

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

CONFRACT : UN SYSTÈME POUR CONTRACTUALISER DES COMPOSANTS LOGICIELS HIÉRARCHIQUES

Philippe Collet, Roger Rousseau

Projet OCL

Rapport de recherche
ISRN I3S/RR-2004-32-FR

Octobre 2004

ConFract : un système pour contractualiser des composants logiciels hiérarchiques

Philippe Collet — Roger Rousseau

Équipe OCL, Objets et Composants Logiciels

*I3S – CNRS – Université de Nice - Sophia Antipolis
Les Algorithmes, Bât. Euclide, 2000 route des Lucioles
BP 121, F-06390 Sophia Antipolis Cedex*

{Philippe.Collet, Roger.Rousseau}@unice.fr

RÉSUMÉ. Cet article présente le système de contractualisation ConFract pour le modèle de composants ouvert et hiérarchique Fractal. Les spécifications, écrites actuellement en CCL-J, un langage d'assertions exécutables adapté à Fractal, sont placées incrémentalement par les concepteurs dans différents contextes : au niveau des interfaces (contrats d'interfaces), ou de la membrane des composants (contrats de composition externe ou interne qui expriment des propriétés compositionnelles). Les contrats sont construits dynamiquement, au moment de l'assemblage ou du réassemblage des composants, lors d'une éventuelle négociation. Ils sont réifiés, à partir des spécifications, en identifiant les responsabilités. L'implémentation repose sur un contrôleur de contrats, qui agit en liaison avec ceux de l'implémentation Julia de Fractal et respecte ainsi les principes de séparation des préoccupations.

ABSTRACT. This article presents the contracting system ConFract for the open and hierarchical component model Fractal. Specifications are currently written in CCL-J, an executable assertions language which is suited to Fractal. Developers can place specifications incrementally in different contexts: on interfaces (interfaces contracts) or on the membrane of components (internal or external composition contracts, which express compositional properties). Contracts are dynamically built, at components (re-)assembly times, during a possible negotiation. They are reified from specifications and identifies responsibilities. The implementation relies on a contract controller, which cooperates with other controllers from the Julia implementation of Fractal, thus respecting the principles of separation of concerns.

MOTS-CLÉS : génie logiciel orienté composant (GLOC), composant hiérarchique, contrat d'interface, contrat de composition, assertion exécutable, ConFract, Fractal, CCL-J, Julia, Java.

KEYWORDS: component-based software engineering (CBSE), hierarchical component, interface contract, composition contract, executable assertion, ConFract, Fractal, CCL-J, Julia, Java.

1. Introduction

Depuis l'appel de McIlroy en 1968, l'approche par composants logiciels a connu une importante évolution. Les composants ont d'abord été des unités de compilation, puis des modules emboîtables avec une interface explicite (Modula2, Ada ...), ensuite des classes liées par des liens d'héritage ou de clientèle (Eiffel, C++ ...), enfin des boîtes noires capables de communiquer sur le réseau à travers plusieurs interfaces (CCM, EJB, .NET ...). Aujourd'hui on voudrait concilier les avantages de toutes ces conceptions du composant logiciel, avec des possibilités de séparation des préoccupations et de choix du niveau d'abstraction. Un composant logiciel « moderne » peut donc être considéré aussi bien, comme une boîte noire qui communique à travers le réseau par des interfaces qui peuvent être assemblées dynamiquement, que comme une hiérarchie de sous-composants, avec d'éventuels partages. Ceci suppose des modèles de composants élaborés comme celui de *Fractal* [BRU 04], avec des possibilités d'ouverture pour étendre les services et permettre l'interception des messages.

Mais de tels modèles bouleversent les processus du génie logiciel traditionnel, en mélangeant les propriétés statiques et dynamiques, fonctionnelles et de qualité de service. Pour maîtriser la fiabilité d'applications à base de tels composants, il faut intégrer les vérifications dans des processus complexes de développement, à base de (re)configuration dynamique et de négociation, à partir de contrats multiformalismes, qui respectent l'organisation hiérarchique et déterminent de manière fine les responsabilités.

Cet article présente le modèle de contractualisation *ConFract* conçu pour *Fractal*, dont le modèle de composants offre une organisation hiérarchique, un assemblage dynamique et des possibilités réflexives, séparées par aspects via des contrôleurs distincts. La section 2 présente la problématique de la contractualisation des composants logiciels « modernes » et la section 3, une description résumée du modèle *ConFract*. Son implémentation est décrite dans la section 4 et la section 5 situe ce travail par rapport aux travaux connexes. Enfin, la section 6 conclut cet article.

