

PROJET RNTL ARCAD*

D1.3 Document d'architecture

Coordonnateur : Thomas Ledoux

Auteurs: Françoise Baude, Eric Bruneton, Thierry Coupaye,
Pierre-Charles David, Thomas Ledoux, Audrey Ocello, Michel Riveill

16 juillet 2004

1 Introduction

L'objectif principal du projet ARCAD est de proposer une infrastructure extensible pour le développement et le déploiement d'applications réparties construites par assemblage de composants. Cette infrastructure doit notamment permettre l'ajout dynamique de propriétés non-fonctionnelles aux composants, la reconfiguration dynamique d'une architecture à base de composants.

Comme il a été clairement exposé dans le livrable D5.3, l'approche développée initialement consistait à considérer l'extension de modèles de composants "industriels" "standards" tels que COM, CCM ou EJB. Aussi, la première phase du projet a connu un travail autour de l'extension du modèle de composants industriel EJB à travers son implantation JOnAS. Cette phase du projet a certes démontré la possibilité de réaliser certaines extensions mais a surtout souligné les limites du modèle EJB (peu structuré, relativement fermé) pour construire une plate-forme extensible supportant la programmation par composants.

La seconde phase du projet s'est alors concentrée autour de la proposition du modèle de composants Fractal et de Julia (sa mise en oeuvre de référence pour le langage Java) distribués aujourd'hui dans le cadre du consortium ObjectWeb. En effet, Fractal fournit un cadre architectural "ouvert" ayant pour vocation de subsumer les modèles industriels comme EJB ou CCM; Julia fournit un support d'exécution extensible, adaptable sur lequel il est possible d'ajouter aisément des propriétés non-fonctionnelles.

*Le projet ARCAD (Architecture Extensible pour Composants Adaptables) a été labellisé en décembre 2000. Les différents partenaires sont France Télécom R&D (équipe ASR - Thierry.Coupaye@francetelecom.com), INRIA (projet Oasis - Denis.Caromel@inria.fr, projet Sardes - Daniel.Hagimont@inria.fr), Ecole des Mines de Nantes (équipe OBASCO - Thomas.Ledoux@emn.fr), le laboratoire I3S commun à l'Université de Nice - Sophia Antipolis et au CNRS (projet Rainbow - Anne-Marie.Pinna@unice.fr). Le coordonnateur du projet est Michel.Riveill@unice.fr

Ainsi, la plate-forme ARCAD consiste en un noyau Fractal/Julia sur lequel vont venir se greffer les différentes contributions des partenaires pour construire un intergiciel adaptable permettant le développement et le déploiement d'applications réparties construites par assemblage de composants.

Ce document est organisé de la façon suivante. La première section présente le modèle de composants Fractal. La deuxième section propose un certain nombre d'extensions développées dans le cadre du projet pour enrichir le noyau Fractal/Julia et proposer un environnement d'exécution *middleware* pour composants.

2 Le modèle de composants Fractal

Cette section décrit tout d'abord le modèle de composants Fractal, ou pour être plus exact, les modèles de traitements et de programmation Fractal ; puis met en valeur les caractéristiques de Fractal qui en font un très bon support pour l'adaptation.

2.1 Modèle de traitements abstrait

Un *composant* est la composition d'un *contrôleur* et d'un *contenu*¹.

Le contenu d'un composant est composé (d'un nombre fini) d'autres composants (appelés sous-composants) qui sont sous le contrôle du contrôleur du composant englobant. Le modèle est complètement récursif et autorise l'imbrication des composants à un niveau arbitraire.

Un composant peut interagir avec son environnement par l'échange de signaux. Un *signal* consiste en un nombre fini d'arguments qui peuvent prendre trois formes : i) un *nom*, ii) une *valeur* ou iii) un composant (incluant l'état complet de son contrôleur et de son contenu). Les noms sont des symboles utilisés pour désigner des entités du modèle. Le modèle abstrait ne place aucune contrainte sur la forme ou la structure des noms. Les noms doivent uniquement être compris comme étant relatifs à un contexte de nommage particulier, i.e. les noms réalisent leur fonction de référencement de manière non ambiguë uniquement dans un contexte donné. Les contextes de nommage peuvent être imbriqués et se recouvrir - autorisant les noms à être valides dans plusieurs contextes. Les valeurs sont typiquement construites à partir de types primitifs et de constructeurs de types de données. Le comportement d'un composant est défini par un ensemble de transitions dans lequel chaque transition spécifie le composant original, un ensemble fini de signaux entrants, un ensemble fini de signaux sortants et un ensemble fini de composants résultants - appelé le *résidu*. Le modèle abstrait n'impose aucune contrainte sur la forme du résidu ou sur la nature des

1. Composants, contrôleurs et contenus sont des entités abstraites ; le modèle abstrait n'impose aucune contrainte sur la réalisation de ces entités - en particulier, il ne spécifie absolument pas que contrôleur et contenu seront réalisés par des entités logicielles distinctes (par exemple deux objets au sens de la programmation orientée objet...) - ceci est spécifié par les modèles d'ingénierie (cf Section suivante).

signaux sortants. En particulier, le composant original peut apparaître comme un argument d'un signal sortant, permettant au composant de migrer spontanément vers un autre composant. Ce mécanisme peut être utilisé, par exemple, pour passer les composants, ce qui peut être utilisé pour la persistance ou la mobilité des composants. Lorsqu'un ou plusieurs composants apparaissent dans le résidu d'une transition en plus du composant originel, celui-ci peut être considéré comme une usine à composants. Si le composant originel apparaît dans le contenu d'un composant du résidu d'une transition, le composant originel peut être compris comme ayant créé son propre méta-composant. Ce mécanisme peut être utilisé pour la création paresseuse de meta-niveaux dans des infrastructures complètement dynamiques.

