

# PROJET RNTL ARCAD\*

## D5.3 - Apports et limites de l'architecture ARCAD

D5.3 - T0+30 : interne - T0+36 : public

Coordonateur : Thierry COUPAYE

Auteurs : Françoise BAUDE, Eric BRUNETON, Thierry COUPAYE,  
Pierre-Charles David, Thomas Ledoux, Matthieu MOREL, Audrey OCELLO,  
Michel RIVEILL

15 juin 2004

## 1 Objectifs de ce document

Ce document consitue le rapport final de réalisation du projet RNTL Arcad mettant en évidence les apports et les limites de l'approche choisie. Son organisation retrace en quelque sorte le déroulement du projet tel qu'il a pu être ressenti par les partenaires du projet ; et tel qu'il est brièvement décrit ici.

La thématique générale du projet était la construction d'une plate-forme extensible supportant la programmation par composants et autorisant l'adjonction plus ou moins transparente du support d'un certain nombre de services techniques (communication en mode déconnecté, migration, sécurité, etc.) et plus généralement de propriétés non fonctionnelles.

L'approche développée initialement consistait à considérer l'extension de modèles de composants "industriels" "standards" tels que COM, CCM ou EJB. Le choix s'est porté sur EJB et sur une de ses implantation, JOnAS, pour différentes raisons dont i) la relative maturité et simplicité du modèle EJB, ii) le langage Java sous-jacent au modèle EJB est utilisé par la plupart des partenaires, iii) la disponibilité d'une implantation open source et la proximité du projet avec le consortium ObjectWeb développant cette implantation.

---

\*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 (equipe OCM - 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 coordonateur du projet est Michel.Riveill@unice.fr

La phase initiale du projet a certes démontré la possibilité de réaliser la plupart des extensions désirées de la plate-forme JOnAS pour le support des propriétés non fonctionnelles ; mais également i) une certaine difficulté à effectuer ces extensions, résultant par des solutions ad-hoc et ii) plus encore une grande difficulté à intégrer, composer ces différentes extensions - du fait d'un manque de structuration (architecture) de la plate-forme JOnAS et plus globalement de manques au niveau du modèle même de composants EJB - notamment en qui concerne la composition architecturale des composants et le support pour leur déploiement et opération (administration).

La seconde phase du projet est alors parti sur d'autres bases centrées i) sur le développement d'un modèle de composants "plus architectural" que les modèles industriels et Julia un support d'exécution extensible de Fractal en Java ; et ii) sur des extensions du modèle et/ou de la plate-forme Julia et leur intégration censée être plus aisée dans ce nouveau contexte.

L'organisation de ce document reprend ce cheminement. La première section présente les limitations des modèles de composants industriels. Cette section introduit une grille d'analyse constituée de quatre critères : i) principes architecturaux, ii) support de propriétés non fonctionnelles, iii) contrôle au déploiement et à l'exécution et iv) implantations. Elle utilise ensuite cette grille pour comparer les modèles EJB et CCM. Les deux sections suivantes utilisent à leur tour cette grille d'analyse pour discuter des apports et limites du modèle de composants Fractal et de son implantation Julia du point de vue de la problématique propre à chaque partenaire.

## 2 Limite des modèles de composants industriels

Après une présentation des motivations et des principes du paradigme composant, cette section se concentre sur quelques travaux représentatif du domaine. Nous décrivons leurs mécanismes de gestion de services systèmes.

Le paradigme composant vient en réponse au problème de complexité de gestion des logiciels. Il considère toutes les étapes du cycle de vie des applications et s'intéresse non seulement à leur facilité de programmation, mais également à leur facilité d'administration.

### Principes architecturaux

**Séparation du code métier et du code technique** Popularisée par le paradigme objet, la programmation modulaire permettant la réutilisation de code existant a été un premier pas vers la simplification du processus de gestion de logiciels complexes. Elle a permis de considérablement réduire les coûts de développement en se concentrant sur l'organisation du code métier des applications, appelé encore code fonctionnel. Le paradigme composant préserve les avantages de la programmation orientée-objet tout en adressant

les activités complexes de déploiement et de contrôle à l'exécution des applications. Il s'intéresse à la gestion du code non fonctionnel des applications qui correspond aux services système utilisés par celles-ci comme la persistance, la sécurité ou les transactions. S'intéressant à la configuration du code non fonctionnel en fonction des environnements d'exécution, le paradigme composant vise, en plus de la réutilisation de code métier dans différentes applications, la réutilisation de ce code dans différents contextes d'exécution.

**Conteneurs de composants** Dans les modèles industriels, la construction d'applications correspond à une phase d'assemblage de composants selon un modèle de composition plat ou hiérarchique. Dans le cas des modèles plats, tels que EJB, .NET, les applications sont basées sur des composants indivisibles, appelés composants primitifs.

L'implantation des composants est prise en charge par le programmeur de composants, ainsi que par le système à composants. Le programmeur fournit le code fonctionnel, alors que le système se charge de la génération du code non fonctionnel. La structure de composant fait coopérer ses deux parties de code en suivant un modèle dans lequel le code non fonctionnel encapsule le code fonctionnel. Ainsi, le code métier peut être réutilisé non uniquement, comme dans les approches à objets, au sein de différentes applications, mais également dans différents environnements d'exécution.

Les modèles à composants existantsinstancient de différentes manières le modèle abstrait de structuration. Par exemple, dans EJB ou CCM, le code de gestion non fonctionnelle est appelé *conteneur* et gère les propriétés de persistance, de sécurité et de transactions.

**Contrôle au déploiement et à l'exécution** Le déploiement d'une application inclut l'installation de ses composants sur des machines d'exécution, sa configuration en fonction de l'environnement concret d'exécution et son lancement. Contrairement à l'approche objet où le code de déploiement fait partie du code des applications et un lancement demande une intervention sur tous les sites d'exécution, dans le monde à composants, le code de déploiement est défini de manière groupée et séparément de l'application. L'interprétation des directives de déploiement est automatisée et est laissée à charge aux environnements d'exécution. Dans EJB et CCM, par exemple, le déploiement est dirigé à l'aide de descripteurs XML.

L'unité sur laquelle portent les règles de déploiement est encore le composant. Elles configurent, en réalité, les conteneurs de composants de manière à ce que leur code fonctionnel puisse être exécuté dans l'environnement d'exécution considéré.

L'exécution des applications à base de composants est liée à deux aspects qui sont, d'une part, le support à l'exécution des instances de composant et, d'autre part, les mécanismes de contrôle du fonctionnement des applications

pendant leur exécution.

- Support d'exécution des instances de composant. Les instances de composant sont, comme dans le paradigme objet, des entités d'exécution concrètes créées à partir de types de composants. Leur exécution est rendue possible grâce à des environnements qui leur fournissent tous les services dont elles ont besoin. Souvent désignés sous le nom de serveurs, ces environnements se chargent de fournir des services système dont les indispensables services de nommage et de communication intercomposant, ainsi que des services de persistance, de sécurité, de transactions, etc. Ces services sont accessibles aux conteneurs qui les adaptent aux besoins des composants.
- Contrôle des applications à l'exécution. Les traitements de contrôle à l'exécution, appelés traitements d'administration, comprennent l'observation et l'éventuelle reconfiguration des applications. Ces reconfigurations assurent le fonctionnement optimal des applications en les adaptant en fonction de variations de l'environnement d'exécution ou en fonction d'évolutions de besoins. Des modèles à composants qui prêtent attention aux aspects d'administration sont par exemple CCM et RM-ODP.