2. Contractualiser des composants logiciels

2.1. Contexte : composants logiciels hiérarchiques

Le modèle de composants *Fractal*¹ [BRU 04] autorise une organisation hiérarchique d'une application, avec des composants qui peuvent être récursivement composés d'autres sous-composants. Les composants peuvent être connectés via des interfaces de rôle client ou serveur. Prenons l'exemple simplifié d'un simulateur de photocopieur noir-et-blanc, comme illustré par la figure 1. Il offre l'avantage d'être facilement compris, tout en offrant des exemples de compositions non triviales. Du point de vue externe, le photocopieur offre deux interfaces serveur, un panneau de com-

1. Nous ne détaillons pas le modèle *Fractal* que nous supposons maintenant assez connu.

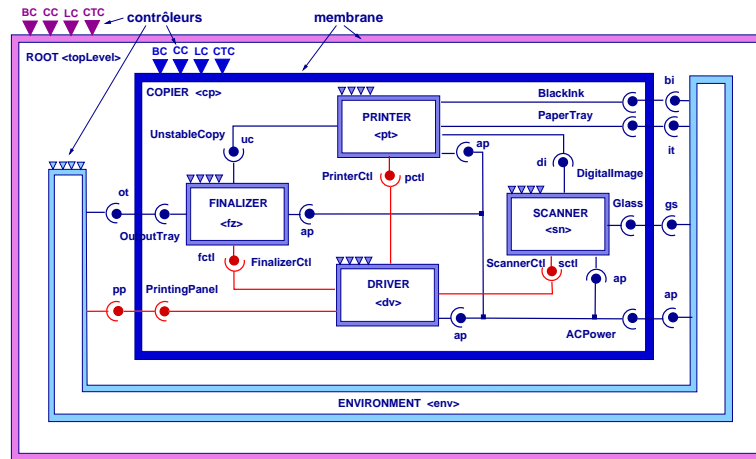


Figure 1 – Architecture interne du photocopieur.

mande (pp) et un bac de réception des copies (ot). Les interfaces client gèrent tout ce que l'environnement doit fournir pour réaliser une photocopie : de l'encre noire (bi), un magasin de papier vierge (it), une vitre d'exposition de l'original (gs) et une alimentation électrique (ap). Au moment de l'assemblage, toutes ces interfaces sont connectées dynamiquement par un composant racine (<topLevel>), à des interfaces de type compatible, mais de rôle inverse, issues d'un composant d'environnement qui simule le comportement de l'utilisateur, pour clore le système. L'organisation hiérarchique du modèle *Fractal* repose sur le concept de *membrane*, qui délimite un espace de visibilité et peut intercepter tous les messages qui la traversent. On peut ainsi relier des propriétés ou des comportements externes à une organisation interne, dans le cas d'un composant composite.

Du point de vue interne, le photocopieur <cp> est un assemblage de quatre sous-composants : <dv>, qui distribue le flux de contrôle interne, <sn>, qui numérise le document original, <pt>, qui imprime les copies demandées et <fz>, qui sèche l'encre ou cuit le toner. L'assemblage est sous le contrôle de la membrane du photocopieur, constituée d'intercepteurs et de contrôleurs, fournis par la plate-forme *Fractal*. Les liaisons internes relient les interfaces d'alimentation électrique, distribuent le contrôle issu de <dv> via les interfaces sctl, pctl, fctl. Le flux de données est matérialisé par la circulation du papier via it, uc, ot, de l'image numérique via gs, di, uc, ot et de l'encre via bi, uc, ot.

Les signatures des interfaces, éventuellement commentées, sont décrites dans le langage support de l'implémentation *Julia* de *Fractal*, actuellement *Java*², comme indiqué sur la figure 2. Ces signatures peuvent faire appel à des interfaces ou des classes de bibliothèque, comme *Sheet*, *SheetImage*, *InkCartridge* ou *Stack* sur cet exemple.

2. Pour alléger les notations, le mot-clé `public` est sous-entendu et les “;” sont omis.

interfaces externes	interfaces internes	interfaces de bibliothèque
<pre> interface OutputTray{ Stack /*Sheet*/ sheets() ... interface PrintingPanel{ void copy (int n) double duration () boolean isPowerOn() boolean isCopying () ... interface BlackInk { InkCartridge cartridge () double minLevel () ... interface PaperTray { Stack /*Sheet*/ tray () boolean isPaperJam () int capacity () ... interface Glass{ Sheet original () boolean originalOnGlass () ... </pre>	<pre> interface UnstableCopy{ Sheet copy () ... interface DigitalImage{ SheetImage image () ... interface ScannerCtl{ void scan () double duration () ... interface PrinterCtl{ void print () double duration () ... interface FinalizerCtl{ void finalize () double duration () ... </pre>	<pre> interface Sheet{ SheetImage image () boolean isStable () ... interface SheetImage{ boolean equals(SheetImage o) boolean isBlank () ... interface InkCartridge { double level () //ml ... interface Stack /*Elem*/{ // Java 2 Platform , v1.4.2 // without genericity ... int size () Object peek () Object pop () Object push(Object o) Object elementAt(int i) ... </pre>

Figure 2 – Signatures partielles des interfaces du photocopieur.

