Processus pour la génération automatique de composants exécutables à partir de contraintes d'architecture

Sahar Kallel*,**, Bastien Tramoni*,
Chouki Tibermacine*, Christophe Dony* et Ahmed Hadj Kacem**

*LIRMM, CNRS et Université de Montpellier, France
sahar.kallel, chouki.tibermacine, bastien.tramoni, dony@lirmm.fr,
**ReDCAD, Université de Sfax, Tunisie
sahar.kallel@redcad.org, ahmed.hadjkacem@fsegs.rnu.tn

Résumé. 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, ou bien une règle de conception générale. Ces spécifications peuvent être exprimées avec un langage standardisé comme OCL. Elles sont la plupart du temps des spécifications brutes sans aucune structure permettant leur paramétrage et réutilisation. Afin de pouvoir les vérifier dans la phase d'implémentation nous proposons dans ce travail une méthode pour traduire automatiquement ces spécifications 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, personnalisables 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 en sortie des composants programmés en COMPO, un langage de programmation par composants réflexif développé par notre équipe.

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 `age`, une contrainte fonctionnelle 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.

Les contraintes d'architecture sont des spécifications où les descriptions d'architecture, et non les états des composants, sont analysées (Tibermacine, 2014). Elles définissent des invariants imposés par des règles de conception générales ou le choix d'un style/patron architectural, comme le style architectural en couches (Shaw et Garlan, 1996), où "les composants se situant dans les couches non-adjacentes ne doivent pas être connectés directement les uns avec les autres". Ceci est un exemple d'une contrainte d'architecture. OCL (Group, 2014) 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).

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 (Zdun et Avgeriou, 2008; Gamma et al., 1994; Buschmann et al., 2007). D'une part, ces contraintes sont des spécifications brutes n'offrant pas assez de structure permettant facilement leur réutilisation. Avec l'expérience qu'on a acquis dans la spécification de ce type de contraintes, nous pouvons dire qu'elles sont généralement composées de parties indépendantes ayant leur propre sémantique et qui sont liées par des opérateurs logiques. D'autre part, 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, sur du code, nous avons deux solutions possibles. La première solution consiste à écrire un nouvel interprète, 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 qu'elle est chronophage, et 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, par exemple). 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 (Ocl, 2008; Octopus, 2006; Demuth, 2004), ils considèrent uniquement les contraintes fonctionnelles et non architecturales.

Dans ce papier, nous proposons de traduire automatiquement 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 (Group, 2011) et fournit en sortie un ensemble de composants exécutables programmés en COMPO (Spacek et al., 2014), qui est un langage à composants développé par notre équipe. Nous proposons de transformer les contraintes d'architecture en "composants-contraintes" (Tibermacine et al., 2011) 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, même à l'exécution (les contraintes d'architec-

ture 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.

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. La section 6 présente une évaluation expérimentale de notre processus. Avant de conclure et exposer les perspectives de ce travail, nous discutons les travaux connexes dans la section 7.

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*" (Chappell, 2004). Ce patron introduit trois catégories de composants : les consommateurs (*consumers*) de services, les fournisseurs (*producers*) de services 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 (avec des formats de messages différents). La contrainte d'architecture qui spécifie les conditions imposées par ce patron est exprimée en OCL en utilisant le métamodèle UML (Group, 2011) dans le Listing suivant.

```
1  context Component inv :
2  let bus:Component
3  = self.realization.realizingClassifier ->select(c:Classifier|c.oclIsKindOf(Component)
4   and c.oclAsType(Component).name='esbImpl')
5  customers : Set(Component)
6  = self.realization.realizingClassifier ->select(c:Classifier|c.oclIsKindOf(Component)
7   and (c.oclAsType(Component).name='cust1' or c.oclAsType(Component).name='cust2'
8    or c.oclAsType(Component).name='cust3'))
9  producers : Set(Component)
10 = self.realization.realizingClassifier ->select(c:Classifier|c.oclIsKindOf(Component)
11  and (c.oclAsType(Component).name= 'prod1' or c.oclAsType(Component).name='prod2'
12 or c.oclAsType(Component).name='prod3'))
13 in
14 -- Le bus doit avoir au moins un port requis et au moins un port fourni
15 bus.ownedPort->exists(p1,p2:Port|
16    p1.provided->notEmpty() and p2.required->notEmpty())
17 and
18 -- Les consommateurs doivent avoir uniquement des ports requis
19 customers->forAll(c:Component|
20 c.ownedPort->forAll(required->notEmpty() and provided->isEmpty()))
21 and
22 -- Les consommateurs doivent être connectés uniquement au bus
23 customers->forAll(com:Component|
24   com.port->forAll(p:Port|p.end->notEmpty()
25     implies
26       self.ownedConnector ->exists(con:Connector |bus.ownedPort->exists( pb:Port|
27            con.end.role->includes(pb)) and con.end->includes(p.end))))
28 and
29 -- Les fournisseurs doivent avoir uniquement des ports founis
30 producers->forAll(c:Component|
31   c.ownedPort->forAll(provided->notEmpty() and required->isEmpty()))
```

```
32 and
33 -- Les fournisseurs doivent être connectés uniquement au bus
34 producers->forAll(com:Component|
35   com.port->forAll(p:Port|p.end->notEmpty()
36     implies
37       self.ownedConnector->exists(con:Connector| bus.ownedPort->exists( pb:Port|
38           con.end.role->includes(pb)) and con.end->includes(p.end))))
```