## 2.1 EJB

**Principes architecturaux** Selon le modèle EJB, une application répartie est constituée de composants appelés Beans qui sont implémentés en Java. Les Beans n'ont qu'une seule interface d'entrée et peuvent être de trois types : session, entité et à messages. Au sein des applications, les Beans de type session et entité sont accessibles à travers une communication synchrone basée sur Java RMI. Les Beans à messages, introduits par la dernière version de la spécification, sont un moyen d'intégration d'un service d'événements dans les EJB. Ils communiquent de manière asynchrone en utilisant JMS.

L'établissement d'un lien vers un Bean consiste en la récupération de sa référence. Elle peut être passée en paramètre par un autre composant ou obtenue à l'aide d'un service de nommage. Les instances de Beans s'exécutent dans des conteneurs EJB qui sont chargés de leurs aspects non fonctionnels. Les conteneurs peuvent héberger plusieurs instances de plusieurs types de Bean. Ils gèrent les transactions, la sécurité et la persistance selon trois types de configurations prédéfinies correspondant aux trois types de composants. Les conteneurs sont inclus dans des environnements d'exécution, des serveurs EJB, qui fournissent les services système nécessaires.

**Gestion d'aspect non fonctionnels** Selon les types de composants, les conteneurs ont les configurations suivantes. Pour les Beans session qui représentant un dialogue avec des clients et qui sont non persistants et non concurrents, les conteneurs ne gèrent que l'état de ce dialogue. Pour les Beans

à messages, définis en tant que composants sans état, pouvant être partagés entre clients, les conteneurs gèrent la concurrence. Pour les Beans entité, représentant des données persistantes, les conteneurs gèrent la concurrence et la persistance. Les conteneurs fournissent une gestion de la persistance par défaut, mais offrent aux Beans la possibilité de la gérer eux-mêmes. Pour tous les types de Beans, les conteneurs fournissent les transactions et la sécurité.

Le conteneur est constitué d'objets d'interposition qui interceptent les invocations sur les Beans et qui gèrent des propriétés non fonctionnelles au moyen de traitements avant et après ces invocations. Les prétraitements peuvent inclure la vérification des droits d'accès qui est propre à la gestion de sécurité, le démarrage des transactions ou la propagation des contextes transactionnels. Les post traitements se chargent de l'enregistrement éventuel des beans sur support persistant et de la terminaison des transactions démarrées par un prétraitement.

Dans le cas de composants entité ou session, les conteneurs fournissent deux objets : un EJB Home et un EJB Object. Les EJB Home sont des usines de composants qui permettent la création, la recherche et la destruction d'instances. Ils sont rattachés à un type de composant. Les EJB Object sont des objets d'interposition qui sont rattachés aux instances d'un type de composant et représentent le seul moyen pour accéder aux composants encapsulés à l'intérieur des conteneurs. Ce sont eux qui contiennent, en plus des fonctions d'emballage et de déballage de messages réseau, les pré et post traitements nécessaires à la gestion des aspects non fonctionnels.

Dans le cas de composants à messages, dont les instances n'ont pas d'état ni d'identité, les conteneurs fournissent un objet d'interposition qui définit un objet de destination. Les messages des clients sont adressés à cet objet qui délègue au conteneur l'envoi d'un accusé de réception et la transmission des messages à une des instances des composants à messages.

L'environnement EJB propose une gestion d'aspects non fonctionnels qui suit les principes de la séparation des aspects. Cependant, il ne prévoit la gestion que des aspects non fonctionnels déjà mentionnés : persistance, transactions, sécurité et concurrence. La spécification définit de manière précise les liens entre ces aspects, la manière dont ils sont intégrés dans les conteneurs et l'ordre de leur traitement.

**Contrôle au déploiement et à l'exécution** Les EJB prévoient l'utilisation de descripteurs de déploiement pour la génération automatique des conteneurs. Ces descripteurs contiennent des informations sur le code fonctionnel, ainsi que sur le code non fonctionnel des composants. En ce qui concerne les fonctionnalités métier des composants, ils déclarent leurs classes d'implémentation, ainsi que leurs interfaces d'accès à distance. En ce qui concerne les aspects non fonctionnels, les descripteurs définissent le type de composant (session, entité, à messages) et donnent des précisions sur la ges-

tion des services système utilisés.

La dernière version de la spécification introduit également la possibilité de décrire, en plus des informations de gestion de composants, des informations relatives aux assemblages des applications. Le déployeur peut définir les dépendances entre Beans en spécifiant les types de Beans qui seront référencés par un Bean donné.

L'environnement EJB ne fournit aucune spécification concernant l'observation et le contrôle d'une application pendant son exécution. Il n'est pas possible d'obtenir de vision globale de l'application, ni d'informations sur l'état d'exécution d'un composant. Une solution possible est d'utiliser une autre spécification fournie par Sun Microsystems qui est l'extension de la plate-forme Java pour l'administration (JMX). Elle permet la programmation d'objets de contrôle pour les besoins applicatifs spécifiques. Dans le cas des EJB, ces objets pourraient faire partie des conteneurs pour fournir des fonctionnalités de gestion additionnelles. Cependant, JMX se place au niveau applicatif et il n'existe pas de spécification de son intégration dans les EJB au niveau conteneur.

**Implantations** La liste des projets implémentant la spécification EJB est longue. Elle inclut aussi bien des travaux en source libre que des produits commerciaux. Parmi les projets en source libre les plus utilisés sont JOnAS et JBoss. Les fournisseurs principaux d'implémentations commerciales sont BEA avec BEA WebLogic, IBM avec WebSphere et Oracle avec Oracle9iAS.

Une grande majorité des projets existants fournissent, en plus des caractéristiques requises par la spécification EJB, des services additionnels. En font partie des services génériques d'administration ainsi que des solutions de gestion de la duplication et de la cohérence. En ce qui concerne la gestion de l'administration, elle est abordée dans des projets comme WebSphere, JBoss ou BEA WebLogic et est basée sur des outils tels que JMX ou SNMP. Toutefois, elle ne s'adresse qu'aux aspects non fonctionnels de base : persistance, sécurité et transactions.

## 2.2 CCM

Spécifié par l'Object Management Group en 2002, CCM est un modèle jeune qui n'a pas l'étendue d'utilisation des EJB ou de sa plate-forme de base CORBA. Toutefois, il a l'avantage de considérer des environnements d'exécution hétérogènes et ses caractéristiques reflètent les dernières recherches autour des composants.

**Principes architecturaux** Les composants CCM sont définis par un ensemble d'attributs et par leurs interfaces fournies et requises, appelés respectivement ports d'entrée et ports de sortie. Les attributs de composants sont utilisés pour la configuration au déploiement ou à l'exécution. Les ports

définissent les points de communication des composants et sont utilisés pour leur invocation et leur interconnexion. En effet, il n'est pas possible d'appeler un composant sans référencer un de ses ports d'entrée.

Pour que les composants d'une application puissent communiquer, ils doivent être interconnectés à l'aide de leurs ports. Les connexions sont établies entre des ports de sortie et des ports d'entrée qui représentent des interfaces identiques et qui sont de la même nature synchrone ou asynchrone. En d'autres termes, les ports ne peuvent pas être connectés si l'interface qui est requise par le composant avec le port de sortie n'est pas fournie par le composant avec le port d'entrée. L'acheminement des communications est pris en charge par le bus CORBA sous-jacent.