2.2. Définition des besoins

Il est bien admis que les propriétés décrites par les signatures des interfaces, même commentées, sont très insuffisantes pour contrôler le bon fonctionnement d'une application [BAC 00]. Il faut disposer de spécifications beaucoup plus complètes des propriétés fonctionnelles et non fonctionnelles (architecture, performances, qualités de service, etc.). Certaines propriétés peuvent être vérifiées de manière anticipée par des analyses statiques ou des preuves. D'autres propriétés, notamment non fonctionnelles, qui font référence à des valeurs connues seulement pendant l'exécution, ne peuvent être vérifiées que dynamiquement. Dans le cas de composant hiérarchique où les assemblages sont dynamiques, le mot *statique* signifie « *juste au début du démarrage du composant* ». Dans le cas du photocopieur, on devrait par exemple pouvoir vérifier statiquement que tous les sous-composants sont connectés à une alimentation électrique ou prouver les propriétés fonctionnelles d'interfaces très stables, comme celles des bibliothèques (Sheet, Stack, etc.). Pour acquérir un degré de confiance élevé, il faut donc accepter divers formalismes de spécification, formels ou semi-formels pour des preuves, des vérifications statiques ou des tests dynamiques.

Les spécifications de propriétés doivent aussi tenir compte de leur domaine de visibilité. Ainsi on peut se placer au point de connexion de deux interfaces client et serveur, comme le font les propositions actuelles de contrats pour les composants [WEI 01, BAR 03]. Dans le cas de la méthode copy du panneau de commande du

photocopieur, cela permet d'énoncer la précondition :

« *que le paramètre n doit être positif* » [1]

ou la postcondition :

« *que le photocopieur n'est plus dans l'état $isCopying$.* » [2]

Ces propriétés sont certes utiles, mais d'une sémantique assez pauvre.

Pour énoncer des propriétés plus pertinentes, il faut **composer** des propriétés externes ou internes en élargissant le domaine visible, mais en respectant le principe d'encapsulation contrôlé par les membranes. Dans le cas de la méthode `copy` du panneau de commande, on voudrait exprimer, comme précondition :

« *qu'il faut assez d'encre sur l'interface bi , assez de papier sur it , un original placé sur gs et l'alimentation électrique branchée sur ap* » [3]

comme postcondition fonctionnelle :

« *que les copies du bac ot sont les feuilles qui se trouvaient dans l'ordre inverse dans le magasin it avant la copie, toutes imprimées de l'image de l'original sur gs et stabilisées* » [4]

et comme postcondition non fonctionnelle :

« *que la durée d'une copie est conforme à la vitesse d'impression du photocopieur, en nombre de pages par minute.* » [5]

D'autre part, dans le cas d'un composant composite, il faut aussi composer ses propriétés internes, visibles sur la surface interne de sa membrane et postées sur ses interfaces internes³ ou sur les interfaces externes de ses sous-composants. Ainsi, pour le composant `<cp>`, on voudrait exprimer, comme postcondition fonctionnelle de `<dv>.pp.copy()` :

« *que les copies du bac ot sont imprimées avec l'image numérisée de $<sn>.di$ sur une feuille de papier vierge $<pt>.it$ et stabilisées* » [6]

ou comme postcondition non fonctionnelle de `<dv>.pp.copy()` :

« *que la durée d'une opération de n copies est la somme de la durée de numérisation et de n fois les durées d'une impression et d'une stabilisation* » [7]

Avec de telles possibilités d'expression on pourrait alors raisonner de manière compositionnelle [BAC 00], par exemple en vérifiant, voire en prouvant, que le comportement externe du photocopieur est bien obtenu par l'organisation et le comportement interne qui l'implémentent.

2.3. Cahier des charges du système ConFract

Les besoins précédents, pour spécifier et vérifier des composants logiciels « modernes », posent un grand nombre de problèmes. Aussi, nous nous sommes limités dans un premier temps au cahier des charges suivant pour le modèle *ConFract* :

– permettre aux concepteurs de placer de manière incrémentale des spécifications fonctionnelles et non fonctionnelles pour des vérifications statiques (à l'assemblage)

3. En *Fractal*, des interfaces internes sont associées systématiquement à des interfaces externes, pour la traversée des membranes des composants composites, comme sur la figure 1.

ou dynamiques (à l'exécution); nous utiliserons d'abord un langage d'assertions exécutables, mais le métamodèle sera conçu pour supporter plusieurs formalismes.

- permettre à ces spécifications d'exprimer des propriétés ponctuelles ou compositionnelles, selon diverses préoccupations (propriété fonctionnelle, contrainte temporelle, consommation de ressource, contrainte d'architecture...), mais en respectant les règles de visibilité et d'encapsulation du modèle *Fractal* ;

- distinguer les **spécifications**, qui jouent le rôle de dispositions, des **contrats** qui sont construits dans une forme réifiée par le système *ConFract* au moment de l'assemblage, éventuellement de manière (re)négociée entre les composants impliqués. L'attribution des responsabilités doit tenir compte des possibilités d'action des participants, afin de traiter les violations selon diverses stratégies : journalisation des violations, panique organisée, renégociation des contrats, réassemblage dynamique, etc.

Ainsi, les contrats prennent ici le sens usuel de « *document négocié entre différentes parties, dont les responsabilités sont clairement établies pour chaque disposition* ».

