# Interfaces riches et contrats à la QML dans le modèle à composants BCM4Java.
# PSTL

Rozovyk Artemiy
rozovyk.artem@gmail.com

Benalla Anis
benalla_anis@yahoo.com

George Mathieu
mathieu.george92@gmail.com

June 5, 2020

## 1 Introduction

Ce document s'inscrit dans la réalisation de notre UE PSTL du master STL de Sorbonne-Université. Celle-ci implique la réalisation d'un projet sur toute la durée du second semestre de la première année du master. Notre projet s'intitule « Interfaces riches et contrats à la QML dans le modèle à composants BCM4Java ». De manière plus concrète, nous avons travaillé sur un modèle de composants, utilisé par la faculté pour la recherche et l'enseignement, notamment l'UE Composant du M1 et l'UE ALASCA du M2, et nous y avons rajouté plusieurs fonctionnalités. Comme le nom du modèle l'indique, il est utilisé pour la réalisation de projets orientés composants écrits en Java.

Notre travail est composé de trois principaux objectifs. Tout d'abord, la réalisation d'un langage permettant la conception d'interfaces riches, semblable à QML [1], que nous décrivons dans les sections à venir. Ce langage nous permet alors d'introduire des notions de programmation contractuelle tels que les pré et postconditions portant sur les arguments et les valeurs de retour d'une entité. Nous introduisons également des notions de qualité de service, c'est à dire des propriétés non-fonctionnelles telles que la disponibilité, durée de réponse, etc.

Nous devions également ajouter un moyen de nous assurer que les interfaces riches offertes et requises possèdent des contrats conformes. Un point important de cet objectif est de pouvoir mettre un place un mécanisme permettant de manipuler les contraintes définies dans notre langage, puis de les transformer en expressions exploitables par une librairie-solveur afin de pouvoir les évaluer.

Enfin, notre dernier objectif est de pouvoir générer[2] du code permettant de vérifier que les contrats sont bien valides à l'exécution. Autrement dit, nous devons ajouter du code permettant de s'assurer du respect des contrats à l'exécution, côté client et serveur.

[1] Quality-of-service Modelling Language

[2] On parle bien de la génération, car les injection de code seront produites de manière dynamique suivant le type de contrainte à valider.

# 2 Thématiques abordées

Notre projet gravite autour de deux thématiques informatiques majeurs : le paradigme de programmation contractuelle ainsi que la programmation orientée composants.

## 2.1 Types abstraits de données

De manière générale, un TAD peut être défini comme étant *une classe d'objets dont le comportement est défini par un ensemble de valeurs d'un ensemble d'opérations*; il s'agit d'un analogue à la structure algébrique en mathématiques.
Les types abstraits de données ont été proposés pour capturer l'essence des types de données par les propriétés des opérations qu'ils proposent plutôt que par leur implantation. Les types abstraits de données peuvent être décrit sous la forme de types algébriques, cela nous permet de pouvoir prouver la validité des appels aux opérations d'un TAD en manipulant des axiomes.

## 2.2 Programmation par composants

La programmation orientée composant utilise une approche modulaire, où un programme est divisé en plusieurs entités logicielle distinctes déployables. Les diverses entités du programme doivent déclarer leurs points d'interconnexion de manière explicite. On définit ainsi une interface offerte comme l'ensemble des services offerts par le composant fournisseur et une interface requise comme l'ensemble des services requis par le composant client. Par son utilisation d'interfaces offertes et requises, la programmation par composants met encore plus l'accent sur les interfaces dans la conception que la programmation par objets. De fait, vu que les composants et leurs interfaces sont conçus indépendamment, cela introduit un problème de conformité entre interfaces offertes et requises. Dans ce projet nous essayons de répondre à ce problème en introduisant des moyens de faciliter le développement par composants, notamment en matière d'interconnexion.

## 2.3 Sémantique axiomatique

Définir une sémantique axiomatique pour un langage de programmation consiste à définir comment chacune des instructions du langage transforme une précondition supposée vraie en une postcondition prouvée vraie. Impossible d'implanter ces notions dans tous les programmes, car assez complexe, il s'agit d'un problème de classe NP. Nous adoptons donc une approche plus pragmatique, notamment la conception et la programmation par contrats.

## 2.4 La conception par contrat

**Définition 1.** ***Conception par contrats:*** *Conception des relations clients/serveurs entre entités logicielles autour d'obligations réciproques formulées par des assertions.*

C'est une approche basée sur la logique mathématique qui sert à prouver qu'un programme informatique est correct. L'aspect théorique de la programmation contractuelle repose principalement sur les travaux de C.A.R. Hoare, notamment via son article de 1969 « An Axiomatic Basis for Computer Programming ». Concrètement, une fonction qui possède un contrat doit respecter les règles établies par le contrat pour que le programme s'exécute correctement. L'article de Hoare définit les prédicats majeurs utilisés par un contrat via ce qu'on appelle le triplet de Hoare qui se représente sous la forme suivante : {P} C {Q}. Cela constitue la base de la programmation contractuelle qui est un des éléments clé de notre projet. « P » représente l'ensemble des préconditions que doit respecter le code « C ». Les préconditions sont des obligations faites à l'appelant, lui demandant de fournir des paramètres et d'appeler le client dans un état tel que ses préconditions sont satisfaites. « Q » représente les postconditions, celles-ci sont des engagements pris par l'appelé, il s'engage à rendre un résultat et de se retrouver dans un état tel que ses postconditions sont satisfaites, sous réserve que les préconditions étaient satisfaites.

Les pré et post conditions servent principalement à vérifier que les paramètres d'entrée ou de sortie ont des valeurs acceptables par le programme. Cela évite au programmeur d'opter pour une approche défensive et de placer de nombreux `if` en début de fonction. Les conditions sont généralement des conjonctions logiques de comparaisons entre une variable ou un attribut (la taille d'une chaîne de caractères par exemple) et un entier.

La programmation par contrat a été introduite concrètement pour la première fois par Bertrand Meyer avec le langage Eiffel en 1985.

**Définition 2.** ***Programmation par contrats:*** *Application des principes de la conception par contrats à des programmes grâce à des mécanismes intégrés dans les langages de programmation ou par des instructions ajoutées manuellement.*

Les spécifications doivent être vérifiables et précises, elles sont définies de façon formelle généralement sous forme d'assertion. Dans notre projet, nous représentons ses assertions via des annotations Java qui contiennent les expressions booléennes à évaluer.

## 2.5 Les propriétés non fonctionnelles

La notion du type abstrait de données a des nombreux bénéfices. En cachant les détails d'implémentation effective elle permet notamment de :

- Rendre la compréhension du code plus facile (Le code de "haut niveau", non obscurci par les détails du code "bas niveau").
- L'implémentation peut être changé sans affecter toute une partie du programme qui utilise ce type abstrait.

- L'utilisation des TAD facilite la réutilisation du code.

Cependant, une grande partie des systèmes d'aujourd'hui a soit une architecture répartie, soit présente des notions de concurrence ou de programmation embarquée / temps réel. Il serait irréaliste de supposer qu'on peut tout simplement ignorer l'implémentation d'un type. La performance d'une opération diffère souvent d'une implémentation à l'autre, les propriétés temporelles telles que temps de traitement moyen, la disponibilité d'une opération constituent des aspects cruciaux qui sont pris en compte par les utilisateurs d'un système.

