Traduction de spécifications de contraintes d'architecture en composants exécutables

Sahar Kallel, Chouki Tibermacine, Bastien Tramoni, Christophe Dony

Traduction de spécifications de contraintes d'architecture en composants exécutables

Sahar Kallel
ReDCAD, Université de Sfax, Tunisie
sahar.kallel@redcad.org

Chouki Tibermacine
LIRMM, CNRS et Université de Montpellier, France
tibermacin@lirmm.fr

Bastien Tramoni
LIRMM, CNRS et Université de Montpellier, France
bastien.tramoni@lirmm.fr

Christophe Dony
LIRMM, CNRS et Université de Montpellier, France
dony@lirmm.fr

Ahmed Hadj Kacem
ReDCAD, Université de Sfax, Tunisie
ahmed.hadjkacem@fsegs.rnu.tn

ABSTRACT

Les contraintes d'architecture sont des spécifications définies par les développeurs dans la phase de conception, qui permettent de vérifier, après une évolution de l'architecture, si sa description est encore conforme aux conditions imposées par un patron ou un style architectural. Ces spécifications peuvent être exprimées avec un langage standardisé comme OCL (*Object Constraint Language*). Afin de vérifier ces contraintes d'architecture dans la phase d'implémentation nous devrions soit: i) réécrire entièrement ces contraintes avec un langage qui est interprétable dans cette phase de développement, ou ii) développer de nouveaux interpréteurs pour le langage de contrainte (OCL, par exemple) qui peuvent analyser le code des programmes sur lesquels les contraintes doivent être vérifiées. Ce dernier choix est difficile à mettre en œuvre et nécessite beaucoup de travail. Cependant, même en optant pour la première solution, la tâche de réécrire entièrement les contraintes dans la phase d'implémentation est fastidieuse et source d'erreurs. Nous proposons dans ce travail une méthode pour traduire les spécifications des contraintes d'architecture en composants exécutables. En plus de les rendre vérifiables en phase d'implémentation, nous avons choisi de cibler les composants logiciels afin de rendre ces contraintes d'architecture réutilisables et composables. Puisque les contraintes d'architecture doivent analyser les descriptions d'architecture, les composants générés utilisent le mécanisme de réflexivité standard fourni par le langage de programmation. Notre implémentation prend en entrée des contraintes OCL spécifiées sur le métamodèle UML. Elle produit des composants programmés en COMPO, un langage de programmation par composants.

Keywords

Contrainte d'architecture, Composant, OCL, Introspection

1. INTRODUCTION: CONTEXTE ET PROBLÉMATIQUE

Les contraintes d'architecture sont des spécifications d'invariants qui sont vérifiables par l'analyse des descriptions d'architecture. Ce genre de contraintes ne doit pas être confondu avec les contraintes fonctionnelles, qui sont vérifiables par l'analyse de l'état des composants exécutables constituant l'architecture. Par exemple, si on considère un modèle UML (une description d'architecture) contenant une classe `Employé` (un composant dans cette architecture) qui a un attribut `âge`, une contrainte fonctionnelle OCL représentant un invariant sur cette classe peut tester les valeurs de cet attribut pour qu'elles soient toujours comprises dans l'intervalle 16 à 70. Cette contrainte sera vérifiée sur toutes les instances de la classe *Employé*. Ce genre de contraintes est intrinsèquement dynamique. Elles ne peuvent être vérifiées que lors de l'exécution. D'autre part, les contraintes d'architecture sont des spécifications où les descriptions d'architecture, et non les états des composants, sont analysées [35]. Elles définissent des invariants imposés par le choix d'un style architectural ou un patron de conception, comme le style architectural en couches [30], où "les composants se situant dans les couches non-adjacentes ne doivent pas être connectés directement les uns avec les autres". C'est un exemple d'une contrainte d'architecture. OCL [18] est un exemple de langage de contraintes. C'est un standard de l'OMG qui permet de spécifier les deux sortes de contraintes: fonctionnelles (les contraintes naviguent dans les modèles UML) et architecturales (les contraintes naviguent dans des métamodèles).

Les contraintes fonctionnelles sont utilisées dans la programmation par contrat pour permettre une définition précise et vérifiable des interfaces des composants logiciels [25]. Les contraintes d'architecture sont utilisées lors l'évolution d'une architecture logicielle pour garantir que les changements n'ont pas d'effets néfastes sur les patrons ou les styles d'architecture appliqués, et donc sur la qualité [36].

Contrairement aux contraintes dans la programmation par contraintes, les contraintes d'architecture ne sont pas des conditions qui doivent être satisfaites par une solution dans un problème d'optimisation, où nous devrions trouver une solution optimale parmi une multitude de solutions. Elles sont des conditions qui sont évaluées pour voir si une solution

donnée (notre description d'architecture) respecte les conditions ou pas. Si les conditions ne sont pas satisfaites, nous ne sommes pas amenés à trouver une autre solution. Nous devons changer la solution actuelle (la description d'architecture), en annulant les changements précédents, par exemple, et puis réévaluer les conditions.

Une multitude de contraintes d'architecture ont été formalisées pour les patrons d'architecture existants proposés dans la littérature et la pratique du génie logiciel [40, 14, 6]. Malheureusement, ces contraintes d'architecture sont actuellement vérifiées seulement sur les artefacts de conception (descriptions statiques de la conception de l'architecture). Pour les vérifier en phase d'implémentation, nous avons deux solutions possibles. La première solution consiste à écrire un nouvel interpréteur, utilisable dans la phase d'implémentation, du langage utilisé pour la spécification des contraintes en phase de conception (OCL, par exemple). Mais cette solution peut être intuitivement écartée parce que : i) elle prend trop de temps, et ii) elle oblige les programmeurs à apprendre un autre langage (le langage utilisé pour spécifier les contraintes en phase de conception) pour spécifier, en phase d'implémentation, de nouvelles contraintes d'architecture. La seconde solution possible consiste à réécrire les contraintes entièrement avec des langages utilisés par les développeurs en phase d'implémentation. Malheureusement, cette tâche de réécriture manuelle de toutes ces contraintes est fastidieuse et source d'erreurs.

L'idée dans ce papier est de générer des programmes exécutables représentant les contraintes d'architecture à partir de leurs spécifications définies en phase de conception, de la même manière que du code peut être généré à partir de modèles UML. La plupart des outils existants permettant de générer du code à partir de modèles ne considèrent pas la génération de code pour les contraintes associées à ces modèles. Pour ceux qui le font, comme [27, 28, 9], ils considèrent uniquement les contraintes fonctionnelles et non architecturales.

Dans ce papier, nous proposons de traduire semi-automatiquement[1] les contraintes d'architecture vers des "programmes exécutables" dans la phase d'implémentation. Nous proposons un processus en deux étapes qui prend en entrée des contraintes d'architecture OCL exprimées sur le métamodèle UML et fournit en sortie un ensemble de composants exécutables programmés en COMPO [32], qui est un langage à composants développé par notre équipe. Nous proposons de transformer les contraintes d'architecture en "composants-contraintes" [39] afin que nous puissions les mettre sur "étagères" et par la suite les rendre réutilisables, personnalisables et composables avec d'autres pour produire des contraintes plus complexes. Ces composants-contraintes utilisent le mécanisme d'introspection fourni par le langage de programmation afin d'analyser les descriptions d'architecture et d'examiner la structure de leurs composants à l'exécution (les contraintes d'architecture peuvent donc être vérifiées après une reconfiguration dynamique de l'architecture). Nous avons utilisé l'introspection afin d'exploiter un mécanisme standard fourni par le langage de programmation, sans avoir à recourir à des bibliothèques externes. Le choix de COMPO est motivé par le fait que ce langage fournit de l'introspection. De plus, nous pouvons décrire le code métier de l'application et les contraintes qui lui sont associées avec le même langage à base de composants, dans un environnement unifié.

