Université Nice SOPHIA ANTIPOLIS

CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

CONTRÔLE D'ADMISSION DE COMPOSANTS AVEC DES CONTRATS COMPORTEMENTAUX

Philippe Collet, Roger Rousseau

Projet OCL

Rapport de recherche
I3S/RR–2002-49–FR

Septembre 2002

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télécopie : (33) 492 942 898
http://www.i3s.unice.fr/I3S/FR/

RÉSUMÉ :

Dans cet article, nous proposons un modèle technique et méthodologique afin de mieux contrôler la compatibilité sémantique des interfaces des composants logiciels lors de leur admission sur une plate-forme. Ces contrôles d'admission portent principalement sur des contrats comportementaux basés sur des assertions exécutables. Leur compatibilité est déterminée, soit par certification, soit par des tests sur la plate-forme. Compte tenu des nombreux paramètres pour faire le meilleur choix d'admission, nous préconisons une politique de contractualisation qui repose sur un processus de négociation.

MOTS CLÉS :

génie logiciel orienté composant, contrat comportemental, assertions exécutables, compatibilité de contrats, certification, négociation

---

ABSTRACT:

In this paper, we propose a technical and methodological model to better control the semantic compatibility of software components' interfaces at the time of their admission on a platform. These admission controls mainly rely on behavioral contracts using executable assertions. Their compatibility is determined, either by certification or by testing on the platform. Considering the numerous parameters to make the best choice of admission, we recommend a contractual policy based on a negotiation process

KEY WORDS :

component-based software engineering (CBSE), behavorial contract, executable assertions, compatibility of contracts, certification, negotiation

Contrôle d'admission de composants avec des contrats comportementaux

Philippe Collet — Roger Rousseau

Équipe 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. *Dans cet article, nous proposons un modèle technique et méthodologique afin de mieux contrôler la compatibilité sémantique des interfaces des composants logiciels lors de leur admission sur une plate-forme. Ces contrôles d'admission portent principalement sur des contrats comportementaux basés sur des assertions exécutables. Leur compatibilité est déterminée, soit par certification, soit par des tests sur la plate-forme. Compte tenu des nombreux paramètres pour faire le meilleur choix d'admission, nous préconisons une politique de contractualisation qui repose sur un processus de négociation.*

ABSTRACT. *In this paper, we propose a technical and methodological model to better control the semantic compatibility of software components' interfaces at the time of their admission on a platform. These admission controls mainly rely on behavioral contracts using executable assertions. Their compatibility is determined, either by certification or by testing on the platform. Considering the numerous parameters to make the best choice of admission, we recommend a contractual policy based on a negotiation process.*

MOTS-CLÉS : *génie logiciel orienté composant, contrat comportemental, assertions exécutables, compatibilité de contrats, certification, négociation.*

KEYWORDS: *component-based software engineering (CBSE), behavorial contract, executable assertions, compatibility of contracts, certification, negotiation.*

1re soumission à *LMO'2003*, le 20/09/2002.

1. Introduction

L'approche par objets n'a pas apporté tous les bénéfices attendus par l'industrie logicielle, à l'échelle prévue. Au cours du développement de cette approche, les mécanismes objets ont gagné en subtilité, mais aussi en complexité : leur totale maîtrise reste l'apanage de développeurs experts. *A contrario*, l'approche par composants vise la réutilisation par un assemblage aisé des composants, par des programmeurs ordinaires, alors que le développement des composants complexes et des plates-formes est laissé aux experts. Cependant, l'approche par composants déplace la complexité d'une hiérarchie de classes vers les points de connexion entre les composants. Les plates-formes tendent à se baser sur des systèmes à objets et à se complexifier. Il faut donc mieux contrôler les aspects sémantiques des composants.

Il y a donc intérêt à garder le meilleur des deux approches, par objets et par composants, dans les nouvelles architectures logicielles : les composants pour le premier niveau d'assemblage et les objets pour l'organisation interne des composants. Selon un rapport du SEI [BAC 00], le principal verrou de l'approche par composants est la difficulté de déduction des propriétés d'un assemblage de composants à partir des composants eux-mêmes, pris individuellement. Résoudre ce problème permettra de garantir certaines propriétés de fonctionnement d'un assemblage, notamment de qualité de service, et surtout, d'effectuer un *raisonnement compositionnel*. Pour atteindre cet objectif, nous proposons d'utiliser des assertions exécutables, organisées sous forme de contrats, pour spécifier les interfaces des composants. Cela permet d'introduire plus de sémantique, de manière incrémentale et adaptée à la plupart des développeurs, et de concilier des possibilités de preuve et de tests.