CCM définit les types de composants, ainsi que leurs implantations, à l'aide de langages déclaratifs. Les types de composants sont décrits en utilisant une extension de l'IDL de CORBA ce qui permet la programmation des composants en différents langages et leur exécution dans des environnements hétérogènes. Quant aux implantations des composants, elles sont décrites à l'aide du langage de description CIDL. Contenant des détails sur la structure logicielle et sur la gestion non fonctionnelle des composants, les descriptions CIDL sont utilisées pour la génération des leurs conteneurs. Comme dans les EJB, ces conteneurs s'exécutent dans de serveurs qui leur fournissent les services système nécessaires. Étant donné que CCM est défini comme une couche au-dessus de CORBA, les services système proposés sont les services de transactions, de sécurité, de tolérance aux fautes, etc. de CORBA.

**Gestion d'aspects non fonctionnels** Contrairement aux conteneurs EJB, un conteneur CCM ne gère les instances que d'un type de composant. Les conteneurs contiennent des usines de composants (des Home) qui se chargent de la création et de la destruction des instances de composant. Ils disposent également d'objets d'interposition pour les interfaces fournies et requises des instances de composant.

Les conteneurs proposent deux interfaces particulières de gestion des objets d'interposition : une d' introspection et une de gestion de ports. L'interface d' introspection fournit des informations sur le type de composant, sur ses interfaces fournies et requises, ainsi que sur l'état de ses connexions à d'autres composants. L'interface de gestion de ports est utilisée pour établir ou détruire des connexions entre composants. En effet, les interconnexions entre composants sont gérées à l'aide de primitives explicites et ne sont pas établies par simple passage de référence comme dans le cas des EJB.

Les interface d' introspection et de gestion de ports, font partie des interfaces externes des conteneur qui incluent également les interfaces fonctionnelles des composants, ainsi que l'interface de l'objet Home. En plus des interfaces externes, les conteneurs disposent d'interfaces internes et d'appels ascendants. Les interfaces internes sont fournies par les conteneurs et

utilisées par les composants pour paramétrer la gestion des services système CORBA. Les interfaces d'appels ascendants sont fournies par les composants et utilisées par les conteneurs et interviennent dans les cas où la gestion des services système est à la charge des composants. Ces cas sont analogues à la persistance gérée par les Beans (BMP) dans les EJB.

La spécification CCM définit quatre instanciations de la structure générale des conteneurs. Ce sont quatre configurations prédéfinies de gestion d'aspects non fonctionnels correspondant à quatre catégories de composant : service, session, processus et entité. Comme dans le cas des EJB, la gestion des aspects non fonctionnels inclut la persistance, les transactions et la sécurité, avec possibilité de gestion de la persistance par le conteneur ou par le composant. Dans le cas des composants service, qui sont des composants sans état, et des composants session, qui ont un état non persistant, les conteneurs ne fournissent que les transactions et la sécurité. Pour les besoins des états persistants des composants processus et entité dont la différence réside dans l'utilisation de clés primaires pour l'identification des instances entité, les conteneurs se chargent, en plus des transactions et de la sécurité, de la gestion de la persistance.

**Contrôle au déploiement et à l'exécution** CCM suit l'exemple des EJB et fournit des descripteurs de déploiement. Toutefois, ces descripteurs concernent non seulement les composants isolés, mais également les assemblages de composants pour former des applications. En ce qui concerne les descripteurs de composants, ils décrivent leur structure logicielle, leur type en termes d'interfaces fournies et requises, ainsi que leur catégorie (processus, entité, session, service). Ces descripteurs sont générés à la compilation de la description des composants, mais peuvent être modifiés pour paramétrer la gestion des aspects non fonctionnels. Quant aux descripteurs d'assemblages, ils décrivent les architectures initiales des applications. Ils spécifient les instances de composants constituant une application, définissent des règles de placement sur des sites d'exécution et donnent les connexions à établir entre instances.

CCM ne gère pas explicitement le contrôle des applications à l'exécution. La plate-forme ne définit pas de gestionnaire global qui peut fournir des renseignements sur les paramètres d'exécution et sur les évolutions des applications par rapport à leur définition initiale dans les descripteurs. Elle ne fournit pas non plus de gestion spécifique de reconfiguration. Toutefois, les interfaces d'introspection et de gestion de ports sont des briques de base permettant l'observation et le contrôle à l'exécution de manière partielle. Par introspection, il est possible de naviguer et de connaître les connexions entre composants. Par l'interface de gestion de ports il est possible de reconfigurer les liens entre composants.

**Implantations** Les implémentations de CCM ne sont que peu nombreuses. Elles comptent à ce jour plusieurs implémentations partielles en source libre comme OpenCCM, EJCCM ou MicoCCM et un premier produit commercial de iCMG. Alors que les implémentations non commerciales ne se concentrent que sur la spécification CCM, le produit proposé par iCMG considère également plusieurs aspects d'administration. Il fournit des outils d'observation de l'environnement d'exécution, des composants et des ressources système et permet la définition et le chargement dynamique de scripts décrivant des traitements de gestion. Il utilise ses capacités d'administration pour gérer la tolérance aux fautes et la répartition de charge.

Le paradigme composant vient en réponse au problème de complexité de gestion des logiciels. Il considère toutes les étapes du cycle de vie des applications et s'intéresse non seulement à leur facilité de programmation, mais également à leur facilité d'administration.

**Exécution** L'exécution des applications à base de composants est liée à deux aspects qui sont, d'une part, le support à l'exécution des instances de composant et, d'autre part, les mécanismes de contrôle du fonctionnement des applications pendant leur exécution.

- Support d'exécution des instances de composant. Les instances de composant sont, comme dans le paradigme objet, des entités d'exécution concrètes créées à partir de types de composants. Leur exécution est rendue possible grâce à des environnements qui leur fournissent tous les services dont elles ont besoin. Souvent désignés sous le nom de serveurs, ces environnements se chargent de fournir des services système dont les indispensables services de nommage et de communication intercomposant, ainsi que des services de persistance, de sécurité, de transactions, etc. Ces services sont accessibles aux conteneurs qui les adaptent aux besoins des composants.
- Contrôle des applications à l'exécution. Les traitements de contrôle à l'exécution, appelés traitements d'administration, comprennent l'observation et l'éventuelle reconfiguration des applications. Ces reconfigurations assurent le fonctionnement optimal des applications en les adaptant en fonction de variations de l'environnement d'exécution ou en fonction d'évolutions de besoins. Des modèles à composants qui prêtent attention aux aspects d'administration sont par exemple CCM et RM-ODP.

Nous rappelons que l'objet de la suite du document n'est pas de détailler le modèle Fractal, ni l'implantation de référence en Java Julia, qui sont décrits dans d'autres livrables Arcad et pour lesquels on trouve divers documents sur le site du consortium open source Objectweb (<http://fractal.objectweb.org>) - notamment la spécification du modèle Fractal qui est le

document de référence du modèle Fractal.

L'objectif est d'énumérer les éléments distinctifs de Fractal et de son implantations Julia ; et de discuter leur apports et limites respectifs.

