Architectures logicielles : contraintes d'architecture

Chouki Tibermacine

Chapitre 2

Architectures logicielles : contraintes d'architecture

Dans ce chapitre, nous présentons un concept complémentaire, mais essentiel, dans la description d'architectures logicielles, qui est celui de contrainte d'architecture. Nous expliquons le rôle précis de ces entités, et leur importance dans l'ingénierie du logiciel à objets, à composants ou à services. Nous décrivons ensuite la façon dont elles sont spécifiées et interprétées. Un architecte peut définir des contraintes architecturales puis les associer à ses descriptions d'architecture pour restreindre la structure de celles-ci et pour ultimement rendre persistant un certain niveau de qualité. Grâce à ces contraintes, il peut notamment imposer le respect d'un patron ou un style architectural afin de garantir un certain niveau de maintenabilité. L'interprétation de ces contraintes lui permettra par la suite de vérifier si ces patrons/styles sont toujours respectés après l'évolution des descriptions d'architecture. Nous ferons un état de l'art, qui se veut complet, des techniques et des langages existants pour exprimer ces contraintes. Nous présentons ensuite un certain nombre de travaux que nous avons récemment conduits, et dans lesquels nous avons développé des langages pour exprimer ces contraintes sur des architectures d'applications à objets, à composants et à services. Nous illustrons ces travaux par différents exemples de contraintes d'architecture représentant des styles et patrons connus, comme le style d'architecture *Pipe and Filter* et les patrons d'architecture *Service Façade* ou *Model-View-Controller*. Nous concluons ce chapitre par un énoncé de quelques questions ouvertes donnant lieu à des travaux de recherche en cours autour de ce concept de contrainte d'architecture.

Chapitre rédigé par Chouki TIBERMACINE.

2.1. Introduction

Au fil des années, les systèmes logiciels n'ont cessé de croître, et leurs taille et complexité sont devenues de plus en plus importantes[1]. Les architectures logicielles jouent donc un rôle prépondérant, et sont devenues un artefact central, dans le cycle de vie de ces systèmes informatiques, car elles permettent aux différents protagonistes d'avoir une idée synthétique de leur organisation. Une architecture logicielle est définie dans [BAS 12] comme « l'ensemble des structures d'un système logiciel, nécessaires pour pouvoir raisonner sur celui-ci. Elle est constituée d'entités logicielles, des relations entre elles, ainsi que des propriétés de ces entités et relations ». Cette définition met en avant le fait que cet artefact rend explicite les composants d'un système logiciel, ainsi que les dépendances entre ces composants[2]. Ceci permet d'avoir un aperçu général de l'organisation de ce système, et de raisonner sur cette organisation pour vérifier certaines propriétés, comme les attributs de qualité.

Les activités autour des architectures logicielles sont diverses et variées. Elles comprennent, entre autres, la documentation [CLE 10], l'évaluation [CLE 02] et le raisonnement [TAN 09]. L'une des activités qui a considérablement retenu l'attention des praticiens de l'ingénierie du logiciel ces dernières années est la documentation des architectures. En effet, dans la littérature et la pratique, une pléthore de modèles, langages et outils ont été proposés pour documenter une architecture logicielle. Cette documentation peut porter sur l'architecture elle-même, et dans ce cas on parle de *description d'architecture*, comme elle peut porter sur les décisions d'architecture [HAR 07, JAN 05, KRU 09, TYR 05] ou sur les raisons (*rationale*, en anglais) de ces décisions [TAN 05].

Documenter les décisions d'architecture est une activité importante dans les processus de développement des logiciels [KRU 09]. En effet, ce type de documentation permet, entre autres, de limiter l'évaporation [BOS 04] de la connaissance architecturale [BOE 08]. Il existe une multitude de modèles pour définir ce type de documentation [FAL 11]. Ces modèles incluent à la fois des spécifications textuelles et des spécifications plus ou moins formelles (interprétables de façon automatique par des outils). Celles-ci incluent, entre autres, la description de la décision elle-même, de son état et des décisions alternatives. Parmi les descriptions les plus importantes que l'on rencontre dans la documentation d'une décision d'architecture, nous retrouvons les contraintes d'architecture.

Une contrainte d'architecture représente la spécification d'une condition que doit respecter une description d'architecture afin de satisfaire une décision d'architecture.

---

1. Voir, par exemple, l'évolution de JUnit : http://edmundkirwan.com/general/junit.html

2. Le terme « Composant » est utilisé au sens le plus large, c'est-à-dire un élément d'architecture qui constitue un système. Il ne s'agit pas des composants au sens de l'ingénierie de logiciels à base de composants (CBSE).

Cette spécification doit être définie avec un langage permettant son interprétation de façon automatique. Par exemple, un architecte peut prendre comme décision d'utiliser le patron MVC (*Model-View-Controller* [REE 79]). Une contrainte d'architecture permettant de vérifier le respect de ce patron dans une description d'architecture consisterait alors à vérifier, entre autres, qu'il n'y ait pas de dépendances entre les composants représentant le modèle et ceux représentant la vue.

Ce type de contraintes n'est pas exprimable uniquement dans la phase de conception, accompagnant par exemple des diagrammes de classes UML. Il est tout à fait possible de les définir en phase d'implémentation. En effet, nous pouvons envisager d'écrire et de vérifier ce type de contraintes sur du code, dans lequel nous pouvons facilement repérer les descriptions d'architecture, comme dans certaines applications à objets, ou dans les applications à base de composants ou à services. La spécification des contraintes en phase d'implémentation permet, au-delà de leur vérification statique, de les interpréter dynamiquement. Il devient donc possible de vérifier leur violation, si à l'exécution de l'application, l'architecture de celle-ci évolue.

Si l'on prend par exemple le cas de la contrainte d'architecture vérifiant une partie du patron MVC, et exprimée sur une application à objets, nous pourrons spécifier une condition stipulant le fait que les objets marqués (dont les classes ont été stéréotypées, si l'on est en phase de conception en UML, ou annotées, si l'on est en phase d'implémentation en Java, par exemple) comme entités du modèle ne doivent pas disposer de références vers les objets marqués comme entités de la vue.

De nombreux langages ont été développés pour spécifier ce type de contraintes. Ceux-ci sont principalement utilisés dans la phase de conception, et sont souvent associés aux langages de description d'architecture [MED 00]. Il existe cependant un certain nombre de langages qui sont utilisés en phase d'implémentation. Un état de l'art sur ces différents langages est présenté dans la section suivante de ce chapitre.

Nous introduisons dans les sections 2.3 à 2.5 un langage que nous avons développé dans le passé, qui s'appelle ACL : *Architecture Constraint Language* [TIB 10]. Nous explorons l'utilisation de ce langage dans des contextes différents de celui pour lequel il a été développé à l'origine, qui est celui des applications à composants[3]. Nous montrons comment ce langage pourrait être utilisé pour écrire des contraintes d'architecture sur des applications à objets, décrites en phase de conception en UML, puis en phase d'implémentation en Java. Nous expliquons également l'utilisation de ce langage avec des applications à services, décrites en phase d'implémentation avec BPEL (*Business Process Execution Language*). Tout cela est présenté dans les sections 2.3 à 2.5.

3. Ici, le sens du terme "Composant" est celui utilisé dans l'ingénierie de logiciels à base de composants (CBSE).

Nous concluons ce chapitre par un énoncé de l'apport de ce travail au domaine des architectures logicielles. Nous clôturons ensuite la discussion par un exposé de nos travaux de recherche en cours autour de ce concept de contrainte d'architecture.

2.2. Etat de l'art

Nous distinguons dans cet état de l'art deux sortes de langages : les langages utilisés pour spécifier des contraintes d'architecture en phase de conception, et qui ont été proposés conjointement, ou directement intégrés, à des langages de description d'architecture, et les langages utilisés en phase d'implémentation.

2.2.1. *Expression de contraintes d'architecture en phase de conception*

Nous présentons l'expression des contraintes d'architecture en phase de conception en deux temps. D'abord, nous exposons les langages et méthodes d'expression de ces contraintes dans les langages de description d'architecture (ADL). Ensuite, nous montrons différentes utilisations du langage OCL pour spécifier ce type particulier de contraintes.

2.2.1.1. *Expression de contraintes d'architecture dans les ADL*

Dans [MED 00], Medvidovic et Taylor proposent un *framework* de classification des langages de description d'architecture développés jusqu'en 1998-2000. Parmi les critères de classification, nous retrouvons les contraintes d'architecture. Dans cet article, les auteurs désignent les contraintes d'architecture, présentées dans ce chapitre, comme les « invariants programmables » spécifiés lors de la modélisation des configurations d'architecture. Seuls quelques langages fournissent cette possibilité. Ces langages sont Aesop [GAR 94], SADL [MOR 95, MOR 97] et Wright [ALL 97].

Aesop permet d'écrire des « contraintes de style » ou de « topologie » (appelées aussi par ces auteurs « règles de configuration ») pour forcer une certaine organisation de l'architecture afin de respecter un style d'architecture [SHA 96], comme le *Pipe and Filter* ou le « client/serveur ». Ces contraintes sont spécifiées sous la forme d'implémentations de méthodes dans des classes abstraites[4] C++ représentant les types d'architecture (*Filter*, *Server*, etc.). Ces classes héritent de toute une interface de programmation (ensemble de fonctions), appelée FAM (*Fable Abstract Machine*). Celle-ci permet entre autres d'ajouter ou supprimer des ports aux composants, ou mettre en place des connecteurs. Ces classes doivent être spécialisées (par sous-typage) pour

4. Même si dans cet ADL, un architecte écrit du code, il n'est toutefois pas considéré en phase d'implémentation. Comme il ne fait qu'écrire des spécifications d'architectures, il est considéré encore en phase de conception. L'implémentation des fonctionnalités fournies par les composants de l'architecture n'est donc pas définie avec ce langage.

créer des composants, des connecteurs ou des configurations respectant le style introduit par ces classes. Toute description d'une nouvelle architecture (ajout de composants, de ports, de connecteurs, etc.) respectant un style passerait par la vérification de ces contraintes en invoquant les méthodes qui les implémentent. Un exemple de contrainte en Aesop, extrait de [GAR 94], est donné dans le listing ci-dessous :

```
1  fam_bool pf_source::attach(fam_port p) {
2    if (!fam_is_subtype(p.fam_type(),PF_WRITE_TYPE))
3    {
4      return false;
5    }
6    else
7    {
8      return fam_port::attach(p);
9    }
10 }
```

Dans cette contrainte, le port reçu en paramètre de la fonction de connexion `attach` est vérifié. Si son type est un sous-type d'un type précis introduit par le style *Pipe and Filter*, alors la connexion est établie.

SADL est un ADL permettant de spécifier des contraintes (appelées par ces auteurs *Well-formedness Rules*) permettant de respecter des styles d'architecture. Il introduit une syntaxe pour exprimer des prédicats dans la logique du premier ordre restreignant les types des éléments constituant une description ou un style d'architecture (composant, connecteur, port, etc.). Comme dans Aesop, dans SADL, des types de base sont définis pour représenter les éléments d'architecture sans aucune contrainte. Toute nouvelle description d'architecture ou style d'architecture devra introduire des sous-types (par héritage) avec d'éventuelles contraintes. Celles-ci sont vérifiées lors de l'interprétation des primitives d'instanciation de composants et connecteurs ou de liaison de ces derniers. Une contrainte portant sur les connecteurs du style *Dataflow*, extraite de [MOR 97], est présenté ci-dessous :

```
1 connects_argtype_1: CONSTRAINT =
2   (/\ x)(/\ y)(/\ z) [ Connects(x, y, z) => Dataflow_Channel(x) ]
3 connects\_argtype_2: CONSTRAINT =
4   (/\ x)(/\ y)(/\ z) [ Connects(x, y, z) => Outport(y) ]
5 connects\_argtype_3: CONSTRAINT =
6   (/\ x)(/\ y)(/\ z) [ Connects(x, y, z) => Inport(z) ]
```

Cette contrainte stipule qu'une connexion entre composants dans ce style d'architecture implique trois éléments d'architecture, un canal (x), qui doit être de type `Dataflow_Channel`, un port (y) de type `Outport` et un autre port (z) de type `Inport`.

Wright fait suite au langage Aesop, qui a été développé dans la même équipe de recherche. Il permet de décrire de façon formelle des architectures, et notamment les connecteurs entre composants dans ces descriptions d'architecture. Il s'appuie sur une notation d'algèbre de processus, une variante de CSP [HOA 78], pour modéliser le comportement des ports et des connecteurs. Ce langage permet d'exprimer

des contraintes pour formaliser des styles d'architecture. La contrainte suivante, par exemple, stipule que la configuration doit avoir une topologie en étoile :

$$\begin{aligned}&\exists center : Components \bullet \\ &\quad \forall c : Connectors \bullet \exists r : Role; p : Port \mid ((center, p), (c, r)) \in Attachments \\ &\wedge \forall c : Components \bullet \exists cn : Connectors; r : Role; p : Port \\ &\qquad \mid ((c, p), (cn, r)) \in Attachments\end{aligned}$$

Le premier prédicat indique qu'il existe un composant ("`center`"), parmi tous les composants de la description d'architecture (le mot-clé `Components`) qui est attaché à tous les connecteurs de la description. Le second prédicat indique que tous les composants doivent être attachés à un connecteur. Ainsi, cette contrainte garantit que tout composant est connecté au composant représentant le centre de l'étoile.

En plus de ces langages, l'ADL Acme [GAR 00] fournit un langage de contraintes séparé (qui n'a pas été cité avec les autres dans la classification de Medvidovic et Taylor [MED 00]) nommé Armani [MON 01]. Ce langage permet d'exprimer ce que les auteurs appellent des « règles de conception », ce qui correspond à des contraintes d'architecture. Armani est un langage permettant d'écrire des prédicats dans la logique du premier ordre. Il introduit également un certain nombre de fonctions pour vérifier, entre autres, le type d'un élément d'architecture (`satisfiesType(e:Element,t:Type):boolean`), ou pour tester des propriétés de graphes (par exemple, `attached(con:Connector,comp:Component):boolean`). Il permet la définition de deux types de prédicats, les « heuristiques » et les « invariants ». Ces deux entités sont définies de la même manière, sauf que les heuristiques ne sont pas destinées à des vérifications de type. L'exemple ci-dessous représente un invariant et une heuristique spécifiés en Armani :

```
1 Invariant Forall c1,c2 : component in sys.Components |
2             Exists conn : connector in sys.Connectors |
3               Attached(c1,conn) and Attached(c2,conn);
4 Heuristic Size(Ports) <= 5;
```

L'invariant précise que tous les composants du système doivent être connectés entre eux. La configuration forme donc un graphe fortement connexe. L'heuristique indique que le nombre de tous les ports doit être inférieur ou égal à cinq.

FScript[5] est un langage de script pour la reconfiguration d'architectures à base de composants Fractal [BRU 04]. Sa syntaxe est proche des langages de programmation. Il se base sur un langage de navigation dans les descriptions d'architecture Fractal qui s'appelle FPath. Ce dernier a une syntaxe inspirée du langage xPath, langage de navigation dans les documents XML. Ce langage permet d'exprimer des contraintes

5. Un tutoriel de ce langage est disponible dans le dépôt SVN suivant : svn://forge.objectweb.org/svnroot/fractal/tags/fscript-2.0.

d'architecture paramétrables. Un exemple tiré du tutoriel de ce langage est donné ci-dessous :

```
1 -- Tests whether the client interface $itf is bound to
2 -- (an interface of) component $comp.
3 function bound-to(itf, comp) {
4   servers = $itf/binding::*/component::*;
5   return size($servers & $comp) > 0;
6 }
```

La contrainte prend la forme d'une fonction, ce qui veut dire dans FScript que le script est sans effet de bord sur la description d'architecture ; le script utilise l'introspection seulement[6]. Cette fonction admet deux arguments, une interface requise (cliente dans la terminologie du modèle de composants Fractal) et un composant. La contrainte permet de tester si l'interface requise reçue en premier argument (`$itf`) est connectée au composant reçu en deuxième argument (`$comp`).