Pour répondre à ces problématiques, dites de **qualité de service** il est nécessaire de définir la notion des propriétés non fonctionnelles.

**Définition 3.** *Une propriété non fonctionnelle désigne une qualité observable sur une entité logicielle.*

On distingue les propriétés statiques (nombre de modules dans un système ou encore la sécurité) et dynamiques (la disponibilité, le débit d'objets passant par le système, le nombre de pannes par an...).
Afin de fournir un niveau adéquat de *qualité de service* le système doit proposer des manières de négocier, d'observer et d'adapter les métriques en question. Pour cela, le système doit spécifier explicitement les niveaux des métriques qu'il opère. À partir du moment où nous pouvons observer et comparer les valeurs de ces métriques, nous pouvons définir un *contrat non fonctionnel*.

**Définition 4.** *Un contrat non fonctionnel est un ensemble de contraintes sur les valeurs de métriques non fonctionnelles défini pour les opérations d'une entité logicielle.*

### 2.5.1 Langage d'expression des propriétés non fonctionnelles

Concrètement, nous avons besoin d'un mécanisme flexible permettant l'expression des contraintes requises et offertes. Nous avons également besoin de pouvoir vérifier la conformité entre les contrats offerts et requis. Finalement, les contrats doivent être vérifiables statiquement (à la conception) et/ou dynamiquement, à l'exécution.

### 2.5.2 QML (Quality of service Modeling Language)

Un exemple de langage qui répond à ces critères a été défini par les travaux de Frølund and Koistinen (1998) où les abstractions suivantes sont introduites:

- **Dimension** - la propriété nommée ayant un domaine de valeurs et pouvant être ordonnée.
- **Type de contrat** - un ensemble de dimensions, représentant un aspect de qualité de service: (performance, fiabilité).

- **Contrat** - l'instance d'un type de contrat qui introduit les contraintes pour les dimensions qu'il présente.
- **Profil** - association d'un contrat aux interfaces, aux opérations, aux arguments ou/et aux valeurs de retour.

**Exemple de QML**

Soit une interface:

```
interface RateServiceI {
    Rate latest(Currency from, Currency to) ;
    Forecast analysis(Currency c) ;
}
```

Nous définissons un ensemble des spécifications suivantes:

```
type Reliability = contract {
    numberOfFailures: decreasing numeric no/year;
    TTR: decreasing numeric sec;
    availability: increasing numeric;
};


type Performance = contract {
    delay: decreasing numeric msec;
    throughput: increasing numeric mb/sec;
};



systemReliability = Reliability contract {
    numberOfFailures < 10 no/year;
    TTR {
        percentile 100 < 2000;
        mean < 500;
        variance < 0.3
    };
    availability > 0.8;
};


rateServerProfile for RateServiceI = profile {
    require systemReliability;
    from latest require Performance contract {
        delay {
            percentile 50 < 10 msec;
            percentile 80 < 20 msec;
            mean < 15 msec
        };
```

```
    };
    from analysis require Performance contract {
        delay < 4000 msec
    };
};
```

### 2.6 Conformité à la QML

En QML, la conformité est définie pour chaque niveau d'abstraction et de manière descendante : depuis les profils jusqu'aux contraintes.
Un profil $Q$ est conforme à un profil $P$ si les contrats dans $P$ associés à une entité $e$, sont conformes aux contrats associés à $e$ dans $Q$.
La conformité des contrats à leur tour est définie en matière de conformité de l'ensemble des contraintes.
Finalement, afin de définir la conformité des contraintes, nous devons pouvoir être en mesure d'affirmer qu'une contrainte est plus restrictive que l'autre.
"Être plus restrictive" dépend de l'importance d'une valeur

- les valeurs **increasing** (montant) ⇒ les grandes valeurs sont plus intéressantes.
- les valeurs **decreasing** (descendant) ⇒ les petites valeurs sont plus intéressantes.

Par exemple, la dimension `availability` est **increasing**, et `delay` est **decreasing**
c'est-à-dire que

"`availability` >0.90" est **plus restrictive** que "`availability` >0.73"
"`delay` <5" est **plus restrictive** que "`delay` <10"

## 3 Les outils utilisés

### 3.1 BCM

Notre objectif est d'implémenter ces différentes notions au modèle de composant BCM4Java. BCM4Java est un modèle minimal au sens où il est défini autour d'un minimum de concepts centraux (composants emboîtables, interfaces offertes et requises proposées sur des ports explicites, connexion des ports par connecteurs explicites, assemblages de composants pouvant être répartis sur plusieurs machines virtuelles et plusieurs ordinateurs), et utilisant des technologies à la fois basiques et stables de Java, comme RMI, les ≪sockets ≫, etc. Il s'agit donc d'un modèle très dépouillé plutôt destiné à faire de la recherche et de l'enseignement, et non à concurrencer des modèles visant une exploitation dans des applications réelles, commerciales.

Dans BCM, un composant est défini comme un objet Java et les méthodes qu'il possède sont les services qu'il utilise ou offre. Pour être utilisé, celui-ci doit implémenter la classe abstraite `AbstractComponent`. Une interface offerte doit nécessairement implémenter

l'interface Java `OfferedI` fournie par BCM. De même, les interfaces requises doivent implémenter l'interface Java `RequiredI`. Les ports doivent implémenter l'une des deux classes abstraites suivantes : `AbstractOutboundPort` et `AbstractInboundPort`. La première est destinée aux ports sortants situés côté client, la seconde est utilisée par les ports entrants situés côté serveur. Un port sortant implémente l'interface requise du composant, il sert de point intermédiaire lors de la connexion entre les composants. De manière similaire, un port entrant implémente l'interface offerte et sert de point d'entrée au composant.

Listing 1: Extrait de l'exemple "basic_cs" issu de BCM qui montre la structure d'un port sortant la méthodes getUri() ) un service requis par le composant

```
public class URIConsumerOutboundPort
extends AbstractOutboundPort
implements URIConsumerI
{

 public URIConsumerOutboundPort(String uri, ComponentI owner) throws
     Exception
 {
  super(uri, URIConsumerI.class, owner) ;
  assert uri != null && owner != null ;
 }

 @Override
 public String getURI() throws Exception
 {
  return ((URIConsumerI)this.connector).getURI() ;
 }
}
```

La connexion exploite également un unique connecteur qui va relier deux ports. Les connecteurs et les ports sont représentés par des objets Java. La connexion peut s'effectuer de deux manières différentes, selon si le programme s'exécute dans une seule JVM [3] ou dans plusieurs. Dans une exécution en mono JVM on utilise des références Java, pour une exécution en multi JVM on utilise RMI [4].

On note que toutes les entités présentes dans le modèle possèdent une URI [5] qui sert d'identifiant pour un objet. Cela nous permet de faire des références à des objets présents dans la mémoire d'autres JVM.

