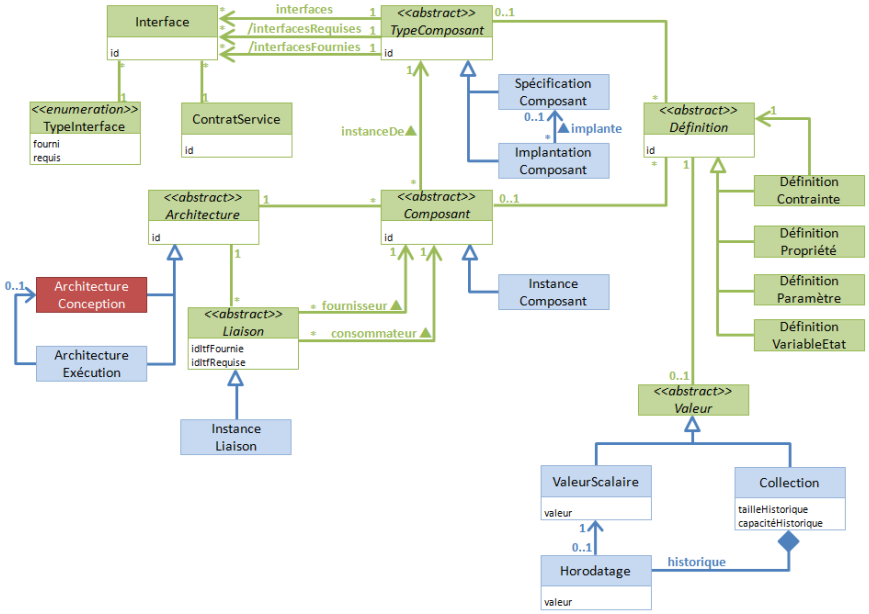

Figure 6. Méta-modèle de l'architecture de l'exécution.

5. Implantation et validation

5.1. Implantation

Notre approche a été mise en œuvre dans le cadre du projet FUI Medical pour l'application Actimétrie présentée précédemment. Notre solution se base sur la plate-forme Cilia Mediation Framework [5] qui permet de simplifier la construction d'applications de médiation. Cilia supporte l'évolution à chaud de son architecture à base de composants orientés services dynamiques et est capable de rendre compte de son état interne. De plus, Cilia embarque l'intergiciel RoSe [6], dédié à la découverte et la communication avec des équipements et des services distants.

L'implantation de notre approche est résumée sur la Figure 7 : la plate-forme d'exécution Cilia avec RoSe, une boucle de contrôle dans laquelle nous avons étendue la connaissance avec les différentes architectures, Cilia IDE qui est un outil de conception et de supervision des architectures.

L'atelier Cilia IDE, basé sur Eclipse IDE, permet de concevoir, de déployer et de superviser des architectures à base de composants Cilia soit au format XML soit de manière graphique. Il perçoit l'état de la plate-forme Cilia via la base de connaissance

5. http://wikiadele.imag.fr/index.php/Cilia
6. http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose

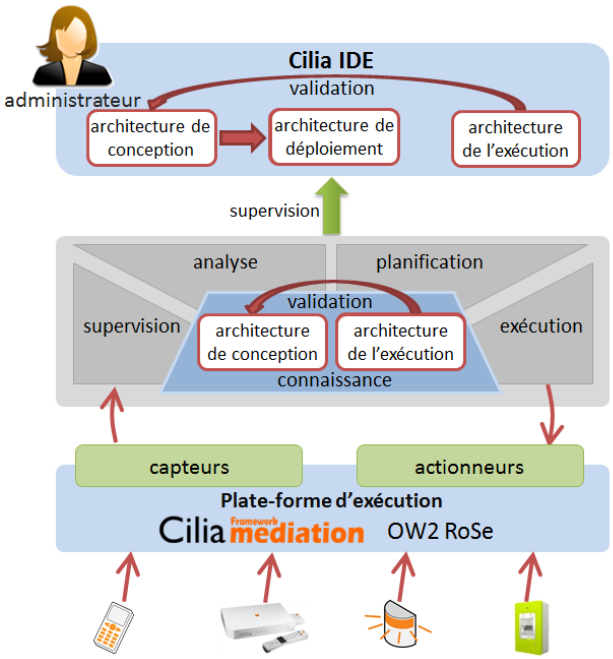

Figure 7. Architecture de l'implantation.

qu'il peut, par ailleurs, alimenter sur la partie conception. Enfin, il est en mesure d'envoyer des ordres de reconfiguration directement à la plate-forme Cilia. Et, il permet de montrer, s'il y a une violation de contraintes entre les architectures, où elle se trouve.