Listing 1 – *La contrainte du patron d'architecture en Bus en OCL/UML*

Dans le Listing 1, dans les lignes 2 à 12, 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 19- 20) et les fournisseurs ont uniquement des ports de sortie (lignes 30- 31). De plus, elle vérifie si le bus a au moins un port d'entrée et un port de sortie (lignes 15- 16) par lesquels les consommateurs et les fournisseurs sont connectés.

3 Approche générale

Nous proposons un processus composé de deux étapes. Dans la première étape, nous modifions le format des contraintes à partir d'une spécification textuelle `brute` qui n'offre pas assez de structure, réutilisation et paramétrage (Listing 1) vers une description d'architecture constituée de "composants-contraintes". Ces composants sont décrits avec un ADL nommé CLACS (Tibermacine et al., 2011) (prononcé Klax). 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 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 (Tibermacine, 2014) 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 réutilisation (par composition) 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. Le premier métamodèle utilisé est celui d'UML dans lequel les contraintes sont initialement définies (comme dans l'exemple présenté dans la section précédente). Le choix d'UML est simplement motivé par le fait qu'il est un standard industriel [1], 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,

1. Même si une étude empirique récente (Petre, 2013) 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.

comme cela a été expérimenté dans (Briand et al., 2005)), 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. Dans le résultat final, il n'y plus de 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. Pour cela, nous proposons un micro-processus de plusieurs étapes. Toutes les étapes prennent comme entrée l'arbre syntaxique abstrait de notre contrainte de départ.

– **Extraction des déclarations de variables :** Parfois, une sous-expression est utilisée plusieurs fois dans une contrainte OCL. Le mot clé `Let`, par exemple, permet de déclarer une expression qui est de fait liée à l'invariant de la contrainte. La première étape de notre approche est d'extraire les déclarations existantes afin de les externaliser en tant que `définitions OCL` (contraintes stéréotypées avec `def` :) qui sont liées au contexte et donc réutilisables dans un autre invariant. Au cours de cette étape, la contrainte subit des changements afin de faire appel aux nouvelles définitions OCL aux endroits appropriés. A ce niveau notre contrainte de début sera comme suit :

```
1  context Component
2  --let expressions extraction
3  def: letBus(): Component = self.realization.
4  realizingClassifier->select(c : Classifier | c.oclIsKindOf(Component)
5   and c.oclAsType(Component).name = 'esbImpl')
6  def: letCustomers(): Set(Component) = self.realization.
7  realizingClassifier->select(c : Classifier | c.oclIsKindOf(Component)
8  and (c.oclAsType(Component).name = 'cust1' or
9   c.oclAsType(Component).name = 'cust2' or
10  c.oclAsType(Component).name = 'cust3'))
11 def: letProducers(): ...
12 inv:
13 letBus().ownedPort ->includes(p1, p2 : Port | p1.provided
14   ->notEmpty() and p2.required ->notEmpty())
15 and
16 letCustomers()->forAll(c:Component|c.ownedPort
17 ->forAll(required->notEmpty()and provided->isEmpty()))
18  and ...
19  and ...
20  and ...
```

Listing 2 – *Contrainte après l'extraction des expressions let*

Dans le listing 2, les variables `customers`, `producers` et `bus` ont été extraites sous forme de `définitions` (lignes 3 à 11). Dans l'invariant les appels aux variables ont été transformés en appels aux définitions correspondantes (Ligne 13 et 16,par exemple).