[3] Java Virtual Machine
[4] Remote Method Invocation
[5] Unique Ressource Identifier

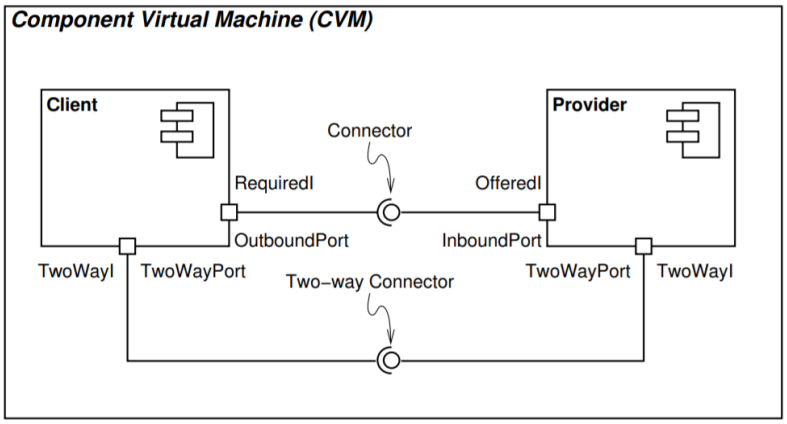



Figure 1: Illustration représentant un cas basique d'architecture BCM, issu du cours numéro 1 de l'UE Composant

### 3.1.1 Implémentation de la conformité des interfaces en BCM

La programmation orientée composant couplée à la programmation contractuelle permet de faire apparaître la thématique de notre projet. Nous voulons nous assurer que les contrats des interfaces requises par le client et les interfaces offertes par le fournisseur soient bien conformes.

Actuellement dans BCM, on considère que deux interfaces sont conformes si toutes les méthodes déclarées dans l'interface requise par le client sont présentes dans l'interface offerte par le fournisseur. On considère qu'une méthode est correctement implémentée si elle possède le même nom, le même nombre de paramètres, ainsi que des types de paramètres et un type de retour conforme avec ceux de l'interface offerte. On remarque également que l'interface requise ne doit pas nécessairement implémenter l'entièreté des méthodes de l'interface offerte. On peut ainsi implémenter partiellement les fonctionnalités proposées par le composant fournisseur. Dans le cas où deux interfaces ne sont pas conformes, il est nécessaire d'effectuer des modifications pour s'assurer qu'elles puissent communiquer décemment. Cela peut se faire par le biais du connecteur.

Voici un exemple extrait de BCM pour représenter plus concrètement la conformité des interfaces.

```
public interface  URIProviderI
extends  OfferedI
{
 public String  provideURI() throws Exception ;
 public String[]  provideURIs(int numberOfRequestedURIs) throws
     Exception ;
}

public interface  URIConsumerI
extends  RequiredI
{
 public String  getURI() throws Exception ;
```

```
 public String[]  getURIs(int numberOfURIs)
 throws Exception ;
}

public class   URIServiceConnector
extends  AbstractConnector
implements URIConsumerI
{

 @Override
 public String  getURI() throws Exception
 {
  return ((URIProviderI)this.offering).provideURI() ;
 }

 @Override
 public String[]  getURIs(int numberOfURIs) throws Exception
 {
  return ((URIProviderI)this.offering).provideURIs(numberOfURIs) ;
 }
}
```

Cela met en évidence le rôle du connecteur qui permet de fluidifier le processus de conformité des interfaces. Dans cet exemple, il permet notamment d'assurer la conformité des interfaces bien que les services aient des noms de méthodes différents. Le connecteur regroupe ainsi la majeure partie des modifications nécessaires pour qu'une connexion intercomposant devienne possible. Actuellement dans BCM, les préconditions, les postconditions et les invariants sont déclarés dans la javadoc des classes et méthodes à titre indicatif. Notre objectif principal est donc de compléter cette conformité déjà existante dans BCM en l'étendant aux divers éléments liés à la programmation contractuelle.

### 3.2 Annotation Java, réflexion

Une **annotation** est une forme de métadonnée syntaxique qui peut être ajoutée au code source. En java on peut annoter les classes, les méthodes, les arguments, les variables ainsi que les paquetages. Classiquement, les informations définies par les annotations servent lors de la compilation. Cependant, il est possible de rendre une annotation persistante à l'exécution en spécifiant la *rétention*=`runtime`.

La réflexion Java est une technique intégrée au langage, qui correspond à des notions **d'introspection** et **d'intercession**, qui sont les capacités d'un programme à examiner son propre état et de modifier son état d'exécution voire sa propre interprétation ou sémantique. Concrètement, la réflexivité Java est implémentée par le concept des **métaobjets**, il s'agit de la méthode `getClass()` héritée de la classe `Object` ce qui la rend disponible dans n'importe quelle classe [6]. L'objet retourné par cette méthode est un

[6]Toute classe Java hérite de la "super-classe" `Object`

sous-type de la classe `Class<?>` qui, parmi d'autres, offre les possibilités suivantes:

- Récupérer l'ensemble des méthodes définis dans un objet.
- Exécuter l'une des méthodes, si instance de cette classe est disponible [7]
- Récupérer l'ensemble des annotations.

### 3.3 Solveur Choco

Pour démontrer que les préconditions requises impliquent les préconditions offertes et vice-versa pour les postconditions, nous utiliserons un moteur de résolution de contraintes. Pour ce projet, le moteur Choco en Java, décrit dans Charles Prud'homme (2016.), sera utilisé.
`Coco Solver` est une librairie *Open source* Java pour la programmation par contraintes qui existe depuis plus de 20 ans et est utilisée par des nombreuses universités et entreprises.

L'idée sera donc de s'intéresser aux expressions du langage des interfaces riches pour générer des contraintes en Choco qui pourront alors être vérifiées. L'essentiel du travail consistera donc à transformer les expressions du langage des interfaces vers les expressions de Choco, puis de lancer le calcul.

Le modèle de base de Choco opère uniquement sur les variables entières. Voici un exemple basique d'utilisation de la librairie native:

```
org.chocosolver.solver.Model m = new Model("Choco Solver Hello World");
// a prends les valeurs dans { 4, 6, 8 }
IntVar a = m.intVar("a", new int[]{4, 6, 8});
IntVar b = m.intVar("b", 0, 2); // b prends les valeurs dans [0, 2]
m.arithm(a, "+", b, "<", 8).post(); // imposer une condition
while (m.getSolver().solve())
    System.out.println("Solution " + i++ + " found : " + a + ", " + b);
```

Le solveur trouve les résultats suivants:

```
Solution 1 found : a = 4, b = 0
Solution 2 found : a = 4, b = 1
Solution 3 found : a = 4, b = 2
```

Malgré la puissance native de la librairie `Choco` , nous aimerions également pouvoir traiter les variables réelles. Pour cela Choco propose l'intégration d'une autre librairie `Ibex` écrite en `C++` et basée sur arithmétique d'intervalles. Son utilisation est assez similaire à la librairie native:

[7] On peut en créer une si nécessaire , également grâce à la réflexion.

```
Model model = new Model("Ibex");
RealVar x_a = model.realVar("X_a", .1d, 4.d, 1.E-1);
RealConstraint c1 = model.realIbexGenericConstraint("{0} > 0.8;", x_a);
RealConstraint c2 = model.realIbexGenericConstraint("{0} < 3 ", x_a);
model.post(c1,c2);
for (Solution s : model.getSolver().findAllSolutions())
    System.out.println(s);
```

```
Solution: X_a=[0.9,0.9687500000000001] ...
```

### 3.4 Javassist

Pour effectuer l'injection de code nécessaire à la vérification des contraintes, nous avons utilisé la librairie Javassist - "The Java Programming Assistant" (December 2019), développée d'abord à l'Université de Tokyo et actuellement maintenue dans le cadre du projet libre JBoss. Celle-ci nous permet de modifier le bytecode Java durant l'exécution d'un programme. On peut ainsi définir ou modifier des classes et des méthodes.