Dans ce contexte, il est nécessaire de contrôler la compatibilité des interfaces lors de l'admission des composants sur une plate-forme. Pour cela, nous définissons la notion de compatibilité pour des interfaces contractualisées, puis nous proposons un modèle technique et méthodologique pour la contrôler. Ce modèle s'appuie, soit sur des certifications, soit sur des tests sur la plate-forme. Compte tenu des nombreux paramètres pour faire le meilleur choix d'admission, nous préconisons une politique de contractualisation qui repose sur un processus de négociation.

La section 2 précise les hypothèses minimales pour ce modèle. La section 3 rappelle les apports de l'approche contractuelle pour les composants. La section 4 développe les règles et les procédures de contrôle pour établir la compatibilité des contrats. Nous terminons cet article par une discussion sur les possibilités de négociation et la conclusion.

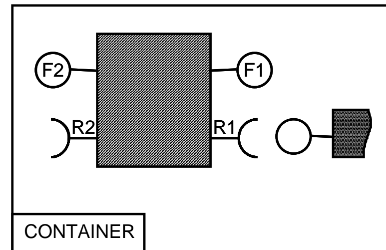
2. Cadre de travail

2.1. *Hypothèses minimales sur le modèle de composants*

Nous prenons comme point de départ la définition d'un composant logiciel donnée par Szyperski [SZY 98] : « un composant logiciel peut être vu comme une unité de

composition avec des interfaces fournies et requises **contractuellement** spécifiées. Cette unité est déployable et toutes ses dépendances externes sont explicites. »

Un composant est une **boîte noire** qui publie des interfaces fournies (par ex. F1 et F2) et requises (par ex. R1 et R2). Le container est chargé de contrôler le cycle de vie des composants, notamment leurs connexions. C'est donc au niveau des interfaces qu'il faut placer le contrôle sémantique d'admission d'un composant sur le container.



Nous traitons les aspects fonctionnels par les deux premiers niveaux de contrats, selon la classification de Beugnard *et al.* [BEU 99], les contrats *de type* et *de comportement*. Pour contrôler la connexion d'une interface fournie à une interface requise, nous associons à chaque catégorie de contrat un référentiel et une notion de compatibilité : un référentiel de type (RT) pour la *T-compatibilité* et un référentiel de comportement (RC) pour la *C-compatibilité*.

Les deux autres niveaux de contrats [BEU 99] — de synchronisation et de contraintes non fonctionnelles —, la prise en compte de mécanismes objets internes aux composants et la gestion événementielle devront être étudiés comme extension de ce modèle. Par ces hypothèses minimales, nos propositions sont d'abord applicables aux modèles de composants actuels, comme CCM et .NET, et aux propositions récentes comme le support des composants dans UML et le modèle Fractal [BRU 02].

2.2. *Référentiel de type (RT)*

Ce référentiel gère les types de manière très classique. Un type d'interface est défini par un nom unique, la liste des noms de ses surtypes et la liste des signatures typées de ses méthodes. La T-compatibilité entre deux interfaces est obtenue en interrogeant le RT qui la détermine directement par les noms de types et leur place dans la hiérarchie. Le RT agit comme un serveur unique et accessible en tout point d'une plate-forme de composants déployés.

Ce système de typage supporte l'héritage multiple, avec d'éventuels conflits résolus lors de l'admission d'une nouvelle interface dans le RT. De même, le paramétrage des types (généricité) et l'introduction dans le RT des types de classes ou d'autres classifieurs au sens UML peut être envisagée, pourvu que la compatibilité de leur type avec celui des interfaces ait un sens. Pour d'évidentes raisons de simplicité, nous renonçons à des systèmes à composants ayant des référentiels de type différents.

3. Approche contractuelle pour les composants

3.1. *Définition d'un contrat comportemental dans les systèmes à objets*

Un contrat peut être défini comme une spécification d'obligations mutuelles entre deux parties au moins. Un *contrat comportemental* doit spécifier les hypothèses et le résultat de ces opérations, par exemple avec l'une des nombreuses approches de spécification formelle qui ont été développées depuis des années. Nous basons les contrats comportementaux sur des assertions exécutables, organisées en préconditions, postconditions et invariants [MEY 92]. Cette approche apporte une solution pragmatique pour la spécification et la vérification à l'exécution de propriétés données, avec de bonnes aptitudes à supporter des spécifications incomplètes lors d'un développement incrémental, ce qui facilite la compréhension, la flexibilité et l'adaptabilité.