3. Aperçu du modèle ConFract

3.1. Spécifications en langage CCL-J

Les spécifications sont écrites dans le langage *CCL-J* (*Component Constraint Language for Java*), qui a été conçu pour *Fractal* en s'inspirant d'*OCL* [OBJ 97]. Les catégories de spécifications actuelles sont classiques [D'S 98], **pre**, **post**, **inv**, **rely** et **garantee**, et correspondent à des périodes de validité dans le processus d'exécution. Chaque catégorie est composée d'une ou plusieurs clauses, identifiées par un numéro ou un label et logiquement reliées par une conjonction implicite.

Les spécifications précisent leur contexte de visibilité par des variantes de la construction **context**. Celle-ci peut citer la méthode d'un type d'interface : **context signature-de-méthode ...**, cf. figure 3 ; un type de composant : **on <COPIER> context ...**, cf. figure 4 ; ou un composant particulier, instance ou template de composant *Fractal* : **on <cp> context ...** cf. figure 5.

CCL-J permet aussi de spécifier des assertions ponctuelles, postées au niveau des types d'interface (sans référence à aucune autre interface) ou au niveau d'unités de bibliothèques du langage sous-jacent, classes ou interfaces dans le cas de *Java*. Ainsi, la partie gauche de la figure 3 traduit en *CCL-J* les spécifications [1] et [2] (page 5) ; la partie droite donne un début de spécification de la classe *Stack* de *Java*.

Mais comme dit précédemment, le plus important est de pouvoir spécifier des propriétés compositionnelles, ce qui est la principale contribution du langage *CCL-J*. La figure 4 donne des exemples de spécifications de compositions externes en *CCL-J*. Sur la partie gauche, on définit le comportement du composant *COPIER*, (équivalent des spécifications [3], [4] et [5], page 5) en une seule construction **on**, en reliant toutes

<pre> // Interfaces de composants Fractal context void PrintingPanel.copy(int n) pre 0 < n // cf spec [1] rely isPowerOn() guarantee isCopying() post !isCopying() // cf spec [2] context PaperTray inv tray().size() <= capacity() </pre>	<pre> // Unités de bibliothèque Java context void Stack.push(Object o) post: peek() == o size() == size()@pre + 1 forEach (int i in 1 .. size()-1 elementAt(i) = elementAt(i)@pre ... </pre>
---	--

Figure 3 – Exemples de spécifications d’interfaces en CCL-J.

<pre> on <COPIER> context void pp.copy(int n) pre // cf spec [3] bi.cartridge().level() >= bi.minLevel() n <= it.tray().size() && ! it.isPaperJam() gs.originalOnGlass() ap.isPluggedIn() post // cf spec [4] forEach i in 1..n : ot.sheets().elementAt(i). equals(gs.original().image()) bi.cartridge.level() <= bi.cartridge.level()@pre it.tray().size() == it.tray().size()@pre - n post // cf spec [5] speed: duration() <= n*60/attributes.CPM // CPM = nb copies/minute end on </pre>	<pre> on <SCANNER> context void scl.scan() pre gs.originalOnGlass() post di.image().equals(gs.original().image()) end on on <SCANNER> context void scl.scan() post let (double md = time() - time()@pre) md-0.01 <= scl.duration() <= md+0.01 end on on <COPIER> context * pp.*(*) pre // cf spec [8] ap.isPluggedIn(); end on </pre>
---	--

Figure 4 – Exemples de spécifications externes en CCL-J.

les interfaces dans les formules logiques. Sur la partie droite, on définit de même le comportement du composant SCANNER, mais en séparant dans deux constructions les aspects fonctionnels ou non, pour illustrer la flexibilité du langage. CCL-J fournit aussi l’opérateur * pour indiquer qu’une liste d’arguments est sans importance ou pour factoriser une propriété commune à plusieurs spécifications. L’exemple en bas à droite de la figure 4 exprime :

« que les préconditions de toutes les méthodes de l’interface pp exigent [8]
que l’interface ap ait l’alimentation électrique branchée. »

Sur cette figure, toutes les propriétés sont postées sur un *type de composant*, car elles sont valables quel que soit l’assemblage interne. La construction **let** permet de factoriser des parties de spécification et ainsi d’alléger les formules logiques.

De même, la figure 5 donne des exemples en CCL-J de spécifications de compositions internes. L’exemple du haut est l’équivalent de la propriété fonctionnelle [6], puis de la propriété non fonctionnelle [7] données page 5. L’exemple du bas de la figure illustre une contrainte d’architecture, en exploitant les facilités d’introspection de *Fractal* :

« que tous les sous-composants de <cp> aient au moins une interface client [9]
de type *ACPower*. »

```

on <cp> context <dv>.pp.copy(int n)
post // cf spec [6]
  (let Stack /*Sheet*/ r = <fz>.ot.sheets() ; // feuilles de sortie
    int o = <fz>.ot.sheets().size()@pre ; // nb feuilles avant copie/ot
    int p = <pt>.it.tray().size()@pre ; // nb feuilles avant copie/it
  )
  forEach(i in 1 .. n:
    r.elementAt(o+i).image().equals(<sn>.di.image()) &&
    r.elementAt(o+i).image().isStable() &&
    r.elementAt(o+i) == <pt>.it.tray().elementAt(p-i+1)@pre)
end on

on <cp> context <dv>.pp.copy(int n)
post // cf spec [7]
  let(double thd = <sn>.sctl.duration() +
    n*(<pt>.pctl.duration() + <fz>.fctl.duration()))
  thd - 1 <= duration() <= thd + 1
end on

on <cp>
inv // cf spec [9]
  let(Component[] subc=Fractal.getContentController(this).getFcSubComponents())
  forEach i in 1..subc.length :
    Arrays.asList(subc[i].getFcInterfaces()).exists(Interface ir :
      Class.forName(ir.getFcItfSignature()).conformsTo(ACPower.class)
      && ir.isFcClientItf());
end on

```

Figure 5 – Exemples de spécifications internes en CCL-J.