Le contrôleur d'un composant incarne le comportement de contrôle associé à ce composant. Le contrôleur d'un composant peut typiquement intercepter les signaux à destination de où en provenance de ce composant ou bien superposer un comportement de contrôle au comportement des sous-composants du composant considéré incluant la désactivation, la sauvegarde de la structure et de l'état du composant (checkpointing) et la réactivation des activités du composant. Chaque contrôleur doit être compris comme réalisant la sémantique particulière et arbitraire de la composition de ses sous-composants. Les capacités de contrôle ne sont pas limitées par le modèle. Elles peuvent en particulier être basées sur l'interception à l'instar de la plupart des modèles de composants industriels (et de la plupart des approches réflexives ou de la programmation par aspects) ou bien encore être nulles.

Un composant peut apparaître dans le contenu de (i.e. être *partagé* par) plusieurs composants englobants distincts. Un composant partagé par plusieurs composants est sous le contrôle (des contrôleurs) de ces composants. La sémantique exacte de la composition est déterminée par un composant englobant tous les composants impliqués (i.e. le composant partagé et les composants qui le partagent).

Les caractéristiques centrales et/ou différentiatrices du modèle abstrait de Fractal sont les suivantes:

- récursion à niveau arbitraire;
- partage de composants avec sémantique arbitraire fixée par composants englobants;
- réflexion (introspection et intercession): un composant (et par extension un système puisque qu'un système est une configuration particulière de composants) fournit une représentation explicite et causalement connectée de sa structure; et intercession/contrôle complètement arbitraire à la fois sur la structure et le comportement des composants;
- communications arbitraires par signaux qui recouvrent tous types de communication/interaction: événements/messages, flux multimedia, invocations d'opérations synchrones et asynchrones, communications locales, distribuées, sécurisées, transactionnelles, etc.

2.2 Modèle de traitements concret, modèles de programmation, modèles d'ingénierie

Le modèle présenté dans la section précédente n'est en réalité qu'une partie de ce que le vocable "Fractal" désigne. En effet, Fractal doit être compris comme une hiérarchie, ou une famille de modèles plutôt que comme un modèle unique de composants comme EJB, CCM ou Avalon (Apache/Jakarta) - la raison principale étant que Fractal se veut un cadre architectural très général ayant comme vocation de subsumer ces modèles. La racine de cette hiérarchie (ou l'ancêtre de la famille) est justement le modèle de traitements abstrait présenté précédemment. A l'exclusion de sa racine unique, la hiérarchie n'est pas immuable. Elle est au contraire extensible. On peut donc uniquement en donner une image à un instant donné. La hiérarchie pourra comprendre différents *modèles de traitements*, différents *modèles de programmation* et différents *modèles d'ingénierie*. Nous décrivons dans la suite uniquement le modèle de traitements concret et le modèle de programmation associés au framework Julia qui peut être utilisé pour la programmation de composants Fractal en Java.

Nous avons vu que le modèle abstrait est très minimal en ce sens que les concepts centraux utilisés sont en nombre très limité: *composant*, *contrôleur*, *contenu*, *signal*, *nom* et *valeur*. Le modèle concret *raffine* le modèle abstrait en lui imposant des *contraintes* quant à la forme des composants, des noms et des signaux manipulés. La contrainte la plus importante concerne la forme des signaux échangés au cours des interactions entre composants: ils peuvent être complètement arbitraires dans le modèle abstrait tandis qu'ils peuvent prendre uniquement des formes bien définies dans le modèle concret. Ces formes correspondent à des interactions de type invocation d'opérations synchrones avec ou sans retour sur des points d'interaction identifiés aux composants: les *interfaces*. Le modèle concret interdit en particulier l'utilisation de composants comme signaux, i.e., seuls des noms de composants peuvent être utilisés comme paramètres d'appel ou de retour. Les interfaces sont les seuls points d'accès aux composants. Le modèle concret introduit également le concept de *liaison* (binding) comme support des interactions. Une liaison est un lien orienté, correspondant à un canal de communication, entre une interface *client* et une interface *serveur*. Le modèle concret introduit donc un typage des composants et interfaces; ainsi qu'un mécanisme d'instantiation à base de *patron* (template), d'*usines à patrons* (template factories) et d'*usines à composants* (component factories). Le modèle concret reste lui aussi minimal. Les concepts utilisés sont: *composant*, *contrôleur*, *contenu*, *nom* et *valeur* qui ont le même sens que dans le modèle abstrait auxquels s'ajoutent *interface* et *liaison*; et enfin *type*, *patron*, *usine* (*factory/template*).

Un *modèle de programmation* Fractal est une projection d'un modèle de traitement dans un langage de programmation donné. Un modèle de programmation *supporte* un modèle de traitements. Il se présente généralement sous la forme d'une API telle l'API Java associée au modèle de traitement concret.

Un modèle d'ingénierie décrit les éléments de la *mise en oeuvre* d'un modèle de traitement supportant un modèle de programmation donné. Les différents

modèles d'ingénierie pouvant être utilisés sont en réalité associés à Fractal mais ne font pas partie stricto sensu de Fractal car ce sont des implantations. Julia, par exemple, est la mise en oeuvre du modèle concret Fractal supportant le modèle de programmation Fractal pour le langage Java. Les entités décrites par le modèle d'ingénierie Julia comprennent les *objets de contrôle*, les *intercepteurs*, les *mixins*, les *générateurs de code*, etc. En effet, Julia est principalement un framework dédié à la programmation des intercepteurs et plus généralement des contrôleurs de composants Fractal.