[1]Une automatisation complète n'est pour le moment pas possible, comme nous le détaillerons plus loin.

La section suivante sera consacrée à la présentation d'un exemple, qui servira à l'illustration de notre travail tout au long de l'article. Dans la section 3, nous présentons brièvement notre approche en indiquant les étapes de traduction des contraintes en composants. Les sections 4 et 5 décrivent ces étapes en détail. Avant de conclure et présenter les perspectives, nous discutons les travaux connexes dans la section 6.

2. EXEMPLE ILLUSTRATIF

Pour mieux comprendre le contexte de notre travail, nous introduisons l'exemple d'une contrainte d'architecture permettant la vérification des conditions topologiques imposées par le patron "*Enterprise Service Bus*" [8]. Ce patron introduit trois catégories de composants : les consommateurs (*consumers*), les fournisseurs (*providers*) et le bus. Le bus est défini comme un adaptateur qui établit la communication entre les consommateurs et les fournisseurs car ils peuvent avoir des interfaces incompatibles. La contrainte d'architecture qui spécifie les conditions imposées par ce patron est exprimée en OCL en utilisant le métamodèle UML dans le Listing suivant. Le métamodèle UML dans lequel la contrainte navigue est présenté dans la Figure 1.

En UML, un composant est une spécialisation d'une classe. Il hérite de toutes les caractéristiques d'une classe (peuvent déclarer des attributs (`properties`) et des opérations, participer à des associations et aux relations d'héritage, etc.). En tant que *encapsulated classifier*, un composant peut définir un ensemble de ports via lesquels il spécifie les interfaces fournies et requises. En tant que *structured classifier*, il peut définir des connecteurs entre des éléments connectables, qui peuvent être des ports, entre autres. Un composant est réalisé par un ou plusieurs classificateurs (*classifiers*). Une spécification détaillée du modèle de composants UML est décrite dans [17].

```
1  context Component inv :
2  let bus:Component
3  = self.realization.realizingClassifier
4  ->select(c:Classifier|c.oclIsKindOf(Component)
5   and c.oclAsType(Component).name='esbImpl'),
6  customers : Set(Component)
7  = self.realization.realizingClassifier
8  ->select(c:Classifier|c.oclIsKindOf(Component)
9   and (c.oclAsType(Component).name='cust1'
10    or c.oclAsType(Component).name='cust2'
11    or c.oclAsType(Component).name='cust3')),
12 producers : Set(Component)
13 = self.realization.realizingClassifier
14 ->select(c:Classifier|c.oclIsKindOf(Component)
15  and (c.oclAsType(Component).name= 'prod1'
16 or c.oclAsType(Component).name='prod2'
17 or c.oclAsType(Component).name='prod3'))
18 in
19 -- Le bus doit avoir au moins un port d'entree
20 -- et un port de sortie
21 bus.ownedPort->exists(p1,p2:Port|
22    p1.provided->notEmpty() and p2.required->
23     notEmpty())
24 and
25 -- Les consommateurs doivent avoir uniquement
26 -- des ports d'entree
27 customers->forAll(c:Component|c.ownedPort
```

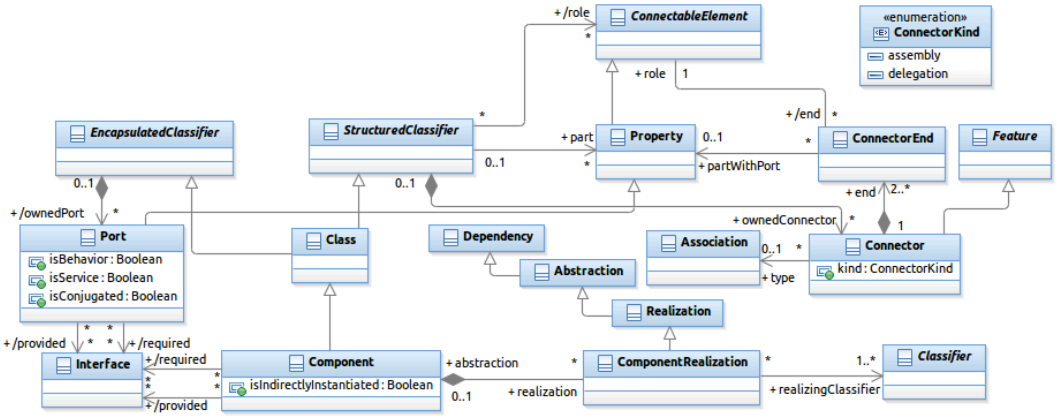

Figure 1: Un extrait du métamodèle UML (Composants et structures composites) [35]

```
28    ->forAll(required->notEmpty()
29      and provided->isEmpty()))
30  and
31  -- Les consommateurs doivent etre connectes
32  -- uniquement au bus
33  customers->forAll(com:Component|
34    com.ownedPort->forAll(p:Port|p.end
35      ->notEmpty()
36      implies
37        self.ownedConnector
38          ->exists(con:Connector|
39            bus.ownedPort->exists( pb:Port|
40              con.end.role->includes(pb)) and
41                con.end->includes(p.end))))
42  and
43  -- Les fournisseurs doivent avoir uniquement
44  -- des ports de sortie
45  producers->forAll(c:Component|
46    c.ownedPort->forAll(provided->notEmpty()
47      and required->isEmpty()))
48  and
49  -- Les fournisseurs doivent etre connectes
50  -- uniquement au bus
51  producers->forAll(com:Component|
52    com.ownedPort->forAll(p:Port|p.end
53      ->notEmpty()
54      implies
55        self.ownedConnector
56          ->exists(con:Connector|
57            bus.ownedPort->exists( pb:Port|
58              con.end.role->includes(pb)) and
59                con.end->includes(p.end))))
```

Listing 1: La contrainte imposée par le patron Entreprise Service Bus en OCL/UML

Dans le Listing 1, dans les lignes 2 à 17, la contrainte filtre l'ensemble des composants internes afin d'obtenir le composant représentant le bus, les composants représentant les consommateurs et les fournisseurs dans la description d'architecture. Le reste de la contrainte vérifie si les consommateurs ont uniquement des ports d'entrée (lignes 27- 29) et les fournisseurs ont uniquement des ports de sortie (lignes 45-47). De plus, elle vérifie si le bus a au moins un port d'entrée et un port de sortie (lignes 21- 23) par lesquels les consommateurs et les fournisseurs sont connectés.