L'expression de la ligne 4 permet de récupérer l'ensemble des composants (stockés dans une variable `servers`) ayant une interface fournie (serveur dans la terminologie Fractal) connectée à l'interface requise `$itf`. L'expression de la ligne suivante réalise l'intersection (opérateur `&`) entre cet ensemble de composants (`servers`) et le composant `$comp` reçu en paramètre. Si l'intersection est non vide, la fonction retourne la valeur « vrai ». Sinon, elle retourne « faux ».

Plus généralement, ce langage s'appuie sur un langage de requête FPath pour réaliser l'introspection. Il fournit un certain nombre d'opérateurs ensemblistes : intersection (&), union (|), taille (`size`), etc. La richesse de ce langage réside dans l'utilisation d'une syntaxe proche de xPath pour écrire des requêtes complexes, et la possibilité d'appeler des fonctions déjà spécifiées précédemment à l'intérieur de ces requêtes.

Le langage AADL [FEI 12] (*Architecture Analysis and Design Language*) est un ADL permettant la description d'architectures (logicielles et matérielles) à base de composants pour les systèmes embarqués et temps réel. Un langage du nom de REAL [GIL 10] (*Requirements and Enforcements Analysis Language*) a été proposé comme langage de contraintes pour AADL. Il permet d'exprimer des contraintes sous la forme de théorèmes s'appliquant sur une collection d'éléments d'architecture (des composants ou des connexions, par exemple). Dans le listing suivant [GIL 10], la contrainte s'applique aux instances de composants de type `Thread` et vérifie leur périodicité. Leur propriété dont le nom est `Dispatch_Protocol` doit avoir comme valeur `periodic` ou `sporadic`.

6. Il existe une autre forme de scripts dans FScript appelée « Action », qui permet de modifier la description d'architecture (effectuer de l'intercession), FScript étant un langage de reconfiguration d'architecture et non un langage de contraintes.

```
1 theorem task_periodicity
2 foreach t in Thread_Set do
3   check((Get_Property_Value(t,"Dispatch_Protocol") = "periodic")
4         or
5         (Get_Property_Value(t,"Dispatch_Protocol") = "sporadic"));
6 end task_periodicity
```

Les contraintes dans ce langage utilisent un mécanisme d'introspection pour, entre autres, obtenir l'ensemble des instances de composants (de type `Thread`, `Data`, etc.), pour obtenir les valeurs de leurs propriétés, pour tester les accès ou les connexions entre instances de composants (`Is_Accessing_To(...)`, `Is_Bound_To(...)`, `Is_Subcomponent_Of(...)`, etc.). Ce langage propose des itérateurs (`foreach`) et des opérations ensemblistes (`Cardinal`, `Max`, etc.) et des opérateurs booléens et de comparaison.

2.2.1.2. *Expression de contraintes d'architecture avec OCL*

Le langage OCL [OMG 12] (*Object Constraint Language*) est le standard de l'OMG (*Object Management Group*) pour exprimer des contraintes sur des modèles UML. L'objectif de ce langage est de fournir un moyen aux développeurs de spécifier des conditions pour affiner la sémantique de leurs modèles. Ce langage de contraintes a été initialement proposé pour spécifier des conditions sur des aspects fonctionnels et pas architecturaux. Par exemple, dans un diagramme de classe, où nous retrouvons une classe `Employé`, ayant un attribut `age` de type entier, une contrainte OCL représentant un invariant sur cette classe peut forcer les valeurs de cet attribut pour qu'ils soient toujours compris dans l'intervalle 16 à 70. Cette contrainte sera vérifiée sur toutes les instances du modèle UML, et donc sur toutes les instances de la classe `Employé`.

Il existe toutefois d'autres utilisations du langage OCL, qui sont au niveau d'un métamodèle et non pas d'un modèle. Ceci permet d'obtenir des contraintes de niveau architectural comme celles discutées dans ce chapitre. Nous listons ci-dessous quelques exemples de spécifications dans lesquelles le langage OCL est utilisé pour exprimer des contraintes d'architecture.

1) La spécification du langage UML [OMG 11] : dans cette spécification, le métamodèle du langage UML est présenté, et afin d'affiner la sémantique de ce métamodèle, des contraintes OCL lui ont été associées. Celles-ci naviguent dans ce métamodèle et imposent des conditions sur les types des métaclasses dans ce métamodèle, les valeurs de leurs attributs, le nombre de leurs instances, leurs interconnexions, etc. Dans la figure 2.1, extraite de [OMG 11], nous montrons un petit extrait du métamodèle UML dans lequel sont spécifiées partiellement les associations entre classes.

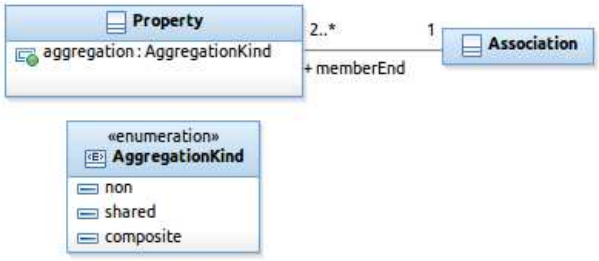

Figure 2.1. *Extrait du métamodèle UML*

Dans cette figure, il est indiqué qu'une association a deux ou plusieurs extrémités, qui sont des instances de la métaclasse `Property`. Chacune de ces instances a un attribut `aggregation`, qui peut recevoir les trois valeurs suivantes : `none` (pas d'agrégation), `shared` (agrégation au sens UML) et `composite` (composition au sens UML). Voici un exemple de contrainte [OMG 11] écrite en OCL sur ce bout de métamodèle.

```
1 -- Only binary associations can be aggregations
2 context Association inv:
3 self.memberEnd->exists(aggregation <> Aggregation::none)
4 implies self.memberEnd->size() = 2
```

La première ligne est un commentaire précisant le rôle de la contrainte. La ligne 2 désigne le contexte de la contrainte. Celui-ci représente la métaclasse dans le métamodèle sur laquelle la contrainte s'applique. Celle-ci sera donc évaluée sur toutes les instances de cette métaclasse. Dans notre exemple, il s'agit de la métaclasse `Association` de la figure 2.1. Dans la ligne 3, la contrainte navigue à travers l'association entre les métaclasses `Association` et `Property` dans le métamodèle pour obtenir les instances de `Property` liées à l'instance de `Association` sur laquelle la contrainte est évaluée. La contrainte vérifie ensuite s'il existe au moins une instance parmi ces instances de `Property` dont la valeur de l'attribut `aggregation` est différente de `none` (vérifier s'il existe au moins une association qui est une agrégation ou composition). Dans la ligne suivante, la contrainte vérifie que, dans ce cas, le nombre d'extrémités de l'association doit être égal à deux (association binaire).

2) Les profils UML : un profil UML est une extension standard au langage UML pour traiter un domaine particulier : les systèmes temps réel, les télécommunications, les systèmes sur puces, les tests de systèmes, etc[7].

Prenons l'exemple du profil UML pour CORBA et les composants CORBA (CCCMP [OMG 08]). Un extrait du métamodèle implémenté par ce profil est donné dans la figure 2.2.

Dans ce métamodèle, il est spécifié qu'une définition de composant déclare un certain nombre de ports, qui peuvent être des *receptacles* (ports requis de type

7. Voir une liste complète des profils UML adoptés par l'OMG sur le site suivant : www.omg.org/technology/documents/profile_catalog.htm.

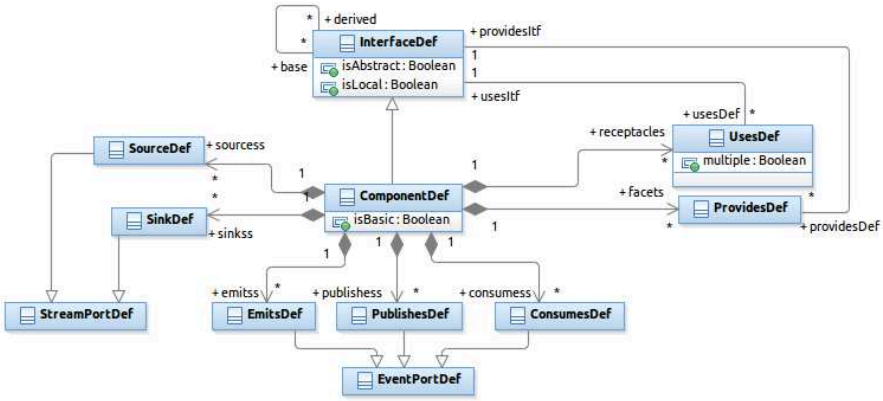

Figure 2.2. *Extrait du métamodèle du profil UML pour CORBA et CCM*

`UsesDef`), des *facets* (ports fournis de type `ProvidesDef`), des ports de type événements (`EmitsDef`, `PublishesDef` et `ConsumesDef`) ou des ports de type flux (`SourceDef` et `SinkDef`).

La contrainte suivante stipule qu'une définition de composant de base ne doit pas déclarer des ports ou hériter des autres définitions de composants :

```
1 context ComponentDef inv:
2 self.isBasic implies
3 facets->isEmpty and receptacles->isEmpty and
4 emitss->isEmpty and publishess->isEmpty and consumess->isEmpty and
5 sinkss->isEmpty and sourcess->isEmpty and
6 base->isEmpty
```

Les lignes 3 à 5 indiquent qu'il ne faut pas qu'il y ait des ports des différents types cités ci-dessus. La dernière ligne précise que la définition de composant ne doit pas être dérivée d'une autre définition de composant (le rôle `base` étant hérité de la métaclasse `InterfaceDef`). Dans ce cas, la définition de composant Corba de base peut seulement posséder une interface `HomeDef` pouvant déclarer des opérations de type `FactoryDef` ou `FinderDef` (qui ne sont pas montrés dans le métamodèle de la figure 2.2 pour des raisons de simplicité).

3) La description des architectures logicielles en UML proposée par Medvidovic *et al.* dans [MED 02] : les auteurs présentent dans cet article trois méthodes différentes pour utiliser UML en tant que langage de description d'architecture. La première méthode consiste à utiliser UML tel quel. La deuxième préconise la mise en place de contraintes d'architecture en OCL sur les métaclasses UML. La troisième méthode propose d'étendre le langage UML. C'est la deuxième méthode qui nous semble la plus intéressante à présenter ici. Les auteurs exposent un ensemble de contraintes OCL s'appliquant sur le métamodèle UML pour respecter les restrictions imposées par certains ADL dans la description d'architectures à base de composants. Les auteurs ont

choisi trois ADL qu'ils ont jugés suffisamment représentatifs, qui sont C2 [MED 96], Rapide [LUC 95] et Wright. Ils ont plus précisément exposé des profils UML de ces ADL, parce qu'ils ont introduit à la fois les contraintes, mais aussi les stéréotypes et les valeurs marquées qui correspondent à ces ADL. Pour le langage C2, par exemple, les auteurs introduisent un stéréotype nommé `C2Component` comme extension de la métaclasse Class. Une contrainte d'architecture attachée à ce stéréotype indique que les composants C2 doivent implémenter exactement deux interfaces, qui doivent être stéréotypées `C2Interface`, l'une d'elle ayant une position au-dessus du composant et l'autre en dessous. Cette contrainte est exprimée en OCL comme suit [MED 02] :

```
1 self.interface->size = 2 and
2 self.interface->forAll(i i.stereotype = C2Interface) and
3 self.interface->exists(i i.c2pos = top) and
4 self.interface->exists(i i.c2pos = bottom)
```

Les auteurs ont au préalable introduit une énumération (utilisée dans les lignes 3 et 4) pour les positions des interfaces (`top` et `bottom`), et les stéréotypes `C2Component`, contexte de cette contrainte, et `C2Interface` utilisé dans la ligne 2.

Il est à noter ici que les contraintes qui sont définies de cette façon, au niveau du métamodèle, s'appliquent à toutes les instances de celui-ci, et donc sur tous les modèles (description d'architectures) définis en utilisant ce métamodèle. Ces contraintes représentent alors des conditions assez fortes, faisant partie de la spécification du langage de description d'architecture. Elles s'appliqueront à toute description d'architecture définie avec ce langage. Ce ne sont donc pas des contraintes d'architecture qui seront vérifiées sur une description d'architecture particulière (celle d'une application X).

2.2.2. *Expression de contraintes d'architecture en phase d'implémentation*

En phase d'implémentation, le développeur est confronté à deux scénarios : i) écrire manuellement le code (les programmes) correspondant à l'architecture décrite lors de la phase de conception, ou ii) générer automatiquement des squelettes de code à partir des descriptions d'architecture et compléter ensuite les parties manquantes. Tout dépend du langage utilisé dans la phase de conception et des outils qui lui sont associés. Dans le scénario idéal, les contraintes accompagnant la description d'architecture sont elles aussi transformées en code ou en contraintes vérifiables sur du code. Malheureusement, à notre connaissance, aucun travail dans la littérature n'a été développé sur ce sujet. Récemment, nous nous sommes penchés sur cette question (voir la conclusion dans la section 2.6), mais le travail que nous avons réalisé ne relève pas du sujet de ce chapitre et ne sera donc pas détaillé ici.

Dans cette section, nous nous focalisons sur les travaux qui ont proposé des langages et des outils pour l'expression de contraintes sur des programmes. Il est à noter ici que les contraintes d'architecture portent parfois sur une granularité assez fine dans

la description d'architecture, comme les modificateurs d'accès des attributs ou les paramètres des méthodes. Ceci est lié au style d'architecture sous-jacent au langage de programmation. Par exemple, les langages de programmation par objets offrent un style d'architecture pour les applications constitué de concepts comme les classes, les prototypes, les attributs, les méthodes, etc. Les contraintes d'architecture sur les applications à objets doivent donc nécessairement porter sur ce niveau de granularité (sur ces concepts et leurs propriétés). Ces contraintes ne relèvent pas du niveau fonctionnel (contraintes sur les valeurs des attributs, par exemple). Elles relèvent plutôt du niveau structurel de ces applications, et donc architectural.

Boris Bokowski dans [BOK 99] a proposé un *framework* qui s'appelle *CoffeeStrainer*. Ce *framework* permet la spécification en Java de contraintes et leur vérification statique sur des programmes Java. Il permet d'exprimer des conditions sur des programmes pour implémenter de bonnes pratiques de conception, comme l'encapsulation ou l'appel systématique d'une méthode de la superclasse lorsque cette méthode est redéfinie dans une sous-classe. Ces contraintes sont définies sous la forme de commentaires particuliers (délimités par la suite de symboles `/*/`) pouvant être placés partout dans une classe. A l'intérieur de ces commentaires, des méthodes sont définies pour implémenter la contrainte en s'appuyant sur une couche réflexive (de métaprogrammation) fournie par CoffeeStrainer. Ces méthodes sont invoquées en analysant l'arbre syntaxique représentant la classe dans laquelle la contrainte a été définie, et donc sur laquelle elle va être vérifiée. Par exemple, la contrainte suivante permet de s'assurer que tous les attributs d'une classe sont privés (principe d'encapsulation des données dans la programmation par objets).

```
1 interface PrivateFields {
2   /*/ public void checkField(Field f) {
3     if(!f.isPrivate()) {
4       reportError(f, "field is not declared private");
5     }
6   }
7 /*/
8 }
```

Si une classe implémente cette interface, la méthode `checkField(Field f)` (ligne 2) sera appelée autant de fois qu'il y a d'attributs dans la classe, en lui passant en argument à chaque fois l'objet `Field` représentant l'attribut. Cette contrainte signale une erreur en appelant une méthode héritée qui s'appelle `reportError(...)` si un des attributs n'est pas privé.