5.2. Validation

Dans le cadre de l'application actimétrie, la perception des habitudes des personnes âgées est effectuée au travers de leurs interactions avec l'écosystème numérique de leur habitat : utilisation de la *set top box*, manipulation d'interrupteurs connectés, passage devant des détecteurs de présence ou activation des détecteurs d'ouverture de porte. Les événements générés par ces équipements sont agrégés et permettent à l'application de localiser les personnes dans leur habitat et de déduire des activités sur leur activité courante. Dans le cadre de cette validation, l'application est séparée en deux parties : la collecte, le filtrage et l'enrichissement des données brutes issues des capteurs de l'habitat, d'un part, et l'analyse de ces données, d'autre part.

La première partie est donc une chaîne de médiation que l'on peut facilement développer avec Cilia. Elle a un tronc commun d'actions par contre la collecte d'informations est adaptée à l'habitat de chacun des utilisateurs, l'écosystème d'équipements étant différent d'un habitat à un autre.

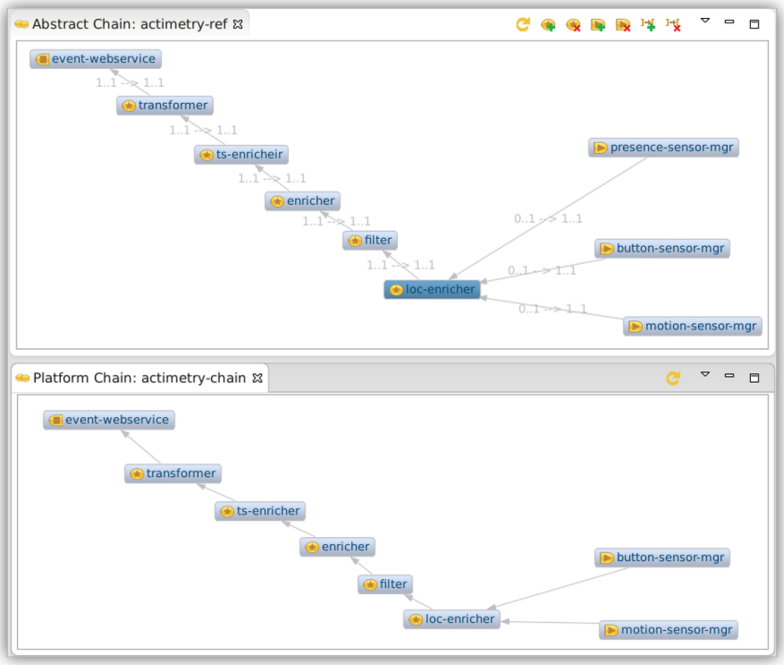

Figure 8. Capture d'écran de Cilia IDE avec deux architectures.

La Figure 8 montre un exemple pour l'application d'actimétrie. La partie haute représente l'architecture de conception qui comprend de la variabilité avec, par exemple, des cardinalités. La partie basse représente l'architecture de l'exécution. Lorsque l'on clique sur un élément, des fenêtres de propriétés permettent d'avoir l'ensemble des caractéristiques des composants. Ces deux vues permettent de détecter facilement les violations de contraintes. Les éléments qui ne respectent pas l'architecture de conception sont représentés en rouge.

6. Travaux connexes

Dans le cadre de l'informatique autonomique, les notions de modèles des besoins/de conception (Elkhodary *et al.*, 2010 ; Baresi *et al.*, 2010) et de modèles des architectures (Garlan *et al.*, 2004 ; Floch *et al.*, 2006) sont très utilisées pour permettre une auto-adaptation des systèmes grâce aux liens entre les modèles. La connaissance est centrale dans la boucle autonomique (Kephart, Chess, 2003). Dans nos travaux, nous avons fait le choix de garder dans la connaissance le modèle de conception et celui d'exécution pour vérifier la conformité entre ces modèles. La difficulté majeure est de savoir ce qui doit être représenté de manière explicite au niveau de la connaissance. Evidemment, plus il y a d'informations explicites, plus il est facile d'implanter et de maintenir une boucle autonomique basée sur cette connaissance (Lalanda *et al.*,

2013). Un avantage de cette approche architecturale est que le modèle architectural peut être utilisé pour vérifier que l'intégrité du système est préservée lors des modifications. En effet, les évolutions sont prévues et appliquées dans un premier temps au niveau du modèle, ce qui permet de montrer quel sera l'état résultant du système et ainsi en déduire les violations éventuelles de contraintes (Oreizy *et al.*, 1998).