3.2. Types de contrat

Le système *ConFract* distingue différents types de contrats selon les spécifications, fonctionnelles ou non, données par les concepteurs (figure 6).

- Les *contrats d'interface* sont établis au point de connexion entre deux interfaces client et serveur (par exemple pp entre <cp> et son environnement <env> ou di entre <pt> et <sn>). Les spécifications qui sont retenues ne citent que les méthodes et les entités qui sont visibles à cette interface.

- Les *contrats de composition externe* sont postés sur la face externe de la membrane d'un composant, par exemple celui sur <cp>, en haut et à gauche de la figure. Ils sont formés des spécifications qui ne citent que des interfaces externes du composant. Ils expriment donc les règles d'utilisation et de comportement externe du composant.

- Les *contrats de composition interne* sont postés sur la face interne de la membrane d'un composant composite, par exemple celui de <cp>, en haut et à droite de la figure. Ils sont formés des spécifications qui ne citent que des interfaces internes du composant et des interfaces externes de ses sous-composants. Ils expriment les règles d'assemblage et de comportement interne d'implémentation.

- Les *contrats de bibliothèque* sont des contrats surtout fonctionnels qui sont définis sur des unités réutilisables du langage sous-jacent, classes ou interfaces dans le cas de *Java*. Ils sont pertinents seulement pour l'implémentation des composants ou de

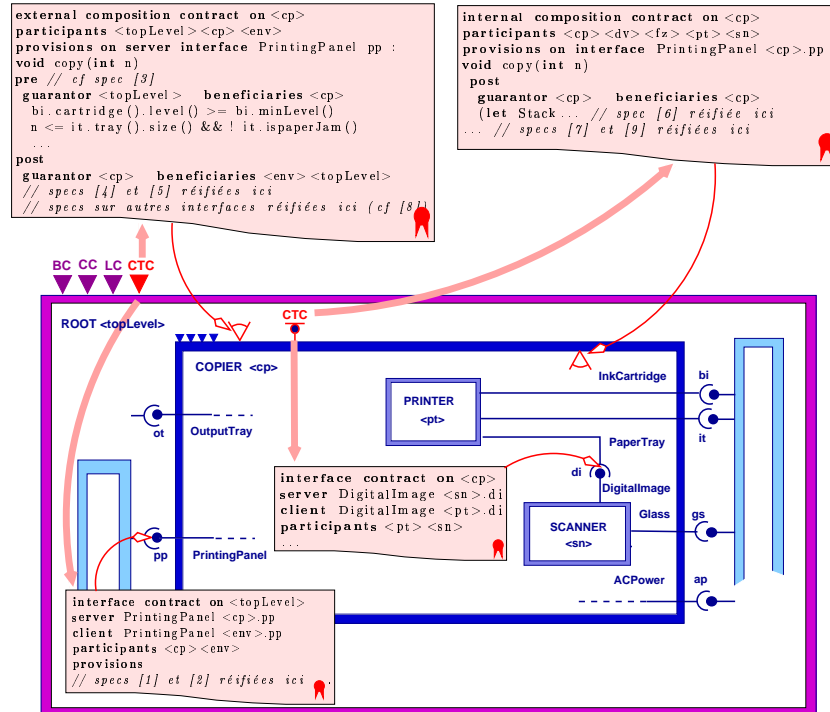


Figure 6 – Exemples de contrats construits à partir des spécifications du photocopieur.

leurs interfaces et ne participent pas aux négociations d'assemblage. Comme ils n'ont rien d'original, nous n'en parlerons plus dans la suite de cet article.

3.3. Responsabilités

Au cours de la réification d'un contrat, le métamodèle de *ConFract* détermine les responsabilités associées à chaque spécification, parmi la liste des composants *participants* au contrat. Lorsque les responsabilités d'une spécification sont toutes déterminées, celle-ci devient une **disposition** du contrat (ou spécification fermée). Ces responsabilités peuvent être : a) le *garant*, l'unique composant qui doit être prévenu en cas de négation de la disposition et qui a la capacité d'agir pour traiter le problème, b) le ou les *bénéficiaires*, les composants qui peuvent compter sur la véracité de la disposition et qui peuvent être prévenus en cas de changement d'état de celle-ci (invalidation ou rétablissement) c) d'éventuels *contributeurs*, qui sont des composants qui participent à la véracité de la disposition, et à qui on peut demander de « faire un effort » pour le rétablissement de cette disposition. Contrairement à la plupart des systèmes de contractualisation, il n'y a pas de notion de « coupable » dans notre modèle, car celui-

ci est plus dynamique et ouvert à négociations. Dans l'exemple du photocopieur, si la précondition de la méthode `copy` qui spécifie « `n <= it.tray().size()` » est fausse, on peut imaginer une négociation pour, soit limiter `n` par le panneau de commande, soit changer l'interface `PaperTray` pour avoir un magasin de papier de plus grande capacité. Pour cette disposition, c'est `<topLevel>` le garant, qui mène la négociation, `<cp>` le bénéficiaire et `<env>` et `<cp>` les contributeurs, par l'intermédiaire des interfaces `it` et `pp`.