Les contraintes exprimables avec CoffeeStrainer peuvent descendre à un niveau de détail assez fin, en vérifiant par exemple le type des instructions qui sont dans les méthodes (est-ce une invocation de méthode, une affectation, etc. ?). Ce niveau de détail n'est pas utile pour l'expression de contraintes d'architecture, qui portent sur des aspects d'une granularité plus importante (attributs et méthodes déclarées, classes étendues, interfaces implémentées, etc.). Pour réaliser cela, `java.reflect`

suffit amplement. Cependant, cette librairie n'est exploitable qu'à l'exécution des programmes, alors que parfois les contraintes d'architecture peuvent être évaluées statiquement. Nous reviendrons sur ce point plus loin dans ce chapitre.

Un langage plus ancien, nommé CCEL (*C++ Constraint Expression Language*), a été proposé dans [CHO 93] pour la spécification de contraintes portant sur la structure des programmes C++. Les contraintes exprimables avec ce langage portent uniquement sur les déclarations dans les programmes (déclarations de classe, de fonctions et de variables), ce qui correspond au type de contraintes que l'on voudrait exprimer. Cependant, ce langage introduit une nouvelle syntaxe, certes inspirée de C++, mais qui nécessite un apprentissage spécifique.

Dans [KLA 96], les auteurs proposent un langage nommé CDL (*Category Description Language*) permettant d'exprimer dans la logique du premier ordre des contraintes d'architecture sous la forme de formules portant sur des arbres représentant la syntaxe de programmes (*parse trees*). Ce langage permet de spécifier des contraintes indépendamment de tout langage. En revanche, pour pouvoir intégrer ces contraintes à un langage concret, ce langage doit être étendu pour permettre l'annotation, avec les noms des contraintes, des programmes écrits avec ce langage et sur lesquelles les contraintes doivent être vérifiées.

DCL (*Dependency Constraint Language* [TER 09]) est un langage de spécification de contraintes d'architecture pour des applications à objets. Ce langage permet d'exprimer des contraintes qui sont vérifiées de façon statique, c'est-à-dire sur le code source des applications à objets, avant leur exécution. *dclchek* est l'outil fourni par les auteurs de ce langage pour vérifier les contraintes DCL sur du code Java.

Ce langage permet tout d'abord d'indiquer les parties de l'application à objets qui représentent des modules (un module = un ensemble de classes et d'interfaces). Ensuite, les contraintes spécifient des conditions que doivent respecter ces modules. Les auteurs de ce langage indiquent qu'il est possible de spécifier deux catégories de contraintes : les « divergences » et les « absences ». Par « divergence », les auteurs veulent dire que le code source d'une application à objets comporte une dépendance qui ne respecte pas la contrainte. Par « absence », les auteurs veulent dire que le code source ne comporte pas une dépendance. Les « divergences » peuvent être de deux sortes :

1) des contraintes du type : « Seules les classes du module A peuvent dépendre des types dans le module B ». Par le mot « dépendre », les auteurs veulent dire accéder (une classe du module A peut accéder aux membres publiques d'une classe du module B), déclarer, créer, implémenter, hériter, etc ;

2) des contraitnes du type : « Les classes du module A ne peuvent pas dépendre des types dans le module B ».

Des exemples de contraintes sont donnés ci-dessous :

```
1 only A can-access B
2 only A can-declare B
3 only A can-useannotation B
4 A cannot-create B
5 A cannot-implement B
6 A cannot-throw B
```

La contrainte de la ligne 2 indique que seules les classes du module A peut déclarer des variables de types définis dans le module B. A l'opposé, la contrainte de la ligne 6 stipule que les classes du module A ne peuvent pas lever des exceptions dont les types sont définis dans le module B.

Les « absences » sont des contraintes du style : « Les classes déclarées dans le module A doivent dépendre des types déclarés dans le module B ». Des exemples de cette catégorie de contraintes sont montrés ci-dessous :

```
1 A must-extend B
2 A must-throw B
3 A must-useannotation B
```

La première contrainte indique que les classes déclarées dans le module A doivent étendre une classe déclarée dans le module B. La seconde contrainte précise que toutes les méthodes des classes du module A doivent lever des exceptions dont les types sont déclarés dans le module B. La dernière contrainte impose l'utilisation dans toutes les classes du module A d'au moins une annotation déclarée dans le module B.

L'expression de contraintes d'architecture est assez limitée dans DCL. En effet, avec ce langage nous pourrons imposer des conditions sur les dépendances entre les types, mais il est impossible, par exemple, d'écrire des contraintes restreignant le nombre d'éléments d'architecture (nombre d'instances, d'attributs ou d'opérations, par exemple), nécessitant des compositions logiques complexes ou encore requérant une analyse dynamique des programmes (pour vérifier par exemple si la valeur d'un attribut ne correspond pas à une référence vers un objet d'un certain type « interdit »).

Un certain nombre de travaux dans la littérature se réfèrent aux contraintes d'architecture comme conditions sur les dépendances structurelles entre éléments de programmes. SCL (*Structural Constraint Language* [HOU 06]) est un langage de contraintes de cette famille permettant l'écriture de ce que les auteurs appellent des « intentions » de conception. SCL est un langage pour la spécification de prédicats dans la logique du premier ordre. Ce langage permet d'analyser la syntaxe de programmes, représentée sous la forme d'un graphe, à travers un certain nombre d'opérations. Par exemple, l'opération `subclasses(class("X"))` retourne l'ensemble des sous-classes de la classe « A ». L'exemple suivant, issu de [HOU 06], introduit une contrainte s'assurant que les méthodes `equals` dans les classes Java ont une signature correcte :

```
1 def Object as class("java.lang.Object")
2   for p: packages, c: classes(p), m: methods(c)
3   (
4     (name(m)="equals" & isPublic(m) & sizeof(params(m))=1)
5     =>
6     (returnType(m) = boolean & type(ith(params(m),0))=Object)
7   )
```

Cette contrainte itère sur tous les packages, et pour chacune des méthodes se trouvant dans les classes de ces packages (boucle `for` de la ligne 2), si la méthode est publique, s'appelle `equals(...)` et admet un argument (ligne 4), elle doit avoir comme type de retour `boolean` et son paramètre doit être de type `Object` (ligne 6).

Les concepteurs de ce langage ont proposé une sorte de simplification du langage OCL mais fournie comme nouveau langage ayant une syntaxe propre. Par exemple, la boucle `for` de la ligne 2 du listing précédent, s'écrirait de façon plus verbeuse en OCL en imbriquant les opérations `forAll`. Ils ont proposé aussi une adaptation directe de ce langage aux langages de programmation procéduraux et à objets statiquement typés. Pour analyser, le code source de programmes, des opérations sont donc directement intégrées au langage, comme `isPublic(...)` ou `methods(...)`. Ces opérations auraient pu être introduites comme navigation dans le métamodèle du langage de programmation concerné par l'analyse, comme nous les avons introduits dans la section précédente portant sur le langage OCL, ou comme nous allons le présenter dans la section suivante de ce chapitre. Cela a le mérite de rendre le langage de contraintes, ayant été présenté comme langage indépendant d'un langage de programmation particulier, paramétrable par ce dernier. Le langage OCL fournit cette possibilité et a le mérite d'être un langage standardisé, connu, bien outillé et facile à apprendre à et à utiliser [BRI 05].

D'autres langages proposés dans la littérature peuvent être groupés dans une même famille, celle des langages issus de Prolog. Par exemple, LogEn [EIC 08] (*Logical Ensembles*) est un langage pour exprimer des ensembles d'éléments composants des programmes, et des contraintes sur les dépendances entre ces ensembles. Ce langage est basée sur un formalisme à la Prolog, nommé DataLog [CER 89], permettant de représenter des programmes sous la forme de relations comme suit :

```
1 type(t1, 'bat.type.ObjectType')
2 type(t2, 'bat.type.IType')
3 interface(t1, t2)
```

Dans la ligne 1, nous avons une déclaration de type dont l'identifiant est `t1` et qui s'appelle `bat.type.ObjectType`. Dans la ligne 3, la relation indique que t1 déclare implémenter l'interface dont l'identifiant est `t2` (déclarée dans la ligne au-dessus).

Typiquement, les contraintes imposées par les patrons de conception [GAM 95] sont exprimables avec ce langage en désignant d'abord les différents rôles dans un

patron par des ensembles, et ensuite en spécifiant les contraintes d'appartenance ou de non-appartenance à ceux-ci, en s'appuyant sur des opérateurs logiques de conjonction ou disjonction. Par exemple, pour le patron *Factory*, les auteurs spécifient trois ensembles représentant :

– tous les éléments de programme d'une certaine application ;

– la classe `Factory` (cet ensemble a l'identifiant suivant dans le listing ci-dessous : `TypesFlyweightFactory`) ;

– les constructeurs des classes dont les objets devront être créés avec la classe `Factory` (cet ensemble a l'identifiant suivant : `TypesFlyweightCreation`).

Ceci est fait en utilisant des relations `part_of(...)` de la même façon que ci-dessus. Ensuite, la contrainte du patron *Factory* imposant la création d'objets d'un certain type qu'à travers la classe `Factory` est définie de la façon suivante :

```
1 violations(S, T, 'TypesFlyweight'):-
2   part_of(T, 'TypesFlyweightCreation'),
3   tnot(part_of(S, 'TypesFlyweightFactory')).
```

Cette contrainte s'assure que toutes les paires S et T qui sont en relation `uses(S,T)` (c'est-à-dire dépendantes les unes des autres), T appartient à l'ensemble des constructeurs, et S n'est pas la classe `Factory`. En d'autres termes, il faut que T (un constructeur) soit utilisé exclusivement à partir de T (la classe `Factory`).

Un interprète de ce langage a été intégré au processus de compilation incrémentale d'Eclipse pour vérifier en permanence les contraintes lorsque le développeur modifie le code source de ses programmes.

Les principaux atouts de ce travail sont les performances intéressantes obtenues lors de l'interprétation des contraintes (efficacité de création des ensembles et vérification des dépendances) et l'incrémentalité de l'approche (son intégration facile et efficace au processus de compilation incrémentale d'Eclipse). Les auteurs de ce travail se focalisent en revanche, exclusivement sur les dépendances structurelles entre éléments de programmes (méthodes, attributs, superclasses, etc.). Les contraintes se résument à la vérification de l'existence ou de l'absence d'une dépendance entre ceux-ci. Ce langage souffre cependant de manque d'expressivité. En effet, des contraintes complexes impliquant des navigations complexes sont difficilement exprimables dans LogEn. *Law Governed Architecture* [MIN 96, MIN 97] est une approche similaire, basée sur Prolog, pour l'expression et la vérification de contraintes de dépendances entre programmes écrits en Eiffel.

Dans [BLE 05], les auteurs présentent une approche proposant un langage propre basé sur Prolog nommé Spine pour écrire sous la forme de contraintes les patrons de conception. De la même manière que dans LogEn, une contrainte s'exprime avec

des relations de type `constructorsOf(...)` ou `isStatic(...)`. L'exemple suivant [BLE 05] représente le patron de conception *Public Singleton* :

```
1  realises('PublicSingleton', [C]) :-
2    exists(constructorsOf(C),true),
3    forAll(constructorsOf(C),
4      Cn.isPrivate(Cn)),
5    exists(fieldsOf(C),F.and([
6      isStatic(F),
7      isPublic(F),
8      isFinal(F),
9      typeOf(F,C),
10     nonNull(F)
11   ]))
```

Dans cette contrainte, on vérifie d'abord que la classe `C` a au moins un constructeur (ligne 2), que tous ses constructeurs sont privés (lignes 3 et 4), et qu'elle possède au moins un attribut qui est statique, public, `final`, dont le type est la classe C et sa valeur n'est pas nulle (lignes 5 à 10).

Ce langage se focalise exclusivement sur les patrons de conception. Basé sur un formalise à la Prolog, il a les mêmes limites que le langage LogEn.

Il existe en pratique plusieurs outils d'analyse statique de code permettent de vérifier des contraintes d'architecture. Une liste non exhaustive de ces outils comprend : Sonar, Checkstyle, Lattix, Macker, Classycle, Architexa et JArchitect. Ces outils varient considérablement dans les fonctionnalités qu'ils implémentent. Certains fournissent les moyens aux développeurs d'écrire des contraintes représentant des styles d'architecture ou des patrons de conception en s'appuyant sur un arbre syntaxique des programmes analysés (comme Checkstyle). D'autres se limitent à la spécification de restrictions de dépendances entre modules (comme Lattix). Ils proposent différentes notations, allant de l'écriture de programmes (comme dans Checkstyle) à la définition de spécifications de manière déclarative (en XML comme dans Macker) et/ou graphique (comme dans Sonar). Ces outils permettent de vérifier ces contraintes à différents niveaux du processus de développement, comme par exemple, lors de la programmation (et donc en s'appuyant sur la compilation à la volée proposée dans Eclipse entre autres), lors des *commit* dans les systèmes de gestion de version de type SVN (comme Checkstyle), ou lors des constructions de projets avec des outils comme Ant et Maven (comme Macker).

La dernière famille de langages, et pas des moindres, est composée des langages offrant la possibilité d'écrire des métaprogrammes (appelés aussi langages réflexifs). Dans ces métaprogrammes, le développeur a la possibilité d'accéder en lecture (introspection) et/ou en écriture (intercession) entre autres à la structure des programmes, qui sont réifiés sous la forme d'objets (dans les langages à objets réflexifs) ou de composants (langages à composants réflexifs, comme Compo [SPA 12]). Java est un exemple de langage de programmation à objets offrant principalement la possibilité de faire de

l'introspection, et Smalltalk est un exemple de langage de programmation à objets offrant la possibilité de faire à la fois de l'introspection et de l'intercession. Pour la spécification de contraintes d'architecture, nous n'avons besoin que de l'introspection, car les contraintes d'architecture ne font qu'analyser la structure des programmes, et n'ont pas d'effet de bord. Cependant, en utilisant ce mécanisme d'introspection, les contraintes d'architecture sont évaluées dynamiquement, c'est-à-dire à l'exécution des programmes sur lesquels les contraintes sont vérifiées. Ceci n'est pas nécessaire pour certaines catégories de contraintes d'architecture dans lesquelles une analyse statique de l'architecture suffit. Dans certains cas (patron MVC, par exemple, présenté dans la section suivante), la contrainte doit vérifier les types des instances créés à l'exécution et leurs interconnexions. Elle doit donc être évaluée dynamiquement.

Dans les sections suivantes, nous allons vous présenter des travaux que nous avons conduits ces dernières années sur la spécification de contraintes d'architecture. Nous allons vous montrer comment nous pouvons exploiter des langages existants et bien connus des développeurs, comme OCL et Java, pour exprimer ce type de contraintes sur des applications écrites dans divers paradigmes. Nous avons choisi les trois paradigmes les plus connus dans l'industrie du logiciel : le paradigme des objets, celui des composants et enfin celui des services.

2.3. Contraintes d'architecture sur des applications à objets

Les contraintes d'architecture exprimées sur des applications à objets permettent entre autres de formaliser les conditions imposées par des styles d'architecture, des patrons de conception, des dépendances entre types, ou toute sorte d'invariant impliquant une vérification de l'architecture de l'application et non son état. Dans la suite de cette section, nous présentons d'abord la spécification de ces contraintes d'architecture sur des applications à objets en phase de conception, puis la spécification de ces contraintes en phase d'implémentation.