Après traduction de cette contrainte en utilisant l'approche présentée dans ce papier, le résultat est un ensemble de descripteurs de composants COMPO. Deux d'entre eux sont présentés dans le listing suivant :

```
1  Descriptor BusArchitecture extends Constraint{
2    internally requires {
3      bus <: BusConstraint;
4      subPort <: PortConstraint;
5      toConnect <: ToConnectBusConstraint;
6    }
7     service verify(...) {
8      |c1 c2 c3 c4 c5|
9      c1 := bus.isBusStructure('esbImpl');
10     c2 := subPort.isPortInputOnly({'cust1',
11               'cust2','cust3'});
12     c3 := subPort.isPortOuputOnly({'prod1',
13               'prod2','prod3'});
14     c4 := toConnect.isConnectToBus('esbImpl',
15               {'cust1','cust2','cust3'},);
16     c5 := toConnect.isConnectToBus('esbImpl',
17               {'prod1','prod2','prod3'});
18       return (c1 & c2 & c3 & c4 & c5);
19    }
20 }
```

Listing 2: Le descripteur BusArchitecture en COMPO

```
1  Descriptor ToConnectBusConstraint
2       extends Constraint{
3   service isConnectToBus(busName,setOfCompNames)
4   {
5    |conns bus result|
6    result=true;
7    bus=context.getDescriptor(busName);
8    setOfCompNames.each([:i|
9         |comp|
10      comp=context.getDescriptor(setOfCompNames
11           .getItemAt(i));
12      conns:=comp.getDescribedConnections();
13      if(!(conns.each([:cd|cd.srcPortDesc()==bus.
14         or([cd.destPortDesc()==bus)]))
15           {result :=result & false};
16      return result;
17   }
18 }
```

Listing 3: Le descripteur ToConnectBusConstraint en COMPO

Il existe au total cinq sous-contraintes dans la contrainte d'architecture (Listing 1). Chacune (lignes 21 - 23, 27 - 29,

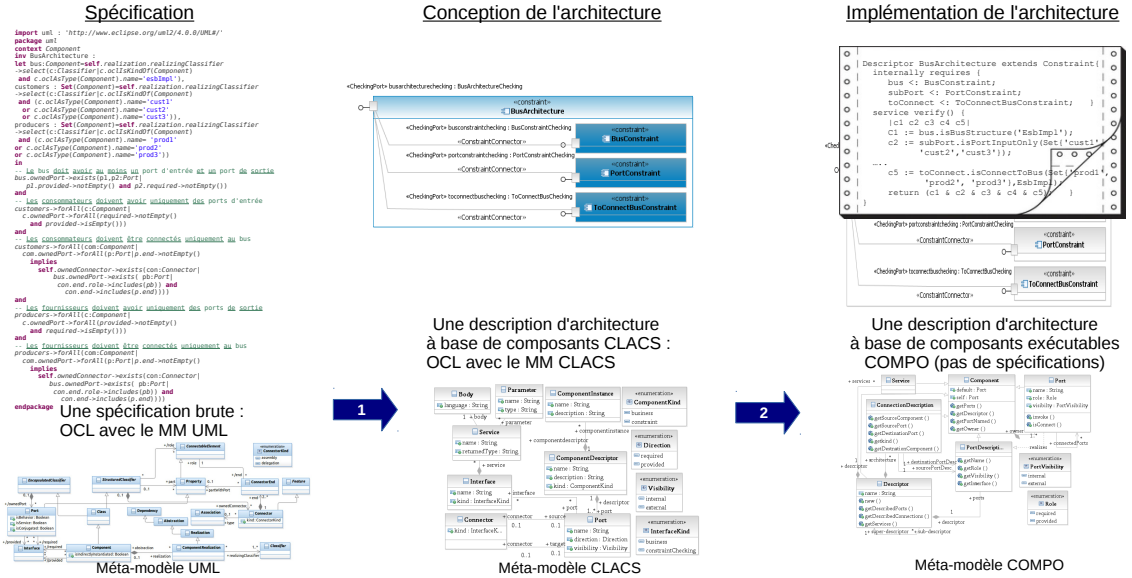

Figure 2: Le processus de traduction

33 - 41, 45 - 47 et 51 - 59) est supposée être définie individuellement dans un descripteur de composant. Mais dans cet exemple, les sous-contraintes 2 et 4 peuvent être regroupées dans le même descripteur de composant (`PortConstraint`) car elles vérifient les mêmes aspects (voir la Section 4, étape 5 pour plus de détails). Elles vérifient si tous les composants d'un ensemble donné (`customers` dans la première sous-contrainte et `producers` dans la deuxième) ont des ports d'un type particulier (d'entrée ou de sortie). Le descripteur `PortConstraint` fournit deux services permettant la vérification de ces deux sous-contraintes. D'autre part, les sous-contraintes 3 et 5 vérifient exactement le même invariant (contrairement aux sous-contraintes 2 et 4) sauf qu'elles sont appliquées sur des ensembles de composants différents (`customers` pour la sous-contrainte 3 et `producers` pour la sous-contrainte 5). Ainsi, dans le résultat de la traduction, il existe un unique descripteur de composant (`ToConnectBusConstraint`) qui est généré pour ces deux sous-contraintes. Ce composant-contrainte fournit un seul service qui est paramétrable avec un ensemble de composants sur lesquels la contrainte doit être vérifiée.

Pour cet exemple, nous allons obtenir, en plus du descripteur du composant principal (`BusArchitecture`), trois composants (au lieu de cinq). `BusArchitecture` décrit un composant composite qui est composé de trois composants. Pour des raisons de simplicité un seul descripteur est présenté dans le Listing 3. C'est le descripteur qui correspond au troisième composant du composite. Il définit un service qui vérifie la contrainte et qui sera exporté par ses ports. Pour vérifier cette contrainte, nous utilisons les méthodes de réflexivité de COMPO (`getDescribedConnections()` par exemple) pour introspecter les éléments d'architecture.

A travers cette "componentification", les descripteurs des composants-contraintes de COMPO peuvent être réutilisables (instanciés plusieurs fois dans différents contextes), composables (les instances peuvent être connectées entre elles ou connectées dans un composant composite pour construire des composants-contraintes plus complexes) et paramétrables (pour vérifier par exemple que `customers` ou `producers` sont connectés uniquement au `bus`, nous pouvons passer les bons arguments au service du descripteur `ToConnectBusConstraint`).

Ces composants-contraintes sont exécutables. Le descripteur du composite (`BusArchitecture`) doit être instancié par un architecte et connecté a un composant métier sur lequel elle/il veut vérifier le patron. Ce composant métier doit donc déclarer un port d'entrée pour vérifier le patron et invoquer le service du composant-contrainte. C'est dans ce composant métier (dans l'invocation du service) que les bons arguments doivent être passés au composant-contrainte.

3. APPROCHE GÉNÉRALE

Nous proposons un processus composé de deux étapes. La Figure 2 donne un aperçu général de ce processus. Dans la première étape, nous modifions le format des contraintes à partir d'une spécification textuelle "brute"[2] (Listing 1) vers une description d'architecture constituée de "composants-contraintes". Ces composants sont décrits avec un ADL nommé CLACS [39] (prononcé Klax), qui sera présenté dans la section suivante. La deuxième étape consiste à générer du code COMPO à partir de CLACS. Ces deux étapes seront décrites en détail dans les sections qui suivent.

Nous n'avons pas effectué une traduction directe de OCL/UML vers COMPO parce que cette traduction nécessite plusieurs transformations en même temps : modification de la syntaxe

[2]Nous désignons par une spécification "brute", une spécification qui n'offre pas assez de structure, réutilisation et paramétrage.

des contraintes, leur décomposition, leur migration vers un nouveau métamodèle, l'introduction d'une structure autour de ces contraintes, entre autres.