2.3 Fractal un modèle permettant l'adaptation

Le modèle Fractal supporte intrinsèquement deux types d'adaptation différents: une adaptation structurelle, qui consiste à choisir, statiquement ou dynamiquement, un assemblage de composants adapté à une situation donnée, et une adaptation "non fonctionnelle" grâce aux capacités d'interception des contrôleurs.

Le modèle concret Fractal définit un ensemble d'interfaces "minimal" offrant des primitives de base pour configurer et reconfigurer dynamiquement les composants. Ces interfaces concernent le contrôle des attributs, des liaisons et des sous composants d'un composant, ainsi que le contrôle de son cycle de vie. Ces primitives peuvent être utilisées "statiquement" pour instancier une nouvelle configuration, soit directement soit via un ADL. Elles peuvent également être utilisées pour reconfigurer dynamiquement une configuration existante, une reconfiguration ayant lieu en trois étapes: suspension de l'activité du composant (en utilisant son contrôleur de cycle vie), reconfiguration proprement dite (c'est à dire ajout, retrait ou modification de liaisons, de sous composants, ou d'attributs), et enfin reprise de l'activité du composant. Ces primitives peuvent enfin être utilisées pour les auto-adaptations dynamiques qui peuvent se faire par des changements de structure de l'application. Par exemple, une application de vidéo à la demande peut s'adapter aux variations du débit disponible en s'auto-reconfigurant, et notamment en insérant ou en retirant des composants de filtrage du flux vidéo, afin d'adapter son débit à celui qui est disponible.

Le modèle Fractal indique que les contrôleurs *peuvent* intercepter les appels entrants et sortants d'un composant, au niveau de ses interfaces, sans plus de précisions. Ceci autorise donc tout type d'interception. Par exemple un appel entrant peut ne pas être intercepté du tout, ou bien être réifié partiellement (sans réifier ses arguments par exemple) ou complètement (on peut alors manipuler le nom de la méthode, ainsi que le nombre et le type des paramètres comme on le souhaite, comme dans les protocoles à méta-objets). Cette interception peut se faire par un objet séparé du contenu du composant, ou bien par du code mélangé au code fonctionnel du composant (la séparation entre contrôleur et contenu au niveau du modèle n'interdit pas de mélanger ces fonctions au niveau de la mise en oeuvre concrète du modèle). Le traitement des appels réifiés peut lui même être quelconque: ils peuvent par exemple être traités de façon synchrone (le niveau de base étant bloqué tant que le niveau méta n'a pas terminé de traiter l'appel réifié) ou asynchrone. Ils peuvent être traités par un seul objet, par une

chaîne de méta objets... Autrement dit le modèle Fractal autorise toutes les techniques réflexives et orienté aspect qui sont connues pour être utiles pour adapter les propriétés non fonctionnelles d'un système, et en particulier d'un composant ou d'une configuration de composant.

Julia, qui est l'implantation de référence de Fractal, offre toutes les primitives de (re)configurations définies dans le modèle Fractal. Julia offre également un canevas pour définir des objets d'interposition variés, pour intercepter les appels au niveau des contrôleurs de composants. Deux générateurs d'objets d'interposition simples sont proposés : un qui réifie seulement le nom des méthodes, l'autre qui réifie aussi les arguments. L'utilisateur peut également définir ses propres générateurs d'objet d'interposition.

3 Utilisation de Fractal pour construire un intergiciel adaptable

Comme nous venons de le voir, Fractal est un modèle intrinsèquement adaptable et Julia en est son implantation de référence. L'ensemble des sous-projets présentés ci-dessous ont pour but d'étendre le noyau Fractal/Julia. L'intégration de ces services est un premier pas pour la construction d'un intergiciel adaptable.

3.1 Fractal RMI (France Telecom R&D)

Fractal RMI est un intergiciel, réalisé à base de composants Fractal, qui permet des communications synchrones distantes entre composants Fractal. Fractal RMI offre plus de transparence que Java RMI, d'une part parce que `Remote` et `RemoteException` ne sont pas obligatoires pour les interfaces accessibles à distance, et d'autre part parce que les talons et squelettes sont générés dynamiquement grâce à ASM (il n'y a donc plus besoin de `rmic`). Par contre, Fractal RMI n'utilise pas le même protocole (JRMP) que Java RMI, et ce parce qu'il est basé sur une "fractalisation" partielle de Jonathan, qui ne propose pas d'implémentation de ce protocole [DTHS99].

Fractal RMI est constitué d'une dizaine de composants Fractal regroupés dans un composant composite qui offre l'interface `NamingContext` (cf. Figure 1). Ce composant composite correspond à ce qu'on appelle l'ORB en CORBA, il y en a une instance par site (ou par machine virtuelle). L'interface `NamingContext` permet de gérer l'espace de nommage réparti utilisé par Fractal RMI : création d'un nom réparti pour un objet local, et création d'une chaîne de liaison vers un objet désigné par son nom réparti.

Parmi les composants internes de Fractal RMI, on trouve des composants implantant des protocoles (et notamment le protocole d'invocation de méthode à distance), un composant pour gérer l'empaquetage et le dépaquetage des messages, un composant pour créer des talons et squelettes, ainsi que des composants de gestion de ressources (cache de tampons mémoire pour lire et écrire les messages, pool de threads, ...). La moitié des composants sont des composants de