– **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 15, 18, 19 et 20 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 dernières peuvent être décomposées de nouveau. La condition d'arrêt de la récursivité est quand la sous-contrainte à décomposer ne contient plus aucun opérateur logique et lorsque sa taille est strictement supérieure à une navigation (taille jugée petinente pour la réutilisation). Cette dernière condition peut parfois être ignorée pour maximiser le nombre de sous-contraintes, comme dans l'exemple ci-dessous.
  Au fur et mesure de cette étape, l'invariant est modifié en faisant appel aux nouvelles définitions générées. A ce stade, nous obtenons un multiensemble (*bag*) de contraintes d'architecture qui retournent des valeurs booléennes. Le Listing 3 représente un extrait de la contrainte durant l'étape de la décomposition.

```
1  context Component
2  def: def1(c:Classifier): Boolean = c.oclIsKindOf(Component)
3  def: def2(c:Classifier): Boolean = c.oclAsType(Component).name = 'esbImpl'
4  def: letBus(): Component = self.realization.realizingClassifier
5  ->select(c : Classifier | def1(c) and def2(c))
6  def: def3(c : Classifier): Boolean = c.oclIsKindOf(Component)
7  ...
8  def: part1(): Boolean = letBus().ownedPort
9  ->includes(p1, p2 : Port | def7(p1) and def8(p2))
10 def: part2(): ...
11 def: def11(p: Port): Boolean = p.end->notEmpty()
12 def: def12(p: Port): Boolean = self.ownedConnector->exists(con : Connector |
13 letBus().ownedPort ->exists(pb:Port| con.end.role ->includes(pb)) and con.end ->
       includes(p.end))
14 def: part3(): Boolean = letCustomers()
15 ->forAll(com : Component | com.port
16   ->forAll(p : Port |def11(p) implies def12(p)))
17 def: part4(): ...
18 def: part5(): ...
19 inv:
20 part1() and part2() and part3() and part4() and part5()
```

Listing 3 – *Contrainte du patron d'architecture en Bus durant la décomposition*

Dans le Listing 3, la contrainte est composée de cinq sous-contraintes OCL principales (part1(), part2(), part3(), part4() et part5(), voir le bas du Listing). Ces sous-contraintes peuvent être décomposées de nouveau en d'autres sous-contraintes de façon récursive. Par example, `def12()` contient l'opérateur "and", donc elle sera encore décomposée. Toutes les sous-contraintes sont définies comme des définitions OCL (`def:`) et présentées avant l'expression `inv:`. Nous pouvons remarquer que certains définitions possèdent des paramètres. La raison de mettre quelques paramètres dans cette étape (la décomposition) est d'avoir la possibilité de décrire toutes les définitions OCL générées avec le même contexte, celui de l'invariant (ligne 1).

– **Suppression des redondants :** Dans cette étape deux cas se présentent. Nous devons supprimer les paramètres, dans la signature d'une définition, qui ne sont jamais utilisés dans son corps. Pour cela, nous utilisons une expression régulière qui permet de vérifier si tous les paramètres d'une définition sont au moins présents une fois dans son corps. Si un paramètre n'est pas retrouvé alors il sera supprimé de la signature de la définition.

Il faut impacter par la suite cette modification à toutes les définitions qui font appel à la définition modifiée. Dans le deuxième cas, nous pouvons avoir des définitions qui sont identiques, c'est à dire qu'elles ont le même nombre et type de paramètres et ayant un corps identique. Dans ce cas, nous en laissons une. Afin de garder la cohérence entre les définitions OCL, nous avons aussi modifier les appels à la définition supprimée par des appels à la définition conservée. Par exemple, dans le Listing 3 `def1()` et `def3()` sont identiques. Maintenant, nous avons un ensemble de définitions OCL qui constituent notre contrainte.

– **Paramétrisation des contraintes :** Cette étape est une étape de généralisation de la contrainte. Elle permet de rendre les définitions réutilisables en leur passant en paramètre des éléments d'architecture spécifiques à un cas d'utilisation. En créant les signatures des services qui intègrent les sous-contraintes, nous ajoutons des paramètres pour chaque valeur littérale définie dans ces sous-contraintes. Le type des paramètres est obtenu à partir de l'arbre syntaxique abstrait de la contrainte.
Dans cette étape, nous avons besoin d'optimiser notre processus, i,e, éliminer quelques définitions redondantes (obtenues dans la phase de paramétrisation). Par exemple, `def4()` dans le Listing 3 est définie dans cette étape comme suit :

```
1 context Component
2 def: def17(c:Classifier, name1:String): Boolean =
3 c.oclAsType(Component).name = name1
4 def: def18(c:Classifier, name2:String): Boolean =
5 c.oclAsType(Component).name = name2
6 def: def19(c:Classifier, name3:String): Boolean =
7 c.oclAsType(Component).name = name3
8 def: def4(c:Classifier, name1:String,name2:String,name3:String):
9 Boolean = def17(c,name1)and def18(c,name2) and def19(c,name3).
```

Listing 4 – *Exemple de paramétrisation*

Nous remarquons que `def2()` (après paramétrisation), `def17()`, `def18()` et `def19()` sont similaires. Elles se diffèrent uniquement par le nom du paramètre. Par conséquent, nous supprimons `def17()`, `def18()` et `def19()` et nous les remplaçons par `def2()` présentée dans le Listing 4. Nous optimisons encore `def4()` de façon qu'elle aura comme paramètres `c:Classifier` et `consumersNames: Set(String)`. Cette optimisation est effectuée en comparant les expressions OCL avant le = (`c.name`) pour chaque valeur littérale.
A la fin de cette étape, notre invariant est complètement décomposé en un ensemble de contraintes sous forme de définitions OCL. Ces contraintes seront enregistrées dans un annuaire afin de les réutiliser pour créer d'autres spécifications de contraintes.