La librairie utilise une classe `ClassPool` pour décrire l'ensemble des classes utilisées par le programme. Celle-ci consiste en une map d'objets `CtClass` qui offrent des opérations très similaires à celles offertes par la classe `Class` de `java.lang.reflect`. L'utilisateur peut ainsi récupérer une classe via son nom pour ensuite y apporter des modifications. Un grand nombre de paramètres sont accessibles, notamment les méthodes déclarées, les superclasses et les interfaces implémentées. Pour effectuer les injections de code, nous avons principalement utilisé les méthodes `insertBefore()` et `insertAfter()`. Elles servent respectivement à injecter du code en tête de la méthode et à la fin de la méthode. Il est important de noter que le code injecté va passer par le compilateur Java inclut dans Javassist et que celui-ci a ses limites. Par exemple, il n'est pas possible de définir des classes internes ou anonymes, les mots clés `continue` et `break` ne sont pas définis et les types génériques ne sont pas supportés. Une fois toutes les modifications faites, on utilise la méthode `writeFile()` pour effectuer les changements dans le fichier de classe.

```
ClassPool pool = ClassPool.getDefault();
CtClass clazz = pool.getCtClass("HelloWorld");
CtMethod m = clazz.getDeclaredMethod(helloWorld);
m.insertBefore("{ System.out.println(\"Hello World\"); }");
clazz.writeFile();
```

Pour l'identification des paramètres, Javassist propose des identifiants spéciaux pour les manipuler. Depuis la version 3.18 de Javassist, l'utilisation directe des paramètres d'entrée est possible dans les méthodes d'injection de code, sans avoir recours aux identifiant. Il reste néanmoins nécessaire d'utiliser l'identifiant `$_` pour représenter la valeur de retour.

L'intérêt d'utilisation de cette librairie et de pouvoir injecter les conditions définies dans les interfaces, afin de mettre en place des moyens de vérification des contraintes à l'exécution.

# 4 Implémentation du système des propriétés non fonctionnelles

Dans cette partie nous allons vous présenter le travail réalisé dans le cadre d'une implémentation d'un système permettant la définition des propriétés non fonctionnelles au sein du framework `BCM4Java`.

### 4.0.1 Expressions acceptés

Dû à des choix d'implémentation dans les parties suivants, les expression qui seront acceptés dans le langage suivent la grammaire suivante:

```
<boolexp>::=<term>{<or><term>}
<term>::=<expression>{<and><expression>}
<expression>::= <num><rel_op><var> | <var><rel_op><num>
<rel_op> ::= '<' | '<=' | '>=' | '>' |
<or>::= '||'
<and>::= '&&'
```

À noter qu'actuellement, l'`expression` doit comporter une valeur numérique à gauche ou à droite, une chaîne de caractères du côté opposé, les deux séparés par un `rel_op`. Ceci est dû à la manière dont les contraintes seront transformées afin d'être traités par le solveur. Effectivement, nous aurions pu introduire un système qui supporterait les expressions à plusieurs variables comme : $x > 3 <= z > k$...
Vu que les contraintes `Choco` sont construites avec la méthode

```
IRealConstraintFactory.realIbexGenericConstraint(String, Variable...)
```

Il serait nécessaire de mettre en place un système intelligent qui détecterait l'ensemble des variables présentes dans l'expression, construire un objet
`Variable[] vars ={ x, z, k }` , ainsi que transformer l'expression en
`String exp= "{0} > 3 <= {1} > {2}"` afin de pouvoir faire l'appel de la méthode ainsi `realIbexGenericConstraint(exp,vars)`.
Nous aurions pu également introduire le parenthésage des expressions. Cela serait possible en construisant d'abord l'AST[8] avec le `BooleanExpressionEvaluator` [9] qui le prend en compte.

### 4.0.2 Métriques de qualité de service

Dans un premier temps nous définissons un langage d'expression des propriétés non fonctionnelles inspiré du `QML`. Ce langage doit inclure :

[8] L'ordre inverse est actuellement implanté.
[9] Que nous décrivons plus dans la section "Adaptation des contraintes au modèle Choco"

- *des expressions logiques similaires à celles de Java pour définir des pré- et postconditions axiomatiques*
- *des expressions de contraintes non fonctionnelles similaires à celles de QML*

Syntaxiquement, le langage que nous définissons sera exprimé sous forme des annotations Java qui seront attachées aux interfaces de `BCM4Java` et dont le contenu sera récupéré par réflexion Java afin d'être traité dans les parties suivantes.

## 4.1 Description du langage de contrats

Pour commencer, nous devons introduire des notions de base de programmation par contrat. Ainsi, les pré et postconditions seront représentées avec les annotations de manière suivante :

```
@Pre(expression = "x > 5 && y > 5 ")
@Post("ret < 100 && ret > 10")
int doSomeOperation(int x, int y) throws Exception;
```

À noter que les expressions écrites dans les préconditions doivent respecter les noms des paramètres et introduire une expression logique valide au sens Java. Pour les postconditions, il est exigé que l'utilisateur fasse référence à la valeur de retour via le mot-clé `ret`.

### 4.1.1 Modélisation des propriétés non fonctionnelles

Afin de faire correspondre les notions présentées dans le papier de Frølund and Koistinen (1998), nous avons procédé ainsi :
Tout d'abord, pour anticiper une vérification dynamique ainsi que pour regrouper les informations correspondantes aux contrats définis par les annotations, nous avons créé la hiérarchie des classes et des interfaces sur la **Figure 2** .
Cette hiérarchie propose une base de définitions des contrats à la QML, notamment grâce aux interfaces. Certaines classes concrètes y sont également présentes à titre d'exemple.
Par conséquent, les dimensions, les types des contrats et les contrats eux-mêmes sont représentés comme des objets java. Quant aux profils, ce sont en fait des associations de contrats aux signatures des interfaces.
Voici quelques exemples:

```
@ContractDefinition(
        name = "systemReliability",
        type= Reliability.class,
        constraints=
                {"numberOfFailures < 5",
                        "availability > 0.9"}
)
@Require(contractName = "systemReliability") //for each method.
public interface URIProviderI extends OfferedI {
```