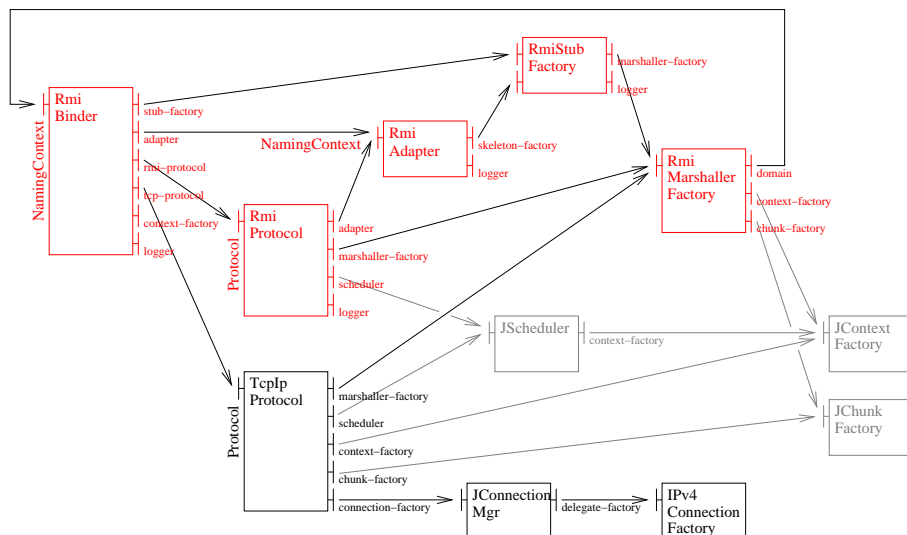


FIG. 1 – Architecture de l'objet de liaison Fractal RMI

Jonathan qui ont été réutilisés tels quels (après "fractalisation"), l'autre moitié est constituée de composants spécifiques à la "personnalité" Fractal RMI (dont le code est fortement inspiré du code des composants de Jeremie, la personnalité "RMI" de Jonathan).

Comme dans Jonathan, l'organisation de Fractal RMI en composants permet une adaptation de l'ORB selon les besoins. On peut par exemple remplacer un composant de gestion de ressources par un autre, voire même un composant implantant un protocole par un composant implantant un autre protocole. En pratique aucune expérimentation concrète n'a été faite pour vérifier cela sur Fractal RMI lui même, mais l'existence de plusieurs personnalités de Jonathan (David, Jeremie et Fractal RMI) montre que cette adaptabilité est bien réelle. Il s'agit toutefois uniquement d'adaptation statique : aucune expérience d'adaptation dynamique n'a été réalisée.

3.2 Communication de groupe, asynchrone avec futur (Oasis)

Comme on l'a vu plus haut, le modèle Fractal ne spécifie pas la façon d'implanter les interactions entre composants ; le modèle de programmation Fractal existe actuellement sous la forme d'une API Java ; le modèle d'ingénierie Julia permet quant à lui d'utiliser cette API dans un cadre assez restreint : tous les composants sont instanciés dans la même JVM, avec synchronisme des demandes et exécutions de services invoqués. Basé sur ProActive, la contribution décrite ci-dessous apporte une réponse à ces limites.

ProActive est un intergiciel conçu autour d'un protocole à méta objets, réalisé entièrement en Java, proposant une API de programmation orienté objet,

parallèle, concurrente, distribuée entre objets actifs. ProActive a pour spécificité: un mécanisme d'appel asynchrone possiblement distant vers un objet actif, avec synchronisation automatique sur le résultat (attente par nécessité); une extension de ce mécanisme vers des groupes, dynamiquement formés, d'objets actifs du même type (communication de groupe typé). Par ailleurs, tout objet actif est migrable d'une JVM à une autre, avec maintien des références permettant de communiquer avec lui.

Une nouvelle mise en œuvre de l'API Fractal intégrée à ProActive a été proposée (où tout composant est constitué d'au moins un objet actif). De la sorte, on dispose de composants conformes au modèle Fractal, dont les interactions sont réifiées par le protocole à méta objets de ProActive et ainsi, sont effectuées de manière distribuée, asynchrone. Les composants sont également migrables. Ils peuvent être regroupés au sein de groupes de composants du même type (prenant au choix soit la forme d'un composant composite qualifié alors de *composant parallèle*, soit la forme d'un groupe d'objets/composants ProActive – utile pour répliquer facilement un même composant dans une optique de tolérance aux pannes par exemple), et une interaction entre un composant et un tel groupe de composants se décompose automatiquement en une interaction parallèle avec chaque composant inclus.

D'un certain point de vue, cette réification des interactions par ProActive procure une nouvelle forme d'adaptation (qui n'est pas structurelle et qui n'exploite pas non plus la facilité d'adaptation non fonctionnelle basée sur les contrôleurs Fractal, cf section 2.3). Un autre point de vue sur le travail réalisé est le suivant: permettre de développer des applications, en particulier visant le domaine du *grid computing*, qui ont besoin d'exploiter les caractéristiques du modèle ProActive (mobilité des activités, asynchronisme avec futur, sécurité des communications, etc.), mais qui sont structurellement adaptables. Une telle faculté serait évidemment beaucoup plus difficile à réaliser si on ne disposait pas du paradigme de programmation par composants, mais uniquement du paradigme de programmation par objets.

Une note au sujet de la manière dont l'intergiciel ProActive est conçu et mis en œuvre: contrairement au sous-projet Fractal RMI, on n'utilise pas de composants Fractal pour la mise en œuvre. Cependant, au dessus du protocole à méta objets ProActive, les différentes caractéristiques comme la distribution, la mobilité, la sécurité, la gestion des groupes, les composants, sont proprement et au maximum isolées les unes des autres. Il serait donc envisageable d'en venir à une implantation à base de composants Fractal en vue d'adapter l'intergiciel lui-même au contexte d'exécution et aux besoins réels des applications supportées.

3.3 Aspect d'auto-adaptation (EMN/Obasco)

Cette section décrit une extension du modèle de composants Fractal pour créer des composants logiciels adaptatifs au contexte.