Les langages d'assertions pour les modèles à objets [OBJ 97, KRA 98] fondent généralement leur expressivité sur la logique des prédicats, augmentée d'opérateurs de quantification universelle et existentielle, voire d'opérateurs d'ordre supérieur (sélection, projection), avec la possibilité d'effectuer des appels de fonction avec liaison dynamique. Cette généralité rend ces langages assez expressifs, mais l'évaluation systématique des assertions peut être coûteuse (quantifications, appels distants, etc.). La liaison dynamique empêche aussi de prouver statiquement certaines propriétés.

3.2. *Sémantique des contrats comportementaux*

L'un des avantages des contrats comportementaux basés sur des assertions exécutables est leur capacité à préciser la sémantique des méthodes et des classes lors du sous-typage : dans un sous-type, les préconditions d'une méthode peuvent être moins fortes, tandis que les postconditions et les invariants peuvent être renforcés [MEY 92]. Ceci s'explique simplement par l'objectif de substitution polymorphe d'objets, sous-types d'une entité réceptrice.

3.2.1. *Vérification des contrats*

Avec des assertions exécutables, les contrats peuvent être vérifiés, soit par des preuves s'ils sont complets avec la logique de Hoare [HOA 71], soit le plus souvent par des tests lors d'une expérience exécutoire, voire de l'exploitation. Ces tests consistent à évaluer les assertions à des instants observables lors de l'exécution du code de l'application, à l'entrée des méthodes pour les préconditions, et à la sortie pour les postconditions et les invariants. L'application en cours de développement sert souvent de lanceur de tests, mais des tests unitaires spécifiques mis en oeuvre à partir des contrats [DEV 99a] sont préférables. Comme l'évaluation systématique de toutes les assertions de toutes les classes et de toutes les méthodes est souvent trop coûteuse, il est possible de mettre en place des techniques semi-automatiques pour sélectionner les assertions les plus pertinentes à évaluer ou pour réduire le niveau d'exhaustivité de certains tests par échantillonnage. Ces techniques utilisent notamment le niveau de

confiance obtenu dans une classe, qui augmente au fur et à mesure de l'évaluation des contrats [COL 99].

3.2.2. *Erreurs détectées*

Plusieurs types d'erreurs peuvent être détectés lors de la compilation ou de l'évalution des contrats. Il est possible de déterminer statiquement si un contrat est *structurellement* correct : toute précondition d'une méthode doit être formée d'éléments visibles au client [MEY 92], car il est vain d'assurer un contrat qu'on ne connaîtrait pas. De même, les assertions surspécifiées des postconditions ou des invariants qui font apparaître des éléments invisibles peuvent être automatiquement masquées. En complément à cette règle structurelle, on peut établir des régles de cohérence plus méthodologiques, par exemple en s'assurant qu'un changement d'état effectué par un service doit être observable à l'aide d'une ou plusieurs fonctions appropriées dans l'interface [MCK 93].

Lors de l'évaluation des contrats, plusieurs formes d'erreur peuvent être détectées. Les plus classiques correspondent à la relation client/fournisseur : si une précondition échoue, le client est fautif car il n'a pas assuré un prérequis du service ; au contraire, si les postconditions échouent, le fournisseur est fautif car il a mal implémenté les résultats qu'il avait proclamés dans les postconditions. La sémantique spécifique des contrats lors d'un sous-typage[1] nécessite des vérifications spécifiques pour s'assurer que le concepteur de la classe héritière respecte bien les règles de substituabilité au niveau des pré et postconditions. Ainsi par exemple, il faut vérifier que les préconditions de la classe ancêtre impliquent celles de la classe héritière. La vérification des préconditions d'une méthode m dans une classe héritière se fait selon la formule :
$(pre_{heritier}(m) \vee pre_{ancetre}(m)) \wedge (pre_{ancetre}(m) \Rightarrow pre_{heritier}(m))$
Si cette implication est fausse, il y a une *erreur de hiérarchie* et c'est le concepteur de la classe héritière qui est fautif [FIN 01]. On procède de façon similaire pour vérifier que les postconditions de la classe héritière impliquent celles de la classe ancêtre et que l'invariant de la classe héritière satisfait aussi ceux de ses ancêtres.