2.3.1. *Contraintes d'architecture en phase de conception*

Nous avons choisi de vous présenter ici des contraintes exprimées avec le langage ACL [TIB 10] sur des modèles de conception d'applications à objets écrits en UML. ACL est une simplification du langage OCL. Dans ACL, seuls les invariants peuvent être exprimés, donc pas de contraintes de type `pre` ou `post` (conditions), ni d'`init`, `derive` ou `body` qui sont possibles dans OCL. Par ailleurs, dans le contexte de la contrainte, l'identifiant utilisé représente la description d'architecture sur laquelle s'applique la contrainte. Le type de l'élément identifié est une métaclasse dans un métamodèle. C'est à partir de cette métaclasse que commence la navigation pour analyser une description d'architecture pour spécifier la contrainte. Nous donnerons ci-dessous des exemples, qui illustreront mieux nos propos.

La figure 2.3 montre un extrait du métamodèle UML, obtenu à partir de la spécification de la superstructure du langage UML, version 2.4.1 [OMG 11].

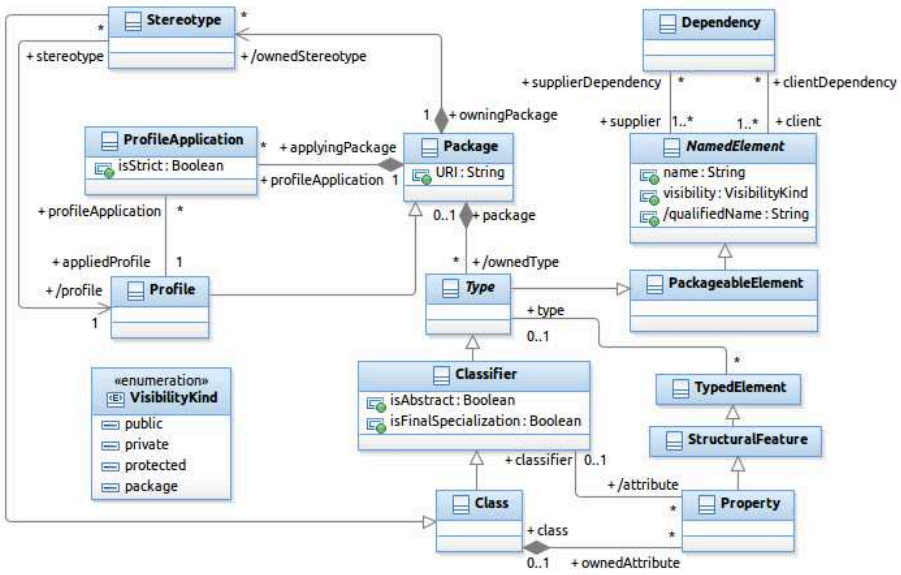

Figure 2.3. *Extrait du métamodèle UML (*Package*,* Property*,* Dependency *et* Profile*)*

Ce métamodèle se focalise sur la description des classes et notamment sur la description des packages, des attributs, des dépendances et des profils. Un package est composé d'un certain nombre de types (voir au centre de la figure 2.3). Ces derniers héritent, par le biais des métaclasses `PackageableElement` et `NamedElement`, de la capacité de participer dans des dépendances (partie à droite en haut de la figure). En bas à droite de la figure, il est indiqué qu'une classe peut déclarer des attributs, qui sont des instances de `Property`. Une classe hérite aussi de `Classifier` le fait qu'elle puisse avoir des attributs qui sont hérités ou importés (association entre `Classifier` et `Property`). La partie à gauche de la figure illustre le fait que l'on peut appliquer un profil à un package, et qu'un profil est constitué d'un certain nombre de stéréotypes.

Nous prenons l'exemple d'une contrainte représentant le patron MVC. Les dépendances entre ces différentes entités sont illustrées dans le diagramme de la figure 2.4. Pour simplifier, nous avons utilisé des packages UML pour représenter les groupements de classes jouant les trois rôles dans ce patron (le modèle, la vue et le contrôleur). Toutefois, cela ne doit pas refléter nécessairement la conception de l'application (les classes de la vue et celles du contrôleur peuvent être groupées dans un même

package). Nous supposons donc que nous disposons de trois stéréotypes nous permettant de désigner les classes dans une application qui représentent la vue (*View*), le modèle (*Model*) et le contrôleur (*Controller*).

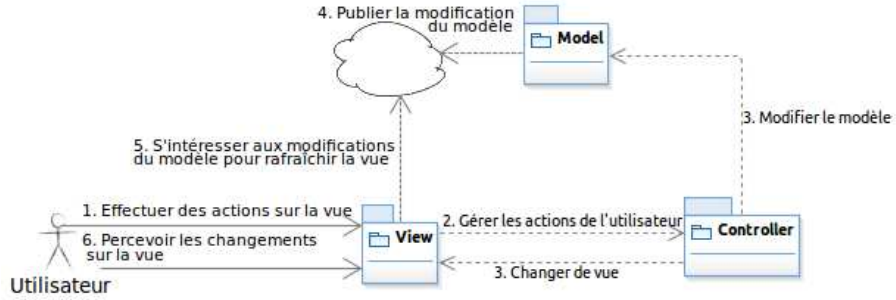

Figure 2.4. *Illustration du patron MVC*

Cette contrainte est composée de trois sous-contraintes, qui vont s'assurer que :

– les classes stéréotypées *Model* ne doivent pas déclarer de dépendances avec les classes stéréotypées *View*. Ceci permet d'avoir entre autres plusieurs vues pour un même modèle, et donc de les découpler. Dans la plupart des implémentations du MVC, les classes du modèle ne dépendent pas directement des classes de la vue, mais plutôt déclarent un attribut ayant comme valeur une collection d'objets écouteurs du changement du modèle. La vue peut jouer le rôle d'écouteur des modifications qui ont eu lieu sur le modèle, afin qu'elle se mette à jour (se rafraîchisse) ;

– les classes stéréotypées *Model* ne doivent pas avoir de dépendances avec les classes stéréotypées *Controller*. Ceci permet d'avoir plusieurs contrôleurs possibles pour le modèle ;

– les classes stéréotypées *View* ne doivent pas avoir de dépendances avec les classes stéréotypées *Model*. Les contrôleurs doivent faire le relais entre la vue et le modèle.

En utilisant ACL, on obtient les contraintes suivantes :

```
1  context MonApplication:Package inv:
2  let model:Set(Type) = self.ownedType->select(t:Type |
3    t.getAppliedStereotypes()->exists(name='Model'))
4  in
5  let view:Set(Type) = self.ownedType->select(t:Type |
6    t.getAppliedStereotypes()->exists(name='View'))
7  in
8  let controller:Set(Type) = self.ownedType->select(t:Type |
9    t.getAppliedStereotypes()->exists(name='Controller'))
10 in
11 self.ownedType->forAll(t : Type |
12   (model->includes(t)
13   implies
14   t.clientDependency.supplier->forAll(tt |
15     view->excludes(tt) and controller->excludes(tt)
16   ))
17   and
18   (view->includes(t)
19   implies
20   t.clientDependency.supplier->forAll(tt |
21     model->excludes(tt)
22   ))
23 )
```

La première ligne du listing déclare le contexte de la contrainte. Elle indique que la contrainte s'applique à tout le package de l'application ; la métaclasse Package (voir figure 2.3) étant alors le point de départ de toutes les navigations dans la suite de la contrainte. Les lignes 2 à 9 servent à collecter les ensembles de classes représentant le modèle, la vue et le contrôleur[8]. Par exemple, dans les lignes 2 et 3, nous partons du package pour rechercher les types définis dans celui-ci. Ensuite, nous sélectionnons uniquement ceux ayant comme stéréotype appliqué `Model` grâce à l'opération `getAppliedStereotypes()` (non spécifiée dans UML/OCL, mais implémentée dans RSA : *Rational Software Architect* d'IBM).

Les sous-contraintes 1 et 2, spécifiées textuellement dans la précédente énumération, sont formalisées avec ACL dans le listing précédent entre les lignes 11 et 16. Dans ces contraintes, on s'assure que si une classe est stéréotypée Model alors elle ne doit pas avoir de dépendance (en naviguant vers la métaclasse `Dependency`) avec des classes stéréotypées View ou Controller. Pour tester si une classe est stéréotypée Model, on vérifie simplement sa présence (grâce à l'opération `includes(...)` de la ligne 12 dans l'ensemble d'objets de type `Class`, nommé `model`, formé dans les lignes 2 et 3 du listing. La dernière sous-contrainte est formalisée de la ligne 18 à la ligne 22.

8. Nous simplifions ici la contrainte en supposant que les classes se trouvent dans un unique package. Si les classes sont définies dans des sous-packages ou des sous-sous-packages, il faudra dans ce cas naviguer de façon récursive dans ces derniers.

Souvent une dépendance entre deux classes se traduit par la déclaration dans la première classe d'au moins un attribut ayant comme type la deuxième classe. On peut également retrouver des paramètres dans les opérations de la première classe qui ont comme type la deuxième classe. Ou encore, dans une opération de la première classe, le type de retour est la deuxième classe. Pour simplifier, nous allons prendre en compte le premier cas : « au moins un attribut dans la première classe a comme type la deuxième classe ». C'est d'ailleurs, le cas le plus fréquent dans la réalisation d'une application selon le patron MVC. La contrainte donnée précédemment peut être raffinée comme suit, sachant que la partie allant de la ligne 1 à 10 du listing précédent ne change pas :

```
1  ...
2  self.ownedType->forAll(t : Type |
3    (model->includes(t)
4    implies
5    if t.oclIsKindOf(Classifier)
6    then
7      t.oclAsType(Classifier).attribute->forAll(a |
8        view->excludes(a.type) and controller->excludes(a.type))
9    else true
10   endif
11   )
12   and
13   (view->includes(t)
14   implies
15   if t.oclIsKindOf(Classifier)
16   then
17     t.oclAsType(Classifier).attribute->forAll(a |
18     model->excludes(a.type))
19   else true
20   endif
21   )
22 )
```

Dans cette contrainte, la vérification de la dépendance se fait sur tous les attributs définis dans les classes. Il est à noter que l'utilisation dans cette contrainte de l'opération `oclAsType(Classifier)` pour permettre la navigation en descendant dans le lien d'héritage entre `Type` et `Classifier` (conversion du type statique des objets `Type` pour naviguer ensuite vers `Property`).

Nous n'avons pas encore fini avec le patron MVC, qui ne peut être vérifié que partiellement en phase de conception. En effet, il est nécessaire de réaliser d'autres vérifications à l'exécution. Celles-ci seront expliquées dans la section suivante.

2.3.2. *Contraintes d'architecture en phase d'implémentation*

Afin de pouvoir vérifier les contraintes d'architecture précédentes en phase d'implémentation, et donc s'assurer que le code produit dans cette phase est toujours conforme aux contraintes définies en amont, nous allons expliquer comment ces

contraintes peuvent être exprimées en Java. Dans un premier temps, nous allons montrer l'expression de ces contraintes dans ACL, mais cette fois-ci sur le métamodèle de Java, pour assurer une transition harmonieuse de la section précédente. Ensuite, nous présentons comment spécifier ces mêmes contraintes en s'appuyant sur le mécanisme d'introspection fourni par Java.

La figure 2.5 montre un métamodèle simplifié du langage Java. Par simplifié, nous voulons dire que ce métamodèle ne représente que les entités dans le langage Java servant à l'écriture de contraintes d'architecture. Nous retrouvons donc les classes, qui ont des attributs (`Field` dans la terminologie Java), des méthodes et des constructeurs. Une classe appartient à un package. Cette métaassociation est navigable dans un sens seulement. Ce qui veut dire qu'à partir d'un package, on ne peut pas savoir quels sont les types qui y sont définis. Tous ces éléments peuvent être annotés (propriété héritée entre autres de la métaclasse `AccessibleObject`). A l'exception des packages, les autres métaclasses ont des `modifiers`, qui peuvent avoir différentes valeurs listées dans l'énumération `Modifier`. Un attribut peut avoir comme valeur pour un objet particulier une référence vers un autre objet. Il est à noter que la sémantique du métamodèle n'est pas suffisamment précise sur ce point. En effet, l'association entre `Field` et `ObjectReference` ne doit être navigable que lorsque l'on part dans une contrainte d'un objet (instance de la métaclasse `Object`), puis vers sa classe (instance de `Class`), et enfin vers l'attribut (instance de `Field`). Si la navigation démarre de `Class`, alors l'accès à `Field` ne doit pas permettre l'accès par la suite à `ObjectReference`. Ce métamodèle a été construit sur la base des classes définies dans l'API `java.reflect` et leurs méthodes permettant de faire de l'introspection. Le manque de sémantique soulevé ci-dessus est lié au fait que d'une part dans Java il n'existe pas un équivalent de `Slot` d'UML. D'autre part, la méthode `get(...)` de la classe `Field` retourne la valeur de l'attribut mais on doit lui passer en argument l'objet pour lequel la valeur de l'« attribut » (slot) doit être retournée. En réalité, ce problème est plus général. Il est dû au fait que dans Java, il n'y a pas un vrai couplage entre les objets et leurs métaobjets (instances de `Class`). Une fois que l'on fait `getClass()` sur un objet, on obtient le métaobjet, mais dans ce métaobjet, il n'y a aucune référence vers l'objet (on perd donc l'accès aux valeurs de ses « attributs »).

Les sous-contraintes du patron MVC introduit dans la sous-section précédente peuvent être exprimées sur ce métamodèle de la façon suivante :

```
1  context Class inv:
2  self.annotation->exists(a:Annotation | a.type.name='Model')
3  implies
4  self.field->forAll(f : Field |
5    not (f.type.annotation->exists(type.name='View')
6    or f.type.annotation->exists(type.name='Controller')))
7  and
8  self.annotation->exists(type.name='View')
9  implies
10 self.field->forAll(not (type.annotation
11   ->exists(type.name='Model')))
```

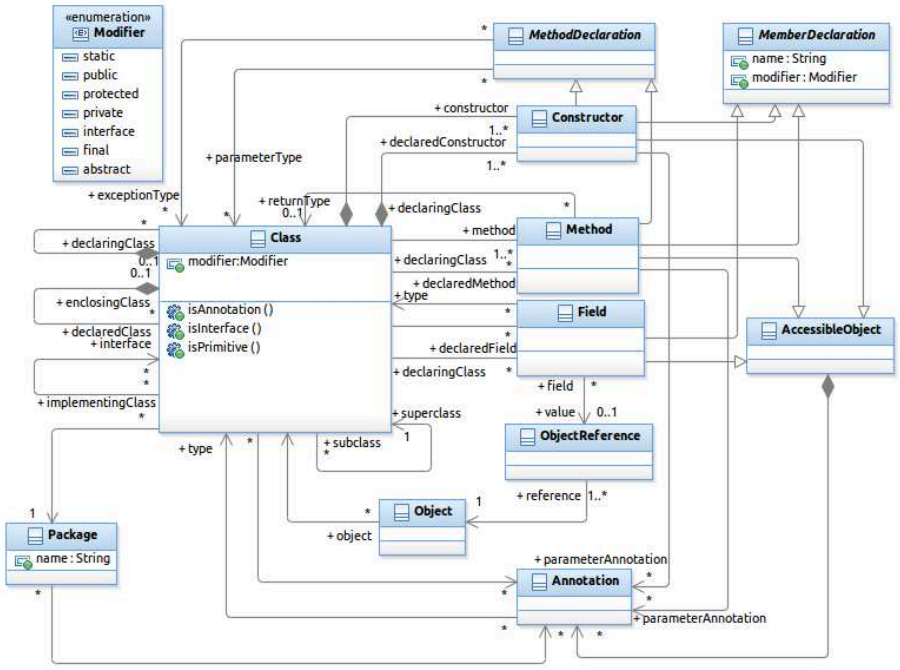

Figure 2.5. *Extrait du métamodèle Java*

Nous supposons ici l'existence de trois annotations, applicables sur des types (classes) et présents à l'exécution (`@Retention(value=RUNTIME)`), correspondant aux trois stéréotypes présentés précédemment. Le contexte de la contrainte ne peut pas être le package de toute l'application. Comme on l'a précisé précédemment, à partir de celui-ci, nous ne pouvons malheureusement récupérer les types qui sont définis dans le package. Nous avons donc précisé comme contexte de la contrainte toute sorte de classe. La contrainte doit donc être vérifiée sur toutes les classes de l'application. La navigation dans le métamodèle Java, montré dans la figure 2.5, est assez simple. Nous analysons ici les attributs (objets de type `Field`) de la classe puis leurs types et les annotations appliquées sur ceux-ci.