Dans notre approche, nous utilisons deux langages : CLACS, un langage de description d'architecture, et COMPO, un langage réflexif de programmation par composants. Dans la littérature, il existe plusieurs langages permettant la spécification de contraintes d'architecture (voir [35] pour un état de l'art). Chacun a ses avantages et son domaine d'application particulier. Mais, CLACS est l'unique langage qui fournit un modèle de composants pour la spécification des contraintes d'architecture. Les contraintes d'architecture modélisées avec ce langage sont des composant-contraintes dans lesquels les invariants sont encore spécifiés en utilisant OCL. Mais ces contraintes OCL naviguent dans le métamodèle de CLACS et non dans celui d'UML. Ceci est indiqué en bas de la Figure 2. Le premier métamodèle est celui d'UML dans lequel les contraintes sont initialement définies (comme l'exemple présenté dans la section précédente). Le choix d'UML est simplement motivé par le fait qu'il est un standard industriel[3], et que OCL est son langage de contraintes original. Nous pouvons considérer ici un référentiel des contraintes d'architecture qui peut être alimenté par la communauté de l'architecture logicielle, en utilisant ces langages généraux de modélisation (faciles à apprendre, comme cela a été expérimenté dans [4]), qui sont UML et OCL.

Le deuxième métamodèle décrit la syntaxe abstraite de CLACS. Comme il a été indiqué précédemment, les contraintes générées dans la première étape du processus sont intégrées dans les composants CLACS. Mais ces contraintes sont encore définies en OCL (d'où l'utilisation d'OCL/CLACS comme le nom du langage de modélisation des composants-contraintes). Cependant, les expressions OCL ici naviguent dans le métamodèle CLACS et non dans celui d'UML, parce que ces contraintes sont vérifiables dans cette phase sur des descriptions d'architecture métiers définies en CLACS.

Le dernier métamodèle est celui de COMPO. Les composants générés dans la dernière étape utilisent le mécanisme d'introspection de COMPO dont l'API est fournie dans ce métamodèle (l'opération `getDescribedConnections()`, par exemple). Dans le résultat final, il n'y pas des spécifications. Elles sont remplacées par un code exécutable en COMPO.

4. TRANSFORMATION DES CONTRAINTES EN COMPOSANTS-CONTRAINTES

Dans cette section, nous décrivons la transformation des contraintes OCL en composants-contraintes CLACS. La Figure 3 montre une partie du métamodèle de CLACS. Un composant CLACS est une instance d'un descripteur de composant (tout comme un objet est l'instance d'une classe). Un composant a un nom, une description et un genre : `business` ou `constraint`. Il déclare des ports qui sont caractérisés par une orientation et une visibilité. Chaque port a une interface qui spécifie un ensemble de signatures de services. Les ports sont liés par des connecteurs. Un connecteur reçoit des invocations de services à travers ses ports sources et les transmet à travers ses ports cibles. Dans notre travail, nous nous concentrons sur les composants de type `constraint` et des interfaces et des connecteurs de type `constraintChecking`.

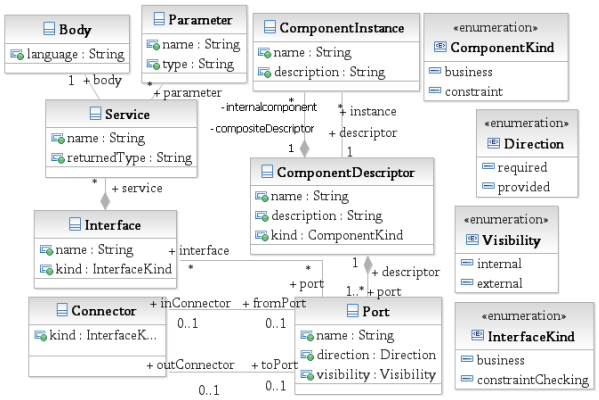

Figure 3: Un extrait du métamodèle CLACS

Les contraintes OCL sont des prédicats, dans la logique du premier ordre. Elles ont une syntaxe simple, intuitive et concrète. Même si les transformations présentées dans ce papier sont appliquées sur OCL, notre travail peut être généralisé à d'autres langages de prédicats équivalents. Cela n'est pas démontré expérimentalement dans notre travail, mais le lecteur peut constater que les transformations effectuées sont généralisables.

Pour générer les composant-contraintes, nous proposons un micro-processus de plusieurs étapes :

1. **Décomposition des contraintes :** Premièrement, nous décomposons les spécifications de contraintes textuelles en un ensemble de sous-contraintes. Cette décomposition est basée sur les opérateurs logiques du plus haut niveau (Lignes 24, 30, 42 et 48 dans le Listing 1). Les opérandes de ces opérateurs sont considérées ici comme les sous-contraintes. Cet ensemble de sous-contraintes est raffiné récursivement en un arbre de sous-contraintes quand ces sous-contraintes peuvent être décomposées de nouveau. La condition d'arrêt de la récursivité est quand la sous-contrainte à décomposer est considérée atomique. Une sous-contrainte est considérée atomique quand elle ne contient pas des opérateurs logiques et quand elle contient au moins deux navigations. Cette dernière condition nous évite de se retrouver avec des contraintes de taille inférieure à la taille des instructions d'invocation de service qui permet leur vérification : lorsqu'une contrainte est embarquée dans un composant-contrainte, sa vérification est effectuée par l'invocation du service fourni par ce composant ; et pour invoquer ce service à l'intérieur d'une contrainte, nous utilisons le port d'entrée de ce composant comme suit:
   `portName.checkingServiceName(...)`
   Cette invocation retourne une valeur booléenne. Cette invocation de service est vue comme une expression

[3]Même si une étude empirique récente [29] a mis en avant le fait qu'UML n'est pas entièrement (mais sélectivement) utilisé par les développeurs en industrie, et qu'il est utilisé de façon informelle, il est communément admis qu'UML est la norme *de facto* dans la modélisation du logiciel, connue par un grand nombre de développeurs.

OCL avec une seule navigation (une seule utilisation de l'opérateur "."). Il n'est pas intéressant d'embarquer une contrainte incluant une seule navigation vers un composant qui sera utilisé à travers une expression de taille équivalente (une invocation de services, avec la même longueur approximativement). De plus, notre expérience avec les contraintes d'architecture nous conduit à affirmer que les expressions de petites tailles ne sont pas utiles pour la réutilisation.

Pour chaque nœud dans l'arbre obtenu avec la décomposition de la contrainte, nous générons un service dans un descripteur de composant CLACS. Ensuite, nous intégrons chaque sous-contrainte dans un service.

2. **Fusion des déclarations :** Dans cette étape optionnelle, nous fusionnons toutes les déclarations (les expressions OCL `let` potentielles) avec les sous-contraintes identifiées. Nous remplaçons les noms des variables (`bus`, `customers` et `producers` dans notre exemple) par les expressions qui sont définies après le symbole `=`. Par exemple, la déclaration de la variable "bus" est fusionnée dans la première sous-contrainte comme suit:

```
1 self.realization.realizingClassifier
2 ->select(c:Classifier |
3  c.oclIsKindOf(Component) and
4  c.oclAsType(Component).name='esbImpl')
5 .ownedPort->exists(p1,p2:Port |
6   p1.provided->notEmpty()
7   and p2.required->notEmpty())
```

Listing 4: La première sous-contrainte fusionnée avec la déclaration de "bus"

3. **Paramétrisation des contraintes :** En créant les signatures des services qui intègrent les sous-contraintes, nous ajoutons des paramètres pour chaque valeur littérale définie dans ces sous-constraintes. De plus, nous créons des paramètres pour chaque variable de quantificateur d'une sous-contrainte, qui est imbriquée dans un quantificateur et qui utilise cette variable, comme `con` et `p` dans la sous-contrainte `con.end->includes(p.end)` (Ligne 59 dans le Listing 1). Le type de ces paramètres est obtenu à partir de l'arbre syntaxique abstrait de la contrainte. C'est dans cette étape que nous fusionnons les descripteurs de composants équivalents. Des descripteurs de composants sont équivalents s'ils fournissent chacun un service embarquant la même contrainte d'architecture (par contre, ces services seront utilisés avec des arguments différents). Par exemple, les sous-contraintes 3 et 5 dans le Listing 1 seront intégrées dans un seul descripteur de composant.