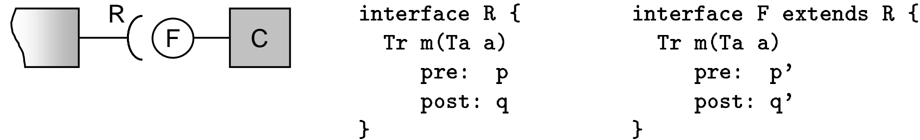
3.3. *Contrats comportementaux pour les composants*

Outre les contrats de type, les plates-formes de composants actuelles définissent souvent des contrats comportementaux entre les composants ou entre un composant et son container, mais en langage naturel. En revanche, de nombreuses études sur la problématique des composants logiciels ont montré la nécessité de contrats fonctionnels et non fonctionnels [SZY 98, BEU 99, BAC 00, SOM 02]. L'étude du SEI [BAC 00] montre aussi que l'on peut utiliser des contrats pour deux objectifs de contrôle séman-

1. Par manque de place, nous limitons notre raisonnement aux pre et postconditions et à l'héritage simple. Les règles pour les postconditions sont directement applicables aux invariants et l'ensemble des résultats peut être généralisé à l'héritage multiple.

tique, (i) de la compatibilité de la connexion des composants au niveau des interfaces et (ii) de la sémantique de l'interaction entre plusieurs composants [HOL 92].

En ne considérant que le premier cas, un composant doit publier des interfaces requises ou fournies, annotées par des contrats comportementaux. Pour illustrer le problème du contrôle d'admission, considérons le cas d'un composant C, qui fournit une interface F, candidate à une connexion sur R, une interface requise par un composant quelconque.




```
interface R {
  Tr m(Ta a)
     pre:  p
     post: q
}
```

```
interface F extends R {
  Tr m(Ta a)
     pre:  p'
     post: q'
}
```

Les interfaces R et F contiennent des méthodes comme `m`, dont le comportement est spécifié par des pré et postconditions. Etablir si une connexion entre F et R est possible revient à savoir si F est *compatible* avec R, avec la même sémantique de substitutionalité que dans les langages à objets contractualisés, lorsqu'on veut vérifier qu'un objet o d'un type $\mathcal{T}_o$ peut être attaché à une entité e d'un type $\mathcal{T}_e$, surtype de $\mathcal{T}_o$. On peut donc utiliser des techniques de vérifications d'assertions similaires à celles utilisées dans les langages à objets. La T-compatibilité peut être établie *a priori* grâce au référentiel de type, mais il nous faut maintenir définir la C-compatibilité au niveau des contrats comportementaux entre interfaces.

4. Contrôle des contrats comportementaux lors de l'admission

4.1. *Principe de fonctionnement*

Les phases de développement et de déploiement se chevauchent, car l'un des objectifs de l'approche par composants est de faciliter la maintenance des applications par des réactualisations à chaud avec la connexion de nouveaux composants. L'assemblage de composants ou leur remplacement nécessite une phase d'admission qui est réalisée par le container de la plate-forme. Les contrôles d'admission visent la garantie que le composant fait ce que l'on attend de lui, aussi bien sur des aspects fonctionnels que non fonctionnels (non étudiés dans cet article).

Lorsque la C-compatibilité ne peut être obtenue directement par l'égalité des types et de la structure des contrats comportementaux, on utilise des règles et des procédures de validation des contrats, avec des contrôles de sous-typage, des tests et des certifications. Comme les tests peuvent être plus ou moins intensifs et complets, la C-compatibilité est exprimée avec différents niveaux de confiance. Certains cas d'incompatibilité peuvent amener à une négociation, car il est parfois nécessaire d'outrepasser certains contrôles, sous la responsabilité de procédures ou d'interventions humaines qui acceptent le risque engendré (voir section 5).

Comme la complexité des langages assertionnels rend impossible la déduction automatique de la subsumption d'une formule assertionnelle quelconque par une autre

(sauf dans quelques cas simples), le référentiel des comportements (RC) s'appuie sur des preuves ou des tests anticipés, voire sur des testse ffectués lors de l'admission. Ces tests devront être accompagnés d'un niveau de confiance, si possible certifié. Il nous faut donc étudier deux problèmes : (i) comment faire fonctionner le RC pour déterminer les compatibilités de connexion ? (ii) comment négocier la connexion d'un composant, si le niveau de compatibilité des contrats n'a pas le niveau de confiance requis par l'application ?

Nous proposons de résoudre ces problèmes en utilisant les concepts suivants :

– la T-compatibilité des interfaces fournies avec les interfaces requises,

– le RC qui pilote le calcul de compatibilité des comportements,

– des règles de compatibilité entre interfaces contractualisées,

– des certifications, pour obtenir une réponse plus rapide et plus sûre,

– des tests de compatibilité en cas d'absence de certificat, basés sur une évaluation des contrats lors de l'admission, éventuellement prolongée lors de l'exécution,

– la négociation en cas d'incompatibilité, qui peut conduire au refus d'admission ou au relâchement des contrats dans des contextes particuliers.