Une des limites de ces approches existantes est le manque d'expressivité possible au niveau des modèles de conception. Les modèles sont définis à base de composants, de connecteurs et de contraintes (*e.g.*, les plates-formes Acme (Garlan, Schmerl, 2002), C2/xADL (Oreizy *et al.*, 1998)) mais il y a relativement peu de variabilité supportée. De plus, une autre limite à l'utilisation de ces plates-formes est le manque d'outils facilitant la gestion complète du cycle de vie. Comme nous l'avons montré, il y a une cohérence entre les architectures de la conception et de l'exécution. Cependant, il n'existe pas aujourd'hui d'outils permettant aux concepteurs, développeurs, administrateurs de travailler sur un même système. L'administrateur doit se baser sur sa connaissance du système et sur les documents de conception qu'on lui fournit.

7. Conclusion

Les éléments manipulés au niveau de l'approche à services ont un niveau de granularité qui facilite le maintien du lien entre les modèles de la conception et de l'exécution. C'est pourquoi cette approche est utilisée aujourd'hui dans de nombreux domaines applicatifs comme pour l'informatique autonomique. De plus, l'approche à services permet de gérer l'hétérogénéité et le dynamisme ce qui est très utile dans le cadre d'environnement changeant et imprévisible.

Dans cet article, nous avons proposé une approche qui permet de simplifier l'administration des applications pervasives en introduisant les modèles de la conception et de l'exécution au niveau de la connaissance d'une boucle autonomique. Pour faciliter la tâche de l'administrateur, nous avons également proposé un algorithme de conformité entre les différents modèles pour signaler si des modifications de l'architecture de l'exécution n'ont pas respecté des contraintes de conception. Ces modèles doivent donc être compréhensibles par l'administrateur et être utiles pour un gestionnaire autonomique qui automatise certaines tâches de l'administrateur. Les modèles que nous avons proposés, ont été définis de telle sorte que l'on garde une continuité entre les différentes activités (de la conception à l'exécution) tout en gardant la spécificité de chacune des étapes du cycle de vie. Cette approche a été implantée sous la forme d'une extension de la plate-forme Eclipse pour la partie modélisation (Cilia IDE). Elle a été testée avec la plate-forme logicielle Cilia qui permet de développer des chaînes de médiation dynamiques dans le cadre du projet FUI Medical.

Bibliographie

Baresi L., Pasquale L., Spoletini P. (2010). Fuzzy Goals for Requirements-Driven Adaptation. In *RE 2010, 18th IEEE International Requirements Engineering Conference, Sydney,*

New South Wales, Australia, September 27 - October 1, 2010, p. 125–134. IEEE Computer Society.

Elkhodary A. M., Esfahani N., Malek S. (2010). FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems. In G. Roman, K. J. Sullivan (Eds.), *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, p. 7–16. New York, NY, USA, ACM.

Escoffier C., Chollet S., Lalanda P. (2014). Lessons learned in building pervasive platforms. In *11th IEEE Consumer Communications and Networking Conference, CCNC 2014, Las Vegas, NV, USA, January 10-13, 2014*, p. 7–12. IEEE.

Favre J.-M. (2004). Foundations of Model (Driven) (Reverse) Engineering: Models - Episode I: Stories of the Fidus Papyrus and of the Solarus. In *Postproceedings of Dagsthul Seminar on Model Driven Reverse Engineering*.

Floch J., Hallsteinsen S., Stav E., Eliassen F., Lund K., Gjorven E. (2006, March). Using Architecture Models for Runtime Adaptability. *Software, IEEE*, vol. 23, n° 2, p. 62-70.

Garlan D., Cheng S.-W., Huang A.-C., Schmerl B., Steenkiste P. (2004, Oct). Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, vol. 37, n° 10, p. 46-54.

Garlan D., Schmerl B. R. (2002). Model-based adaptation for self-healing systems. In *Proceedings of the First Workshop on Self-Healing Systems, WOSS 2002, Charleston, South Carolina, USA, November 18-19, 2002*, p. 27–32. ACM.

Kephart J. O., Chess D. M. (2003). The vision of autonomic computing. *Computer*, vol. 36, n° 1, p. 41–50.

Lalanda P., Bourcier J., Bardin J., Chollet S. (2010, février). Building smart home pervasive services. In *Smart Home Systems*, p. 1-17. InTech.

Lalanda P., McCann J. A., Diaconescu A. (2013). *Autonomic Computing - Principles, Design and Implementation*. Springer–Verlag.

Morand D., García I., Lalanda P. (2011). Autonomic enterprise service bus. In *IEEE 16th conference on emerging technologies & factory automation, ETFA 2011, Toulouse, France, september 5-9, 2011*, p. 1–8. IEEE.

OMG. (2002, avril). *Meta-Object Facility (MOF^{TM}) Specification.* (version 1.4)

Oreizy P., Medvidovic N., Taylor R. N. (1998). Architecture-based runtime software evolution. In *International conference on software engineering*, p. 177–186. IEEE Computer Society.

Papazoglou M. P. (2003). Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the 4th international conference on web information systems engineering*, p. 3–12.