La contrainte exprimée en ACL ci-dessus peut être implémentée en Java ainsi :

```
1  // Nous supposons ici que les classes de l'application
2  // ont été déjà chargées d'une certaine façon
3  public boolean invariant(Class<?>[] classesMonApplication) {
4    for(Class<?> uneClasse : classesMonApplication) {
5      if(uneClasse.isAnnotationPresent(Model.class)) {
6        Field[] attributs = uneClasse.getDeclaredFields();
7        for(Field unAttribut : attributs) {
8          if(unAttribut.getType().isAnnotationPresent(View.class) ||
9          unAttribut.getType().isAnnotationPresent(Controller.class))
10           return false;
```

```
11          }
12        }
13        if(uneClasse.isAnnotationPresent(View.class)) {
14          Field[] attributs = uneClasse.getDeclaredFields();
15          for(Field unAttribut : attributs) {
16            if(unAttribut.getType().isAnnotationPresent(Model.class))
17              return false;
18          }
19        }
20      }
21      return true;
22    }
```

La méthode `invariant(...)` admet en paramètres les objets de type `Class` représentant chacune des classes de l'application[9]. Nous ne pouvons malheureusement pas partir de l'objet de type `Package` représentant le package de l'application, parce que dans java.reflect, celui-ci ne référence pas les classes qui se trouvent dans le package. Il s'agit d'un simple objet comportant des informations sur le package (son nom, par exemple).

Il est à noter ici, qu'à la différence des approches d'analyse statique de code, la vérification de la contrainte et donc l'exécution de cette méthode nécessite au moins le chargement de toute l'application par le chargeur de classes afin d'obtenir les objets de type `Class` représentant les différentes classes de l'application, puis les mettre dans un tableau qui est passé en argument lors de l'invocation. Dans ce chapitre, nous nous focalisons sur la spécification de contraintes d'architecture et non sur leur vérification.

Revenons maintenant sur un point qui a été soulevé lors de l'introduction des différentes sous-contraintes formalisant les restrictions sur les dépendances imposées par le patron MVC. Dans la première sous-contrainte, nous avons indiqué que les classes du modèle ne doivent pas déclarer de dépendances avec les classes de la vue ; ce qui en fait une contrainte sur les types statiques. Cependant, à l'exécution, selon les implémentations de ce patron, on peut se retrouver, avec une référence vers un objet de la vue, dans un objet du modèle. En effet, ce qui a été schématisé par un nuage dans la figure 2.4 peut être implémenté par le patron `Observer`. Dans ce cas, un objet du modèle stocke une collection d'objets écouteurs des modifications sur le modèle (la collection peut être statiquement typée par une interface, par exemple, nommé `EcouteurModificationModel`). Mais à l'exécution, cette collection comportera bien des objets de la vue, dont les classes implémentent l'interface précédente (`EcouteurModificationModel`). Donc, dynamiquement on se retrouve avec des dépendances entre les classes du modèle et les classes de la vue, alors que statiquement cette dépendance n'est pas visible. Cela doit être admis pour la première sous-contrainte seulement. En revanche, ça ne doit pas être vrai pour les deux autres

9. Nous désignons par les classes de l'application, les classes faisant partie du domaine métier de l'application. Cela exclut les classes des librairies utilisées par l'application.

sous-contraintes, qui ne doivent pas déclarer des dépendances, statiquement et dynamiquement. Ceci se traduit par le code Java suivant qui vient s'ajouter à la contrainte précédente :

```
1  // Nous supposons ici la réception en paramètre d'un tableau
2  // constitué des différents \index{Objet}objets qui composent l'application
3  public boolean invariant(Object[] objetsMonApplication) {
4    for(Object unObjet : objetsMonApplication) {
5      Class<?> uneClasse = unObjet.getClass();
6      Field[] attributs = uneClasse.getDeclaredFields();
7      for(Field unAttribut : attributs) {
8        // Vérifications du précédent listing ...
9        boolean accessAttrModifie = false;
10       if(! unAttribut.getType().isPrimitive()) {
11         try {
12          if(! unAttribut.isAccessible()) {
13            unAttribut.setAccessible(true);
14            accessAttrModifie = true;
15          }
16          Class<?> cl = unAttribut.get(unObjet).getClass();
17          if(cl.isAnnotationPresent(Controller.class)) return false;
18         }
19         catch(IllegalAccessException e) {
20           e.printStackTrace();
21         }
22         finally {
23           if(accessAttrModifie) unAttribut.setAccessible(false);
24         }
25       }
26     }
27   }
28   return true;
29 }
```

Cette fois-ci nous nous appuyons dans cette contrainte sur les objets qui composent l'application et non sur les objets de type `Class` qui représentent les classes de l'application. Ici, nous allons plus loin dans l'exécution, parce que cela suppose que l'application ait été chargée, mais aussi lancée, pour obtenir les objets qui composent l'application et aller chercher les valeurs de leurs *slots* (définis par les attributs déclarées dans les classes de ces objets). Cette obtention des valeurs des slots des objets est faite sur la ligne 16. Celle-ci est précédée d'un certain nombre de vérifications pour s'assurer que l'attribut de la classe de l'objet n'est pas de type primitif et est accessible (il a une accessibilité publique, sinon la modifier). C'est le type de la valeur slot en question qui est vérifié, pour s'assurer qu'il ne s'agit pas d'une classe annotée `Controller`. La dernière sous-contrainte formalisant le patron MVC peut être raffinée de la même façon. Ceci n'est pas montré dans le listing précédent pour des raisons de simplicité et de brièveté.

La spécification de cette contrainte au niveau conception est possible aussi, en supposant que l'on ait un modèle représentant les instances qui composent l'application. Cette contrainte exprimée en ACL va porter sur le métamodèle UML des instances. Un extrait de celui-ci est donné dans la figure 2.6. Dans ce métamodèle, il est indiqué qu'une spécification d'instance a un `Classifier` qui la définit et comporte un

certain nombre de slots. Ces derniers ont un `StructuralFeature` (un `Property` par exemple) qui les définit. Ils ont des valeurs désignées par des `ValueSpecification`. Ces dernières peuvent être de différents types : `InstanceValue` (une référence vers une instance), `LiteralSpecification` (un entier, réel, etc.), etc. La métaclasse qui nous intéresse est `InstanceValue` (montré dans le métamodèle de la figure 2.6). Celle-ci est relié à `InstanceSpecification` pour désigner la spécification de l'instance référencée dans le slot.

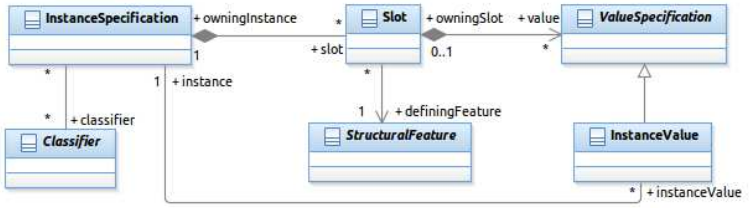

Figure 2.6. *Extrait du métamodèle UML (Instances)*

La contrainte ACL appliquée sur ce métamodèle UML pourrait être définie comme suit :

```
1  context InstanceSpecification inv:
2  -- Nous supposons ici que nous avons déjà récupéré
3  -- les ensembles de classes model, view et controller
4  -- comme ça a été fait précédemment
5  if model->includes(self.classifier)
6  then
7    self.slot->forAll(s : Slot |
8      -- Mêmes vérifications que précédemment
9      -- en utilisant s.definingFeature pour accéder
10     -- à l'attribut (StructuralFeature) qui définit le slot
11     if s.value.isOclKindOf(InstanceValue)
12     then
13       controller->excludes(s.value.oclAsType(InstanceValue)
14         .instance.classifier)
15     else true
16     endif
17   )
18 else true
19 endif
```

Dans cette contrainte, le contexte de la contrainte est InstanceSpecification. Nous supposons donc que la contrainte doit être vérifiée sur toutes les spécifications d'instances qui composent l'application. Nous avons spécifié dans la contrainte le fait que si le classifier de la spécification d'instance est stéréotypé `Model`, alors nous testons si un de ses slots contient une référence vers une instance (ligne 11), c'est-à-dire le slot ne contient pas une valeur de type primitif. Dans ce cas, nous accédons au `Classifier` de la valeur stockée dans le slot. Celui-ci ne doit pas être stéréotypé `Controller`.

Ici aussi, pour des raisons de simplicité et brièveté, la dernière sous-contrainte formalisant le patron MVC n'est pas raffinée.

La même contrainte sur le métamodèle Java peut être définie de la façon suivante :

```
1  context Object inv:
2  if self.class.annotation->exists(name='Model')
3  then
4    self.class.field->forAll(f:Field |
5      -- Mêmes vérifications que précédemment
6      -- en utilisant f.type pour accéder
7      -- au type de l'attribut
8      if f.value <> null
9      then f.value.object.class.annotation = 'Controller'
10     else true
11     endif
12   )
13 else true
14 endif
```

Dans cette contrainte, le contexte est un objet (instance de `Object`) composant l'application. Nous naviguons ensuite vers sa classe, puis vers les valeurs de ses attributs (voir lignes 8 et 9). Cette expression ACL ne formalise pas la dernière sous-contrainte du patron MVC.

Nous avons montré dans cette section comment les contraintes d'architecture peuvent être écrites simplement en Java lors de la phase d'implémentation, sans aucune extension de langage (à part les annotations qu'il faudra définir, au besoin, pour les patrons ou styles d'architecture formalisés). Nous n'avons cependant pas expliqué comment et à quel moment ces contraintes sont évaluées. Différentes solutions sont possibles, comme par exemple l'injection automatique de code de vérification de ces contraintes à la fin des constructeurs des classes concernées. Cela constitue un travail sur lequel nous allons nous pencher dans le futur. Par ailleurs, nous sommes en train de développer une méthode et un outil pour la génération de code Java (comme celui des listings ci-dessus) à partir de contraintes d'architecture exprimées en ACL sur le métamodèle UML (comme celles données précédemment).

2.4. Contraintes d'architecture sur des applications à composants

Après avoir exploré la façon dont les contraintes d'architecture peuvent être spécifiées sur des applications à objets, nous allons dans cette section exposer la manière dont ces contraintes peuvent être exprimées dans des applications à base de composants. Les composants sont considérés ici comme une évolution du concept d'objet (ou classe, ceci sera détaillé après) pour apporter plus de modularité aux architectures de ces applications. En effet, dans le développement par composants, il est préconisé de déclarer de façon explicite, en plus des fonctionnalités fournies, les fonctionnalités requises d'une application. Ceci permet de découpler les composants d'une application et donner plus de souplesse à leur connexion. Pour construire une nouvelle

application, il faudra créer des instances de composants et satisfaire leurs fonctionnalités requises en les connectant à d'autres instances qui fournissent ces fonctionnalités. Nous allons procéder en deux étapes, comme dans la section précédente. Nous allons tout d'abord présenter la spécification de contraintes d'architecture sur des applications en phase de conception, puis exposer comment cela peut être fait en phase d'implémentation.

2.4.1. *Contraintes d'architecture en phase de conception*

Nous avons choisi le standard UML comme langage de modélisation des applications en phase de conception. Nous allons expliquer comment les contraintes d'architecture peuvent être spécifiées sur des applications à base de composants modélisées avec ce langage, largement répandu dans le milieu académique, mais surtout aussi dans le milieu industriel.

La figure 2.7 montre un extrait du métamodèle UML spécifiant les éléments de modélisation autour des composants logiciels. Le modèle de composants UML présenté englobe à la fois la spécification des composants et des structures composites en UML [OMG 11]. La métaclasse `Component` est une spécialisation de `Class`. Cela lui confère toutes les capacités d'une classe (participer dans une relation d'héritage, par exemple). Par ailleurs, la métaclasse `Class` hérite désormais de `EncapsulatedClassifier` et `StructuredClassifier`. Cela veut dire qu'un composant (spécialisation de `EncapsulatedClassifier`) peut avoir des `ports` qui peuvent déclarer des interfaces fournies et des interfaces requises (voir en bas à gauche de la figure 2.7). Cela permet aux composants d'encapsuler (et donc cacher à leur environnement) les éléments qui les constituent. Ils exposent leurs fonctionnalités *via* ces ports, qui sont les points de communication avec l'environnement (les autres composants). Un composant (spécialisation de `StructuredClassifier`) peut déclarer des `parts` qui sont des propriétés qui référencent les instances qui constituent sa structure interne (on dit alors que le composant a une structure composite). Ces mêmes instances peuvent jouer des `rôles` dans des connexions (être attachées à des extrémités de connecteurs `ConnectorEnd`). Un composant peut posséder des connecteurs qui relient des éléments « pouvant être connectés » (instances de type `ConnectableElement`). Ca peut être le port d'un « part » dans une structure composite d'un composant (association entre `ConnectorEnd` et `Property` dans le métamodèle de la figure 2.7) ou bien le port du composant englobant[10] (ayant la structure composite). Enfin, un composant peut avoir un ou plusieurs `Classifier` (classes, par exemple), qui le « réalisent ». Par *Realization* d'un composant, la

10. Ces caractéristiques ne portent pas exclusivement sur les composants. Les classes aussi peuvent avoir des ports et des structures composites.

spécification UML désigne l'ensemble des classifiers qui implémentent le comportement de ce composant. Ceux-ci peuvent être d'autres composants (`Component` est une spécialisation de `Classifier`). Ceci rend le modèle de composants UML hiérarchique.

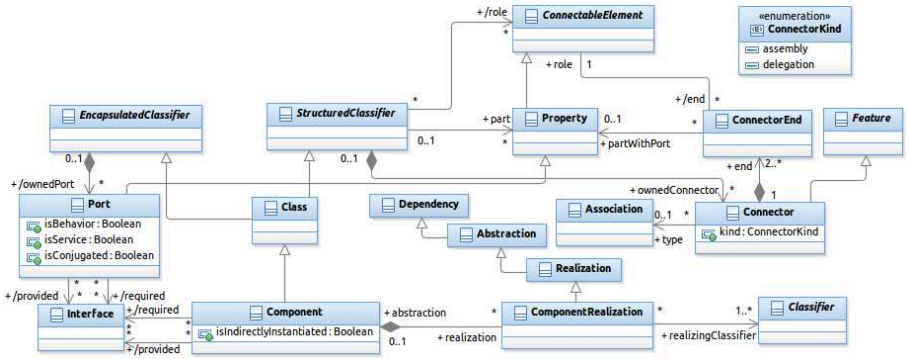

Figure 2.7. *Extrait du métamodèle UML (composants et structures composites)*

Nous allons dès à présent présenter de façon informelle (textuelle) la contrainte que l'on voudrait spécifier dans cette section. Il s'agit d'une contrainte qui formalise les conditions structurelles imposées par le style d'architecture `Pipe & Filter` [SHA 96]. Dans ce style, les conditions qui doivent être formalisées sont les suivantes :

– il n'y a qu'un seul composant possédant un ou plusieurs ports en entrée (déclarant des interfaces fournies seulement) qui sont connectés au composant englobant ou qui ne sont pas connectés du tout (il s'agit du composant de l'extrémité gauche dans la figure 2.8). Ce même composant doit avoir au moins un port en sortie (déclarant des interfaces requises seulement) connecté. Les ports en sortie de ce composant doivent tous être soit connectés aux autres composants de même niveau hiérarchique que ce composant ou bien pas du tout connectés ;

– il n'y a qu'un seul composant possédant un ou plusieurs ports en sortie qui sont connectés au composant englobant ou qui ne sont pas du tout connectés (il s'agit du composant de l'extrémité droite dans la figure 2.8). Ce même composant doit avoir au moins un port en entrée connecté. Les ports en entrée de ce composant doivent tous être soit connectés aux autres composants de même niveau hiérarchique que ce composant ou bien pas du tout connectés ;

– les autres composants possèdent des ports en entrée et en sortie, dont au moins un port en entrée et un port en sortie qui sont connectés ;

– les connecteurs entre chaque paire de composants doivent aller dans la même direction. Ceci veut dire qu'il n'y a pas de connecteur reliant le port en entrée du

premier et le port de sortie du second, et un autre connecteur qui relie le port de sortie du premier au port d'entrée du second ;

– les connecteurs entre tous les composants doivent aller dans la même direction. Cela veut dire que pour chaque paire de composants connectés, il n'y a pas un troisième composant, qui est connecté, par ses ports en entrée, au premier composant et, par ses ports en sortie, au deuxième composant.