4.2. *Référentiel de comportements (RC)*

Les interfaces requises ou fournies sont identifiées dans le RC par leur nom de type, unique dans le RT associé. Les composants sont aussi identifiés de manière unique dans le RC dans un espace de nommage spécifique. C'est le RC, accessible en tout point de la plate-forme, qui répond aux questions de compatibilité des interfaces. Dans les cas favorables, son travail est simplifié par des certificats associés aux composants et aux interfaces, et délivrés par un serveur de certification (SC) (voir figure 1).

Toute connexion d'une interface présuppose évidemment qu'elle soit utilisable. Pour préciser les contrôles à effectuer, nous donnons les définitions préliminaires suivantes :

– une interface requise est *utilisable*, *ssi* elle est fonctionnellement *cohérente* ;

– une interface fournie est *utilisable*, *ssi* elle est *cohérente* et que l'implémentation fournie par son composant est *correcte* ;

– un composant est *utilisable ssi* toutes ses interfaces requises et fournies sont *utilisables*.

Pour trouver le meilleur compromis entre rigueur et flexibilité, les contrôles d'utilisabilité seront réalisés de préférence par un SC réputé, qui pourra procéder à des vérifications et validations (V & V) très complètes, sur de longues périodes d'utilisation et renforcées par la collaboration des RC. À défaut de certificat, c'est le RC lui-même qui fera des vérifications plus sommaires et plus automatisées (sans preuves par exemple). La certification des contrôles vise donc deux objectifs, (i) de gain de

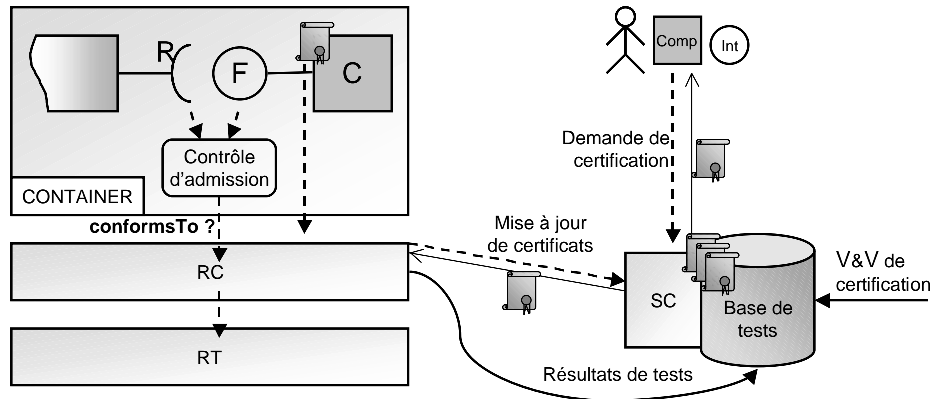


Figure 1. *Principe de fonctionnement du RC et du SC.*

temps à l'admission, par des preuves ou des tests anticipés (donc avec tout le temps nécessaire), (ii) d'obtention de niveaux de confiance et de qualité plus élevés. On se place ici dans le domaine de certification d'aspects critiques de l'informatique, comme la sécurité, le code mobile, etc. où la réputation du SC joue un rôle important dans la sûreté de fonctionnement. Examinons maintenant les contrôles à effectuer, d'abord de manière simple, puis certifiée.

4.2.1. *Vérifications de cohérence et de correction*

Une *vérification simple de cohérence* d'une interface (requise ou fournie) par un SC ou un RC nécessite au moins :

– l'existence d'un type unique dans le RT associé au SC ou au RC de contrôle,

– le test des assertions par au moins un composant connecté à l'interface fournie,

– l'absence d'erreur de hiérarchie (pre, post et inv) au sens de Findler et Felleisen [FIN 01].

Une *vérification simple de correction* de l'implémentation d'une interface par un composant nécessite au moins :

– l'existence d'un identifiant unique du composant dans son espace de noms,

– la cohérence simple de l'interface fournie, comme définie précédemment,

– les tests usuels de correction d'une implémentation, avec l'armement des assertions pré, post, inv les plus efficaces pour détecter des erreurs et l'emploi d'un jeu de tests pour générer des situations à contrôler. La sélection des contrôles d'assertion peut s'inspirer de nos travaux antérieurs sur l'évaluation efficace d'assertions quantifiées [COL 99] en s'appuyant sur des catégories d'assertions. La fourniture des jeux de tests peut s'effectuer selon les mêmes principes que les classes auto-testables [DEV 99b].

D'après les définitions d'utilisabilité données précédemment, les vérifications simples de cohérence et de correction permettent d'obtenir des *vérifications simples d'utilisabilité*.