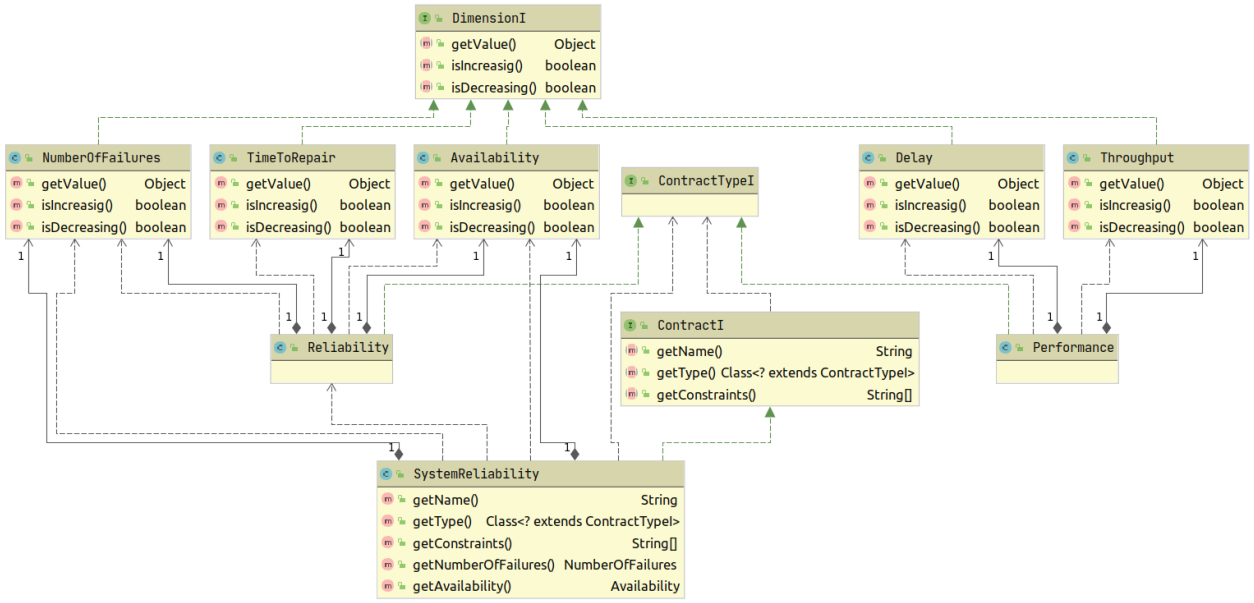



Figure 2: Hiérarchie d'objets QML.

Cet exemple montre comment déclarer un contrat qui peut ensuite être requis par les méthodes de l'interface courante. À noter que les contraintes doivent porter sur le nom exact d'une dimension défini dans le type du contrat. Deuxièmement, l'annotation `@Require` déclare que ce contrat est requis pour l'ensemble des opérations du système.
Ainsi, l'utilisateur doit pouvoir déclarer plusieurs contrats réutilisables. Ceci est possible en utilisant le principe d'une annotation répétable disponible depuis `Java SE 8`.
Voici la définition de @ContractDefinition :

```
@Repeatable(ContractDefinition.List.class)
public @interface ContractDefinition {

    String name();
    Class<? extends ContractTypeI> type();
    String[] constraints();

    @Retention(RetentionPolicy.RUNTIME)
    @Target({ElementType.TYPE})
    @interface List {
        ContractDefinition[] value();
    }
}
```

Parmi les attributs d'une annotation, comme le nom, le type et les contraintes, on définit un type interne possédant lui-même un attribut de type tableau d'annotations du type englobant. L'annotation @Repeatable tout au début, prend en argument cette classe interne ce qui rend possible la répétition d'une annotation d'un même type.

Donc, le fait de pouvoir avoir à la fois une seule annotation (La liste est vide dans ce cas, et `getAnnotation(ContractDefinition.class)` rend une valeur non nulle) et des annotations multiples, nous devons nous adapter afin de traiter les deux cas.

```
Class<? extends ComponentServiceI> clazz = ...
ContractDefinition[] clientContractDefs =
 clazz.getAnnotation(ContractDefinition.class) == null ?
  clazz.getAnnotation(ContractDefinition.List.class).value() :
  new ContractDefinition[]{clazz.getAnnotation(ContractDefinition.class
    )};
```

À l'intérieur d'une interface nous pouvons annoter les méthodes de manière suivante:

```
@RequireContract(
            contractType= Performance.class,
            constraints= {"delay < 3000"} )
public String getURI() throws Exception ;

@Require(contractName = "systemRepairability")
    void otherOperation() throws Exception;
```

Le rôle du `@RequireContract` est de proposer une manière alternative de déclarer un contrat et l'attacher directement à une méthode. Quant à `@Require`, elle permet d'associer un contrat déjà déclaré.

## 4.2 Vérification de conformité à la connexion

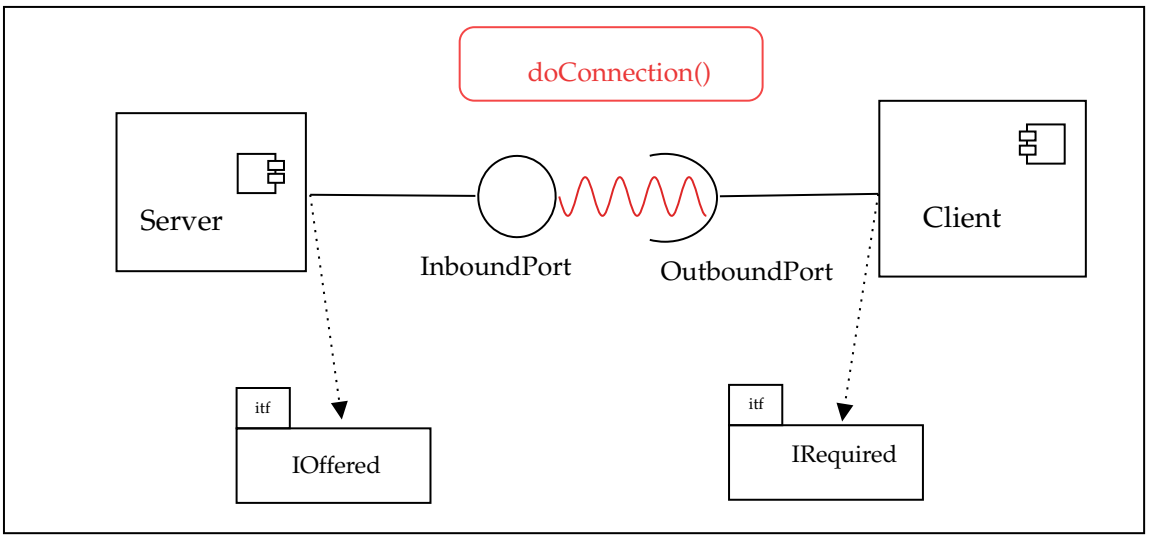
La conformité entre interfaces offertes et requises est la notion permettant de garantir que les contraintes sont compatibles les unes avec les autres. Plus précisément, si les préconditions de l'interface requise sont satisfaites alors celles de l'interface offerte doivent nécessairement l'être aussi. Similairement, si les postconditions de l'interface offerte sont satisfaites, alors celles de l'interface requise doivent nécessairement l'être aussi. Il y a donc une notion d'implication logique bidirectionnelle. Pour les préconditions, les requises impliquent les offertes alors que pour les postconditions, c'est l'inverse, les offertes impliquent les requises.
Par exemple, les préconditions requises et offertes suivantes sont conformes:

$\texttt{size} > 10 \Rightarrow \texttt{size} > 5$

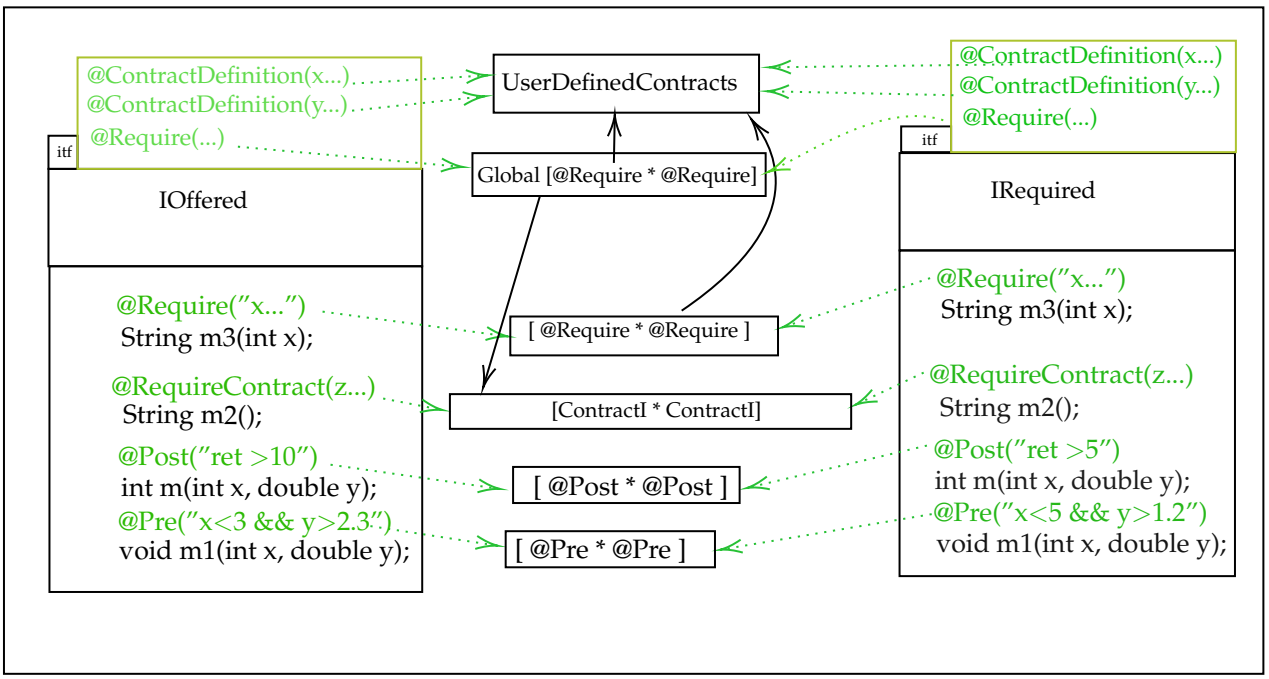
Concrètement, nous allons chercher à récupérer les contraintes définies dans les deux interfaces et vérifier leurs compatibilités. D'abord sur le plan structurel, c'est-à-dire que si un service requis par client spécifie un contrat, le même contrat doit exister dans la spécification du service du serveur. Finalement, nous allons chercher à prouver une implication suivant le type des contraintes en utilisant le solveur Choco. Pour ce faire nous devons faire un travail de regroupement et de transformation des contraintes définies dans les interfaces de chaque composant. Le but est d'obtenir une structure que peut être facilement converti en modèle de contraintes de la librairie Choco.

### 4.2.1 Récupération et regroupement de l'ensemble de contraintes.

Afin d'assurer la conformité entre les deux interfaces en BCM4Java, nous devons faire une vérification le plus tôt possible. Après un certain nombre de recherches et suite au conseil de l'encadrant du projet, nous avons décidé que l'endroit le plus approprié serait la connexion des ports, notamment la méthode `doConnexion()` de la classe abstraite `AbstractOutBoundPort`:

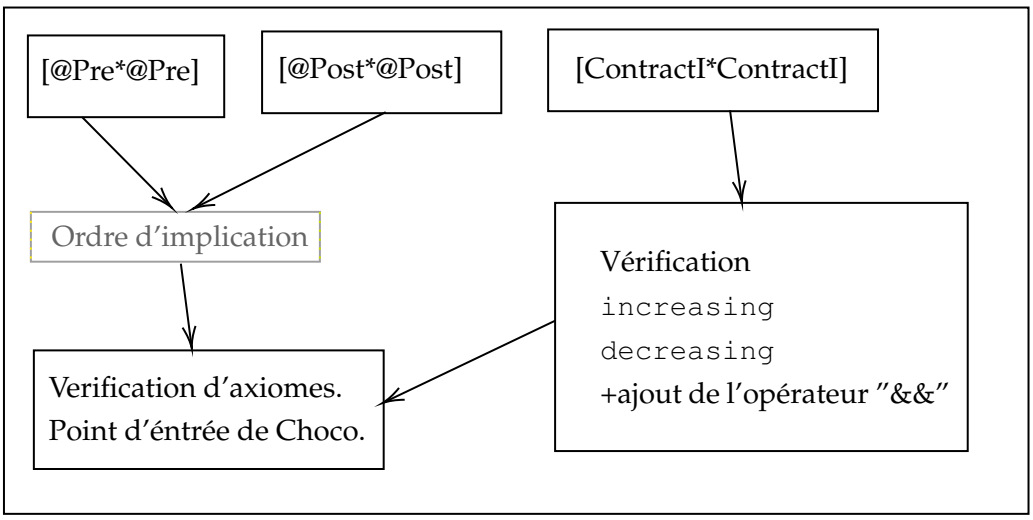



Ayant l'accès aux interfaces, nous pouvons procéder à la récupération des annotations définies dans les interfaces en se servant de la réflexivité java:




Ce schéma montre le principe de regroupement des contraintes en liste de couples de chaque type correspondant. Dans un premier temps, les définitions des contrats sont récoltées des deux côtés depuis les annotations au niveau de l'interface. L'objet obtenu correspond à une liste d'associations `<Nom,@ContractDefinition>`. Cet objet sert de