– **Groupement des services :** Chaque composant-requête CLACS (query-component) encapsulera une définition OCL qui retourne une valeur dont son type est différent de booléen et chaque composant-contrainte (constraint-component) encapsulera une définition OCL qui retourne uniquement les valeurs booléennes. En fait, parmi les définitions générées, chacune qui correspond à l'expression let sera embarquée dans un descripteur de composant-requête comme `letConsumers()` et chacune parmi les autres sera embarquée dans un descripteur composant-contrainte. Avec ce principe, nous obtenons un grand nombre de composants. De ce fait, nous avons proposé d'optimiser cette étape et de mettre ensemble les définitions OCL qui sont de même type et partageant les mêmes aspects dans un descripteur de composant. Pour mesurer la similarité

entre les définitions, nous avons implémenté un processus automatique en analysant les arbres syntaxiques abstraites des corps des définitions. 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* (Tai, 1979)) est mesurée entre chaque paire de sous-arbres. Si cette mesure est inférieure à un seuil [2], nous considérons que les deux sous-contraintes sont similaires. Prenons l'exemple de part2() et part4() (correspondants aux sous-contraintes dans les lignes 19 à 20 et 30 à 31 du Listing 1), ces deux définitions partagent le même aspect qui est l'élément architectural (un port). Les arbres syntaxiques 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 le reste des deux sous-arbres, nous pouvons remarquer qu'il y a une similarité entre eux (seulement deux opérations d'édition (substitution de nœuds) : `required` et `provided` sont inversés). Donc, ces deux définitions sont regroupées comme deux opérations dans un même descripteur de composant.

- **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). Ces mappings ont été définies en utilisant la même template comme dans (Tibermacine et al., 2006b). Pour des raisons de limitation de places, nous ne présentons pas ces mappings. Il faut noter que le mot clé `self` [3] est remplacé par `context`, qui est évalué par une référence au port requis connecté au méta-descripteur du composant métier sur lequel la contrainte va être vérifiée. Cette résolution de connexion est effectuée lorsque la vérification est lancée.
- **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 est générée. Cette description d'architecture contient tous les composant-contraintes et les composant-requêtes (instances) nécessaires connectés entre eux. Ces composants intègrent les contraintes d'architecture refactorisées [4] qui naviguent dans le métamodèle de CLACS.

La Figure 1 illustre la description d'architecture complète des composants-contraintes et composants-requêtes générés à partir de notre exemple (Listing 1). Il y a trois expressions `let` dans la contrainte d'architecture (Listing 1), chacune (lignes 2 - 4, 5 - 8, 9 - 12) est supposée être définie séparément d'une façon basique dans un descripteur de composant-requête. Cependant les expressions let 2 et 3 sont similaires. Elles sont donc représentées par un seul composant-requête (`ParticipantsIdentification`). Ils existent cinq composant-contraintes à droite de la figure. Ces composants représentent les définitions OCL qui sont extraites à partir de notre contrainte de début. Ces définitions sont appelées tout au long de la contrainte et ils serviront potentiellement pour d'autres contraintes. Il y a en total cinq sous-

2. La valeur de ce seuil sera fixée empiriquement.
3. self se trouve dans la contrainte initiale écrite dans le métamodèle UML
4. une contrainte est refacotrisée si elle a subi toutes les étapes décrites précédemment.