Par manque de place nous ne décrivons ici que les responsabilités des pre- et post-conditions des contrats de composition. En considérant la figure 1, les responsabilités du contrat de composition externe de l'imprimeur `PRINTER <pt>` sont données par le tableau suivant :

Rôle de l'interface	Construction	Garant	Bénéficiaires
server : <i>pctl, uc</i>	pre	<code><cp></code>	<code><pt></code>
server : <i>pctl, uc</i>	post	<code><pt></code>	<code><cp></code> , <code><dv></code> (<i>pctl</i>), <code><fz></code> (<i>uc</i>)
client : <i>ap, di, bi, it</i>	pre	<code><pt></code>	<code><cp></code> (<i>ap, it, bi</i>), <code><sn></code> (<i>di</i>)
client : <i>ap, di, bi, it</i>	post	<code><cp></code>	<code><pt></code>

Par exemple, sur le composant `<pt>`, pour la postcondition d'une méthode sur son interface serveur `uc`, le garant est le composant lui-même — puisqu'il implémente la méthode et fournit l'interface — et les bénéficiaires sont `<cp>` qui englobe `<pt>` et `<fz>` qui est connecté à l'interface `uc`. Comme le contrat de composition externe représente les règles d'utilisation du composant `<pt>`, il est logique d'attribuer la responsabilité de ce contrat au composant `<cp>` qui englobe `<pt>`, parce que c'est `<cp>` qui a la maîtrise de son assemblage interne. Quant aux responsabilités associées au contrat de composition interne de `<cp>`, elles sont immédiates car `<cp>` est à la fois garant et seul bénéficiaire dans tous les cas, puisque seul responsable de son implémentation.

3.4. Fermeture progressive des contrats

Lorsqu'un composant est introduit dans un assemblage, *ConFract* crée un contrat de composition interne s'il est composite, et un contrat de composition externe s'il a des spécifications liées à plusieurs de ses interfaces. Pour chaque spécification liée à des contrats de composition, un patron (*template*) de spécification est créé et attaché aux contrats de composition. Chaque patron est en attente de tous ses contributeurs pour « se fermer ». Lorsqu'un nouveau sous-composant est ajouté à une composition, tous les patrons qui y participent ont leurs responsabilités qui sont complétées. Lorsque tous les contributeurs d'un patron sont connus, celui-ci est fermé et devient une disposition. Lorsque tous les patrons des spécifications d'un contrat de composition interne sont fermés, le contrat est lui aussi *fermé*, car toutes les responsabilités sont identifiées pour chacune de ses dispositions et le composant peut-être lancé.

Pour un contrat d'interface entre client et serveur, le cycle de vie est très simple, car il y a seulement deux participants au contrat. Il est créé lors de la connexion entre les interfaces et est automatiquement fermé, puisque les responsabilités sont connues.

4. Mise en oeuvre du système

Le système *ConFract* est actuellement intégré à *Fractal* à l'aide de *Julia* [BRU 04], son implémentation de référence en *Java*. Il utilise ainsi les mécanismes de contrôleurs, de *mixins* et d'intercepteurs, afin que son intégration respecte le principe de séparation des préoccupations.

4.1. Le contrôleur de contrats

Les différents contrats sont gérés par des *contrôleurs de contrats* (CTC sur la figure 6), placés sur la membrane de chaque composant. Chaque contrôleur de contrats d'un composant composite prend en charge le cycle de vie et l'évaluation :

- du contrat de composition interne du composite sur lequel il est placé,
- du contrat de composition externe de chacun des sous-composants,
- du contrat d'interface de chaque connexion située dans son contenu.

Lors de la création d'un composant composite, l'initialisation de son contrôleur de contrats crée son contrat de composition interne. Les autres contrats seront construits et mis à jour par des *mixins*.

4.2. Mixins

En fonction des manipulations et des actions qui surviennent sur les composants, la gestion des contrats est assurée par différents *mixins* placés sur les contrôleurs *Fractal* suivants (cf. figure 6) :

– *Binding controller* (BC). Comme ce contrôleur gère la création et la suppression des connexions entre composants, un mixin notifie le contrôleur de contrats englobant des connexions (resp. déconnexions) afin d'instancier (resp. de supprimer) le contrat d'interface correspondant.

– *Content controller* (CC). Ce contrôleur gère l'insertion de sous-composants à l'intérieur d'un composite. Un mixin notifie le contrôleur de contrats à chaque insertion, afin qu'il construise le contrat de composition externe du nouvel entrant *C*. Le contrôleur de contrat peut aussi fermer les spécifications du contrat de composition interne qui font référence à *C*. Les actions inverses sont réalisées lors du retrait d'un sous-composant.