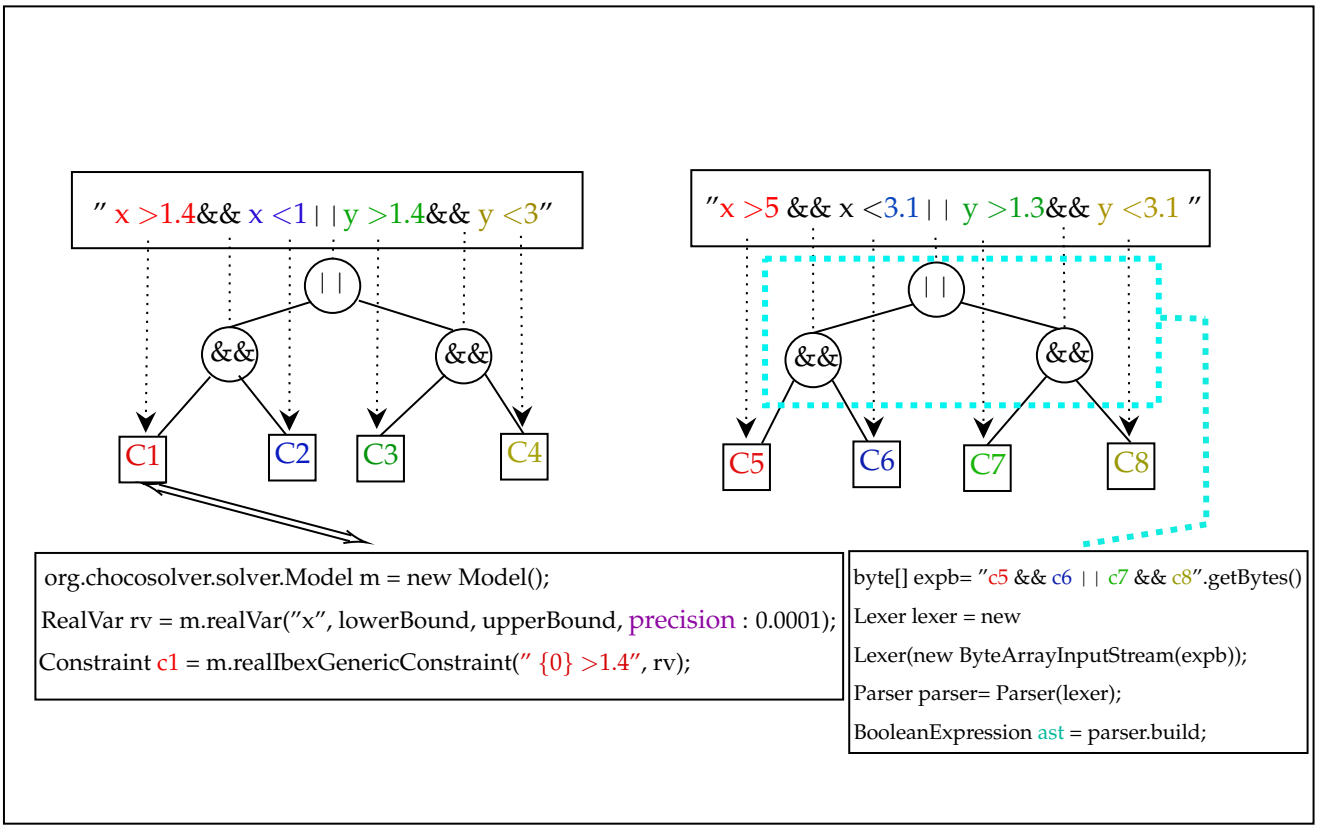
référence aux déclarations du type `@Require` globales ainsi que locales à une méthode. Les déclarations locales d'un nouveau type de contrat au niveau d'une méthode via `@RequireContract` attachent également une instance de contrat de ce type à cette dernière. Le tableau de couples de contrats ainsi créé est également enrichi avec les contrats globaux. Les pré et postconditions sont également regroupés en liste des couples[10]. Finalement nous faisons une dernière organisation des contraintes afin de déterminer le bon ordre d'implication (les préconditions du client impliquent les préconditions du serveur, et l'inverse pour les postconditions). Quant aux contrats de qualité de service, ils sont de même nature que les postconditions, ainsi les contraintes du serveur doivent impliquer celles du client. De plus, les implications des contraintes de qualité de service doivent être conformes à l'importance de leurs dimensions (`decreasing/increasing`) Le schéma suivant montre cette transformation:




### 4.3 Adaptation des contraintes au modèle Choco

Une fois que les contraintes sont récupérées et organisées en liste de couples, nous pouvons procéder à la décomposition des chaînes de caractères correspondantes. Le but ici est d'arriver à une représentation arborescente des contraintes afin d'utiliser efficacement le solveur `Choco`.
Tout d'abord chaque inégalité intérieure de l'expression est transformée en `Constraint` de `Choco`. Pour cela on identifie le nom de la variable et on instancie une `RealVar` en indiquant les bornes min et max que peut prendre la variable [11], nous indiquons également la précision qui sera pris en compte lors de la résolution des contraintes. Finalement l'expression est associée à la variable réelle et la contrainte est créée. Les contraintes ainsi produites sont gardés dans une `Map<String,Constraint>` afin d'être réutilisées dans l'étape suivante.

[10] À cette étape nous pouvons supposer qu'il y a une correspondance structurelle au niveau des définitions sinon une exception serait lancé lors de composition.
[11] Nous prenons min = (la plus petite valeur numérique contenue dans l'expression) -1 et max = (la plus grande valeur numérique contenue dans l'expression)+1




En principe, la composition des contraintes `Choco` (avec `or, and, not..`) forme une structure arborescente. Par conséquent nous devons mettre en place une manière de transformer une expression booléenne en arbre correspondant qui sera ensuite parcouru afin de construire l'arbre final des contraintes `Choco`. Pour ce faire, nous avons utilisé une solution existante qui résout un problème proche du nôtre. Il s'agit de BooleanExpressionEvaluator développée par notre collègue italien Nicola Malizia, que nous avons adapté à notre problème (notamment les terminaux ne sont pas des `true` ou `false` mais des `String` quelconques).
Ensuite, l'arbre est effectivement transformé en modèle `Choco`

```
Constraint makeConstraint(Model m,
                          Map<String, Constraint> cmp,
                          BooleanExpression ast){
    if (ast instanceof Terminal) { return cmp.get(ast.toString()); }
    if (ast instanceof And) {
            return
            m.and(makeConstraint(m, cmp, ((And) ast).getLeft()),
                  makeConstraint(m, cmp, ((And) ast).getRight());
    }
    if ... instanceof Or
        ... m.or(...
```

Ayant les deux arbres de contraintes `Choco`, notamment celle du "serveur" $A_s$ et celle du "client" $A_c$) nous cherchons à prouver que l'implication $(A_s \implies A_c)$ est vraie. Pour

cela nous voulons construire la négation de l'implication : $(A_s \wedge \neg A_c)$. Si une solution est trouvée pour cette négation, cela signifie que l'implication est fausse.

À cette étape, étant déjà très avancés dans le développement de la solution, nous avons constaté qu'il existe un dysfonctionnement de la librairie, dû probablement à un bug de composition des contraintes `Choco` (notamment l'utilisation de l'opérateur `not` ) et les variables réelles ayant comme base une librairie externe Ibex. Nous avons créé une issue github et avons eu le retour du développeur principal, Charles Prud'homme, qui promet de corriger le problème dans la version à venir.

En attendant, nous contournons ce problème manuellement, en faisant l'inverse des opérateurs `<, >, <=, >=` ainsi qu'en inversant l'AST :

```
... if (ast instanceof And) {
            return m.or(makeConstraint(...getLeft()),...getRight())
... if (ast instanceof Or) {
            return m.and(...
```

Ayant résolu ce problème, nous procédons au test consécutif de toutes les contraintes. Par conséquent, une fois que toutes les paires de contraintes sont testées et valides, nous pouvons affirmer que les interfaces sont bien conformes

## 4.4 Vérification dynamique des contraintes

Une fois la conformité des interfaces établie, rien n'assure qu'à l'exécution les contrats seront effectivement satisfaits. L'approche classique en programmation contractuelle est d'insérer du code qui va vérifier les contraintes à l'exécution. Dans BCM4Java, le code en question peut être inséré dans les ports entrants et sortants des composants ainsi que dans les connecteurs.
Pour générer du code Java et l'insérer dans des classes existantes nous utilisons l'outil `Javassist`.

### 4.4.1 Gestion dynamique des pré et postconditions

Pour l'insertion des contraintes, nous avons décidé d'utiliser les ports, ceux-ci implémentant directement les interfaces offertes ou requises par le composant.

On a redéfini la méthode `onLoad()` de la classe `Translator` qui est appelée pour chaque classe à leur chargement dans la JVM. Cette méthode nous permet d'effectuer des modifications dans une classe avant qu'elle ne soit chargée. On vérifie ensuite si la classe chargée est un port ou non, c'est-à-dire qu'elle hérite d'une des classes abstraites propres aux ports (`AbstractOutboundPort` ou `AbstractInboundPort`). Pour chaque interface implémentée par le port, on effectue ensuite une injection de code de conformité dans les méthodes du port.