Ces contraintes sont schématisées dans la figure 2.8. Les composants dans ce style sont nommés des filtres et les connecteurs des pipes. Dans ce chapitre, nous simplifions l'architecture de l'application en considérant le fait que celle-ci a été conçue exclusivement selon ce style. Les applications du monde réel sont souvent construites en combinant plusieurs styles. Dans ce cas, la contrainte aura une formalisation différente, qui pourrait s'appuyer par exemple sur des stéréotypes `Pipe` et `Filter` appliqués sur les composants. La contrainte doit s'assurer que les conditions énumérées ci-dessus s'appliquent uniquement aux composants stéréotypées, qui participent donc à la mise en œuvre du style d'architecture. Faute de place, nous ne pourrons malheureusement pas développer ce cas ici.

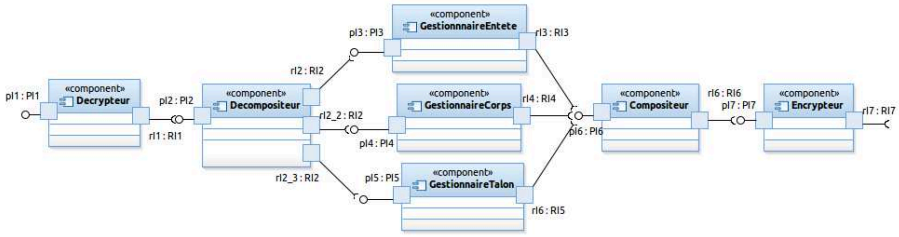

Figure 2.8. *Illustration du style d'architecture Pipe and Filter*

Nous pourrons écrire les différentes sous-contraintes en utilisant ACL et en s'appuyant sur le métamodèle de la figure 2.7. La première sous-contrainte peut être spécifiée de la façon suivante :

```
1  -- Nous supposons ici que les composants formant l'application
2  -- sont encapsulés dans un composant ayant une structure composite.
3  -- Ceci n'est pas montré dans la figure illustrant le style
4  -- Pipe and Filter
5  context MonApplication : Component inv:
6  let internalComps : Set(Component) =
7  self.realization.realizingClassifier->select(c:Classifier |
8    c.oclIsKindOf(Component))->oclAsType(Component)
9  in
10 internalComps->one(c : Component |
11   self.ownedConnector->exists(con:Connector |
12     con.end.role->exists(r1,r2 | r1.oclIsKindOf(Port)
13       and (r1.oclAsType(Port) = c.ownedPort)
14       and r1.oclAsType(Port).provided->notEmpty()
15       and r1.oclAsType(Port).required->isEmpty()
16       and r2.oclIsKindOf(Port)
17       and self.ownedPort->includes(r2.oclAsType(Port))
```

```
18      )
19    )
20    and
21    self.ownedConnector->exists(con:Connector |
22      con.end.role->exists(r1,r2 | r1.oclIsKindOf(Port)
23        and (r1.oclAsType(Port) = c.ownedPort)
24        and r1.oclAsType(Port).required->notEmpty()
25        and r2.oclIsKindOf(Port)
26        and self.ownedPort->excludes(r2.oclAsType(Port))
27      )
28    )
29  )
```

Dans cette contrainte, nous créons tout d'abord un ensemble (`internalComps`) constitué des composants qui forment la structure interne du composant représentant l'application. Nous rappelons que c'est cette dernière qui a une architecture organisée selon le style *Pipe and Filter*. Ensuite, nous nous assurons qu'il n'existe qu'un seul composant dans cet ensemble (`internalComps`) qui a au moins un connecteur qui le relie avec le composant englobant à travers un port qui n'a que des interfaces fournies (et pas requises). Il est à noter ici que la vérification de la présence de connecteurs entre composants passe par la consultation de l'ensemble des connecteurs de l'application (`self.ownedConnector`). En effet, nous ne pouvons obtenir les connecteurs attachés au port d'un composant en partant de ce composant. Cela correspond bien au découplage entre composants préconisé par le développement par composants : un composant ne connaît pas les autres composants auxquels il est connecté, il ne fait qu'invoquer des opérations sur son port requis, il ne dépend pas d'un composant particulier. Dans la deuxième partie de la contrainte (à partir de la ligne 20), on vérifie que ce composant unique est connecté aux autres composants de l'application par ses ports en sortie (déclarant des interfaces requises).

La deuxième sous-contrainte énumérée précédemment peut être formalisée de la même façon en inversant les interfaces requises et les interfaces fournies.

La troisième sous-contrainte est spécifiée en ACL dans le listing suivant :

```
1  ...
2  and
3  internalComps->select(c |
4    not self.ownedConnector->exists(con:Connector |
5      con.end.role->exists(r1,r2 | r1.oclIsKindOf(Port)
6        and (r1.oclAsType(Port) = c.ownedPort)
7        and r2.oclIsKindOf(Port)
8        and self.ownedPort->excludes(r2.oclAsType(Port))
9      )
10   )
11   and c.ownedPort->select(provided->notEmpty()
12     and required->notEmpty()))
13 ->size() = internalComps->size() - 2
```

Dans cette contrainte, on compte le nombre de composants qui participent dans les connexions mais qui ne sont pas connectés au composant englobant. Ce nombre doit

être égal au nombre total de composants de l'application moins les deux composants qui sont dans les extrémités.

Maintenant, nous allons voir comment la quatrième sous-contrainte peut être définie dans ACL :

```
...
and
internalComps->forAll(c1,c2 : Component |
  self.ownedConnector->select(con:Connector |
    con.end.role->oclIsKindOf(Port) and
    con.end.role->oclAsType(Port)->one(c1.ownedPort) and
    con.end.role->oclAsType(Port)->one(c2.ownedPort))
  ->forAll(con : Connector | con.end.role->select(oclIsKindOf(Port))
    ->oclAsType(Port)->exists(p1,p2:Port |
      (c1.provided->includesAll(p1.provided)
        and c2.required->includesAll(p2.required))
      xor (c2.provided->includesAll(p1.provided)
        and c1.required->includesAll(p2.required))))
)
```

Ici, nous nous assurons que dans l'ensemble des connecteurs de l'application, si un connecteur relie deux composants c1 et c2, il faut que tous les autres connecteurs entre c1 et c2 soient toujours entre ports en entrée (déclarant des interfaces requises) de c1 et ports de sortie (déclarant des interfaces fournies) de c2, ou inversement. Il ne faut pas qu'il y ait en même temps des connecteurs orientés de c1 vers c2 et de c2 vers c1.

La dernière sous-contrainte, et sans doute la plus complexe, est définie ci-dessous :

```
...
and
internalComps->forAll(c1,c2:Component |
  let conns : Set(Connector) =
  self.ownedConnector->select(con:Connector |
    con.end.role->oclIsKindOf(Port) and
    con.end.role->oclAsType(Port)->one(c1.ownedPort) and
    con.end.role->oclAsType(Port)->one(c2.ownedPort))
  in -- conns = {connecteurs entre c1 et c2}
  conns->notEmpty()
  and
  internalComps->excludes(c3 : Component |
    c3 <> c1 and c3 <> c2 and
    if conns.end.role->oclAsType(Port)
      ->exists(p | c1.ownedPort.required->includesAll(p.required))
    then --Connecteur(s) c1 vers c2
      not (
        self.ownedConnector.end.role->select(oclIsKindOf(Port))
        ->oclAsType(Port)->exists(p1,p2 |
          c3.ownedPort.required->includesAll(p1.required) and
          c2.ownedPort.provided->includesAll(p2.provided))
       or
        self.ownedConnector.end.role->select(oclIsKindOf(Port))
        ->oclAsType(Port)->exists(p1,p2 |
          c1.ownedPort.required->includesAll(p1.required) and
          c3.ownedPort.provided->includesAll(p2.provided))
      )
    else -- Connecteur(s) c2 vers c1
```

```
29        not (
30          self.ownedConnector.end.role->select(oclIsKindOf(Port))
31          ->oclAsType(Port)->exists(p1,p2 |
32            c2.ownedPort.required->includesAll(p1.required) and
33            c3.ownedPort.provided->includesAll(p2.provided))
34         or
35          self.ownedConnector.end.role->select(oclIsKindOf(Port))
36          ->oclAsType(Port)->exists(p1,p2 |
37            c3.ownedPort.required->includesAll(p1.required) and
38            c1.ownedPort.provided->includesAll(p2.provided))
39        )
40      endif
41    )
42  )
```

Dans cette dernière partie de la contrainte, nous vérifions que pour toute paire de composants connectés (c1, c2) dans la liste des composants internes, il n'existe pas un troisième composant (c3) qui est connecté, *via* ses ports en entrée, à c1 et, *via* ses ports en sortie, à c2, si les connecteurs vont de c1 vers c2, et inversement si les connecteurs vont de c2 vers c1. Ceci garantit que tous les connecteurs dans l'application sont orientés dans la même direction.

Nous avons simplifié dans cette formalisation l'utilisation du modèle de composants UML. Nous supposons que le développeur n'a pas défini un même port avec des interfaces requises et fournies auquel sont attachés plusieurs connecteurs (entrants et sortants).

2.4.2. *Contraintes d'architecture en phase d'implémentation*

Dans la littérature, il existe plusieurs langages (comme ArchJava [ALD 02], ComponentJ [SEC 08] ou SCL [FAB 08]) ou *frameworks* (comme Spring, OSGi ou Fractal/Julia [BRU 04]) de programmation par composants. Pour pouvoir écrire des contraintes, il nous faudrait, en revanche, un langage ou un *framework* offrant des capacités d'introspection. Là aussi, plusieurs langages s'offrent à nous. Nous avons choisi un langage, qui a été développé dans notre équipe dans le cadre de la thèse de Petr Spacek [SPA 12]. Ce langage s'appelle Compo [SPA 12]. Il a été développé dans un cadre plus large que la spécification de contraintes d'architecture. Cependant, le choix a été porté sur ce langage pour les raisons suivantes :

1) il fournit un support pour la spécification de contraintes d'architecture de façon explicite ;

2) il est réflexif, presque à tous les niveaux (tout est réifié, sauf le corps des services) ;

3) la réification des entités de ce langage est réalisée par des composants, et non par des objets, comme le font les autres langages à composants.

Grâce à ce dernier point, nous proposons aux développeurs un environnement homogène, où ils ne définissent que des descripteurs de composants ; ils n'auront jamais à choisir entre les objets ou les composants pour définir tel ou tel concept du domaine métier de leur application.

Un extrait du métamodèle de Compo est donné dans la figure 2.9. Dans ce métamodèle, il y a une distinction claire entre un descripteur de composants et une instance de composants (appelée simplement `Component`). Dans UML, `Component` hérite de `Class`, ce qui en fait un descripteur de composants. Le fait d'hériter de la métaclasse `Class` lui confère la capacité d'être instanciée (voir la méta-association entre `Classifier` et `InstanceSpecification` dans le métamodèle de la figure 2.6). En UML, une instance de composants n'a donc rien de particulier comparativement avec une instance de classes.

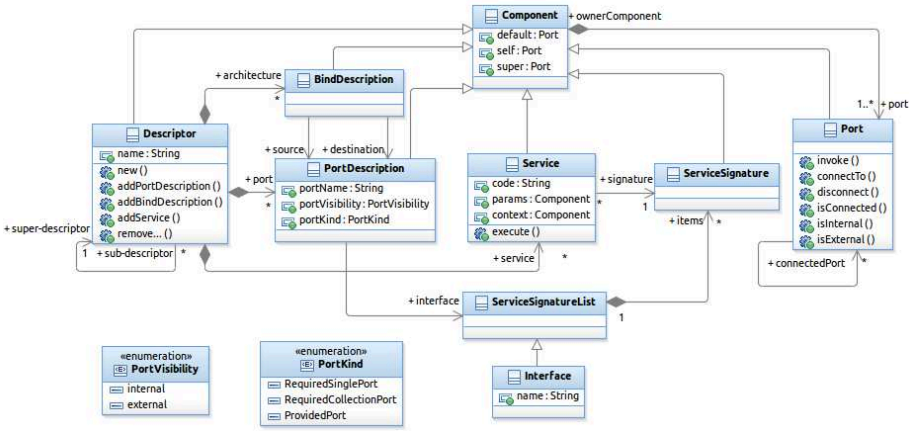

Figure 2.9. *Extrait du métamodèle de Compo*

Dans Compo, il y a une dichotomie généralisée entre les descripteurs et leurs instances : descripteur de composant et composant, descripteur de port et port, etc. Dans un descripteur de composant Compo (métaclasse `Descriptor`), le programmeur peut déclarer un certain nombre de descripteurs de ports. Ces derniers doivent indiquer la liste des signatures de services (un service est l'équivalent d'une opération en UML), qui peut être groupée dans une interface nommée ou dans une interface anonyme (`ServiceSignatureList`). Ces services ont une signature. Un connecteur (`BindDescription`)) met en relation un descripteur de port source et un descripteur de port destination. Un port réalise un descripteur de port, et peut être de type Collection ou de type simple (*Single*). Un port peut être, par ailleurs, fourni ou requis. Il peut être interne ou externe. Par exemple, un port requis interne sert à connecter les

instances de composants à leur composant englobant. Enfin, nous avons intégré l'héritage dans ce langage. Un descripteur de composant peut hériter d'un autre descripteur (son *super-descriptor*[11]).

Les contraintes imposées par le style *Pipe and Filter* peuvent être programmées dans Compo de la façon suivante :

```
1  Descriptor PipeAndFilter extends Constraint
2  {
3      internally requires {
4          scOne <: SubConstraintOne;
5          scTwo <: SubConstraintTwo;
6          scThree <: SubConstraintThree;
7          scFour <: SubConstraintFour;
8          scFive <: SubConstraintFive;
9      }
10     architecture {
11         delegate contextscOne to contextself;
12         delegate contextscTwo to contextself;
13         delegate contextscThree oneDelegReq to contextself;
14         delegate contextscFour to contextself;
15         delegate contextscFive to contextself;
16     }
17     service verify() {
18         |c1 c2 c3 c4 c5 |
19         c1 := scOne.verify();
20         c2 := scTwo.verify();
21         c3 := scThree.verify();
22         c4 := scFour.verify();
23         c5 := scFive.verify();
24
25         return (((c1.and([c2])).and([c3])).and([c4])).and([c5]);
26     }
27 }
```

Dans ce listing, nous nous appuyons sur un modèle de spécifications de contraintes d'architecture sous la forme de composants, initialement introduit dans [TIB 11]. Les différentes sous-contraintes imposées par le style *Pipe and Filter* sont alors définies par plusieurs descripteurs de composants. Dans ce listing, nous déclarons le descripteur de composants, nommé `PipeAndFilter`, qui contient l'ensemble des instances de composants qui vérifient les différentes sous-contraintes, et qui sont au nombre de cinq (déclarées dans les lignes 4 à 8). Ces composants sont connectés à leur composant englobant (lignes 11 à 15). Ensuite, nous démarrons les vérifications en invoquant dans le service `verify` du composant `PipeAndFilter` le service `verify()` sur chacun des ports des différents composants internes. Le résultat final doit correspondre à la conjonction (au sens de la logique booléenne) des différents résultats (booléens) retournés par les composants internes (ligne 25).