– *Life-cycle controller* (LC). C'est la méthode `start` de ce contrôleur qui démarre un composant. Due à l'ouverture de *Fractal*, le seul moment où l'on est certain qu'un

composant est complètement assemblé est juste au début de `start`, le moment précis où l'on peut effectuer des vérifications « statiques » (cf. section 2.2). Le contrôleur de contrats du composant (resp. du composant englobant) vérifie alors que son contrat de composition interne (resp. externe) est fermé. Enfin, les dispositions des contrats qui sont vérifiables statiquement, comme les invariants de composants, sont alors évaluées.

4.3. *Intercepteurs*

Pour l'évaluation des dispositions de contrats, des intercepteurs *Julia* sont utilisés. Chaque interface *Fractal* spécifiée par une disposition de contrats reçoit un intercepteur sur l'entrée et/ou la sortie des méthodes. Dans le cas de *CCL-J*, lorsqu'une méthode est appelée sur une interface *Fractal*, le contrôleur de contrats évalue alors les préconditions du contrat d'interface et vérifie l'absence d'erreurs de hiérarchie [FIN 01]. Ensuite, les préconditions du contrat de composition externe du composant qui reçoit l'appel sont évaluées, afin de garantir au composant que son environnement est conforme à ses hypothèses. Enfin, les préconditions du contrat de composition interne sont évaluées. Des vérifications, dans l'ordre inverse, sont effectuées au retour de la méthode pour les postconditions et les invariants.

5. Travaux connexes

L'utilisation des assertions a débuté pour la spécification et la preuve de programmes. Des contrats logiciels à base d'assertions exécutables ont d'abord été utilisés pour le langage Eiffel [MEY 92]. Depuis, de nombreuses extensions de langages à objets ont été proposées pour adapter et étendre ces *contrats objets*, notamment pour *Java* avec *iContract* [KRA 98], *ST-class* [DEV 02] et *JML* [LEA 99] (cf. [PLö 02] pour une comparaison des prototypes et de leur intégration). *JML* combine d'ailleurs des assertions exécutables et des constructions de programme abstrait, afin de construire des modèles exécutables qui utilisent des fonctions d'abstraction sur les classes spécifiées. Cette forme avancée de spécification aurait tout intérêt à être étendue au contexte des composants logiciels. Le langage *CCL-J* contient actuellement moins de constructions que *JML*, car nous avons voulu privilégier sa simplicité pour valider notre modèle de contrats ; *CCL-J* dispose cependant de possibilités non présentées dans cet article, comme l'utilisation de variables de modèles abstraits. Dans toutes les approches évoquées, les contrats objets, de clientèle ou d'héritage, sont implicites ; ils ne sont qu'un moyen d'attribuer la responsabilité d'une erreur de programmation, contrairement aux contrats définis dans *ConFract* qui offrent des possibilités de traitement beaucoup plus étendues, notamment par des négociations [CHA 04].

Les contrats de composition fournis par *ConFract* peuvent être comparés aux contrats de collaboration d'objets proposés dans [HEL 90]. La notion de vues dans la collaboration est similaire aux rôles des participants dans nos contrats. Toutefois, dans le système *ConFract*, les contrats de composition sont portés par des composants

— ce qui permet de les répartir dans la hiérarchie — et sont générés automatiquement en fonction des actions d’assemblage et de connexion.

Des adaptations de l’approche par contrats dans les systèmes à objets ont déjà été proposées pour les composants logiciels. Des contrats pour les composants *.NET* ont été développés [BAR 03], mais ils se contentent d’utiliser le langage *AsmL* d’annotation par programme abstrait pour spécifier séparément les interfaces. Dans le monde UML, l’expression de contrats a fait l’objet de plusieurs propositions : [WEI 01], qui spécifie des contrats fonctionnels avec OCL [OBJ 97] et non fonctionnels avec QML [FRØ 98], et plus récemment [DEF 04], qui propose une variante des contrats de [WEI 01], mais que l’on peut vérifier par résolution de contraintes. Bien que les contrats soient plus explicites dans ces travaux que dans ceux du monde objet, leur caractérisation et leur cycle de vie restent relativement faibles. De plus, ces travaux effectuent une complète séparation entre les aspects fonctionnels et non fonctionnels alors que souvent, comme nous l’avons vu sur les exemples, il est nécessaire de pouvoir les combiner.

6. Conclusion

Les plates-formes récentes comme *Fractal*, combinent les avantages des différentes approches de composants logiciels, les assemblages dynamiques et répartis d’intergiciels et l’organisation hiérarchique des approches modulaires et à objets. Mais cela pose de nouveaux défis pour maîtriser la correction et la robustesse des propriétés fonctionnelles ou de qualité de service. Pour résoudre ces problèmes, le système *ConFract* propose une approche contractuelle où les contrats sont construits dynamiquement à partir des spécifications, au moment des assemblages. Différents types de contrats sont identifiés, notamment pour exprimer des compositions de propriétés externes ou internes. Les responsabilités sont identifiées de manière fine au niveau de chaque disposition, ce qui permet de bien organiser le rattrage des erreurs et ouvre la voie à d’éventuelles négociations. L’implémentation actuelle de *ConFract* suit une approche par séparation des préoccupations en exploitant les intercepteurs et contrôleurs de *Fractal*.