## 3 Apports de Fractal

### 3.1 France Telecom R&D

#### Principes architecturaux

**Structuration du modèle** Le modèle Fractal est un système général, minimal et extensible de *relations* entre des *concepts*. Fractal n'est pas dédié à un langage ou un environnement d'exécution particulier. C'est un modèle général qui ne présuppose pas une sémantique ou une granularité particulière associé aux composants - à l'instar des modèles de composants industriels EJB ou CCM qui modélisent des composants "métiers", de plutôt "gros grain", dont des "conteneurs" fournissent de manière plus ou moins transparente certains services d'infrastructure (propriétés non fonctionnelles). Fractal apparait également moins "figé" qu'EJB, CCM, OSGi ou Avalon. C'est au contraire un modèle minimal et extensible : "tout est à la carte" dans Fractal - y compris le(s) système(s) de types ou les capacités réflexives exhibés par les composants. Le modèle est principalement défini par cinq concepts : *composant*, *interface*, *liaisons*, *contrôleur*, *contenu*. Notons que si les concepts d'interface et de liaison sont relativement classiques (cf. e.g. ODP), Fractal généralise leur usage et 1) induit ainsi une séparation nette entre interfaces et implantations des composants ; et 2) rend les liaisons manipulables programmiquement (les liaisons ne sont plus "noyées" ou "enfouies" dans le code ), ce qui favorise intrinsèquement les capacités de (re)configuration.

**Récursion** Fractal est un modèle intrinsèquement récursif. Les composants Fractal sont auto-similaires (d'où le nom de "Fractal") : quelle que soit l'échelle à laquelle on les observe, ils apparaissent toujours comme étant la composition d'un contrôleur et d'un contenu - le contrôleur portant les interfaces qui seules permettent d'interagir avec les composants. L'auto-similarité des composants s'exprime également dans le mécanisme d'import/export d'interfaces ; ainsi que dans la modélisation explicite de *composants de liaisons* pour les liaisons distantes, sécurisées, etc. - ce qui rend en grande partie inutile le concept de *connecteur* présent notamment dans la plupart des langages de description d'architecture (ADL). L'auto-similarité est essentielle pour l'ingénierie des systèmes logiciels complexes (middleware et OS par exemple) vis-à-vis du passage à l'échelle.

**Réflexion** Fractal est un modèle intrinsèquement réflexif. Les contrôleurs permettent une introspection totale de la structure des composants, i.e.,

une réification de la topologie des assemblages de composants et permettent la navigation au sein de ces assemblages. Les contrôleurs exercent également un contrôle arbitraire sur leur contenu, i.e. exhibent des capacités d'intercession arbitraire. La réflexion structurelle est essentielle pour l'administration et plus généralement pour le contrôle (monitoring, reconfiguration, etc.) des logiciels déployés, en particulier des logiciels en exécution (voir également paragraphe "Contrôle au déploiement et à l'exécution" ci-après). La réflexion comportementale est particulièrement intéressante notamment vis-à-vis de la gestion de propriétés non fonctionnelles.

**Partage** Une caractéristique de Fractal est la possibilité pour un composant donné de faire partie du contenu de plusieurs composants englobants (le modèle n'est pas strictement hiérarchique) - on parle alors de *partage* de composants. Le partage de composants participe de *l'encapsulation* des composants au sein du modèle et se révèle particulièrement bien adapté à des composants modélisant des *ressources*.

**Contrôle au déploiement et à l'exécution** Le contrôle au déploiement et à l'exécution constitue la motivation fondamentale et l'apport majeur de l'utilisation de Fractal. Fractal est en effet un modèle essentiellement *structurel (architectural)* qui permet de *partitionner* un système complexe en sous-éléments manipulables. Les composants - et c'est une exigence du modèle - sont des entités présentes à l'exécution ; ce qui évite une "dissolution des composants dans le code" comme cela est possible dans d'autres approches (UML, MDA, ADL) plus centrées sur la modélisation que sur l'opération <sup>1</sup>u sens d'"opérateur de réseaux d'informations"... des systèmes logiciels.

Le *déploiement* est un processus complexe qui fait suite au processus de *développement* et qui couvre l'ensemble des activités réalisées après ce processus de développement : livraison (e.g. téléchargement), installation, configuration, activation, initialisation, administration, reconfiguration, désactivation, désinstallation. Fractal, en tant que modèle, n'offre pas de support complet pour l'ensemble de ces activités. Il définit principalement un ensemble d'API qui supporte dynamiquement la configuration, l'activation/désactivation et la reconfiguration (interfaces de contrôle Factory, LifeCycleController, BindingController, ContentController). Néanmoins, l'effort d'outillage entrepris autour de Fractal (Fractal ADL, GUI, RMI, JMX) permet, lui, un support plus complet et plus utilisable ("user friendly").

D'une manière générale, les apports techniques de Fractal vis-à-vis du déploiement et de l'exécution concernent :

- l'adaptabilité statique (environnements de déploiement arbitraires, évolutions des besoins, évolutions techniques, évolutions organisationnelles)

---

<sup>1</sup>A

- et dynamique (évolutions des environnements de déploiement) des systèmes logiciels ;
- l'administrabilité de ces systèmes par une instrumentation possiblement grandement automatisée (possiblement selon différents standards) et un contrôle uniforme quel que soit le "grain" considéré ;
- et plus généralement une plus grande homogénéité : couverture de tout type de logiciels et couverture de tout le cycle de vie : développement, configuration, déploiement et opération (administration et maintenance).

**Gestion d'aspects non fonctionnels** La gestion d'aspects non fonctionnels n'est pas la première motivation (historiquement parlant) du développement de Fractal - du moins telle qu'elle est incarnée par les conteneurs EJB ou CCM. La spécification du modèle et les implantations actuelles n'offrent pas de cadre ou solutions spécifiques pour cette problématique. Néanmoins, remarquons que, dans une compréhension plus large de la problématique, les contrôleurs définis dans la spécification couvrent bel et bien la gestion de deux aspects non fonctionnels : la structure (contrôle des liaisons et des contenus) et le cycle de vie - qui apparaissent comme des prérequis à la gestion d'aspects plus complexes tels persistance, sécurité, etc. Remarquons par ailleurs, que le contrôle exercé sur les composants étant arbitraire, et que le modèle étant extensible ; des solutions pourront probablement être proposées pour cette problématique - et permettront, accessoirement, d'avancer vers la définition de *personalités* selon divers standards : EJB, JDO...

### Implantation Julia

**"Complétude"** Julia offre un support "complet" de Fractal en ce sens qu'elle fournit une (voire plusieurs) implantation de toutes des interfaces définies dans la spécification de Fractal, offrant ainsi un support d'exécution de composants Fractal correspondant au niveau de conformité le plus élevé défini par la spécification. Cette complétude, du point de vue du programmeur de composants, est renforcée par l'outillage associé - notamment Fractal RMI qui permet de programmer des composants Fractal répartis.

**Modularité et extensibilité** Julia se présente avant tout comme un framework de programmation et de composition de contrôleurs Fractal en Java. Les contrôleurs Fractal étant des entités abstraites réalisant diverses fonctions de contrôle (liaison, contenu, cycle de vie, nommage...), on peut même plutôt parler de "composition d'aspects de contrôle" en référence à la programmation par aspects, plutôt que de "composition de contrôleurs". Les aspects de contrôle sont également des entités abstraites implantées dans Julia par des *objets contrôleurs*. Julia offre une bibliothèque d'objets contrôleurs,

c'est-à-dire des objets implantant les interfaces de contrôle Fractal (actuellement plusieurs pour `BindingController` et `LifeCycleController`, une pour `ContentController`, une pour `NameController` qui n'est pas obligatoire dans la spécification...). Cette bibliothèque peut être étendue par le programmeur et les objets contrôleurs composés pour construire des contrôleurs de composants arbitraires. Les objets contrôleurs de deux types : *intercepteurs* et *objets de contrôle*. Julia fournit un mécanisme pour associer des intercepteurs aux interfaces ; et surtout un mécanisme de *mixin* pour composer les aspects de contrôle. D'une certaine manière, on peut dire que le développement de Julia est motivé par les mêmes soucis architecturaux que Fractal mais à un niveau d'abstraction plus bas (typiquement du niveau de la programmation par aspects).