3.3.1 Introduction

Motivations Avec la prolifération des plates-formes d'exécution aux caractéristiques techniques parfois très différentes (du téléphone portable aux grilles de calcul) et la démocratisation des réseaux - notamment sans fil - dans lesquels la disponibilité des ressources (CPU, mémoire, batterie, ressources logicielles, ...) peut varier aléatoirement en cours d'exécution, nous sommes plus que jamais confrontés à la dynamique des environnements dans lesquels nos applications doivent fonctionner. Devant cet état de fait, nous proposons une approche générale pour la construction d'applications auto-adaptables devant s'ajuster automatiquement aux évolutions de leur environnement dans le but de continuer à fonctionner correctement.

Profession de foi Le besoin de construire des applications qui s'adaptent au contexte n'est pas nouveau. Cependant, cela est généralement fait de façon ad hoc, en tentant d'anticiper au moment du développement les futures conditions d'exécution de l'application, et en intégrant directement dans celle-ci le code nécessaire pour détecter ces évolutions et appliquer les reconfigurations appropriées. D'un point de vue génie logiciel, cette orientation n'est pas satisfaisante puisque l'observation du contexte et la décision de reconfiguration se trouvent mêlées au code métier, ce qui complexifie le développement, limite les possibilités d'adaptation et réduit surtout la réutilisation de la partie métier. L'adaptation est une "préoccupation transverse", qui *cross-cut* le code métier d'une application, et qui doit être traité séparément du reste. Cette approche inspirée par les concepts de séparation des préoccupations (*separation of concerns*) [HL95] et programmation par aspects [KLM⁺97] nous invite à repenser le cycle de vie des applications et à considérer l'adaptation aux évolutions du contexte comme un "aspect".

Ainsi, dans un premier temps, le développeur d'une application va se concentrer sur sa problématique métier. Puis, une fois connu l'environnement de déploiement et/ou d'exécution, il est possible de définir séparément la logique d'adaptation et de développer l'aspect correspondant. Ce dernier sera finalement tissé (*weavé*) au déploiement et/ou à l'exécution au code métier, produisant ainsi une application auto-adaptable.

Approche choisie Concrètement, notre objectif est de proposer aux programmeurs d'application un modèle de développement, ainsi qu'un framework et des outils associés, permettant de :

- mettre en oeuvre cette séparation entre code métier et logique d'adaptation ;
- exprimer le code métier dans un langage, une architecture potentiellement adaptable (c.à.d. flexible et structuré à la fois), et autorisant aussi la reconfiguration dynamique ;
- exprimer la logique d'adaptation dans un langage d'aspect, autorisant un certain nombre de contrôles sur les adaptations réalisables ;

- ré-assembler le code métier et la logique d’adaptation pour obtenir finalement une application auto-adaptable ;
- fournir un service de *context-awareness* permettant de représenter le domaine contextuel à superviser et permettant de détecter les événements qui doivent déclencher l’adaptation.

3.3.2 Extension de Fractal

Comme il a été indiqué dans la section 2.3, Fractal est un bon candidat pour développer des applications auto-adaptables et ceci pour plusieurs raisons :

- Fractal est un petit modèle, structuré et réflexif ;
- Fractal offre des primitives de base pour configurer et reconfigurer dynamiquement les composants (paramétrisation, modification des liaisons entre composants et des relations de composition, contrôle du cycle de vie) ;
- Fractal est un modèle *ouvert* et permet à l’utilisateur de définir ses propres contrôleurs.

Politiques d’adaptation L’objectif des politiques d’adaptation est de définir la logique d’adaptation à appliquer aux composants constituant une application. Au cours de son exécution, chaque composant Fractal de l’application peut se voir affecter (ou retirer) une ou plusieurs politiques d’adaptations décrivant quand et comment réaliser une adaptation. Le composant Fractal devient alors sensible au contexte et auto-adaptable.

Concernant la forme de ces politiques, nous nous sommes inspirés du paradigme ECA (Événement – Condition – Action) issu du domaine des bases de données actives [DGG95]. Une politique est donc constituée d’un ensemble de règles réactives, chacune constituée de trois éléments :

1. la spécification d’un type d’événement (éventuellement composite) qui active la règle lorsqu’il se produit ;
2. une condition (expression booléenne) servant de garde ;
3. une action de reconfiguration, éventuellement composée d’une séquence de plusieurs actions primitives parmi celles disponibles.

Le langage d’aspect basé sur ce paradigme et servant à écrire les politiques d’adaptation est actuellement en cours de spécification et sera largement présenté dans la thèse de Pierre-Charles David prévu au second semestre 2004.

Service de context-awareness Pour pouvoir adapter une application aux évolutions de son contexte d’exécution, il faut bien évidemment connaître ce contexte. L’objectif de ce service est donc d’observer l’environnement en question et d’être capable d’y détecter les circonstances qui doivent déclencher une adaptation. Ce service sera notamment en charge de l’acquisition des données brutes, la représentation et la structuration des données, le raisonnement sur ces données. Plusieurs domaines contextuels sont actuellement à l’étude (contextes matériel, logiciel). Un protocole push/pull fera l’interface entre le service de context-awareness et les politiques d’adaptation.

Quelques mots sur les événements Le service de context-awareness peut offrir des informations très complètes et précises concernant l'environnement d'une application. Cependant, ces informations ne correspondent qu'à un instantané de l'état de l'environnement à un moment donné. Pour aller plus loin, nous avons besoin de pouvoir raisonner sur les évolutions de cet environnement au cours du temps. Nous introduisons pour cela la possibilité de spécifier et détecter des événements composites [Cou02], décrivant l'enchaînement d'événements primitifs au cours du temps. À partir des événements primitifs, des opérateurs de composition classiques (séquences, alternative, conjonction) permettent d'exprimer des conditions complexes concernant l'évolution de l'environnement de l'application.