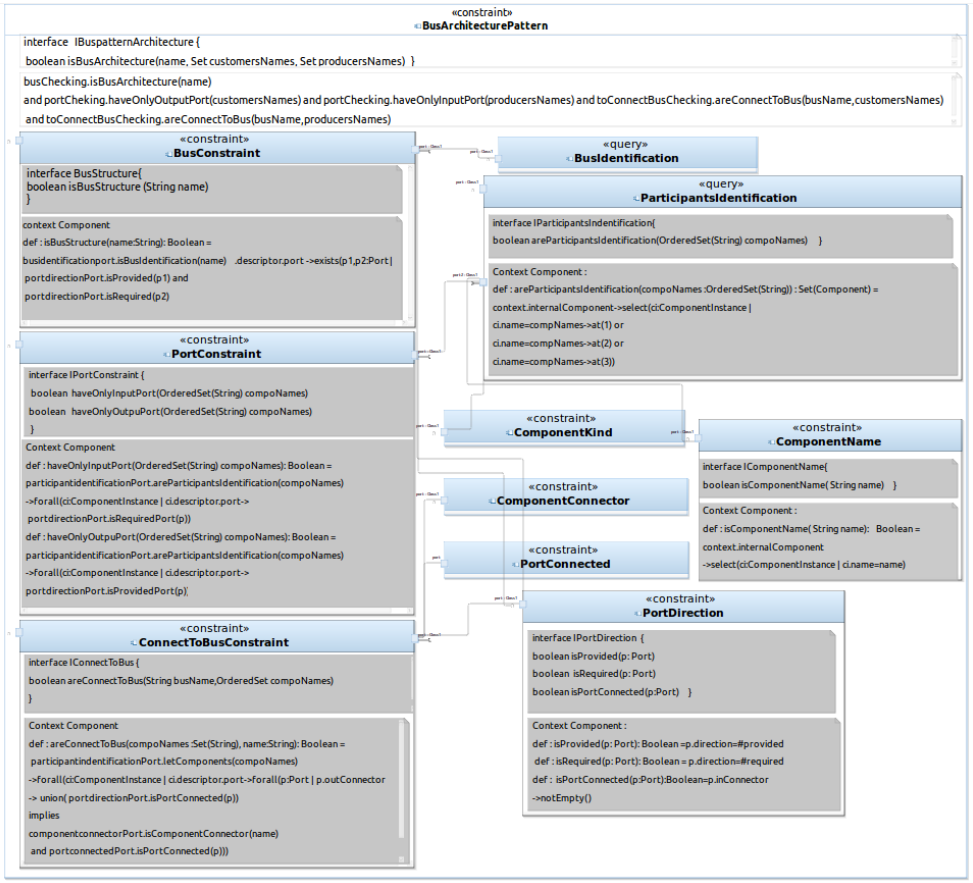

FIG. 1 – *Description d'architecture CLACS du patron d'architecture en Bus*

contraintes dans la contrainte d'architecture. De la même façon, chacune (lignes 15 - 16, 19 - 20, 23 - 27, 30 - 31 et 34 - 38) est supposée être embarquée dans un composant-contrainte. Mais dans cette example, les sous-contraintes 2 et 4 sont regroupées dans un même descripteur de composant (`PortConstraint`) parce qu'elles vérifient les mêmes aspects. Elles vérifient si tous les composants (*customers* dans le premier et *producers* dans le deuxième) ont des types de ports spécifiques (input ou output). Le descripteur `PortConstraint` déclare deux opérations qui servirent à vérifier ces deux contraintes. De l'autre coté, les sous-contraintes 3 et 5 vérifient exactement le même invariant (contrairement aux sous-contraintes 2 et 4), sauf qu'elles d'appliquent sur un ensemble différent de composants (*customers* pour la sous-contrainte 3 et *producers* pour la sous-contrainte 5). Ainsi, il existe un seul descripteur (`ConnectToBusConstraint`) qui est généré pour ces deux sous-contraintes. Il fournit une seule opération qui a comme paramètre l'ensemble des composants sur lequel la contrainte doit être vérifiée. Nous pouvons voir (en haut de la figure) la contrainte vérifiée par le composite, où il y a les cinq appels des opérations vers les trois composants internes (présentés à gauche

de la figure). Ces composants internes font appel aux opérations déclarées dans les autres composants par le nom de leur port de sortie.

A travers cette `composantification`, les descripteurs de composant-contraintes et les composant-requêtes CLACS peuvent être réutilisables (instanciés plusieurs fois dans des différents contextes), composables (les instances peuvent être connectés entre elles ou connectés dans un composant composite pour créer d'autres composants complexes) et personnalisables (il suffit de passer les bons paramètres pour vérifier parfois plusieurs contraintes dans un même composant). A ce niveau, les composants peuvent être vérifiées statiquement en phase de conception [5]. Afin de pouvoir les vérifier sur du code dans la phase d'implémentation, ou dynamiquement à l'exécution du code, nous avons besoin de les transformer en composants exécutables. La vérification des contraintes dans ce cas utilisera la réflexivité (l'introspection) fournie par le langage de programmation.

5 Génération de composants exécutables à partir des composants contraintes

COMPO (Spacek et al., 2014) offre un support permettant de fournir un paradigme unifié pour les composants métiers et les contraintes, avec un même langage de programmation et un même environnement de développement (même éditeur, même interpréteur et même débogueur). Les utilisateurs de COMPO peuvent 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 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.