**Efficacité** Les différentes implantations d'objets de contrôle fournies par Julia peuvent être combinées de différentes manières, qui permettent de couvrir un *continuum* entre configuration statique et reconfiguration dynamique. Ce qui signifie qu'on peut instancier des composants soit avec des contrôleurs minimaux, ayant une empreinte mémoire faible et très peu ou même pas du tout d'impact sur les temps d'exécution, mais n'offrant pas de fonction d'introspection et de reconfiguration, soit au contraire avec des contrôleurs plus évolués, permettant l'introspection et la reconfiguration dynamique, mais ayant un impact plus important sur les performances et empreinte mémoire des applications.

Julia offre deux mécanismes d'optimisation, intra et inter composants. Le premier mécanisme permet de réduire l'empreinte mémoire d'un composant en fusionnant les objets de contrôle en un seul. Le second mécanisme permet d'optimiser les chaînes de liaisons entre composants : ce mécanisme permet de traverser les contrôleurs sans intercepteurs sans aucun surcoût, sans pour autant perdre la méta information sur les liaisons.

Julia est constitué d'un runtime, d'une bibliothèque de mixins, et de générateurs de code pour générer les interfaces des composants et les intercepteurs, pour mélanger les mixins choisis et en faire des objets de contrôle complets... La version actuelle de Julia inclut ainsi des générateurs de classes pour générer les classes `AttributeController` et `InitializationContext` (sorte d'usine à composants), les intercepteurs et les interfaces de composants, ainsi que pour fusionner des classes de contrôleurs, et pour mélanger des classes mixins. Julia fournit également des transformateurs de code pour ajouter les aspects "trace" et "cycle de vie" dans un intercepteur, ainsi que deux transformateurs abstraits pouvant être spécialisés facilement (l'un réifiant complètement tous les appels, l'autre réifiant seulement le nom de la méthode appelée). Les générateurs de code peuvent être utilisés soit statiquement (avant le lancement de l'application), soit dynamiquement, c'est à dire que les classes sont générées pendant l'exécution de l'application, au fur

et à mesure des besoins. Pour que la génération dynamique soit efficace, tous les générateurs de Julia travaillent directement au niveau du bytecode Java (par une utilisation d'ASM).

Citons enfin pour conclure cette section, du point de vue général de l'ingénierie des systèmes logiciels complexes, les apports suivants attendus d'un usage généralisé de Fractal :

- facilité de développement et d'intégration des plates-formes en développement, au déploiement, en opération ;
- productivité des développeurs et intégrateurs par développement de méthodes et outillage standards ;
- pérennité des plates-formes : adaptabilité et réutilisabilité (référentiels d'entreprise), maintenabilité et évolutivité, possibilité de support des standards (EJB, JMX...) par déclinaison en *personnalités*.

### 3.2 Apport de Fractal pour l'auto-adaptation (EMN)

L'objectif de nos travaux est de faciliter le développement d'applications auto-adaptatives, capables de modifier leur configuration (au sens large) en réaction aux évolutions de leur environnement d'exécution. Notre approche consiste, à partir d'une application *adaptable* conçue initialement, à permettre l'ajout dynamique de *politiques d'adaptations*, capables d'observer l'environnement, d'y détecter les changements significatifs pour l'application, et de mettre en œuvre les reconfigurations nécessaires pour adapter celle-ci. L'association d'une application adaptable et des politiques d'adaptations constitue une application *auto-adaptative*.

Pour être adaptable, une application doit offrir un bon compromis entre flexibilité (nécessaire pour permettre des reconfigurations) et rigidité (n'importe quelle reconfiguration ne doit pas être autorisée). L'utilisation d'un modèle de composant dynamique répond bien à cette problématique. Elle autorise la reconfiguration de l'application tant que l'on reste dans le cadre des contraintes architecturales exprimées par l'assembleur de l'application, qui garantissent son bon fonctionnement. De plus, en limitant les interactions entre composants via des interfaces bien définies, l'utilisation d'un modèle de composant permet de limiter l'impact de reconfigurations locales.

Fractal offre toutes les fonctionnalités nécessaires à la construction d'applications adaptatives. Le modèle intègre explicitement la notion d'architecture, permettant aux programmeurs de décrire la structure de leur application de façon assez riche : composition hiérarchique, interfaces requises et fournies, y compris optionnelles... Une application construite avec Fractal est automatiquement reconfigurable dynamiquement, dans la mesure où ces reconfigurations sont compatibles avec les contraintes exprimées par l'assembleur de l'application. De plus, Fractal étant réflexif, ces reconfigurations n'ont pas besoin d'être codées explicitement dans l'application, mais

peuvent être appliquées de façon générique, contrairement à ce qui se passe avec d'autres modèles de composants dynamiques comme ArchJava [?] par exemple.

Plus spécifiquement, les possibilités de reconfigurations fournies par Fractal que nous utilisons pour adapter les applications sont la paramétrisation (reconfiguration des composants à travers l'interface `attribute-controller`), et les reconfigurations structurelles (modifications des liaisons entre interfaces de composants).

### 3.3 Rainbow

L'objectif de nos travaux est de faciliter le développement d'applications auto-adaptatives, capables de modifier leur configuration (au sens large) en réaction aux évolutions de leur environnement d'exécution. Notre approche consiste, à partir d'une application *adaptable* conçue initialement, à permettre l'ajout dynamique de *politiques d'adaptations*, capables d'observer l'environnement, d'y détecter les changements significatifs pour l'application, et de mettre en œuvre les reconfigurations nécessaires pour adapter celle-ci. L'association d'une application adaptable et des politiques d'adaptations constitue une application *auto-adaptative*.

Pour être adaptable, une application doit offrir un bon compromis entre flexibilité (nécessaire pour permettre des reconfigurations) et rigidité (n'importe quelle reconfiguration ne doit pas être autorisée). L'utilisation d'un modèle de composant dynamique répond bien à cette problématique. Elle autorise la reconfiguration de l'application tant que l'on reste dans le cadre des contraintes architecturales exprimées par l'assembleur de l'application, qui garantissent son bon fonctionnement. De plus, en limitant les interactions entre composants via des interfaces bien définies, l'utilisation d'un modèle de composant permet de limiter l'impact de reconfigurations locales.

Fractal offre toutes les fonctionnalités nécessaires à la construction d'applications adaptatives. Le modèle intègre explicitement la notion d'architecture, permettant aux programmeurs de décrire la structure de leur application de façon assez riche : composition hiérarchique, interfaces requises et fournies, y compris optionnelles... Une application construite avec Fractal est automatiquement reconfigurable dynamiquement, dans la mesure où ces reconfigurations sont compatibles avec les contraintes exprimées par l'assembleur de l'application. De plus, Fractal étant réflexif, ces reconfigurations n'ont pas besoin d'être codées explicitement dans l'application, mais peuvent être appliquées de façon générique, contrairement à ce qui se passe avec d'autres modèles de composants dynamiques comme ArchJava [?] par exemple.