En plus de devoir réagir aux évolutions de son environnement, une application peut aussi être amenée à s'adapter en réaction à des changements internes, comme par exemple la création d'un nouveau composant, l'invocation d'une certaine méthode, etc. Ce second cas est traité de la même façon par notre système, en introduisant un nouveau type d'événements primitifs, dits *endogènes* (par opposition aux événement *exogènes* venant de l'extérieur), correspondant aux événements se produisant à l'intérieur même de l'application (reconfigurations, envois & réception de messages...).

Bien entendu, il est possible lors de la spécification d'un événement composite, de mélanger des événements endogènes et exogènes.

Composants Fractal auto-adaptables L'intégration des politiques d'adaptation dans le modèle de composants Fractal se fait de façon très naturelle grâce à l'extensibilité du modèle Fractal et de son implémentation Julia. Lors de la description de l'architecture d'une application, voire lors de son déploiement, certains des composants constituant cette application peuvent être "marqués" comme devant être rendu auto-adaptables. Cela se traduit concrètement par l'ajout à chacun de ces composants d'une nouvelle interface de contrôle, appelée `adaptation-controller`, similaire aux interfaces standards Fractal comme `content-controller`. Cette nouvelle interface offre une API simple pour ajouter ou retirer dynamiquement des politiques d'adaptation au composant.

3.4 Vers la construction d'un modèle intégrant aspects et contrats pour la sûreté des compositions (Rainbow)

3.4.1 Quelques notions sur les contrats

Les propriétés d'un programme peuvent s'exprimer à l'aide d'assertions que l'on appelle également contrats. Une assertion dans un programme est une expression booléenne qui doit être satisfaite pour que le code associé soit exécuté correctement. Les assertions viennent des travaux de Floyd [R.W67] et de Hoare [C.A69]. Elles servent essentiellement dans la spécification des programmes, des objets et plus récemment dans la spécification des composants. Les programmeurs utilisent des assertions dans la description ou le raffinement de contraintes de typage définies dans les interfaces de classe. Le travail le plus connu dans ce

domaine est probablement celui d'Eiffel [B.92]. Eiffel utilise des assertions dans des pré et post-conditions et dans des invariants de classes pour décrire des contrats entre l'utilisateur et une classe. Les contrats ont été introduits dans les objets par Helm [RID90] de façon à compenser le manque de moyen d'expression pour les relations entre les objets. Ils ont été utilisés pour spécifier des compositions de comportements. Ils permettent une approche orthogonale vis-à-vis de la structuration donnée par la notion de classe.

Selon Clemens Szyperski, un composant logiciel est une unité de composition proposant une spécification de ses interfaces contractuelles et des dépendances de contexte explicites. Un composant logiciel peut être déployé de façon indépendante et être sujet à composition avec un tiers.

Un client qui souhaite utiliser un composant a besoin de connaître la sémantique, c'est-à-dire la structure de l'appel et le comportement de chaque méthode contenue dans le composant. La compréhension des relations entre les méthodes contenues dans une interface est un plus pour l'utilisation (et la réutilisation) effective et correcte du composant proposant ces méthodes. De ce fait, on trouve en général deux approches dans la définition des contrats, la troisième citée plus bas étant plus rare.

Un premier niveau de contrat correspond au typage des paramètres des opérations. Il s'agit de décrire exactement les variables passées en paramètres ainsi que les relations entre ces paramètres et l'état du composant, puis l'état des paramètres de retour après exécution de la méthode. Cette approche peut s'exprimer à l'aide des contrats tels que définis par Meyer. La seconde approche en terme de contrat permet l'expression de l'ordonnancement des actions du composant, donc en général l'expression de l'ordre d'exécution des opérations. Cette expression peut se faire à l'aide d'automates ou de logique temporelle. Par exemple, dans UML 2.0, un automate de service est défini, celui-ci devant se composer avec l'automate de service du composant requis. Finalement, la dernière approche correspond à la prise en compte de la qualité de service associée au composant. Pour l'instant, ce type de contrat est rarement décrit.

Plus précisément, une taxonomie a été définie par A. Beugnard et al. [AJND99]. Une spécification est un contrat pour un composant logiciel qui décrit les propriétés selon quatre niveaux qui sont :

- basique: les propriétés syntaxiques des noms des méthodes et le type des paramètres et les propriétés sémantiques simples (e.g. les langages de définition d'interfaces avec les IDL).
- comportemental: les propriétés qui peuvent être spécifiées avec des pré-conditions, des post-conditions, et des invariants, en incluant les contraintes sur l'historique. Ces propriétés sont des propriétés liées à une méthode, cette méthode étant vue comme une unité atomique.
- synchronisation: il s'agit ici des propriétés concernant les interactions des composants. Ces propriétés ne peuvent pas s'exprimer uniquement à l'aide de pré et post-conditions sur une méthode. Cela correspond à la prise en compte des interactions du composant avec l'extérieur, mais ayant une influence sur son comportement interne. On peut décrire leur exécution

sous forme de suite d'étapes. Le principe de l'atomicité, ici, ne peut être conservé, l'indéterminisme des appels et la concurrence de l'exécution des opérations est à prendre en compte dans la description de ces propriétés.

- qualitatif : toutes les propriétés non fonctionnelles, telles que la qualité de service, le temps de réponse correspondent au quatrième niveau de contrat.