5. L'environnement CLACS peut être téléchargé ici :
https ://github.com/saharkallel/clacs.

```
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
```

Algorithme 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;`

TAB. 1 – *Code COMPO généré pour les quantificateurs OCL*

Si le nœud courant est un quantificateur, la procédure `generateQuantifier(current)` est appelée. Elle génère un service privé représentant le quantificateur et après, dans le corps du service de la contrainte, elle crée une invocation de ce nouveau service privé et enre-

Contrainte en CLACS	Contrainte en COMPO

`self.internalComponent`
`->`
`select(ci:ComponentInstance|`
` ci.name='cust1' or`
` ci.name='cust2' or`
` ci.name='cust3')`
	`interComps:=`
	`context.getInterComponents();`
	`interComps`
	`->select([ci:Component|`
	`  ci.getName()='cust1' or`
	`  ci.getName()='cust2' or`
	`  ci.getName()='cust3']);`

`->`
`forAll(ci:ComponentInstance|`
`  ci.descriptor.port ->...)`
	`/*generateQuantifier()*/`
	`resultforall1:=`
	`forall1(interComps);`

`ci.descriptor.port`	`ports:=` `ci.getDescribedPorts();`

`->forAll(...)`	`service isPortOutputOnly (....){`
	`....`
	`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; } }`

TAB. 2 – *Exemple de génération de code source COMPO à partir de composant-contraintes OCL/CLACS*

gistre 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-construits 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 d'itération 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 appe-

lons `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`.

Il est évident que la traduction automatique ne permet pas d'obtenir un code optimal (non verbeux). 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.

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.

6 Évaluation du Processus

L'objectif de cette évaluation est de répondre à la question de recherche suivante : "Est-ce que la méthode de traduction automatique fournit des résultats satisfaisants comparativement à une traduction manuelle, faite par un spécialiste ?"

Afin de répondre à cette question, nous avons identifié un certain nombre de métriques nous permettant de mesurer l'efficacité de la méthode. Ces métriques sont la précision et le rappel, tels qu'ils sont définis dans le domaine de recherche d'information (*Information Retrieval*). Les composants obtenus lors de la traduction seront rangés dans deux catégories :

- Les faux positifs : ce sont des contraintes qui sont considérés comme un composant-contrainte par le prototype mais pas par un spécialiste.
- Les vrais positifs : ce sont les composants-contraintes qui ont été générés par notre méthode et par le spécialiste aussi.

Afin de réaliser ces mesures, nous avons collecté un ensemble de contraintes OCL formalisant quelques patrons d'architecture (Zdun et Avgeriou, 2008). La taille de ces contraintes est donnée dans la Figure 2.

Comme nous pouvons le voir, nous avons utilisé des contraintes de tailles variables pour réaliser notre expérimentation. Les patrons d'architectures formalisés par ces contraintes sont divers. C1 spécifie le patron `client-serveur`, C2 caractérise le patron `Layered Hybrid`, C3 à C6 caractérisent différentes version du patron `Pipes and Filters` et C7 définie le patron `Pipeline`. Nous avons aussi récupéré un catalogue de composants-contraintes qui a été défini par un doctorant de l'IRISA ne faisant pas partie de l'équipe. Ce catalogue a été construit à la main il y a 2 ans. Il inclut les composants-contraintes représentant chacune des contraintes de la Figure 2.

Pour analyser nos résultats, nous allons uniquement observer la précision, le rappel étant impertinent ici, parce que la méthode de traduction génère toujours les composants-contraintes qui seraient créés par un spécialiste, plus d'autres composants-contraintes. La précision est

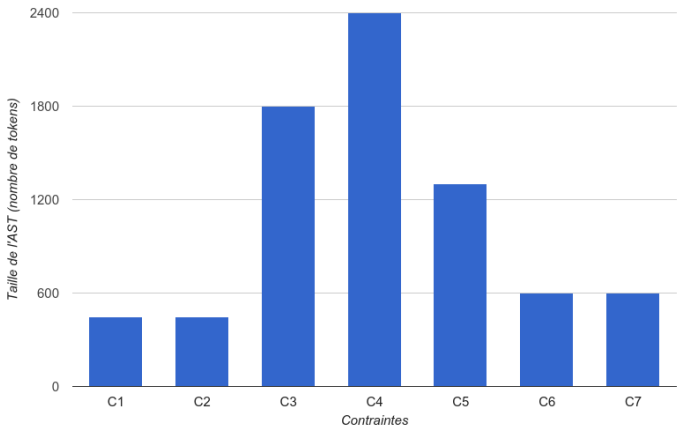

FIG. 2 – *Taille de l'AST des contraintes en nombre de tokens*

calculée de la manière suivante :

$$P = vp/(vp + fp) \tag{1}$$

où vp correspond au nombre de vrais positifs et fp au nombre de faux positifs.