4. **Validation des signatures de services :** La quatrième étape consiste à valider l'ensemble des sous-contraintes et les signatures de services. Nous demandons au développeur de valider les sous-contraintes et les paramètres identifiés, et de définir les noms des services, des interfaces et des ports, afin d'obtenir des composants-contraintes bien décrits (avec des identificateurs significatifs). L'identification des composants-contraintes peut détecter plusieurs paramètres. Certains d'entre eux peuvent être inutiles pour la *componentification* des contraintes. Cela ne peut pas être détecté automatiquement, et pour cette raison le développeur est impliqué pour valider les résultats.

5. **Groupement des services:** Les services qui vérifient les mêmes "aspects" sont intégrés ensemble dans le même descripteur de composant. Par "vérifier des aspects similaires", nous voudrions dire : vérifier les connexions, tester les types, ou d'autres propriétés d'un élément architectural donné (un port ou un connecteur, par exemple). Pour le moment, ce groupement est effectué manuellement par l'utilisateur. Mais nous sommes en train de développer un processus automatique pour analyser les arbres syntaxiques abstraits des sous- contraintes. Chaque paire d'arbres est comparée. Ces arbres doivent partager une racine commune et au minimum un sous-arbre commun (obtenu par un parcours hiérarchique en largeur – *breadth-first traversal*). Cela garantit, dans une certaine mesure, que les sous-contraintes définissent des prédicats sur le même type d'éléments architecturaux qui sont obtenus à travers les navigations dans la contrainte (reflétés par ces sous-arbres). Pour le reste du sous-arbre, une mesure de distance d'édition (*tree edit distance* [33]) est mesurée entre chaque paire de sous-arbres. Si cette mesure est inférieure à un seuil[4], nous considérons que les deux sous-contraintes sont similaires. Donc, ces services seront embarqués dans le même descripteur de composant. Par exemple, les sous-contraintes 2 et 4 vérifient le même aspect qui est un type de l'élément architectural (un `Port`). Les deux arbres de ces deux sous-contraintes ont une racine commune qui est un composant et un sous-arbre commun généré à partir de l'expression :
`.ownedPort->includes(p1,p2:Port| )`
Pour les sous-arbres générés à partir du reste des deux sous-contraintes, nous remarquons qu'il existe une similarité entre eux (uniquement deux opérations d'édition (deux substitutions de nœuds) : `required` and `provided` sont inversés). Donc, ces deux sous-contraintes sont groupés comme deux services dans un même descripteur de composant.

Ces deux dernières étapes ne sont pas complètement séparées. Il y a des allers-retours entre ces étapes afin de valider l'architecture à base de composants de la contrainte (les interfaces des composants-contraintes).

6. **Refactoring des contraintes :** Les contraintes sont, par la suite, refactorisées pour qu'elles : i) utilisent les paramètres formelles validés, et ii) invoquent les services crées où les sous-contraintes sont embarquées. Les arguments sont passés, dans ces invocations, selon ce qui est spécifié dans la contrainte de départ (les littéraux ou les variables identifiés dans l'étape 3, ou utiliser les valeurs des paramètres validés dans l'étape 4). Dans notre exemple, la première sous-contrainte est refactorisée et nous obtenons une première contrainte, donnée dans le Listing 4. Par la suite, cette contrainte sera modifiée encore une fois pour prendre en compte le paramètre validé `String busName` et devient comme suit:

```
1 self.realization.realizingClassifier
```

[4]La valeur de ce seuil sera fixée empiriquement.

```
2 ->select(c:Classifier |
3   c.oclIsKindOf(Component)
4   and c.oclAsType(Component).name=busName)
5 .ownedPort->exists(p1,p2:Port|
6   p1.provided->notEmpty()
7   and p2.required->notEmpty())
```

Listing 5: La première sous-contrainte avec un paramètre validé

La contrainte précédente est vérifiée par le premier composant-contrainte interne. Dans le composant- contrainte composite (`BusArchitecture`), cette contrainte est remplacée par l'invocation de service suivante:
`busChecking.isBusArchitecture(busName)`
où `busChecking` est le nom du port du premier composant-contrainte interne du composite et `busName` est le paramètre formel du service fourni par le composite.

7. **Migration de métamodèle :** Finalement, nous transformons les navigations des contraintes écrites dans OCL/UML vers OCL/CLACS. Cela est effectué en utilisant un ensemble simple de mappings déclaratifs définis entre les deux métamodèles (UML et CLACS). Pour des raisons de limitation de places, nous ne présentons pas ces mappings. Un exemple de cette migration est présenté dans la Figure 4 (voir les contraintes présentées à gauche et à droite).

8. **Génération de la description d'architecture CLACS :** A partir de l'arbre obtenu dans la première étape, une description de l'architecture à base de composants sera générée. Cette description d'architecture contient tous les composant-contraintes (instances) nécessaires connectés entre eux. Ces composants intègrent les contraintes d'architecture refactorisées qui naviguent dans le métamodèle de CLACS.

La Figure 4 illustre la description d'architecture complète des composants-contraintes générés à partir de notre exemple précédent (Listing 1). Nous pouvons voir (en haut de la figure) la contrainte vérifiée par le composite, où il y a les cinq invocations de services vers les trois composants internes (présentés en bas de la contrainte).

Dans la Figure 4, les contraintes à droite sont spécifiées en OCL et naviguent dans le métamodèle CLACS (contrairement à celles qui sont à gauche, qui naviguent dans le métamodèle UML). De plus, le mot clé `self` est remplacé par `context`, qui est évalué par une référence au port requis connecté au composant métier sur lequel la contrainte va être vérifiée. Cette résolution de connexion est effectuée lorsque la vérification est lancée.

A ce niveau, les composants-contraintes peuvent être vérifiées statiquement en phase de conception[5]. Afin de pouvoir les vérifier dynamiquement dans la phase d'implémentation, nous avons besoin de les transformer en composants exécutables. La vérification dynamique des contraintes utilisera la réflexivité (l'introspection) fournie par le langage de programmation.

[5]L'environnement CLACS peut être téléchargé ici : https://code.google.com/p/clacs/.

5. GÉNÉRATION DE COMPOSANTS EXÉCUTABLES À PARTIR DES COMPOSANTS-CONTRAINTES

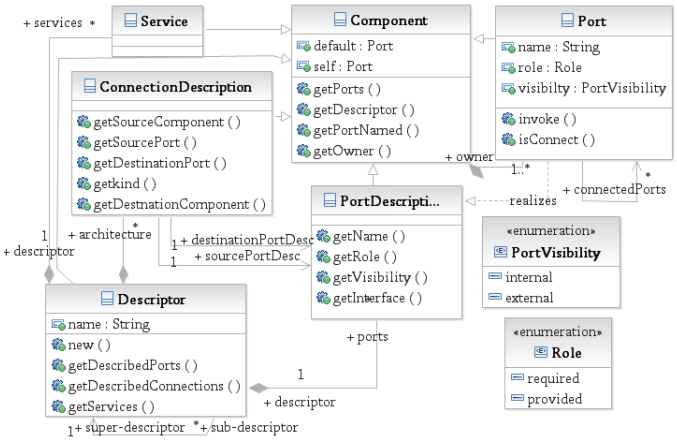

Figure 5: Un extrait du métamodèle COMPO présentant l'intégration de l'introspection [32]

Un extrait du modèle de composant COMPO est décrit par le métamodèle de la Figure 5 (seuls les concepts, les attributs et les opérations utilisés dans ce papier sont présentés dans cette figure et dans le texte qui suit). Les composants sont des instances de descripteurs. Un descripteur définit la structure (ports et description d'architecture interne) et le comportement (implémentation des services) de ses instances. Un composant composite possède des composants internes, *i.e.* les composants auxquels il est connecté via ses ports requis internes et qui sont inaccessibles en dehors du composant. Un port est un point de connexion. Il a un rôle (requis/fournis), une visibilité (interne/externe), un nom et une interface.