4.2.2. *Certifications de cohérence et de correction*

Les certificats sont délivrés par un SC et peuvent être placés dans les interfaces ou dans les composants eux-mêmes, comme des filigranes numériques (*digital watermarks*). Bien que les RC n'aient pas vocation à certifier, ils peuvent transmettre des résultats de tests à leur SC. Cela permet de supporter plusieurs formes de certification, soit par des tests réalisés par une entité centrale, soit par une approche coopérative et distribuée aux points d'utilisation, similaire aux approches de développement en logiciel libre ou aux bêta-tests commerciaux. Il est aussi nécessaire que le SC fournisse des certificats de différents niveaux décroissants : *prouvé*, *validé*, *vérifié*.

Un *certificat de cohérence* d'une interface nécessite au moins une vérification simple de cohérence, complétée par les V & V suivantes :

– l'assurance qualité par des relectures expertes de la couverture fonctionnelle, des spécifications informelles en langage naturel, de la facilité d'emploi, de la place de son type dans la hiérarchie du RT, etc.

– des tests d'assertions sur une implémentation de l'interface et des tests sur la hiérarchie supérieure de cette interface (substituabilité des pré et postconditions).

Un *certificat de correction* d'une implémentation nécessite au moins une vérification simple de correction, complétée par les V & V suivantes :

– l'interface fournie est certifiée cohérente, comme décrit ci-avant,

– les tests de correction de l'implémentation par le composant se font avec des jeux de tests étendus et l'armement de toutes les assertions.

D'après les définitions d'utilisabilité données précédemment, les certifications de cohérence et de correction permettent d'obtenir des *certificats d'utilisabilité*.

4.3. ***Contrôles de compatibilité de connexion***

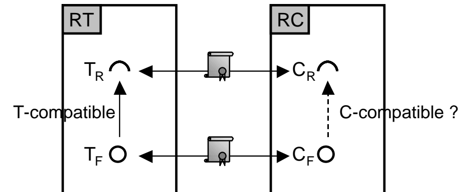
Pour savoir si une interface fournie `F`, implémentée par un composant `C` est compatible avec une interface requise `R`, nous supposons que le RC donne une fonction de signature `int conformsTo(Interface F, Component C, Interface R)` dans le même esprit que celles déjà utilisées dans les systèmes de type ou dans des architectures contractuelles (JAMUS [SOM 02]). Cette fonction rend un entier qui indique le degré de compatibilité ou d'incompatibilité, déterminé à partir des cas explicités ci-après.[2]

2. Nous simplifions volontairement le résultat de cette fonction, qui pourrait être un rapport de conformité contenant diverses informations sur les certificats ou tests effectués, leur niveau de confiance, etc. pour alimenter une éventuelle négociation.

Les règles de compatibilité sont simplifiées par l'emploi de la T-compatibilité et des certificats. La fonction `conformsTo` commence par vérifier que les interfaces R et F sont connues du RC, donc aussi du RT, et que l'interface F est T-compatible avec l'interface R. Si cette condition est fausse, l'incompatibilité est déclarée, de manière certaine si les deux interfaces sont connues du RC. Comme le SC est amené à maintenir une hiérarchie entièrement cohérente d'interfaces contractualisées, toute interface certifiée est C-compatible et T-compatible avec toutes celles qui sont au dessus dans la hiérarchie des types, lesquelles sont aussi forcément certifiées. Cela supprime l'étude d'un cas de compatibilité.[3] Les règles et procédures de calcul de la C-compatibilité procèdent en deux étapes : la détermination d'une C-compatibilité *présumée* par une analyse des autres cas, puis la validation de l'adéquation des niveaux de confiance attendus par la plate-forme avec ceux fournis par les éventuels certificats.

Cas n° 1 : les interfaces R et F sont certifiées utilisables par le même SC

La C-compatibilité présumée résulte simplement ici de la T-compatibilité, par l'isomorphisme entre le RT et le RC qui est garanti par les certificats.



Cas n° 2 : l'interface R est certifiée utilisable, l'interface F ne l'est pas.

Il faut d'abord s'assurer que F est utilisable *pour R*. La non certification d'utilisabilité de l'interface F peut résulter de l'absence de certification de sa cohérence et/ou de la correction de son implémentation. Mais on ne peut avoir une interface F correcte et non cohérente, d'après la définition de la correction qui exige la cohérence.