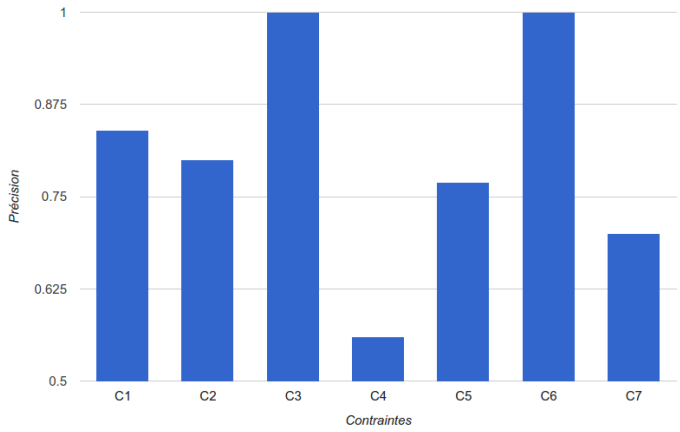

FIG. 3 – *Diagramme de précision des contraintes*

Les résultats sont présentés dans le diagramme 3. Ces résultats s'expliquent par le fait que le découpage de l'invariant réalisé par notre prototype est très proche de la façon dont sont spécifiés les patrons d'architecture. La baisse de la précision sur la contrainte C4 est due au découpage des définitions qui va trop en profondeur et nous fournit des composants-contraintes inexploitables dans un but de réutilisation.

L'un des biais de cette expérimentation est la présence de nombreuses variantes d'un même patron comme le *Pipe and Filter*. Ceci favorise la réutilisation de composants-contraintes entre les différents patrons, et améliore donc la pertinence des composants générés.

7 Travaux connexes

Les travaux connexes peuvent être classés dans différentes catégories : i) les langages et les outils pour la spécification des contraintes en phase de conception et d'implémentation, ii) les techniques de transformation des prédicats/contraintes, iii) les méthodes de génération de code à partir de prédicats, et iv) les méthodes de réutilisation de contraintes. 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 (Tibermacine, 2014). Tous les langages proposés dans la littérature 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 fournissent des moyens très limités) pour paramétrer et/ou réutiliser des contraintes d'architecture.

Hassam *et. al.* (Hassam et al., 2010) ont proposé une méthode de transformation de contraintes OCL lors du *refactoring* des modèles UML, en s'appuyant sur des transformations de modèles. Ils utilisent une méthode d'annotation du modèle UML source pour obtenir un modèle cible annoté. Par la suite ils prennent les deux annotations pour former une table de *mapping* qui sera utilisée finalement avec RoclET (Jeanneret et al., 2006) pour transformer les contraintes OCL du modèle source en des contraintes OCL conformes au modèle cible. Nous trouvons l'idée d'utiliser un compilateur OCL existant plus simple. Leur solution de transformation des contraintes est lourde à mettre en place, elle requiert des connaissances sur les outils et les langages de transformation de modèles. Dans (Ferdjoukh et al., 2013), 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 leur sont associées. Cabot *et. al.* (Cabot et al., 2007) 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 (DresdenOCL (Demuth, 2004)) pour analyser les contraintes. Mais, dans notre approche, 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, Octopus et Dresden OCL. 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 (Briand et al., 2004) proposent une approche pour transformer les contraintes fonctionnelles en Java. Un autre travail (Hamie, 2004) propose une méthode de traduction des contraintes fonctionnelles en JML (*Java Modeling Language*). Dans un travail précédent (Kallel et al., 2014), 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 cet article une traduction des contraintes vers des composants. Dans un autre

travail (Tibermacine et al., 2006b), 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 (Tibermacine et al., 2010)).

Dans (That et al., 2013), Ton That *et. al.* proposent un catalogue de patrons d'architecture sous forme de composant-contraintes. Ce travail a été effectué manuellement. Dans notre approche, les transformations des contraintes sont faites automatiquement. Nous avons utilisé un sous-ensemble du résultat de ce travail comme un "oracle" pour notre évaluation.

8 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 (Tibermacine et al., 2006a). 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 (Spacek, 2013) ou des frameworks d'adaptation dynamique, et plusieurs implémentations dans le domaine des *models@run.time (Bencomo et al., 2014)*. 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 automatiquement des composants-contraintes réutilisables 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 (Erl, 2008) et une méthode pour les interpréter dans les phases de conception et d'implémentation, comme services. Ceci a pour objectif ultime 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.

Références

Bencomo, N., R. France, B. H. C. Cheng, et U. Aßmann (2014). *Models@run.time : Foundations, Applications, and Roadmaps*, Volume 8378 of *LNCS*. Springer.

Briand, L. C., W. Dzidek, et Y. Labiche (2004). Using aspect-oriented programming to instrument ocl contracts in java. Technical report, Carlton University, Canada.