On note qu'après chaque écriture dans un fichier de classe, Javassist gèle la classe pour empêcher d'autres ajouts car la JVM ne peut pas recharger une classe. Étant donné que l'on travaille avant le chargement de la classe, on utilise la méthode `defrost()` pour dégeler la classe et effectuer plusieurs insertions, une par interface implémentée.

Pour chaque interface, on récupère toutes les méthodes qu'elle implémente. On récupère ensuite les annotations utilisées par toutes les méthodes considérées. Pour chaque annotation, on vérifie s'il s'agit d'une précondition ou d'une postcondition, étant donné qu'elles nécessitent des traitements différents au niveau de l'injection. On doit également récupérer les annotations issues des superclasses de l'interface implémentée. On insère ainsi l'ensemble des conditions définies par l'interface implémentée et les interfaces qu'elle implémente. On note qu'il est nécessaire d'exclure la classe `Object` des superclasses considérées.

Les préconditions doivent être respectées avant l'exécution de la méthode, on doit donc les insérer avec la méthode `insertBefore()`.

```
// expression represente ici l'ensemble des preconditions a verifier
cm.insertBefore("if (!(" + expression + "))" + "throw new fr.sorbonne_u.
    components.exceptions.PreconditionException(\""+expression+"\");");
```

Les postconditions devant être vraies à la fin de l'exécution de la méthode, on exploite la fonction `insertAfter()`. Cependant il est impératif de considérer la valeur de retour. Dans les déclarations des postconditions, on utilise la valeur de retour avec la variable `ret`. On doit ainsi la remplacer avec l'identifiant `$_` utilisé par Javassist pour manipuler les valeurs de retour d'une fonction.

```
// expression represente ici l'ensemble des postconditions a verifier
String expression = ((Post)annotation).value();
String modifiedExpression = expression.replaceAll("\\b" + "ret" + "\\b",
     "\\$_");
cm.insertAfter("if (!(" + modifiedExpression + "))" + "throw new fr.
    sorbonne_u.components.exceptions.PostconditionException(\""+
    expression+"\");");
```

### 4.4.2 Un travail préliminaire à la gestion dynamique des propriétés non fonctionnelles

Quant à la vérification dynamique des propriétés non fonctionnelles, leur gestion nécessite souvent de maintenir des valeurs statistiques comme la disponibilité moyenne ou nombre de pannes par mois. De plus un contrat peut être constitué de plusieurs *dimensions* à prendre en compte. Par conséquent, nous ne pouvons pas nous contenter d'une simple injection de code dans les méthodes en question. Dans ce projet nous nous contentons de faire un travail préliminaire qui permettra de regrouper les contrats non fonctionnelles dans un attribut `Map<Method, List<ContractI>>` de la classe `AbstractComponent`. Ceci

est dans le but de donner aux développeurs la possibilité d'avoir conscience de l'ensemble de contrats qu'une méthode doit assurer. L'implémentation du système de maintien et de vérification des contrats reste à la charge du développeur.
Cette opération est faite via Javassist de la manière suivante:

```
for(CtConstructor c : outboundPort.getDeclaredConstructors()){
    c.insertAfter("this.getOwner().getCtrctMap().putAll((java.util.Map)"
        +DynamicConformance.class.getCanonicalName()+
            ".getItfContractMap(this.getImplementedInterface()));");
}
```

Nous considérons tous les constructeur du port sortant. Après le travail d'initialisation du constructeur, nous injectons une ligne de code qui:

- Récupère l'instance du composant propriétaire du port `this.getOwner()`
- Récupère l'instance de l'attribut `Map<Method, List<ContractI>>` avec `getCtrctMap()` se trouvant dans le composant.
- Remplit cette instance en faisant un appel à une méthode statique `getItfContractMap()` de la classe `DynamicConformance` qui prend en argument une interface et retourne une `Map` des associations Method/ContractI.
- La `Map` ainsi obtenue est ajoutée à l'attribut du composant via `putAll`.

### 4.5 Tests réalisés

Les tests que nous avons effectués sont séparés en deux parties distinctes. D'une part nous avons un exemple d'implémentation concret qui permet de vérifier que les injections de code fonctionnent, ainsi que la vérification des contrats à l'exécution. D'autre part, nous avons un fichier de test unitaire pour s'assurer du bon fonctionnement du solver.

L'exemple concret est situé dans `src/fr/sorbonne_u/components/qos/exemple/basic_cs/`. Il s'agit d'un exemple basique d'architecture BCM auquel on a ajouté des interfaces riches. Par ce fait, les fichiers principaux à étudier sont les interfaces `URIProviderI` et `URIConsumerI` situées dans le sous-répertoire `interfaces`. Les interfaces contiennent des exemples de contrats variés possédant des préconditions et postconditions sur les paramètres des méthodes et les valeurs de retour, ainsi que des contrats riches portant sur des propriétés non fonctionnelles.

Le fichier de test unitaire est situé dans `src/fr/sorbonne_u/components/qos/solver/test/`. Chaque test déclare explicitement une condition côté client et côté serveur. L'objectif est de savoir si la condition serveur implique la condition client. Les deux conditions sont passées au solveur qui nous indique si oui ou non les deux conditions sont bien conformes.

```
 @Test
public void testCorrectConnection2() {
server = "x > 1.4 && x <1 || y > 1.4 && y <3";
client = "x > 5 && x <3.1 || y > 1.3 && y <3.1";
boolean implies = ChocoSolver.verifyAll(server, client);
assertTrue(implies);
System.err.println("Correct: (" + server + ") -> (" + client + ")"); }
```

## 5 Conclusion

Dans ce rapport, nous avons exposé les différentes thématiques clés de notre projet, principalement la programmation par composant et la programmation contractuelle. Nous avons travaillé sur une problématique en considérant les thématiques abordées qui était de parvenir à mêler la conformité de contrats à un modèle à composants. Cette problématique était séparée en trois objectifs distincts.

- Réussir à définir un langage d'interfaces riches semblable à QML;
- Implémenter les mécanismes de vérification de conformité issus de ce langage à BCM avec le moteur Choco;
- Pouvoir générer du code permettant de vérifier les contrats pendant l'exécution du code avec l'outil Javassist.

Ce projet nous a permis de grandement améliorer nos compétences en programmation contractuelle, qui était un des éléments clés du semestre et non pas seulement du PSTL. Cela nous a également permis d'apprendre à maîtriser un outil de génération de code avec l'outil Javassist. Le projet nous a également permis d'acquérir des notions en matière de programmation par contraintes et de les mettre en œuvre avec la librairie Choco. Enfin, cela nous a permis de développer du code sur un projet d'envergure destiné à être réutilisé, principalement par les étudiants des UE Composants en M1 STL et ALASCA en M2.

## References


Charles Prud'homme, X. L., Jean-Guillaume Fages. (2016.). Choco solver documentation. *TASC, INRIA Rennes, LINA CNRS UMR 6241,COSLING S.A.S.,*.

Frølund, S., & Koistinen, J. (1998, December). Quality-of-service specification in distributed object systems. *Distributed Systems Engineering*, *5*(4), 179–202. Retrieved 2020-04-30, from https://iopscience.iop.org/article/10.1088/0967-1846/5/4/005 doi: 10.1088/0967-1846/5/4/005

The java programming assistant. (December 2019). *http ://www.javassist.org*.