Ce qui rend COMPO réflexif est le fait que `Descriptor`, `Port` et `Service`, entre autres, tous héritent de la métaclasse `Component`. L'introspection est fournie principalement par les méthodes accesseurs de COMPO qui sont définies dans ses métaclasses, comme `getDescriptor()` et `getSourceComponent()`. Par exemple, la métaclasse `Descriptor` fournit des opérations pour obtenir les ports et les connexions. Les opérations (getter) dans chaque métaclasse du métamodèle peuvent être utilisées pour spécifier les contraintes d'architecture.

COMPO offre un support permettant de fournir un paradigme uniforme pour les composants métiers et les contraintes avec un même langage et dans un même environnement.[6] (même éditeur, même interpréteur et même débogueur). Les utilisateurs de COMPO pourraient construire les composants-contraintes en créant des sous-descripteurs du descripteur `Constraint` (des descripteurs qui héritent de ce dernier) qui est le descripteur de base pour tous les descripteurs des composants-contraintes. Chaque descripteur de composant-contrainte déclare un port `context` qui sera connecté au composant métier sur le quel la contrainte est vérifiée. Par défaut, chaque descripteur de composant-contrainte fournit

[6]L'environnement de COMPO peut être téléchargé ici: http://www.lirmm.fr/~spacek/compo/.

Figure 4: Composants-contraintes pour l'architecture en bus

un service qui retourne une valeur booléenne pour vérifier son contexte courant.

Afin de générer ce type de descripteurs, nous considérons quelques règles de base. Par exemple, les opérateurs arithmétiques, logiques ou de comparaison restent inchangés. Ils sont les mêmes dans OCL et COMPO. Les opérations et les quantificateurs OCL sont transformés en des services internes privés.

La génération de code COMPO est effectuée par l'algorithme 1. Cet algorithme représente les étapes principales dans le processus de génération de code. Le point initial de notre processus est un AST généré à partir de la contrainte qui est intégrée dans le composant-contrainte. Nous effectuons un parcours hiérarchique en profondeur (*depth-first pre-order*) sur cet arbre. Nous analysons le nœud courant et nous effectuons un ensemble d'opérations selon le type de nœud rencontré. Si le nœud courant est un élément du métamodèle CLACS, nous le remplaçons par son équivalent en COMPO, en utilisant des mappings (La procédure `getMapping(current)`). Cette procédure transforme chaque rôle ou navigation définis dans le métamodèle CLACS par l'invocation d'un service d'introspection (accesseur) équivalent défini dans COMPO. Par exemple, `port` devient `getDescribedPorts()`. Tous les mappings sont définis indépendamment de cet algorithme.

Si le nœud courant est un quantificateur, la procédure (`generateQuantifier(current)`) est appelée. Elle génère un ser-

```
Data: Abstract Syntax Tree generated from OCL
      constraint written on CLACS metamodel
Result: A primitive constraint descriptor in COMPO
PROCEDURE generateCompo(oclExpression)
while AST traversal of oclExpression not finished do
    read current;
    if current is not ocl term then
        store current;
    end
    if current is a metamodel element then
        getMapping(current);
    end
    if current is a quantifier then
        generateQuantifier(current);
        generateCompo(parameterOf(current));
    end
    if current is an operation then
        /* isEmpty(),etc                          */
        generateOperation(current);
        if current has parameters then
            /* includes(..),etc                   */
            generateCompo(parameterOf(current));
        end
    end
end
END
```

Algorithm 1: Algorithme de génération de code COMPO

OCL	COMPO
`forAll(ex:OclExpression): Boolean`	`Collection.each([:c| if(!exInCOMPO) return false; ]); return true;`
`exists(ex:OclExpression): Boolean`	`Collection.each[[:c| if(exInCOMPO) return true; ]); return false;`

Table 1: Code COMPO généré pour les quantificateurs OCL

vice privé représentant le quantificateur et après, dans le corps du service de la contrainte, elle crée une invocation de ce nouveau service privé et enregistre la valeur de retour. Par exemple, si `current` est `...->forAll(p:Port | ...)`, `generateQuantifier(current)` crée :
`resultforall1 := forall1(param);`
`param` est créé à partir du nœud de l'AST généré pour le quantificateur (voir Table 2 colonne 2 ligne 3). Ensuite, l'implémentation de ce service (`forall1`) est intégrée. Cette intégration est faite par la procédure `generateCompo(..)`. Le corps de ce service est généré d'une façon récursive. Quand l'AST du quantificateur est traversé, nous appelons toujours la même procédure qui utilise des squelettes (*templates*) de code pré-construites pour chaque quantificateur OCL. Dans le cas d'un quantificateur imbriqué (deux quantificateurs sont définis l'un à l'intérieur de l'autre), le deuxième quantificateur utilise fréquemment les variables du premier pour définir ses expressions. Dans ce cas, nous stockons les variables du premier quantificateur (les paramètres du service qui correspond au premier quantificateur) afin de les passer parmi les paramètres du service correspondant au deuxième.

Le même mécanisme est utilisé pour les opérations de collection OCL. La procédure `generateOperation(current)` génère un service privé dans le code (la même démarche utilisée pour `generateQuantfier(current)`). Si l'opération possède des paramètres comme `includes(...)`, nous appelons `generateCompo(..)`. Ces services ont, par défaut, des squelettes d'implémentation, donc nous ne les générons pas à partir de rien.

Pour illustrer cette génération de code, nous appliquons l'algorithme décrit précédemment sur notre exemple. Pour des raisons de simplicité, nous présentons uniquement le code généré pour le premier service du descripteur `PortConstraint`.

Dans le Tableau 2, nous décrivons pour chaque expression OCL son équivalent en COMPO. La troisième colonne décrit étape par étape la génération de code du corps du service du composant en se basant sur l'algorithme précédent.

Nous avons remarqué que le code généré est syntaxiquement différent du code optimal prévu pour notre exemple (le code présenté dans la section 2). Mais ils sont sémantiquement équivalents. Il est évident que la traduction automatique ne permet pas d'obtenir un code ayant une complexité optimale. Elle fournit cependant une aide précieuse pour les développeurs, qui pourront plutôt se concentrer sur l'implémentation de la logique métier de leur application. Ils peuvent optimiser ce code après, si nécessaire.