Briand, L. C., Y. Labiche, M. Di Penta, et H. D. Yan-Bondoc (2005). An experimental investigation of formality in uml-based development. *IEEE Tran. SW. Eng. 31*(10), 833–849.

Buschmann, F., K. Henney, et D. C. Schmidt (2007). *Pattern-Oriented Software Architecture, Volume 5, On Patterns and Pattern Languages*. Wiley.

Cabot, J., R. Clarisó, et D. Riera (2007). 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*, pp. 547–548. ACM.

Chappell, D. (2004). *Enterprise Service Bus : Theory in Practice*. O'Reilly Media.

Demuth, B. (2004). The dresden ocl toolkit and its role in information systems development. In *Proc.ISD'2004*.

Erl, T. (2008). *SOA design patterns*. Pearson Education.

Ferdjoukh, A., A.-E. Baert, A. Chateau, R. Coletta, et C. Nebut (2013). A csp approach for metamodel instantiation. In *Proc. of IEEE ICTAI 2013*, pp. 1044,1051.

Gamma, E., R. Helm, R. Johnson, et J. Vlissides (1994). *Design patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley.

Group, O. O. M. (2011). 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/`.

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

Hamie, A. (2004). Translating the object constraint language into the java modelling language. In *Proceedings of the 2004 ACM symposium on Applied computing*, pp. 1531–1535. ACM.

Hassam, K., S. Sadou, R. Fleurquin, et al. (2010). Adapting ocl constraints after a refactoring of their model using an mde process. In *Proc. of BENEVOL 2010*, pp. 16–27.

Jeanneret, C., L. Eyer, S. Markovi, et T. Baar (December 2006). Roclet – refactoring ocl expressions by transformations. In *Proc. of ICSSEA'06, Paris, France*.

Kallel, S., C. Tibermacine, M. R. Skay, C. Dony, et A. Hadj Kacem (2014). 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.

Ocl (2008). Eclipse ocl. http ://www.eclipse.org/modeling/mdt/ ?project=ocl.

Octopus (2006). Ocl tool for precise uml specifications. http ://octopus.sourceforge.net.

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

Shaw, M. et D. Garlan (1996). *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall.

Spacek, P. (2013). *Design and Implementation of a Reflective Component-Oriented Programming and Modeling Language*. Ph. D. thesis, University of Montpellier, France.

Spacek, P., C. Dony, et C. Tibermacine (2014). A component-based meta-level architecture and prototypical implementation of a reflective component-oriented programming and modeling language. In *Procs.CBSE'14*, Lille, France. ACM Press.

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

That, M. T. T., C. Tibermacine, et S. Sadou (2013). Catalog of architectural patterns characterized by constraint components, version 1.0. Technical Report IRISA/ArchWare-2013-TR-01.

Tibermacine, C. (2014). *Software Architecture 2*, Chapter Architecture Constraints, pp. 37–90. New York, USA : John Wiley and Sons, Inc.

Tibermacine, C., R. Fleurquin, et S. Sadou (2006a). On-demand quality-oriented assistance in component-based software evolution. In *Proc. (CBSE'06)*, Vasteras, Sweden, pp. 294–309. Springer LNCS.

Tibermacine, C., R. Fleurquin, et S. Sadou (2006b). Simplifying transformations of architectural constraints. In *Proceedings of the ACM Symposium on Applied Computing (SAC'06)*, Dijon, France. ACM Press.

Tibermacine, C., R. Fleurquin, et S. Sadou (2010). A family of languages for architecture constraint specification. *In the Journal of Systems and Software (JSS), Elsevier 83*(5), 815–831.

Tibermacine, C., S. Sadou, C. Dony, et L. Fabresse (2011). Component-based specification of software architecture constraints. In *Proc.CBSE'11*, pp. 31–40. ACM.

Zdun, U. et P. Avgeriou (2008). A catalog of architectural primitives for modeling architectural patterns. *Information and Software Technology 50*(9-10), 1003–1034.

Summary

Architecture constraints are specifications defined by developers at design-time and checked on design artifacts (architecture descriptions, like UML models). They enable to check, after an evolution, whether an architecture description still conforms to the conditions imposed by an architecture pattern, style or any design principle. One possible language for specifying such constraints is the OMG's OCL. Most of these architecture constraints are formalized as "gross" specifications, without any structure enabling parameterization or reusability. In order to check them at the implementation stage, we propose in this work a method for translating automatically these specifications into executable components. In addition to making them checkable at the implementation stage, we chose to target software components in order to make these architecture constraints reusable, parametrizable and composable specifications. Since architecture constraints need to analyze architecture descriptions, the generated components use the standard reflective (meta) level provided by the programming language to introspect architectures. Our implementation takes as input OCL constraints specified on the UML metamodel. It produces components programmed in Compo, our "home-made" reflective component-oriented programming language.