Plus spécifiquement, les possibilités de reconfigurations fournies par Fractal que nous utilisons pour adapter les applications sont la paramétrisation (reconfiguration des composants à travers l'interface `attribute-controller`),

et les reconfigurations structurelles (modifications des liaisons entre interfaces de composants).

### 3.4 Apport de Fractal dans le cadre des applications multimedia (Sardes)

Nos travaux s'inscrivent des applications multimedia distribuées avec l'objectif de fournir un support pour la construction d'applications dynamiquement configurable et reconfigurable. Pour cela, nous avons adopté le modèle Fractal qui nous a offert les avantages suivants :

**Modularité et extensibilité du control :** Le control de composants Fractal et modulaire dans le sens ou ses aspects (liaisons, contenu, cycle de vie) sont définis par des interfaces distinctes (implementées séparément et composées selon le type de composant). De plus, il est possible d'étendre le control d'un composant en intégrant de nouveaux contrôleurs.

**Récursion :** Une application peut être hiérarchisé en un nombre arbitraire de niveaux. Ceci nous a apporté un grand avantage pour la configuration/reconfiguration d'applications complexes (exemple, vidéoconférence multi-tiers). Ce type d'application peut en effet être décomposé en plusieurs parties moins complexes, et donc, facilement configurable. D'autre part, la hiérarchisation permet d'intégrer les règles de reconfiguration à différents niveaux. En effet, selon le type, une reconfiguration peut s'appliquer sur l'ensemble de l'application (cas d'un re-déploiement) ou bien sur un sous-ensemble précis de composants. Dans ce dernier cas, il est préférable que des reconfigurations soient prise en charge localement (i.e. par un composite matérialisant le sous-ensemble de composants) de façon complètement transparente au(x) niveau(x) supérieur(s). Dans notre cas, les reconfigurations dites structurelles (par exemple le changement d'un encodeur vidéo) sont intégrées dans les composites (à différents niveaux), alors que les reconfigurations du comportement d'un composant primitive sont gérées par ce même composant (changement d'attributs clés).

**Partage :** L'une des particularités du modèle Fractal est que, tout en étant hiérarchique, il offre la possibilité de partage de sous-composants entre des composites. Ce concept de partage nous a apporté une flexibilité lors de la configuration de nos applications.

### 3.5 OASIS

L'objectif de nos travaux est d'aider à la programmation et au déploiement d'applications réparties complexes et à haute performance, telles que exemplifié par le *grid computing*. Il est donc nécessaire dans un tel contexte de maîtriser la répartition, le passage à l'échelle, le parallélisme, le couplage de codes. Plus précisément, les applications visées présentent des besoins en terme de portabilité, de sécurité, notamment des communications, de super-

vision, de répartition de charge de calcul (via du redéploiement éventuel, en faisant usage de la mobilité des calculs), d'optimisation des communications (via de la co-allocation de calculs communicants pour optimiser les échanges de données, ou encore via de l'optimisation des chaînes de communications pour joindre n'importe quelle entité), de tolérance aux pannes (via de la gestion des déconnexions momentanées et reprise après reconnexion, via de la reprise des calculs par recouvrement arrière si le calcul n'est plus en état de continuer).

Dans ce contexte, l'utilisation du modèle Fractal a permis un certain nombre d'apports à la programmation d'applications basée sur la bibliothèque ProActive. En effet, il nous a permis de définir et développer une extension à ProActive qui fournit la notion de composants pour le *grid*.

Tout d'abord, ces apports sont liés au modèle Fractal à proprement parler, et plus précisément à ses propriétés de récursion et de réflexion.

**Récursion** Cela permet de définir de l'inclusion à tout niveau d'imbrication. Ceci est important pour maîtriser la complexité due au nombre importants d'entités actives en présence, qu'il nous faut visualiser, piloter, déployer, etc.

**Réflexion** Cela permet de rendre les liaisons manipulables, et donc autorise la reconfiguration dynamique. La reconfiguration dynamique est cruciale pour permettre la tolérance aux pannes par reprise de processus (il faut bien se lier à la nouvelle entité qui remplace l'entité défaillante).

**Ports collectifs** Le modèle Fractal introduit la notion de port collectif, c'est-à-dire le fait qu'il puisse exister plusieurs instances de la même interface cliente. Cet aspect est important en vue de déclencher des appels parallèles de services du même type. Il se trouve que l'implantation de cette notion a pu être aisément réalisée avec l'API de groupe d'objets typé présente dans ProActive.

**Gestion des aspects non fonctionnels** Le modèle Fractal offre déjà quelques propriétés non fonctionnelles telles celles qui relèvent de la structure (liaisons et contenus) et du cycle de vie. Du fait que le cadre de gestion des aspects non fonctionnels est non contraignant, nous avons pu facilement faire cohabiter d'autres aspects non fonctionnels propres à nos besoins, fondés sur une réification des appels de méthodes : communication asynchrone avec futur automatique, communication vers des groupes d'objets typés, sécurisation des communications, etc

## 4 Limites de Fractal

### 4.1 France Telecom R&D

**Typage et comportement** Aucun système de types n'est imposé dans Fractal et tout système de types peut potentiellement entrer dans le cadre Fractal. Néanmoins, le système de types "basique" proposé dans la spécification, supporté par Julia et le plus utilisé dans les expérimentations en cours autour de Fractal, est probablement trop élémentaire pour des systèmes ayant des contraintes de correction, de fiabilité - et au d'une manière générale d'un certain niveau de *confiance*. Des systèmes de types plus sophistiqués permettraient de spécifier et garantir différents types de contraintes ou invariants structurels associés aux assemblages de composants.

En suivant cette idée, on peut noter que le modèle Fractal - en tout cas tel que reflété par les API actuellement définies dans la spécification - est essentiellement *structurel* (manipulation des liaisons et des contenus). On peut donc manquer de concepts pour raisonner sur le comportement, i.e. la sémantique des composants.

Cette limitation est largement conjoncturelle puisque des travaux sur le sujet sont d'ors et déjà en cours sur le sujet : l'IS-Rainbow sur le typage, FTR&D sur spécification de comportement de composants par logique temporelle et observation du respect des comportements spécifiés à l'exécution, FTR&D et l'IS-OCL (P. Collet et R. Rousseau) autour du modèle de contractualisation de composants ConFract et exploration de la contractualisation assertionnelle par pré/post conditions et invariants. A terme, ces travaux pourraient être intégrés dans le modèle Fractal (i.e. la spécification) dans un système de types étendus et/ou un modèle de contractualisation des composants.

**Activités** Fractal manque d'articulation entre composants et *activités* (threads, transactions, processus de workflow...). Le terme "activité" n'est jamais employé dans la spécification de Fractal, et de fait, Fractal n'offre pas de support pour la programmation d'applications concurrentes à base de threads ou mettant en oeuvre des transactions ou des processus. Par ailleurs, Julia est "mono-threadé" en ce qui concerne les activités de reconfiguration. Des réflexions ont été menées dans le contexte d'ObjectWeb dans les groupes de travail "composants" et "transactions" mais cette problématique reste largement ouverte...

**Cohérence des reconfigurations** Fractal manque de support pour garantir la cohérence des reconfigurations dynamiques. Cette cohérence - outre la définition/formalisation de ce concept - nécessiterait la prise en compte explicite de l'état des composants : à la fois l'"état des données" (valeurs des attributs des composants et variables d'état) et l'"état de contrôle"

lié aux activités parcourant les composants. On voit que ce problème très délicat est intimement lié au deux points précédents (typage et activités).