Le langage *CCL-J* de spécifications par assertions exécutables permet déjà d’énoncer des propriétés compositionnelles, fonctionnelles ou non, mais devra être étendu pour faciliter les négociations, améliorer son expressivité pour les contraintes non fonctionnelles et permettre une évaluation plus efficace avec des techniques comme celles que nous avons déjà expérimentées [COL 99]. L’intégration des mécanismes de négociation est en cours [CHA 04] et permettra de traiter les violations d’assertions selon diverses stratégies. Le métamodèle *ConFract* est ouvert à différents formalismes de spécifications et nous prévoyons, d’expérimenter des langages formels pour certifier *a priori* certaines propriétés par des preuves et de formaliser de nouveaux processus de développement logiciel par assemblage dynamique de composants.

Remerciements

Ce travail a été financé par France Télécom sous le contrat de collaboration n° 422721832I3S. Les auteurs remercient Éric Bruneton, Thierry Coupaye, et Nicolas Rivière pour leurs remarques constructives sur le modèle *ConFract* et Alain Ozanne pour son implémentation.

7. Bibliographie

- [BAC 00] BACHMAN F. *et al.* « Technical Concepts of Component-Based Software Engineering », rapport n° CMU/SEI-2000-TR-008, vol. 2 mai 2000, Carnegie Mellon Software Engineering Institute.
- [BAR 03] BARNETT M., SCHULTE W., « Runtime verification of .NET contracts », *Journal of Systems and Software*, vol. 65, n° 3, 2003, p. 199-208.
- [BRU 04] BRUNETON E., COUPAYE T., LECLERCQ M., QUÉMA V., STEFANI J.-B., « An Open Component Model and Its Support in Java », *ICSE 2004 - CBSE7*, vol. 3054 de LNCS, Springer Verlag, mai 2004, fractal.objectweb.org.
- [CHA 04] CHANG H., COLLET P., « Vers la négociation de contrats dans les composants logiciels hiérarchiques », rapport n° ISRN I3S/RR-2004-33-FR, I3S, CNRS/UNSA.
- [COL 99] COLLET P., ROUSSEAU R., « Efficient Implementation Techniques for Advanced Assertion Languages », *L'Objet*, vol. 5, n° 3-4, 1999, p. 417-442, Hermes Science.
- [DEF 04] DEFOUR O., JÉZÉQUEL J.-M., PLOUZEAU N., « Extra-Functional Contract Support in Components », CRNKOVIC I., STAFFORD J., SCHMIDT H., WALLNAU K., Eds., *ICSE 2004 - CBSE7*, vol. 3054 de LNCS, Springer Verlag, mai 2004, p. 217-232.
- [DEV 02] DEVEAUX D., « Self-Testable Classes Requirements », Technical report, mars 2002, SCoT Project : "ITR Bretagne Program", www.stclass.org.
- [D'S 98] D'SOUZA D. F., WILLS A. C., *Object, Components and Frameworks with UML : The Catalysis Approach*, Addison-Wesley Publishing Co. (Reading, MA), 1998.
- [FIN 01] FINDLER R. B., FELLEISEN M., « Contract Soundness for Object-Oriented Languages », *Proceedings of OOPSLA'2001*, 2001.
- [FRØ 98] FRØLUND S., KOISTINEN J., « Quality of Service in Distributed Object Systems Design », *4th USENIX COOTS Conference, Santa Fe (New Mexico)*, Avril 1998.
- [HEM 90] HELM R., HOLLAND I. M., GANGOPADHYAY D., « Contracts : Specifying Behavioral compositions in Object-Oriented Systems », MEYROWITZ N., Ed., *OOPSLA/E-COOP'90*, Ottawa, octobre 1990, p. 169-180.
- [KRA 98] KRAMER R., « iContract - The Java Design by Contract Tool », *TOOLS USA'98*, IEEE Computer Society Press (New York), Aug. 1998.
- [LEA 99] LEAVENS G. T., BAKER A. L., RUBY C., « JML : A Notation for Detailed Design », *Behavioral Specifications of Businesses and Systems*, Kluwer, 1999, p. 175-188.
- [MEY 92] MEYER B., « Applying "Design by contract" », *IEEE Computer*, vol. 25, n° 10, 1992, p. 40-51.
- [OBJ 97] OMG, « Object Constraint Language Specification », version 1.1, n° ad/97-08-08, septembre 1997, www.software.ibm.com/ad/ocl.
- [PLÖ 02] PLÖSCH R., « Evaluation of Assertion Support for the Java Programming Language », *Journal of Object Technology, TOOLS USA'2002*, vol. 1,3, 2002, p. 5-17.
- [WEI 01] WEIS T., BECKER C., GEIHS K., PLOUZEAU N., « A UML Meta-model for Contract Aware Components », *UML'2001*, vol. 2185 de LNCS, octobre 2001, p. 442-456.