6. TRAVAUX CONNEXES

Les travaux connexes peuvent être classifiés dans différentes catégories : i) des langages et des outils pour la spécification des contraintes en phase de conception et d'implémentation, ii) les techniques de transformation des prédicats/contraintes, et iii) des méthodes de génération de code à partir de prédicats. Un état de l'art sur les langages utilisés pour la spécification des contraintes d'architecture en phase de conception et d'implémentation est présenté dans [35]. Ces langages sont pour certains des notations intégrées dans des ADLs existants, comme Armani [26] pour Acme [15], FScript[7] pour Fractal ADL [5] ou REAL [16] pour AADL [12]. D'autres sont des notations avec le style de programmation logique, comme LogEn [10] ou Spine [2], ou des notations avec (ou pour) le style de programmation à objets, comme CDL [24], DCL [34] ou SCL [21]. Dans la pratique, il existe différents outils pour l'analyse statique de code qui permettent la spécification de contraintes d'architecture, comme Sonar (`http://www.sonarqube.org/`), Lattix (`http://lattix.com/`), Architexa (`http://www.architexa.com/`), et Macker (`https://innig.net/macker/`), entre autres. Tous ces langages et outils n'offrent pas la possibilité de transformer ou de générer du code à partir de spécifications OCL ou un autre langage de prédicat. De plus, ils ne fournissent pas de moyens (ou

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

Contrainte en CLACS	Contrainte en COMPO	Exemple illustratif
`self.internalComponent`	`interComps:= context.getInterComponents();`	`service isPortOutputOnly(OrderedSet{'cust1','cust2','cust3'}){ |result| interComps:=context.getInterComponents(); return result; }`
`-> select(ci:ComponentInstance| ci.name='cust1' or ci.name='cust2' or ci.name='cust3')`	`interComps ->select([ci:Component| ci.getName()='cust1' or ci.getName()='cust2' or ci.getName()='cust3']);`	`service isPortOutputOnly(OrderedSet{'cust1','cust2','cust3'}){ |result| interComps:=context.getInterComponents(); interComps->select([ci:Component|ci.getName()='cust1' or ci.getName()='cust2' or ci.getName()='cust3']); return result; }`
`-> forAll(ci:ComponentInstance| ci.descriptor.port ->forAll(p:Port| p.kind=#required))`	`/*generateQuantifier()*/ resultforall1:= forall1(interComps);`	`service isPortOutputOnly(OrderedSet{'cust1','cust2','cust3'}){ |result| interComps:=context.getInterComponents(); interComps->select([ci:Component|ci.getName()='cust1' or ci.getName()='cust2' or ci.getName()='cust3']); resultforall1:=forall1(interComps); result:=result and resultforall1; return result; } service forall1(interComps){ //To be completed }`
`ci.descriptor.port`	`ports:= ci.getDescribedPorts();`	`service forall1(interComps){ |ports| interComps.each([:ci|ports:=ci.getDescribedPorts(); //To be completed }`
`->forAll(p:Port| p.kind=#required)`	`resultforall2:= forall2(ports);`	`service forall1(interComps){ |ports resultforall2| interComps.each([:ci| ports:=ci.getDescribedPorts(); and resultforall2:= forall2(ports); if(!resultforall2) return false;]); return true; } /*generateCompo(..)*/ service forall2(ports){ ports.each([p:| if(!(p.getKind()='required')) return false;]); return true; }`

Table 2: Exemple de génération de code source COMPO à partir de composant-contraintes OCL/CLACS

fournissent des moyens très limités) pour paramétrer et/ou réutiliser des contraintes d'architecture.

Hassam *et al.* [20] ont proposé une méthode de transformation de contraintes OCL lors du *refactoring* des modèles UML, en s'appuyant sur des transformations de modèles. Leur approche consiste à préparer un ensemble ordonné des opérations pour le *refactoring* des contraintes OCL après le *refactoring* du modèle UML. Pour cela ils utilisent d'abord une méthode d'annotation du modèle UML source pour obtenir un modèle cible annoté en s'appuyant sur une transformation de modèles. Par la suite ils prennent les deux annotations pour former une table de *mapping* qui sera utilisée finalement avec RoclET [22] pour transformer les contraintes OCL du modèle source en des contraintes OCL conformes au modèle cible. Nous trouvons que l'idée d'utiliser un compilateur OCL existant est plus simple. Leur solution de transformation des contraintes est lourde à mettre en place, elle nécessite des connaissances sur les outils et les langages de transformation de modèles.

Dans [13], les auteurs proposent une approche pour générer (instancier) des modèles à partir des mètamodèles en prenant en compte les contraintes OCL. Leurs approche est basée sur CSP (*Constraint Satisfaction Problem*). Ils définissent des règles formelles pour transformer les modèles et les contraintes qui lui sont associées. Cabot *et al.* [7] travaillent aussi sur la transformation de modèles UML/OCL vers CSP afin d'analyser les attributs de qualité des modèles. Ces approches sont similaires à notre processus de transformation étant donné que les artefacts transformés/manipulés sont les mêmes (des métamodèles et des spécifications OCL). Ils utilisent le même compilateur OCL que nous (Dresden-OCL [9]) pour analyser les contraintes. Mais, dans notre ap-

proche, nous effectuons de plus une génération de code afin de rendre les contraintes exécutables avec le code métier de l'application en phase d'implémentation. Contrairement à CSP, cela ne nécessite pas un outil externe pour l'interprétation des contraintes. De plus, nous transformons uniquement les contraintes. Dans les autres approches, tout doit être transformé en CSP pour être résolu (les contraintes + les modèles/les métamodèles).

Dans la pratique de l'ingénierie des modèles, il existe des outils pour la traduction des contraintes en code source Java, comme Eclipse OCL (`http://wiki.eclipse.org/OCL`), Octopus (`http://octopus.sourceforge.net/`) et Dresden OCL (`http://www.dresden-ocl.org/index.php/DresdenOCL`).Ces outils transforment les contraintes fonctionnelles et non architecturales. Ils transforment ce type de contraintes en des programmes à objets qui n'utilisent pas le mécanisme d'introspection. Briand *et al.* dans [3] proposent une approche pour transformer les contraintes fonctionnelles en Java en utilisant la programmation par objets. Un autre travail [19] propose une méthode de traduction des contraintes fonctionnelles en JML (*Java Modeling Language*). Dans un travail précédent [23], nous avons développé une méthode pour la transformation des contraintes d'architecture en des métaprogrammes Java. Mais dans ce travail, le résultat des transformations n'est pas une contrainte d'architecture réutilisable et personnalisable. C'est pourquoi nous proposons dans ce papier une traduction des contraintes vers des composants. Dans un autre travail [37], nous avons développé une méthode basée sur la serialisation des contraintes OCL en XML et leur transformation en XSLT, de la phase de conception vers la phase d'implémentation. Le résultat final ici est un ensemble de contraintes qui nécessitent un interpréteur OCL. De plus, les résultats obtenus après la spécification sont encore des spécifications "brutes" qui n'offrent pas de possibilités de réutilisation et de personnalisation. Par ailleurs, les contraintes dans la phase d'implémentation ne peuvent pas être vérifiées à l'exécution (seule l'analyse statique de descriptions d'architecture, dans le modèle de composants CORBA, peut être effectuée [38]).