**Formalisation** Un effort de formalisation (sémantiques formelles) serait probablement très bénéfique afin i) de découvrir d'éventuelles incohérences ou incomplétudes dans le modèle, ii) d'aider à mettre en oeuvre des procédures de test de conformité des implantations de Fractal, et iii) d'augmenter ainsi la confiance des utilisateurs vis-à-vis d'infrastructures logicielles à base de composants Fractal. Des travaux sont en cours à l'INRIA autour de la définition d'un calcul de processus réparti d'ordre supérieur ("Kell calculus") qui englobe Fractal. Des formalisations correspondant plus directement Fractal pourraient être définies.

Notons qu'une (ou même plusieurs) formalisation ne garantirait bien sûr pas seule cette confiance. Cette notion, assez large, couvre notamment des points évoqués précédemment comme le typage et la contractualisation, des reconfigurations cohérentes et bien sur des propriétés non fonctionnelles comme la sécurité au premier chef...

**Personnalités** Un aspect de la volonté de construire Fractal en tant que modèle général est lié à ses potentielles déclinaisons en *personalités* selon les standards du marché (EJB, CCM, JDO, Avalon...). Une personnalité peut être entendue comme un modèle de programmation (API) spécifique, possiblement accompagnée de concepts additionnels à Fractal, voire de composants Fractal prédéfinis (e.g. des composants "conteneurs"). A l'heure actuelle, ce concept de *personnalité* est encore très largement sous-spécifié et non utilisable.

### Limites de l'implantation Julia

- Julia offre des mécanismes à la AOP (intercepteurs et mixins) pour la composition d'aspects de contrôle qui, au regard des derniers travaux du domaine (JAC, Prose) peuvent être considérés comme ad-hoc (conventions de nommage) et relativement élémentaires (au regard de la dynamique ou des interactions entre aspects par exemple);
- La configuration de la plate-forme Julia pour un système donné est basée sur la manipulation d'un fichier de configuration (julia.cfg) utilisant un format (à la LISP) peu évident à manipuler; et surtout qui n'autorise qu'une configuration statique;
- Julia offre des générateurs d'intercepteurs abstraits facilement spécialisables, c'est à dire sans avoir besoin de connaître le bytecode Java, et qui permettent de programmer tout type d'intercepteur pre, post ou around. Cependant, pour obtenir des performances optimales, il est parfois nécessaire d'utiliser des générateurs d'intercepteur spécifiques,

qui ne peuvent être réalisés qu'en ayant une connaissance fine du bytecode Java.

- le niveau d'optimisation ne peut pas être changé de façon arbitraire en cours d'exécution. Il est par exemple possible de changer dynamiquement le niveau d'optimisation intra composant d'un composant, d'ajouter ou d'enlever dynamiquement un ou plusieurs objets et interfaces de contrôle d'un composant, mais uniquement en créant un nouveau composant avec le même état que l'ancien (et en utilisant des "interfaces cachées" spécifiques à Julia) puis en remplaçant l'ancien composant par le nouveau à l'aide des interfaces de contrôle Fractal (ContentController, BindingController...). Ces changements dynamiques ne sont donc possibles que pour les composants fournissant au moins ces interfaces de contrôle.

**Limites de l'outillage** Les limites de l'outillage actuellement disponible autour de Fractal/Julia sont les suivantes (en suivant le cycle de vie des logiciels) :

- manque d'environnement de programmation (IDE) pour la programmation des composants (i.e. du code des primitifs, les outils existants mentionnés ci-dessous (ADL, GUI) concernent uniquement la configuration des composants);
- manque de support pour la construction de référentiels persistants de composants;
- Fractal ADL : manque de support pour le déploiement réparti;
- manque de support pour la gestion de configurations (versions, variantes, révisions);
- Fractal GUI : manque de support pour le déploiement réparti, pas de support pour l'exécution (supervision, administration);

## 4.2 Limites de Fractal pour l'auto-adaptation (EMN)

Par défaut, les seuls types de reconfiguration supportées par Fractal sont la paramétrisation (**attribute-controller**) et la reconfiguration structurale (*rebinding*). Ces mécanismes ne permettent que d'effectuer des adaptations anticipées au moment du développement : les paramètres reconfigurables et les interfaces exposées par un composant sont choisis par le programmeur de celui-ci. Pour permettre des adaptations non-anticipées, nous avons besoin d'un mécanisme plus général pour modifier de façon transparente un composant Fractal.

La réflexion comportementale, qui n'est pas supportée par défaut dans Fractal, permet justement de modifier de façon non anticipée le comportement d'un composant, en permettant de redéfinir la sémantique de l'envoi de message. Grâce à la nature extensible du modèle Fractal et de son implémentation de référence (Julia), nous avons pu ajouter le support de la réflexion

comportementale sous la forme d'un *MetaObject Protocol* adapté au modèle Fractal (voir le livrable D4.3 pour une description du fonctionnement du MOP).

### 4.3 Rainbow

**Typage, comportement, contractualisation** Contrairement aux langages de description d'architecture (ADL) ou aux approches à base d'interactions logicielles, la communication entre les composants dans Fractal est peu d'externalisée. Même si les connexions (bindings) entre composants sont explicitées à l'aide des interfaces clientes, les interactions entre composants sont enfouies dans le code d'implantation des composants primitifs. D'un point de vue exécution cette approche permet d'obtenir des performances optimale, d'un point de vue reconfiguration et flexibilité des assemblages ceci peut limiter les reconfigurations puisque les liaisons entre composants ne peuvent pas être manipulées directement. Bien entendu, il est toujours possible si nécessaire de construire un environnement logiciel permettant de visualiser et de rendre directement manipulable les liaisons entre composants au prix, du respect d'un modèle de programmation et d'un surcoût lors de l'exécution.

Dans Fractal, comme dans la plupart des modèles de composants, l'évolution du type des composants n'est pas prise en charge et un composant doit toujours connaître a priori l'ensemble des types de composants avec lesquels il est susceptible d'interagir. Ceci nuit à la flexibilité du modèle et ne permet pas une évolution non anticipée des composants : tout ce qui peut être potentiellement utilisé doit forcément être spécifié à l'avance sous la forme d'interfaces "optionnelles".

Fractal permet le remplacement dynamique d'un composant par un composant "compatible". La notion de compatibilité repose actuellement uniquement sur une compatibilité en terme du respect des différentes interfaces. Il nous semble souhaitable de pouvoir inclure à terme au sein de la plateforme Fractal d'autres travaux permettant de compléter cette compatibilité syntaxique par une compatibilité sémantique en terme d'utilisation. La proposition ConFract menée au sein de l'I3S par P. Collet et P. Rousseau est une première étape en ce sens.

**Composition d'aspects élémentaire** Le comportement initial d'un composant Julia peut être modifié en utilisant des contrôleurs pour intégrer un certain nombre de services (sécurité, log, ...). Les contrôleurs associés à un composant sont définis dans un fichier de configuration qui est lu au chargement de l'application. A l'exécution, l'ensemble des contrôleurs associés à un composant ne peut pas être modifié. La composition de services reste donc une tâche statique. De plus cette composition ne correspond en fait qu'à un ordonnancement relatif à l'ordre de déclaration des intercepteurs

dans le fichier de configuration. Par conséquent, il est impossible de gérer un enchevêtrement de services plus fin.