11. Ce mécanisme d'héritage entre descripteurs de composants étend l'héritage classique entre classes, notamment en offrant la possibilité d'étendre les ports requis [SPA 12].

Pour des raisons de simplicité, nous donnons ici les listings de deux descripteurs de composants. Il s'agit des descripteurs qui formalisent la première et la quatrième sous-contrainte :

```
1  Descriptor SubConstraintOne extends Constraint
2  {
3      service verify() {
4          |retval|
5          retval := true;
6          intComps := context.getPorts().select([:p |
7              &p.isRequired().and([&p.isInternal()]);
8          ]);
9          intComps.each([:ic |
10             ic.getPorts().each([:x |
11                 if(&x.isProvided().and([&p.isExternal()])) {
12                     | count |
13                     &x.getConnectedPorts().each([:cp |
14                         if(&cp.isProvided().and([&p.isExternal()])) {
15                             if(&cp.getOwner() == context.yourself())
16                             { retVal := retVal.and([true]); }
17                             else
18                             { retVal := retVal.and([false]); }
19                         }
20                     ]);
21
22                 }
23                 if(&x.isRequired().and([&p.isExternal()])) {
24                     &x.getConnectedPorts().each([:cp |
25                         if(&cp.isProvided().and([&p.isExternal()])) {
26                           if(&cp.getOwner().getOwner() == context.yourself())
27                             { retVal := retVal.and([true]); }
28                             else
29                             { retVal := retVal.and([false]); }
30                         }
31                     ]);
32                 }
33             ]);
34         ]);
35         return retVal;
36     }
37 }
```

Dans ce listing, nous avons un descripteur de composant implémentant le service `verify()` qui teste la première sous-contrainte du style *Pipe and Filter*. Ce service identifie d'abord des références vers les composants internes (plutôt les ports de ceux-ci) en s'appuyant sur les ports requis internes du composite. L'accès au métaniveau se fait avec l'opérateur « & » (voir ligne 7, par exemple). Ceci permet de savoir si un port est requis ou fourni, ou s'il est interne ou externe. Ceci permet également d'accéder aux connecteurs qui lient le port au port d'un autre composant (ligne 13). La suite de la contrainte correspond à la contrainte ACL définie dans le premier listing de cette section, où l'on vérifie l'existence d'un unique composant connecté à son composant englobant (fournissant le port `context` dans le listing ci-dessus) *via* son ou ses ports fournis ; les autres ports requis de ce composant interne doivent être connectés aux autres composants internes.

```
1  Descriptor SubConstraintFour extends Constraint
2  {
3      service verify() {
4          conns := context.getDescriptor().getDescribedConnections();
5          conns.each([:conn |
6              |dest source |
7              source := conn.getSourcePortComponent();
8              dest := conn.getDestinationPortComponent();
9              conns.each([:conn2 |
10                 if((conn2.getSourcePortComponent() == dest).and([
11                     conn2.getDestinationPortComponent() == source]))
12                 { return false; }
13             ]);
14         ]);
15         return true;
16     }
17 }
```

La quatrième sous-contrainte programmée en Compo ci-dessus vérifie l'absence du scénario suivant : si nous avons des connecteurs `conn` et `conn2` entre composants, si `conn` a comme destination un composant identique au composant représentant la source de `conn2` (voir ligne 10) alors `conn` a comme source la destination de `conn2` (ligne 11). Ceci reflète l'existence de connecteurs entre une paire de composants qui vont dans des directions différentes, ce qui est interdit par le style *Pipe and Filter*.

Nous remarquons ici que les contraintes d'architecture exprimées en Compo sous la forme de composants améliorent la réutilisation. Ces différents « composants-contraintes » peuvent être assemblés pour former un unique composant-contrainte formalisant le style *Pipe & Filter*. Certains parmi ces composants-contraintes peuvent être réutilisés dans la formalisation d'autres styles comme le *Pipeline* ou le style en couches. Cela éviterait aux architectes de réécrire une partie des sous-contraintes.

2.5. Contraintes d'architecture sur des applications à services

Un service est un groupement d'opérations dans une seule entité, « boîte noire », implémentée dans n'importe quel langage et déployée sur n'importe quelle plate-forme. Cette entité ne maintient pas d'état entre les invocations de ces différentes opérations par un programme client. Lorsqu'un service est accessible *via* le protocole HTTP (conjointement avec SOAP ou pas), on parle de service *web*.

Dans la spécification UML, il existe un profil avec un certain nombre de stéréotypes standards. Parmi ces stéréotypes, nous retrouvons une extension de la métaclasse `Component` qui s'appelle `service`, qui désigne « les composants fonctionnels sans état » [OMG 11]. Ceci est donc assimilé à la notion de service discutée dans cette section.

Parmi les contraintes d'architecture exprimables sur des applications à services, nous retrouvons les patrons SOA [ERL 09]. Ces patrons, comme le *Service Façade*,

imposent un certain nombre de restrictions au niveau de la structure d'une application. Ces restrictions doivent être formalisées afin de garantir, lors de l'évolution, le maintien des attributs qualité qui sont associés au patron.

Dans cette section, nous allons montrer comment on exprime les contraintes d'architecture sur des applications à services conçues et implémentées avec un seul langage, nommé BPEL (standardisé par Oasis sous le nom WSBPEL : *Web Services Business Process Execution Language* [OAS 07]). Le choix de BPEL est motivé par le fait que c'est un langage standardisé largement utilisé par les développeurs de systèmes d'information pour modéliser et mettre en œuvre leurs processus métier en s'appuyant sur des services *web*. Une application à services est donc assimilée dans ce chapitre à un processus BPEL ayant plusieurs services *web* partenaires.

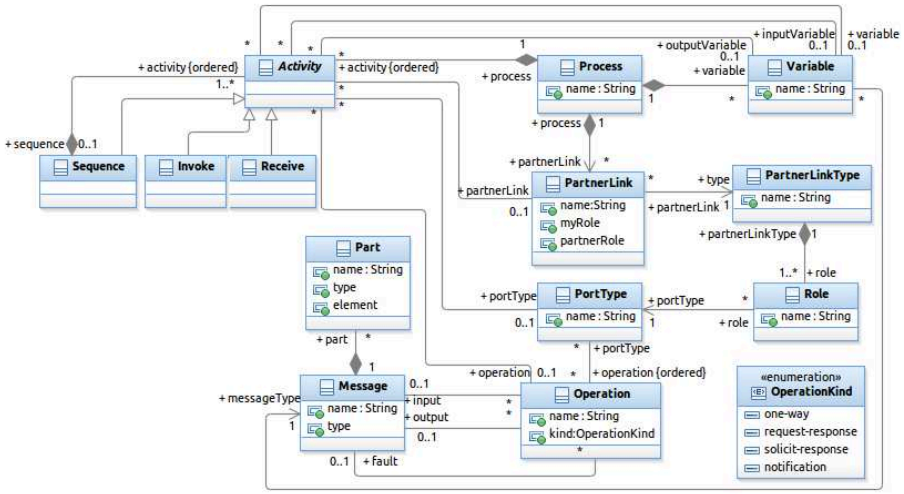

Figure 2.10. *Extrait des métamodèles BPEL et WSDL (Process, Activity, PartnerLink et PortType)*

La figure 2.10 montre un extrait des métamodèles des langages BPEL (partie supérieure de la figure) et WSDL[12] (partie inférieure). Un processus est composé d'un certain nombre d'activités ordonnées (qualificatif `ordered` sur le rôle `activity` dans l'association entre `Processus` et `Activity` dans la figure). Ces activités peuvent être de différentes sortes. Nous en avons listé seulement trois dans le métamodèle (pour le besoin des contraintes définies ci-dessous) : `Invoke` pour invoquer les opérations de services *web* partenaires (fournisseurs), `Receive` pour recevoir les réponses de

12. WSDL (*Web Service Description Language*) est le standard du W3C pour la description d'interfaces de services *web* : www.w3.org/TR/wsdl20/.

ceux-ci ou bien pour recevoir les requêtes des services *web* partenaires (clients), et `Sequence` qui est une activité composite (voir la composition avec `Activity` dans le métamodèle). Cette dernière sert à mettre en œuvre des séquences d'activités. Un processus peut définir un certain nombre de variables, qui peuvent contenir des messages de requête ou de réponse de services *web*. Il déclare un certain nombre de liens vers des services *web* partenaires (`PartnerLink`) qui indiquent les services *web* clients ou fournisseurs utilisés par le processus. Chaque *PartnerLink* désigne des rôles pour le processus (`myRole`) et pour le service partenaire (`partner role`). Un rôle indique un `PortType` qui représente la structure WSDL dans laquelle sont décrites les opérations fournies ou requises par le processus, ainsi que les messages échangés.

Il existe dans la littérature et la pratique une pléthore de patrons d'architecture pour les applications à services, connus sous le nom de patrons SOA (ou *Service-Oriented Architecture patterns* [ERL 09]). Parmi ces patrons, nous avons choisi le patron *service façade* [ERL 09]. Ce patron préconise la définition d'un service unique pour publier/fournir les opérations réalisées par un processus BPEL. Ce patron a comme principal objectif de faire abstraction des détails (parfois complexes) des services exposés par un certain nombre de fournisseurs en proposant un service unique. Un exemple « jouet » de processus BPEL conçu avec ce patron est donné dans la figure 2.11.

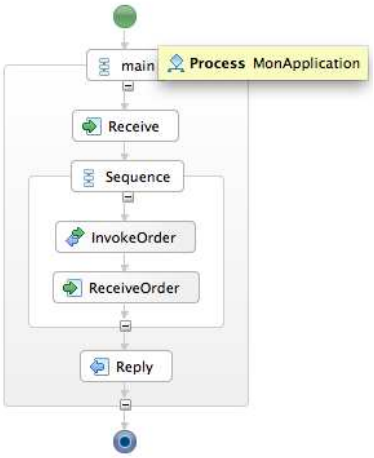

Figure 2.11. *Illustration simple du patron service façade*

Dans cette figure, le service façade est réalisée grâce au *partner link* relié à la première activité dans le processus (l'activité de type `Receive`) et son activité `Reply` correspondante, qui se trouve à la fin du processus. Au milieu du processus, nous

trouvons un certain nombre d'activités `Invoke`[13] pour invoquer les opérations des services fournies par de tierces parties.

La contrainte d'architecture imposée par ce patron consiste en plusieurs sous-contraintes :

– le processus a comme première activité une activité de type `Receive`, liée à un *partner link* représentant le client de l'application (du processus) ;

– le processus a comme comme dernière activité une activité de type `Reply` ayant le même `partner link`, le même `port type` et la même opération que l'activité *Receive* de la sous-contrainte précédente ;

– les seules activités de type `Receive` admises au milieu du processus doivent être précédées d'activités de type `Invoke` correspondantes. Ces dernières sont utilisées pour invoquer les opérations de services partenaires (fournies par de tierces parties). Les activités `Receive`, autorisées ici, serviront alors à la réception des messages retournées par les opérations des services invoquées.

Par faute de place, nous avons volontairement simplifié la contrainte sur ce processus BPEL. En effet, si nous voulions être complets dans la spécification des contraintes imposées par le patron *service façade*, il aurait fallu ajouter d'autres types d'activités, qui pourraient interagir avec un client du processus, comme les activités de type `Pick` et `OnEvent`. Ici, nous n'avons traité que les activités de type `Receive`.

Cette contrainte est formalisée en ACL sur le métamodèle de la figure 2.10 dans les différents listings ci-dessous. La première sous-contrainte est spécifiée comme suit :

```
1  context MonApplication : Process inv :
2  let fst : Activity =
3  if self.activity->first().activity = null -- n'est pas une activité composite
4  then self.activity->first()
5  else if self.activity->first().oclIsTypeOf(Sequence)
6    then self.activity->first().oclAsType(Sequence).activity->first()
7    else null
8    endif
9  endif
10 in
11 if fst <> null
12 then fst.oclIsTypeOf(Receive)
13 else false
14 endif
```

Dans cette première sous-contrainte, nous identifions d'abord quelle est la première activité déclarée dans le processus[14]. Les différents tests (`if`) permettent de

13. Pour des raisons de simplicité, dans la figure 2.11, un seul `Invoke` est montré.

14. Etant donné que les activités dans le processus sont ordonnées (voir le métamodèle), la navigation vers `Activity` depuis `Process` retourne un `OrderedSet`, ce qui permet d'utiliser des opérations comme `first()` pour accéder à la première activité du processus.

traiter le cas où la première activité n'est pas une activité simple mais plutôt composite, comme les `Sequence`. Dans ce cas, il faudra identifier la première activité à l'intérieur de cette séquence. La vérification effectuée dans la ligne 12 permet de s'assurer que c'est une activité de type `Receive`.

Ici aussi, nous simplifions la spécification de la contrainte en considérant uniquement les `Sequence` comme activités composites possibles à l'intérieur du processus, En pratique, d'autres types d'activités composites sont possibles. Dans ce cas, il suffit simplement d'ajouter des tests dans la contrainte pour les prendre en compte (dans le `else` de la ligne 7). Dans cette contrainte, nous ne considérons qu'un seul niveau de profondeur dans les services composites. Si nous voulons prendre en compte plusieurs niveaux de profondeur (pour la première activité dans le processus, ce qui est rare), il suffit de faire un parcours récursif (voir ci-dessous pour un exemple).

La deuxième sous-contrainte ressemble à la première, il suffit de remplacer `first()` par `last()` pour accéder à la dernière activité, et vérifier que le type de l'activité correspond à `Reply`. A cela, il faudra ajouter ce qui suit pour vérifier que les propriétés de cette activité et celles de la première activité (`Receive`) ont des noms identiques :

```
1  ...
2  if lst <> null -- En supposant que lst contient la dernière activité
3  then lst.oclIsTypeOf(Reply)
4    and lst.oclAsType(Reply).partnerLink.name
5      = fst.oclAsType(Receive).partnerLink.name
6    and lst.oclAsType(Reply).portType.name
7      = fst.oclAsType(Receive).portType.name
8    and lst.oclAsType(Reply).operation.name
9      = fst.oclAsType(Receive).operation.name
10 else false
11 endif
```

La troisième sous-contrainte peut être écrite en ACL de la façon suivante :

```
1  ...
2  and
3  let activities : OrderedSet(Activity) =
4  self.activity->excluding(fst)->excluding(lst)->closure(activity)
5  in
6  activities->forAll(a : Activity |
7  if a.oclIsTypeOf(Receive)
8  then
9    activities->exists(aa : Activity |
10     activities->indexOf(aa) < activities->indexOf(a)
11     and aa.oclIsTypeOf(Invoke)
12     and aa.oclAsType(Invoke).partnerLink.name
13       = a.oclAsType(Receive).partnerLink.name
14     -- and ... (même chose pour le portType et operation)
15   )
16 else true
17 endif)
```

Un processus BPEL peut être constitué d'activités composites qui contiennent à leur tour d'autres activités composites, et ainsi de suite, jusqu'à une certaine profondeur. Dans la sous-contrainte définie ci-haut, nous identifions donc de façon récursive toutes les activités qui sont dans le processus, en excluant la première (`fst`) et la dernière activité (`lst`). Cette récursivité est effectuée grâce à l'opération OCL`closure(...)` de la ligne 4. Ensuite, nous nous assurons que s'il y a une activité de type `Receive` parmi ces activités, celle-ci doit être précédée d'une activité de type `Invoke` qui lui correspond, c'est-à-dire qui a le même `partner link`, `port type` et `operation` (lignes 10 à 14).