De nombreux outils permettant la description et la vérification de contrats ont été proposés avec les environnements à objets et plus particulièrement Java. On peut citer quelques travaux tels que JMSAssert [MS02] qui gère des classes annotées avec des conditions. iContract [R.98] utilise un langage d'assertion qui est compatible avec OCL et un pré-processeur de code source dans le but d'associer des contrats. Contract Java [BM01] vérifie des conditions au niveau de l'interface. Il montre comment la programmation orientée composant propose une certaine assurance par la définition de contrat que lorsque la définition de classes est plus restreinte. Leurs interfaces n'incluent pas de variable. Ainsi, les conditions se restreignent à des prédicats sur des paramètres. Le langage JML [GK00] définit le moyen de décrire des spécifications pour des composants Java. Le langage insiste sur le sous-typage comportemental comme étant un élément de raffinement. De plus, le langage fait la différence entre sous-typage fort et faible. Le langage AsmL, défini par M. Barnett et W. Schulte [MW02b, MW02a] définit un langage de spécification exécutable pour définir des interfaces comportementales de composants. Ce langage permet aux clients de comprendre le comportement du composant sans avoir accès au code source. Ils intègrent dans leur langage la notion de sous-typage comportemental de façon à assurer la substitution des composants et permettre la spécification de caractéristiques avancées telles que les types génériques, la sémantique transactionnelle, les contraintes d'historiques ou encore les invariants. Ils proposent également une vérification de ces spécifications à l'exécution. Ces travaux se placent dans le cadre de la plate-forme à base de composants .NET.

Le système ConFract [PR] a été développé au sein de l'équipe OCL du laboratoire I3S pour fournir un modèle abstrait de contractualisation pour les composants hiérarchique du modèle Fractal. Dans ConFract, les contrats sont construits pendant la phase d'assemblage des composants en utilisant les descriptions de schémas d'architecture, de signatures d'interface et de spécifications. Les spécifications sont ainsi clairement séparées des contrats. A partir des spécifications de chacune des parties, les dispositions du contrat sont construites. Chaque disposition d'un contrat permet d'identifier précisément les participants parmi lesquels on distingue le garant (l'entité responsable de la contrainte) et des bénéficiaires (les entités qui veulent s'appuyer sur la contrainte validée). Le système ConFract supporte deux types de contrats qui portent sur les relations de type point-à-point (contrat d'interface) ou de contenance (contrat de composition) entre composants. Potentiellement, des formalismes de spécification relativement différents peuvent être utilisés dans le système pour former des contrats mais dans sa version actuelle, le système ConFract est fourni avec un langage d'assertions dédié aux composants hiérarchiques. Ce langage repose sur des assertions exécutables, organisées en préconditions, postconditions et

invariants, et qui peuvent être vérifiées lors de l'exécution des composants.

A notre connaissance, seul le système Pipa [JM03] fournit une forme de contrats pour un système d'aspects. Pipa fournit en effet un mécanisme d'assertions pour le langage d'aspects AspectJ en utilisant le langage JML évoqué précédemment. Dans ce système, un advice peut être spécifié par des pré et postconditions et un invariant peut être associé à un aspect. La traduction de spécification Pipa/AspectJ vers JML/Java assure l'évaluation des contrats comportementaux à l'exécution. Les contrats proposés ici ne sont qu'une forme embryonnaire du contrat d'aspects que nous souhaitons développer, car ils n'expriment que des propriétés locales à l'aspect. De plus, aucun contrat assurant la sûreté de la composition d'aspects n'est exprimable et aucun méta-modèle ne permet d'abstraire facilement ces propositions pour avoir une approche orientée modèle.

3.4.2 Un modèle intégrant aspects et contrats pour la sûreté des compositions

Les modèles à composants prennent de plus en plus souvent en charge les adaptations dynamiques (ajout/retrait de fonctionnalités, modification des assemblages de composants, modification du comportement des composants, ...) permettant ainsi une évolution de plus en plus aisée des applications. Or, les modifications liées aux adaptations affaiblissent fréquemment la sûreté de l'exécution des applications (appel à une méthode inconnue, comportements contradictoires ou aléatoires, composant requis dans un assemblage manquant ou de type incorrect, introduction de cycles et de points d'interblocages ou de non déterminisme, ...). Pour assurer la sûreté des adaptations, les modèles à composants existants tels que CCM [OMG], Sofa [FDR98], Noah [BFCE⁺04], Molene [MF00], JAC [RLLG01] fournissent essentiellement des solutions ad hoc dont la mise en oeuvre est souvent informelle ou à la charge du développeur d'applications.

Ayant identifié ces problèmes, nous souhaitons fiabiliser le processus d'adaptation dynamiques des composants. Pour cela, notre proposition consiste à identifier : 1) les critères d'une adaptation dite « sûre », 2) le moment où une adaptation peut se produire et 3) comment les modifications associées à une adaptation doivent être effectuées. Nous pensons que ces questions peuvent être traitées en déterminant ce que nous appelons des « propriétés de sûreté » que les composants et les adaptations doivent respecter.

Notre approche est basée sur un méta modèle, décrit en UML [OMG03], sur lequel nous nous appuyons pour exprimer les propriétés de sûreté formellement à l'aide du langage de contraintes OCL [JA99]. Ce méta modèle et les propriétés qui lui sont associées nous permettent de projeter des adaptations et des preuves de propriétés dans les modèles à composants existants manquant d'expressivité en terme d'adaptabilité ou en terme de sûreté des adaptations qu'ils offrent.

3.4.3 Démarche

Un aspect peut être vu comme un composant que l'on souhaite assembler à un programme, à un composant ou encore à une hiérarchie de composants déjà existants. Ce qui change dans l'assemblage d'un aspect vis-à-vis d'un assemblage de composants est le type d'interaction que cet aspect va avoir avec les autres éléments du programme. L'interaction n'est pas de type client/serveur comme dans l'univers objet ou composant, mais peut être par exemple une modification de comportement ou de structure. Il s'agit donc de préciser le lieu, le moment et le moyen par lesquels l'aspect va interagir avec son environnement d'assemblage. Avec les techniques à aspects, on parlera alors de tissage essentiellement parce qu'ici l'assemblage ne se résume pas à une mise en relation prévue d'un service fourni avec un service requis, mais à l'ajout non sollicité d'un comportement ou d'une structure.