Si F n'est pas certifiée cohérente, le RC doit faire des vérifications simples (comme définies en 4.2.2) de cohérence et de correction de son implémentation et vérifier que les contrats de F sont compatibles avec ceux de R. Si F est certifiée cohérente, seules les vérifications de correction sont à effectuer, car la vérification de compatibilité des contrats est déjà garantie. Aussi, seules les assertions définies dans F doivent être évaluées. Si F et R sont égales, on doit juste évaluer les assertions des interfaces, cela suffit à vérifier que le composant de F est C-compatible avec R et qu'il l'implémente correctement. A défaut de jeu de tests, on peut utiliser le composant qui fournit R.

Cas n° 3 : les interfaces sont certifiées utilisables, mais par deux SC différents.

Dans ce cas, aucun des deux SC ne peut garantir à lui seul la C-compatibilité. On se ramène au cas n° 2 en renonçant à la certification de l'interface fournie F.

3. Celui d'une interface F certifiée cohérente et T-compatible avec une interface R qui ne serait pas certifiée.

Cas n° 4 : Aucune des deux interfaces n'est certifiée utilisable

Dans ce cas défavorable, c'est le RC qui a la charge de vérifier simplement que R et F sont utilisables et que F est C-compatible avec R. Pour cela, on connecte l'interface F à R et on arme toutes les catégories d'assertions (préconditions, postconditions et invariants) de F et de R. Pour ceux qui sont certifiés, seules les préconditions sont nécessaires ; pour les autres, toutes leurs catégories d'assertions doivent être armées.

5. Négociation des contrats comportementaux

En cas d'incompatibilité d'une interface fournie avec une interface requise, nous préconisons une négociation qui peut conduire au refus d'admission, au relâchement des contrats ou de leur degré de confiance, à l'appel à d'autres composants alternatifs, etc. La négociation qui mêle des aspects fonctionnels et non fonctionnels est un problème ouvert qui prendra de plus en plus d'importance avec le développement du génie logiciel basé sur les composants et qui devra être appréhendé de manière globale, en considérant toutes sortes de contraintes. Dans cette section, nous ne faisons qu'amorcer la discussion.

L'idée de négocier, à propos de fiabilité, peut choquer puisqu'elle conduit à accepter de ne pas détecter certaines catégories d'erreurs potentielles. Mais dans une approche qui repose en partie sur des tests, on ne peut jamais garantir le risque zéro. Le besoin de négocier les contrats au moment de l'admission d'un composant vient principalement des contraintes de temps et plus généralement des contraintes non fonctionnelles qui perturbent les contraintes fonctionnelles. Les certifications fonctionnelles se feront évidemment sans négociation, les plus complètes possible, parce qu'on aura les moyens et le temps nécessaire. De manière idéale, les certifications pourront utiliser des systèmes de preuves. Pour les composants non certifiés, il faudra faire les tests les plus importants dans un délai imposé. Compte tenu du couplage faible des composants, le temps disponible pourra être de l'ordre de la seconde, acceptable par rapport à la durée des nombreux autres traitements effectués par une plate-forme avant de lancer une application. Avec la vitesse des processeurs actuels, cela permet déjà d'évaluer des millions d'assertions...

Il est également souhaitable d'encadrer la liberté de négociation par une *politique de contractualisation* définie par des règles de génie logiciel, qui seront paramétrées par la criticité de l'assemblage, les niveaux de confiance ou d'incompatibilité, etc. Lors des contrôles d'admission de composants non certifiés, voire pendant l'exécution de l'application, différents cas de négociation sont envisageables :

– l'acceptation d'une compatibilité structurelle des types [CAR 88] lorsque la T-compatibilité des types identifiés par leur nom échoue,

– la fourniture de paramètres des tests à effectuer, comme les niveaux de confiance ou de compatibilité. Ainsi, certaines clauses d'assertions étiquetées moins essentielles ne seront pas évaluées, et l'évaluation de quantifications procédera par échantillonnage comme dans [COL 99].

– l'affaiblissement du niveau de compatibilité recherché, en renonçant à certains services non essentiels, en relâchant des conditions requises.

– l'annulation temporaire de contrats fonctionnels pour assurer une connexion dans des contextes acceptables : développement, essais de déploiement, accord préalable de l'utilisateur...

Par ailleurs, certains cas préssentis comme négociables peuvent être traités automatiquement. C'est le cas, par exemple, d'une interface fournie de même nom que l'interface requise R, mais compatible avec R en fournissant des postconditions supplémentaires dans ses méthodes [BEU 99].