Nous avons montré dans cette section que la simple combinaison d'OCL et un métamodèle des langages BPEL et WSDL nous a permis de formaliser les contraintes d'architecture imposées par un patron SOA. Nous avons fait le même exercice sur d'autres patrons (résultats en cours de publication), et nous pensons que toute la difficulté réside dans le fait d'identifier les bonnes sous-contraintes textuelles (avant leur formalisation) qui représentent un patron donné. Il s'agit d'un problème classique dans les langages de spécification formelle. Il faut s'assurer que l'ensemble de ces sous-contraintes soit complet et que chacune de celles-ci soit rigoureusement définie. Ceci garantit que leur simple vérification permet de savoir si une application à service est conforme à un patron SOA.

2.6. Conclusion

Une architecture logicielle définit l'organisation « à gros grain » d'un système logiciel. La complexité croissante de ces systèmes logiciels a motivé la proposition, depuis plusieurs décennies, d'une multitude de travaux autour de la documentation des architectures logicielles. Cette documentation a porté à la fois sur le produit des architectes (le quoi : la description d'architecture), mais aussi sur les décisions prises lors de l'élaboration de ce produit et les raisons de ces décisions (le comment et le pourquoi). C'est sur ce deuxième aspect que nous nous sommes concentrés dans ce chapitre, où nous soutenons l'idée que la description d'architecture doit être accompagnée d'une spécification « formelle » (au sens interprétable et vérifiable automatiquement), qui décrit les contraintes imposées par les décisions architecturales. Nous avons illustré nos propos avec un certain nombre de contraintes formalisant quelques styles et patrons d'architecture (comme exemples de décisions architecturales). Nous avons montré comment ces contraintes peuvent être exprimées avec des langages simples utilisés même parfois pour décrire les architectures elles-mêmes (cas de Java et Compo). L'objectif de ces contraintes d'architecture est double. D'une part, elles permettent, en tant que documentation supplémentaire, d'améliorer la compréhension de l'architecture. D'autre part, elles permettent, en tant que spécifications vérifiables automatiquement, de fiabiliser l'évolution, pour savoir, entre autres, après avoir effectué des

changements sur l'architecture si la nouvelle architecture respecte toujours les décisions prises précédemment (travail que nous avons conduit sur les applications à composants [TIB 06a]).

Par ailleurs, nous avons montré que la spécification de ces contraintes ne se limite pas aux architectures d'applications à composants (travail réalisé il y a quelques années [TIB 10]). Elles ont une utilité à la fois dans les applications à objets et à services. De plus, nous avons montré leur utilisation, non pas en phase de conception seule, mais en phase d'implémentation (accompagnant du code) aussi.

Nous avons développé un certain nombre d'interprètes pour les différents langages exposés dans ce chapitre. Tous ces interprètes font de l'analyse statique des descriptions d'architecture (écrites la plupart du temps dans des dialectes XML). Par ailleurs, nous nous sommes intéressés dans le passé à la transformation de contraintes écrites en ACL d'un métamodèle à un autre pour des applications à composants [TIB 06b]. Nous avons montré par exemple comment nous pourrons transformer automatiquement des contraintes d'architecture écrites avec ACL sur des applications modélisées avec des composants UML en contraintes écrites avec ACL sur des applications implémentées avec Fractal/Julia.

Nous pouvons énoncer comme perspectives autour de ce concept de contraintes d'architecture les idées suivantes :

– la spécification de contraintes indépendamment d'un paradigme donné en utilisant un métamodèle de graphes (une description d'architecture étant considérée comme un graphe avec nœuds et arrêtes), puis leur transformation vers les différents paradigmes ;

– la proposition d'un langage de haut niveau pour la réutilisation de contraintes par spécialisation (les contraintes du style *Pipeline* sont une spécialisation des contraintes du style *Pipe and Filter*, par exemple) ;

– la génération de code à partir de contraintes d'architecture ;

– la mise en œuvre de ces contraintes dans un environnement de développement intégré.

2.7. Bibliographie

[ALD 02] ALDRICH J., CHAMBERS C., NOTKIN D., « ArchJava: Connecting Software Architecture to Implementation », *Proceedings of the 22rd International Conference on Software Engineering (ICSE'02)*, ACM, p. 187-197, 2002.

[ALL 97] ALLEN R., A Formal Approach to Software Architecture, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, mai 1997.

[BAS 12] BASS L., CLEMENTS P., KAZMAN R., *Software Architecture in Practice, 3e édition*, Addison-Wesley, Boston, 2012.

[BLE 05] BLEWITT A., BUNDY A., STARK I., « Automatic verification of design patterns in Java », *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE'05)*, ACM, p. 224-232, 2005.

[BOE 08] DE BOER R. C., FARENHORST R., « In search of "architectural knowledge" », *Proceedings of the 3rd international workshop on Sharing and reusing architectural knowledge*, SHARK'08, ACM, p. 71-78, 2008.

[BOK 99] BOKOWSKY B., « CoffeeStrainer: Statically-Checked Constraints on the Definition and Use of Types in Java », *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Toulouse, France, Springer-Verlag, p. 355-374, 1999.

[BOS 04] BOSCH J., « Software Architecture: The Next Step », *Proceedings of the 1st European Workshop on Software Architecture (EWSA'04)*, vol. 3047 de *Lecture Notes in Computer Science*, Springer, p. 194-199, 2004.

[BRI 05] BRIAND L. C., LABICHE Y., DI PENTA M., YAN-BONDOC H. D., « An Experimental Investigation of Formality in UML-Based Development », *IEEE Transactions on Software Engineering*, vol. 31, n° 10, p. 833-849, octobre 2005.

[BRU 04] BRUNETON E., THIERRY C., LECLERCQ M., QUÉMA V., JEAN-BERNARD S., « An Open Component Model and its Support in Java », *Proceedings of the ACM SIGSOFT International Symposium on Component-based Software Engineering (CBSE'04). Held in conjunction with ICSE'04*, Edimbourg, Royaume-Uni, mai 2004.

[CER 89] CERI S., GOTTLOB G., TANCA L., « What You Always Wanted to Know About Datalog (And Never Dared to Ask) », *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, n° 1, p. 146-166, mars 1989.

[CHO 93] CHOWDHURY A., MEYERS S., « Facilitating Software Maintenance by Automated Detection of Constraint Violations », *Proceedings of the International Conference on Software Maintenance (ICSM'93)*, p. 262-271, 1993.

[CLE 02] CLEMENTS P., KAZMAN R., KLEIN M., *Evaluating Software Architectures, Methods and Case Studies*, Addison-Wesley, Boston, 2002.

[CLE 10] CLEMENTS P., BACHMANN F., BASS L., GARLAN D., IVERS J., LITTLE R., NORD R., STAFFORD J., *Documenting Software Architectures, Views and Beyond, Seconde édition*, Addison-Wesley, Boston, 2010.

[EIC 08] EICHBERG M., KLOPPENBURG S., KLOSE K., MEZINI M., « Defining and continuous checking of structural program dependencies », *Proceedings of the 30th international conference on Software engineering (ICSE'08)*, ACM, p. 391-400, 2008.

[ERL 09] ERL T., *SOA Design Patterns*, Prentice Hall, Upper Saddle River, 2009.

[FAB 08] FABRESSE L., DONY C., HUCHARD M., « Foundations of a Simple and Unified Component-Oriented Language », *Journal of Computer Languages, Systems & Structures*, vol. 34/2-3, p. 130-149, 2008.

[FAL 11] FALESSI D., CANTONE G., KAZMAN R., KRUCHTEN P., « Decision-making techniques for software architecture design: A comparative survey », *ACM Computing Surveys (CSUR)*, vol. 43, n° 4, p. 33:1-33:28, octobre 2011.

[FEI 12] FEILER P. H., GLUCH D. P., *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, Addison-Wesley Professional, Boston, 2012.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns: Elements of Reusable Object-Oriented Sofware*, Addison-Wesley Professional, Upper Saddle River, 1995.

[GAR 94] GARLAN D., ALLEN R., OCKERBLOOM J., « Exploiting Style in Architectural Design Environments », *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Nouvelle-Orleans, Louisiane, Etats-Unis, p. 175-188, 1994.

[GAR 00] GARLAN D., MONROE R. T., WILE D., « Acme: Architectural Description of Component-Based Systems », dans G. T. LEAVENS, M. SITARAMAN (dir.), *Foundations of Component-Based Systems*, p. 47-68, Cambridge University Press, Cambridge, 2000.

[GIL 10] GILLES O., HUGUES J., « Expressing and enforcing user-defined constraints of AADL models », *Proceedings of the 5th UML and AADL Workshop (UML and AADL 2010)*, 2010.

[HAR 07] HARRISON N., AVGERIOU P., ZDUN U., « Using Patterns to Capture Architectural Decisions », *Software, IEEE*, vol. 24, n° 4, p. 38-45, IEEE, juillet-août. 2007.

[HOA 78] HOARE C. A. R., « Communicating sequential processes », *Communications of the ACM*, vol. 21, n° 8, p. 666-677, ACM, août 1978.

[HOU 06] HOU D., HOOVER H., « Using SCL to specify and check design intent in source code », *IEEE Transactions on Software Engineering*, vol. 32, n° 6, p. 404-423, 2006.

[JAN 05] JANSEN A., BOSCH J., « Software Architecture as a Set of Architectural Design Decisions », *Proceedings of of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, 2005.

[KLA 96] KLARLUND N., KOISTINEN J., SCHWARTZBACH M. I., « Formal Design Constraints », *Proceedings of the 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Jose, Californie, Etats-Unis, ACM Press, p. 370-383, 1996.

[KRU 09] KRUCHTEN P., CAPILLA R., DUENAS J. C., « The Decision View's Role in Software Architecture Practice », *IEEE Software*, vol. 26, n° 2, p. 36-42, 2009.

[LUC 95] LUCKHAM D. C., KENNEY J. L., AUGUSTIN L. M., VERA J., BRYAN D., MANN W., « Specification and Analysis of System Architecture Using Rapide », *IEEE Transactions on Software Engineering*, vol. 21, n° 4, p. 336-355, 1995.

[MED 96] MEDVIDOVIC N., OREIZY P., ROBBINS J. E., TAYLOR R. N., « Using Object-Oriented Typing to Support Architectural Desing in the C2 Style », *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'96)*, San Francisco, Californie, Etats-Unis, p. 24-32, octobre 1996.

[MED 00] MEDVIDOVIC N., TAYLOR N. R., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, n° 1, p. 70-93, 2000.

[MED 02] MEDVIDOVIC N., ROSENBLUM D. S., REDMILES D. F., ROBBINS J. E., « Modeling Software Architectures in the Unified Modeling Language », *ACM Transactions On Software Engineering and Methodology*, vol. 11, n° 1, p. 2-57, 2002.

[MIN 96] MINSKY N. H., « Law-Governed Regularities in Object Systems. Part I: An Abstract Model », *Theory and Practice of Object Systems*, vol. 2, n° 4, p. 283-301, 1996.

[MIN 97] MINSKY N. H., PRATIM PAL P., « Law-Governed Regularities in Object Systems. Part II: a Concrete Implementation », *Theory and Practice of Object Systems*, vol. 3, n° 2, p. 87-101, 1997.

[MON 01] MONROE R. T., Capturing Software Architecture Design Expertise with Armani, Rapport, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvanie, Etats-Unis, 2001.

[MOR 95] MORICONI M., QIAN X., RIEMENSCHNEIDER R. A., « Correct Architecture Refinement », *IEEE Transactions on Software Engineering*, vol. 21, n° 4, p. 356-372, avril 1995.

[MOR 97] MORICONI M., RIEMENSCHNEIDER R. A., Introduction to SADL 1.0: A language for specifying software architecture hierarchies, Rapport, Computer Science Laboratory, SRI International, 1997.

[OAS 07] OASIS, Web Services Business Process Execution Language Version 2.0, Oasis Website: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf, 2007.

[OMG 08] OMG, UML Profile for CORBA and CORBA Components (CCCMP), Version 1.0 Specification, Document formal/08-04-07, Object Management Group Web Site: www.omg.org/spec/CCCMP/1.0/PDF, 2008.

[OMG 11] OMG, Unified Modeling Language Superstructure, Version 2.4.1 Specification, Document formal/2011-08-06, Object Management Group Web Site: www.omg.org/spec/UML/2.4.1/Superstructure/PDF, 2011.

[OMG 12] OMG, Object Constraint Language Specification, Version 2.3.1, Document formal/2012-01-01, Object Management Group Web Site: www.omg.org/spec/OCL/2.3.1/PDF, 2012.

[REE 79] REENSKAUG T., THING-MODEL-VIEW-EDITOR an Example from a planning system, Rapport, Xerox Parc, mai 1979.

[SEC 08] SECO J. C., SILVA R., PIRIQUITO M., « ComponentJ: A Component-Based Programming Language with Dynamic Reconfiguration », *Computer Science and Information Systems*, vol. 5, n° 2, p. 65-86, 2008.

[SHA 96] SHAW M., GARLAN D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, 1996.

[SPA 12] SPACEK P., DONY C., TIBERMACINE C., FABRESSE L., « An Inheritance System for Structural & Behavioral Reuse in Component-based Software Programming », *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE'12)*, Dresde, Allemagne, ACM Press, septembre 2012.

[TAN 05] TANG A., BABAR M. A., GORTON I., HAN J., « A Survey of the Use and Documentation of Architecture Design Rationale », *Proceedings of the 5th IEEE/IFIP Working*

Conference on Software Architecture (WICSA'05), Pittsburgh, Pennsylvanie, Etats-Unis, novembre 2005.

[TAN 09] TANG A., HAN J., VASA R., « Software Architecture Design Reasoning: A Case for Improved Methodology Support », *IEEE Software*, vol. 26, n° 2, p. 43-49, 2009.

[TER 09] TERRA R., DE OLIVEIRA VALENTE M. T., « A dependency constraint language to manage object-oriented software architectures », *Software Practice and Experience*, vol. 39, n° 12, p. 1073-1094, 2009.

[TIB 06a] TIBERMACINE C., FLEURQUIN R., SADOU S., « On-Demand Quality-Oriented Assistance in Component-Based Software Evolution », *Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06)*, Vasteras, Suède, Springer LNCS, p. 294-309, juin 2006.

[TIB 06b] TIBERMACINE C., FLEURQUIN R., SADOU S., « Simplifying Transformations of Architectural Constraints », *Proceedings of the ACM Symposium on Applied Computing (SAC'06), Track on Model Transformation*, Dijon, France, ACM Press, p. 1240-1244, avril 2006.

[TIB 10] TIBERMACINE C., FLEURQUIN R., SADOU S., « A Family of Languages for Architecture Constraint Specification », *Journal of Systems and Software (JSS), Elsevier*, vol. 83, n° 1, p. 815-831, 2010.

[TIB 11] TIBERMACINE C., SADOU S., DONY C., FABRESSE L., « Component-based specification of software architecture constraints », *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering (CBSE'11)*, ACM, p. 31-40, 2011.

[TYR 05] TYREE J., AKERMAN A., « Architecture Decisions: Demystifying Architecture », *IEEE Software*, vol. 22, n° 2, p. 19-27, mars/avril 2005.

Index