De plus, l'un des attraits de l'approche par aspects est le fait que le tissage d'un aspect ne va pas s'appliquer à un seul point précis de l'environnement, mais sur un ensemble de points ayant une relation conceptuelle, que l'on appelle une coupe. Le tissage ne sera donc pas localisé sur un point d'ancrage unique, mais sur un ensemble de points transversaux. Il faudra alors prendre en compte cette globalité qui touchera non pas uniquement des propriétés locales du programme, mais des propriétés globales. Des règles de composition locales et globales seront alors à prendre en compte.

Lors des opérations de tissage nous devons alors décrire le programme qui va être modifié, l'aspect lui-même et, finalement, la modification.

3.5 Conclusion

La plate-forme ARCAD est une infrastructure modulaire conçue autour d'un noyau Fractal/Julia et étendue par les différentes contributions des partenaires du projet. En analysant les extensions proposées, on peut s'apercevoir que l'infrastructure initiale a été enrichie de différentes manières :

- La proposition d'une bibliothèque de composants Fractal proposant un nouveau service "sur" Fractal (Fractal RMI) ;
- L'intégration d'un "backend" existant (ProActive) pour profiter des services de ce dernier (asynchronisme, migration) ;
- L'intégration d'un nouveau contrôleur Fractal et d'un service de context-awareness pour l'ingénierie de l'adaptation (aspect d'auto-adaptation) ;
- La proposition de propriétés de sûreté sur les composants basée sur un méta-modèle et des règles OCL.

L'intégration de ces extensions entre elles n'est qu'en partie réalisée. Mais de par la nature différente de ces extensions, il n'existe pas vraiment d'obstacle technique à leur intégration. La plate-forme ARCAD est dans ce sens plus une architecture à la carte dans laquelle on chargerait des *plugins* (comme pour Eclipse) sur Fractal/Julia.

Références

- [AJND99] Beugnard A., Jezequel J.M., Plouzeau N., and Watkins D. Making component contract aware. *Computer*, pages 32(7):38–44, july 1999.
- [B.92] Meyer B. Eiffel: The language. *Object-Oriented Series*, 1992.
- [BFCE⁺04] M. Blay-Fornarino, A. Charfi, D. Emsellem, A-M. Pinna-Dery, and M. Riveill. Software interaction. *Journal of Object Technology*, 2004.
- [BM01] Findler R. B. and Felleisen M. Contract soundness for object-oriented languages. In *Proceedings of OOPSLA'01*. ACM SIGPLAN, 2001.
- [C.A69] Hoare C.A.R. An axiomatic basis for computer programming. In *Communication of ACM*, pages 12(10):576–583, october 1969.
- [Cou02] Simon Courtenage. Specifying and detecting composite events in content-based publish/subscribe systems. In *22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, Vienna, Austria, July 2002. IEEE.
- [DGG95] K. R. Dittrich, S. Gatzju, and A. Geppert. The active database management system manifesto: A rulebase of a ADBMS features. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, volume 985, pages 3–20. Springer-Verlag, 1995.
- [DTHS99] B. Dumant, F. Dang Tran, F. Horn, and J.B. Stefani. Jonathan: an open distributed platform in Java. *Distributed Systems Engineering Journal*, vol.6, 1999.
- [FDR98] Plasil F., Balek D., and Janecek R. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, May 1998. IEEE CS Press.
- [GK00] Leavens G. and Dahra K. K. Concepts of behavioral subtyping and a sketch of their extensions to component-based systems. In Leavens G. and Sitaraman M., editors, *Proceedings of Foundations of Component-Based Systems*, pages 113–135. Cambridge University Press, 2000.
- [HL95] Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, Massachusetts, 24 1995.
- [JA99] Warmer J. and Kleppe A. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 1999.
- [JM03] Zhao J. and Rinard M. Pipa: A behavioral interface specification language for AspectJ. In *Proceedings of FASE'03*, 2003.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*. Springer-Verlag, June 1997.

- [MF00] Segarra M.T. and André F. A framework for dynamic adaptation in wireless environments. In *Proceedings of TOOLS Europe 2000*, Mont St. Michel, St. Malo, France, 2000.
- [MS02] Man Machine Systems. Jmsassert. available from <http://www.mnsindia.com> at [/JMSAssert.html](#), 2002.
- [MW02a] Barnett M. and Schulte W. The ABCs of specification: AsmL, behaviour and components. *Informatica*, 2002.
- [MW02b] Barnett M. and Schulte W. Contracts, components and their runtime verification. Technical Report MSR-TR-2002-38, Microsoft Research, Microsoft Corporation, Redmond, 2002.
- [OMG] OMG. CORBA 3.0 new components chapters. OMG TC Document [ptc/2001-11-03](#).
- [OMG03] OMG. Unified modeling language specification. OMG TC Document [formal/03-03-01](#), 2003.
- [PR] Collet P. and Rousseau R. Contracting hierarchical components. (submitted for publication).
- [R.98] Kramer R. iContract - the java design by contract tool. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 26)*. IEEE CS Press, 1998.
- [RID90] Helm R., Holland I., and Gangopadhyay D. Contracts: Specifying behavioral compositions in object-oriented system. In *Proceedings of OOPSLA ECCOOP '90*, pages 25:169–180. SIGPLAN Notices, october 1990.
- [RLLG01] Pawlak R., Seinturier L., Duchien L., and Florin G. JAC: A flexible and efficient solution for aspect-oriented programming in java. In Yonezawa A. and Matsuoka S., editors, *Reflection*, volume LNCS 2192, pages 1–24. Springer-Verlag, 2001.
- [R.W67] Floyd R.W. Assigning meanings to programs. In *Proceedings Symposium on Applied Mathematics*, pages 19:19–31, 1967.