6. Conclusion

Dans cet article, nous avons proposé un modèle adapté aux plates-formes actuelles de composants pour vérifier la compatibilité des interfaces fournies avec les interfaces requises lors de la phase d'admission. Les interfaces sont spécifiées de manière contractuelle par des préconditions, postconditions et invariants décrits par des assertions expressives et exécutables. La compatibilité est déterminée par un référentiel de comportements qui complète le référentiel de types et utilise un serveur de certification. Ce modèle autorise donc deux approches complémentaires, de V & V anticipées (preuves, tests poussées, etc.) par le serveur de certification d'une part, et de tests à chaud des comportements d'autre part. Dans les cas d'incompatibilité, nous montrons les possibilités de négociation.

A notre connaissance, depuis l'ouvrage de synthèse de Szyperski [SZY 98] et la classification des contrats proposée par Beugnard *et al.* [BEU 99], la problématique des contrats fonctionnels pour les composants n'a guère été développée. Bien qu'aucune réalisation ne valide encore notre modèle, celui-ci repose sur des solutions existantes, déjà validées dans d'autres contextes. Un contrat de recherche externe avec France Télécom R&D qui vient de commencer nous permettra d'appliquer ces idées à la plate-forme Fractal [BRU 02]. Nous devrons aussi formaliser les concepts proposés, affiner le niveau de confiance, ainsi que les processus de certification et de négociation. En considérant à terme des contrats d'interaction et des aspects non fonctionnels, ces recherches devraient fournir les éléments pour un raisonnement compositionnel basé sur la contractualisation.

7. Bibliographie

[BAC 00] BACHMAN F.,BASS L., BUHMAN C., COMELLA-DORDA S., LONG F., ROBERT J., SEACORD R., WALLNAU K., « Technical Concepts of Component-Based Software Engineering », rapport n° CMU/SEI-2000-TR-008, Mai 2000, Carnegie Mellon Software Engineering Institute, Volume 2.

[BEU 99] BEUGNARD A., JÉZÉQUEL J.-M., PLOUZEAU N., WATKINS D., « Making Components Contract Aware », *IEEE Computer*, vol. 32, n° 7, 1999.

[BRU 02] BRUNETON E., COUPAYE T., STEFANI J.-B., « Recursive and Dynamic Software Composition with Sharing », *ECOOP Workshop on Component-Oriented Programming*, Malaga, Spain, juin 2002.

[CAR 88] CARDELLI L., « Structural Subtyping and the Notion of Power Type », *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, 1988, p. 70–79.

[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.

[DEV 99a] DEVEAUX D., FLEURQUIN R., FRISON P., JÉZÉQUEL J.-M., TRAON Y. L., « Composants objets fiables : une approche pragmatique. », *L'objet*, vol. 5, n° 3-4, 1999, p. 469-494, Hermes Science.

[DEV 99b] DEVEAUX D., JÉZÉQUEL J.-M., « Des classes autotestables », *LMO'1999 (Langages et Modèles à Objets)*, Hermes Science Publications, Janv. 1999, p. 203-215.

[FIN 01] FINDLER R. B., FELLEISEN M., « Contract Soundness for Object-Oriented Languages », *Proceedings of OOPSLA'2001*, 2001.

[HOA 71] HOARE C., « Proof of Program : FIND », *Communications of the ACM*, vol. 14, n° 1, janvier1971, p. 39-45.

[HOL 92] HOLLAND I. M., « Specifying reusable Components using Contracts », LEHRMANN MADSEN O., Ed., *ECOOP'92*, vol. 615 de *LNCS*, 1992, p. 287-308.

[KRA 98] KRAMER R., « iContract - The Java Design by Contract Tool », SINGH M., MEYER B., GIL J., MITCHELL R., Eds., *Tools 26, USA'98*, IEEE Computer Society Press, 1998.

[MCK 93] MCKIM J. C., MONDOU D. A., « Class Interface Design : Designing for Correctness », *The Journal of Systems and Software*, vol. 23, n° 2, novembre1993, p. 85-94.

[MEY 92] MEYER B., « Applying "Design by contract" », *IEEE Computer*, vol. 25, n° 10, octobre1992, p. 40-51.

[OBJ 97] OBJECT MANAGEMENT GROUP I., « Object Constraint Language Specification », version 1.1, rapport n° ad/97-08-08, septembre1997.

[SOM 02] SOMMER N. L., GUIDEC F., « JAMUS, une plate-forme sécurisée pour le code mobile », *LMO'2002 (Langages et Modèles à Objets)*, Hermes Science Publications, *L'objet*volume 8, numéro 1-2/2002, janvier 2002, p. 203-215.

[SZY 98] SZYPERSKI C., *Component Software — Beyond Object-Oriented Programming*, Addison-Wesley Publishing Co. (Reading, MA), 1998.