7. CONCLUSION AND FUTURE WORK

Les contraintes d'architecture sont des spécifications de prédicats qui apportent une aide précieuse aux développeurs pour préserver les styles et les patrons d'architecture dans une application donnée après avoir évolué sa description d'architecture [36]. Actuellement, ces contraintes d'architecture sont vérifiées statiquement sur les artefacts de conception. Mais la description d'architecture existe aussi dans la phase d'implémentation, à l'intérieur des programmes, et même à l'exécution. Ceci est attesté par l'existence d'un grand nombre de langages de programmation par composants [31] ou des frameworks d'adaptation dynamique, et plusieurs implémentations dans le domaine des *models@run.time [1].* Si les programmes sont évolués statiquement ou si l'architecture est modifiée lors de l'exécution (à travers une adaptation dynamique, par exemple), les contraintes d'architecture peuvent ne plus être respectées. Il est donc important de pouvoir les vérifier à cette phase du cycle de vie de l'application. Nous avons présenté dans ce papier un processus de génération de code à partir de spécifications de contraintes d'architecture. Notre processus est composé de deux étapes. la première consiste à générer semi-automatiquement des composants-contraintes à partir des spécifications textuelles "brutes" des contraintes. Elles sont décrites par un ADL nommé CLACS. La deuxième étape génère des composants exécutables permettant la vérification des contraintes dans la phase d'implémentation et à l'exécution. Les descripteurs de composants générés utilisent les capacités réflexives du langage de programmation. Ces descripteurs sont définis avec un langage à base de composants nommé COMPO.

Comme perspectives à ce travail, nous projetons de migrer ce processus de transformation vers un autre paradigme de développement, qui est celui des architectures à services (SOA). Nous voudrions proposer un langage pour spécifier les contraintes des patrons SOA [11] et une méthode pour les interpréter dans les phases de conception et d'implémentation. Ceci nous permettra au final de généraliser cette approche, *i.e.* de spécifier les contraintes d'architecture indépendamment d'un paradigme donné, en utilisant des prédicats appliqués sur des graphes. Par la suite, nous effectuerons, en fonction des besoins, des transformations automatiques vers les architectures à objets, à composants ou à services.

8. REFERENCES

[1] N. Bencomo, R. France, B. H. C. Cheng, and U. Aßmann. *Models@run.time: Foundations, Applications, and Roadmaps*, volume 8378 of *Lecture Notes in Computer Science*. Springer, 2014.

[2] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of design patterns in java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE'05)*, pages 224–232. ACM, 2005.

[3] L. C. Briand, W. Dzidek, and Y. Labiche. Using aspect-oriented programming to instrument ocl contracts in java. *Technical Report, Carlton University, Canada*, 2004.

[4] L. C. Briand, Y. Labiche, M. Di Penta, and H. D. Yan-Bondoc. An experimental investigation of formality in uml-based development. *IEEE Transactions on Software Engineering*, 31(10):833–849, October 2005.

[5] E. Bruneton, C. Thierry, M. Leclercq, V. Quéma, and S. Jean-Bernard. An open component model and its support in java. In *Proceedings of the ACM SIGSOFT International Symposium on Component-based Software Engineering (CBSE'04). Held in conjunction with ICSE'04*, Edinburgh, Scotland, may 2004.

[6] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture, Volume 5, On Patterns and Pattern Languages*. Wiley, April 2007.

[7] J. Cabot, R. Clarisó, and D. Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 547–548. ACM, 2007.

[8] D. Chappell. *Enterprise Service Bus: Theory in Practice*. O'Reilly Media, 2004.

[9] B. Demuth. The dresden ocl toolkit and its role in information systems development. In *Proc. of the 13th International Conference on Information Systems Development (ISD'2004)*, 2004.

[10] M. Eichberg, S. Kloppenburg, K. Klose, and

M. Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering (ICSE'08)*, pages 391–400. ACM, 2008.

[11] T. Erl. *SOA design patterns*. Pearson Education, 2008.

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

[13] A. Ferdjoukh, A.-E. Baert, A. Chateau, R. Coletta, and C. Nebut. A csp approach for metamodel instantiation. In *ICTAI 2013, IEEE Internationnal Conference on Tools with Artificial Intelligence*, pages 1044,1051, 2013.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, October 1994.

[15] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[16] O. Gilles and J. Hugues. Expressing and enforcing user-defined constraints of aadl models. In *Proceedings of the 5th UML and AADL Workshop (UML and AADL 2010)*, 2010.

[17] O. O. M. Group. Unified modeling language (uml), v2.4.1, superstructure specification: Omg document formal/2011-08-06. OMG Website: `http://www.omg.org/spec/UML/2.4.1/`, August 2011.

[18] O. O. M. Group. Object constraint language (ocl), v2.4, specification: Omg document formal/2014-02-03. OMG Website: `http://www.omg.org/spec/OCL/2.4/`, February 2014.

[19] A. Hamie. Translating the object constraint language into the java modelling language. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1531–1535. ACM, 2004.

[20] K. Hassam, S. Sadou, R. Fleurquin, et al. Adapting ocl constraints after a refactoring of their model using an mde process. In *BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*, pages 16–27, 2010.

[21] D. Hou and H. Hoover. Using scl to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[22] C. Jeanneret, L. Eyer, S. Markovi, and T. Baar. Roclet – refactoring ocl expressions by transformations. In *19th International Conference on Software & Systems Engineering and their Applications (ICSSEA'06), Paris, France*, December 2006.

[23] S. Kallel, C. Tibermacine, M. R. Skay, C. Dony, and A. Hadj Kacem. Génération de méta-programmes java à partir de contraintes d'architecture ocl. In *Proceedings of the French Speaking Conference on Software Engineering (CIEL'14)*, Paris, France, June 2014.

[24] N. Klarlund, J. Koistinen, and M. I. Schwartzbach. Formal design constraints. In *Proceedings of the 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 370–383, San Jose, CA, USA, 1996. ACM Press.

[25] B. Meyer. *Touch of Class*. Springer, June 2013.

[26] R. T. Monroe. Capturing software architecture design expertise with armani. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2001.

[27] Ocl. Eclipse ocl. http://www.eclipse.org/modeling/mdt/?project=ocl.

[28] Octopus. Ocl tool for precise uml specifications. http://octopus.sourceforge.net.

[29] M. Petre. Uml in practice. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, pages 722–731. IEEE Press, May 2013.

[30] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[31] P. Spacek. *Design and Implementation of a Reflective Component-Oriented Programming and Modeling Language*. PhD thesis, University of Montpellier, France, December 2013.

[32] P. Spacek, C. Dony, and C. Tibermacine. A component-based meta-level architecture and prototypical implementation of a reflective component-oriented programming and modeling language. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE'14)*, Lille, France, June-July 2014. ACM Press.

[33] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, July 1979.

[34] R. Terra and M. T. de Oliveira Valente. A dependency constraint language to manage object-oriented software architectures. *Software Practice and Experience*, 39(12):1073–1094, 2009.

[35] C. Tibermacine. *Software Architecture 2*, chapter Architecture Constraints. John Wiley and Sons, New York, USA, 2014.

[36] C. Tibermacine, R. Fleurquin, and S. Sadou. On-demand quality-oriented assistance in component-based software evolution. In *Proceedings of the 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06)*, pages 294–309, Vasteras, Sweden, June 2006. Springer LNCS.

[37] C. Tibermacine, R. Fleurquin, and S. Sadou. Simplifying transformations of architectural constraints. In *Proceedings of the ACM Symposium on Applied Computing (SAC'06), Track on Model Transformation*, Dijon, France, April 2006. ACM Press.

[38] C. Tibermacine, R. Fleurquin, and S. Sadou. A family of languages for architecture constraint specification. *In the Journal of Systems and Software (JSS), Elsevier*, 83(5), 2010.

[39] C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse. Component-based specification of software architecture constraints. In *Proceedings of the 14th ACM Sigsoft symposium on Component based software engineering (CBSE'11)*, pages 31–40. ACM, 2011.

[40] U. Zdun and P. Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Information and Software Technology*, 50(9):1003–1034, 2008.