**Formalisation** La programmation par composition peut introduire des cycles. Certains sont explicitement voulus par le programmeur mais d'autres apparaissent par le biais de la réutilisation d'assemblages existants. Actuellement, Julia permet de détecter les cycles de contenance (un composant se contient lui-même directement ou par transitivité). Ceci permet d'éliminer les cycles dans les appels de méthodes résultant de cycles de contenance. Par contre, les cycles d'appels de méthodes ne résultant pas de cycles de contenance ne sont pas détectés. Le programmeur a donc la charge d'assurer que les cycles introduits sont bien souhaités. Cette étape (non automatisée) peut être relativement facile à réaliser par le programmeur à la "construction" de l'application. Cependant, elle peut devenir extrêmement coûteuse et source d'erreurs lorsqu'elle doit être répétée à chaque adaptation des assemblages de composants où de nouveaux cycles peuvent potentiellement apparaître. D'où l'intérêt d'un mécanisme d'aide complémentaire permettant la détection automatique de cycles dans les appels de méthodes. Il ne s'agit pas de détecter les cycles comme des erreurs mais d'avertir le programmeur sur les potentiels risques.

**Cohérence des reconfigurations, manque d'IDE** Pour procéder à une adaptation dans le framework Julia, le programmeur doit explicitement arrêter les composants qui sont concernés par l'adaptation et une fois que les modifications sont opérées, il doit les redémarrer. La sélection des composants à stopper dépend du type d'adaptation visé. Par exemple, le remplacement complet d'un composant requière d'arrêter tous les clients du composant ce qui n'est pas le cas lors du remplacement d'une implantation. Actuellement, cette tâche doit être effectuée par un programmeur, elle est donc source d'erreur. Il nous semblerait intéressant que les mécanismes de base offerts par la plate-forme Julia pour la reconfiguration dynamique puissent être complétés par un ensemble d'outils de plus haut niveaux permettant :

- de garantir que l'ensemble des modifications demandées aux différents composants ont réellement abouti et en totalité. Un ensemble de modification doit pouvoir être effectué de manière atomique afin de faire passer le système d'un état cohérent dans un autre état cohérent, sans que les étapes intermédiaires fréquemment incohérentes ne puissent être rendue visibles.
- d'offrir une approche déclarative, utilisant très certainement les mécanismes programmés de bas niveau, dans le but de pouvoir éventuellement valider avant même la mise en place des différentes modifications que l'ensemble des règles de reconfiguration qui devront être effectuées constitue bien un ensemble 'homogène' n'introduisant pas

d'erreur dans l'architecture de l'application. Par exemple, il peut être intéressant que les vérifications de typage sur les connexions soient validées (ou invalidées) avant d'avoir redémarré les composants afin de ne pas mettre en danger l'application si l'adaptation n'est pas sûre.

Des travaux de telles natures sont en cours d'implémentation dans la plateforme satin dont les principes sont décrits dans [?]

#### 4.4 Limites de Fractal dans le cadre du multimedia (Sardes)

**Activité :** Le modèle Fractal ne distingue pas explicitement les composants actifs des composants passifs. Pour notre part, nous avons eu recours à cette notion d'activité de composants principalement pour des fins de reconfiguration de l'application. (Ceci nécessite en effet l'introduction de composants d'observation des ressources ou des composants 'fonctionnels' de l'application.)

**ADL :** Absence d'aspects dynamiques dans le langage de description d'architecture. Les aspects dynamiques concernent d'une part les propriétés initiales de l'application et d'autre part, son évolution pendant l'exécution (politiques de reconfiguration). Ceci étant jugé essentiel pour atteindre nos objectifs, nous avons défini un langage de spécification permettant de spécifier, outre la structure et les propriétés initiales de l'application, les règles de reconfigurations assurant son évolution. Cependant, ce langage est fortement lié aux applications multimédia et ne peut être utilisé pour d'autres types d'applications. Une extension de l'ADL actuel dans ce sens est envisageable.

#### 4.5 OASIS

**Typage et comportement** On peut noter l'absence de garantie sémantique sur les méthodes de l'API et surtout, sur la validité comportementale d'assemblages.

**Activités** Il est clair qu'il manque une articulation précise entre composant et activité. Pour notre part, nous réussissons naturellement à réaliser une telle articulation car l'on se borne à introduire une unique activité en charge de servir les appels de méthodes distants en mode FIFO par défaut. Cependant, la définition d'activités dans les composants est une voie qui devrait être explorée, notamment dans l'optique d'une spécification comportementale.

**Cohérence des reconfigurations** Ceci est problématique, notamment à cause du fait que la définition de la présence ou l'absence d'état d'un composant n'existe pas. On ne peut pas distinguer dans Fractal des composants *stateless* de composants *statefull*. Remplacer dynamiquement un composant *stateless* ne nous semble pas compromettre la cohérence de la reconfiguration.

Par contre, le remplacement d'un composant *statefull* nous semble plus délicat. En effet, il devient alors nécessaire d'initialiser le remplaçant de sorte à ce qu'il possède le même état que celui qu'il remplace, ce qui peut contraindre à ce que les deux composants soient non seulement du même type, mais de la même classe. Tout au moins, ceci devant être réalisé par programmation, c'est en effet source d'erreur. Par ailleurs, l'atomicité des reconfigurations n'est pas garantie ; or, dans un contexte réparti où les composants sont hébergés sur plus d'un site, il n'est pas certain qu'une reconfiguration atomique soit facile à assurer. Un autre point limitatif du modèle provient du fait que l'on n'a pas facilement une représentation de la configuration globale d'un ensemble de composants constituant l'application (l'ADL ne peut constituer qu'une représentation de la configuration *initiale*). De ce fait, si un composant qui se trouve être serveur, mais aussi client d'autres composants, décidait de manière unilatérale et autonome de se faire remplacer (*auto-reconfiguration*), on pourrait être confronté à un problème : si un composant serveur est changé, les clients n'en sont pas informés, et par conséquent, ces clients ne pointent plus vers le bon composant. Ainsi, l'utilisation de la propriété de reconfiguration dynamique en vue de rendre le système tolérant aux pannes ne se borne, dans l'état, qu'à la reprise de composants pilotée par une entité externe (telle un composant hiérarchique englobant) qui elle a une vision globale de la partie du système impliquée.

### **Modes de communication en passant par les interfaces collectives**

Comme précisé dans les apports du modèle Fractal, la notion de ports collectifs est importante pour le calcul parallèle, et a pu être facilement mise en œuvre en utilisant l'API de groupe d'objets typé présente dans ProActive. Cette API permet de spécifier si les paramètres de l'invocation doivent être diffusés ou distribués aux différents objets cibles de l'appel. Or, dans le modèle Fractal, le mode de transmission des paramètres de l'appel d'une communication collective n'est pas proprement spécifié. On croit comprendre que le mode consiste en une diffusion de la même information à tous les composants connectés au port collectif. Ainsi, nous n'avons pas pu – au niveau de notre implantation du modèle Fractal –, exploiter la possibilité de pouvoir distribuer les paramètres en vue d'exécuter les services en parallèle mais sur des données différentes. (Cette possibilité n'est alors offerte que par programmation ad-hoc).

**Limites de l'outillage** L'ADL Fractal présuppose l'utilisation de templates. Or, nous n'en faisons pas usage. D'autre part, nous avons introduit un nouveau type de composant (composant parallèle) et nous voulons déployer tous les composants en faisant mention de leur localisation via le concept de nœud virtuel. L'ADL Fractal devrait donc être extensible, en tout cas, nous n'avons pas pu l'utiliser tel quel en l'état.