Le système de composants Fractal

Thierry Coupaye, Vivien Quéma, Lionel Seinturier, Jean-Bernard Stefani

Chapitre 3

Le système de composants Fractal

Les approches à base de composants apparaissent de plus en plus incontournables pour le développement de systèmes et d'applications répartis. Il s'agit de faire face à la complexité sans cesse croissante de ces logiciels et de répondre aux grands défis de l'ingénierie des systèmes : passage à grande échelle, administration, autonomie.

Après les objets dans la première moitié des années 1990, les composants se sont imposés comme le paradigme clé de l'ingénierie des intergiciels et de leurs applications dans la seconde moitié des années 1990. L'intérêt de la communauté industrielle et académique s'est d'abord porté sur les modèles de composants pour les applications comme EJB, CCM ou .NET. À partir du début des années 2000, le champ d'application des composants s'est étendu aux couches inférieures : systèmes et intergiciels. Il s'agit toujours, comme pour les applications, d'obtenir des entités logicielles composables aux interfaces spécifiées contractuellement, déployables et configurables ; mais il s'agit également d'avoir des plates-formes à composants suffisamment performantes et légères pour ne pas pénaliser les performances du système. Le modèle de composants Fractal remplit ces conditions.

Ce chapitre présente les principes de base du modèle Fractal (section 3.1). Les plates-formes mettant en œuvre ce modèle sont présentées dans la section 3.2. L'accent est mis sur deux d'entre elles, Julia (section 3.2.1) et AOKell (section 3.2.2). Les autres plates-formes existantes sont présentées brièvement en section 3.2.3. La section 3.3 présente le langage Fractal ADL qui permet de construire des assemblages de composants Fractal. La section 3.4 présente quelques bibliothèques de composants disponibles pour le développement d'applications Fractal, dont Dream (section 3.4.1) dédiée au développement d'intergiciels. La section 3.5 compare Fractal à des modèles de composants existants. Finalement, la section 3.6 conclut ce chapitre.

3.1 Le modèle Fractal : historique, définition et principes

Le modèle de composants Fractal est un modèle général dédié à la construction, au déploiement et à l'administration (e.g. observation, contrôle, reconfiguration dynamique) de systèmes logiciels complexes, tels les intergiciels ou les systèmes d'exploitation.

Le modèle de composants Fractal a été défini par France Telecom R&D et l'INRIA. Il se

présente sous la forme d'une spécification et d'implémentations dans différents langages de programmation comme Java, C, C++, SmallTalk ou les langages de la plate-forme .NET. Fractal est organisé comme un projet du consortium ObjectWeb pour le *middleware open source*. Les premières discussions autour du modèle Fractal, initiées dès 2000 à France Telecom R&D dans la lignée du projet Jonathan [Dumant et al. 1998], ont abouti en juin 2002 avec la première version officielle de la spécification et la première version de Julia, qui est l'implémentation de référence de cette spécification. La spécification a évolué pour aboutir en septembre 2003 à une deuxième version comportant un certain nombre de changements au niveau de l'API. Dès le départ, un langage de description d'architecture, Fractal ADL, a été associé au modèle de composants. Basé sur une syntaxe *ad hoc* au départ, il a évolué et sa version 2, définie en janvier 2004 et implémentée en mars 2004, est basée sur une syntaxe extensible.

Les caractéristiques principales du modèle Fractal sont motivées par l'objectif de pouvoir construire, déployer et administrer des systèmes complexes tels que des intergiciels ou des systèmes d'exploitation. Le modèle est ainsi basé sur les principes suivants :

- **composants composites** (i.e. composants qui contiennent des sous-composants) pour permettre d'avoir une vue uniforme des applications à différents niveaux d'abstraction.
- **composants partagés** (i.e. sous-composants de plusieurs composites englobants) pour permettre de modéliser les ressources et leur partage, tout en préservant l'encapsulation des composants.
- **capacités d'introspection** pour permettre d'observer l'exécution d'un système.
- **capacités de (re)configuration** pour permettre de déployer et de configurer dynamiquement un système.

Par ailleurs, Fractal est un modèle *extensible* du fait qu'il permet au développeur de personnaliser les capacités de contrôle de chacun des composants de l'application. Il est ainsi possible d'obtenir un continuum dans les capacités réflexives d'un composant allant de l'absence totale de contrôle à des capacités élaborées d'introspection et d'intercession (e.g. accès et manipulation du contenu d'un composant, contrôle de son cycle de vie). Ces fonctionnalités sont définies au sens d'entités appelées contrôleurs

Il est ainsi possible de distinguer différents rôles dans les activités de développement autour du modèle Fractal :

- les **développeurs de composants applicatifs** s'intéressent à la construction d'applications et de systèmes à l'aide de Fractal. Ils développent des composants et des assemblages de composants à l'aide de l'API Fractal et du langage de description d'architecture Fractal ADL. Ces composants utilisent des contrôleurs et des plates-formes existants.
- les **développeurs de contrôleurs** s'intéressent à la personnalisation du contrôle offerts aux composants. Ils développent de nouvelles politiques de contrôle comportant plus ou moins de fonctionnalités extra-fonctionnelles et permettant d'adapter les applications à différents contextes d'exécution (par exemple avec des ressources plus ou moins contraintes). Le développement de contrôleurs est conduit à l'aide des mécanismes offerts par les plates-formes : par exemple, *mixin* pour Julia (voir section 3.2.1) ou aspect pour AOKell (voir section 3.2.2).
- les **développeurs de plates-formes** fournissent un environnement permettant

d'exécuter des applications et des systèmes écrits à l'aide de composants Fractal. Un développeur de plate-forme fournit une implémentation des spécifications Fractal dans un langage de programmation et des mécanismes pour personnaliser le contrôle. On trouve ainsi des plates-formes dans différents langages comme Java ou C (voir section 3.2).

La section suivante introduit les éléments disponibles pour le développement d'applications et de systèmes avec Fractal. La section 3.3 reviendra sur le langage Fractal ADL et expliquera notamment comment il est possible de l'étendre. La section 3.1.2 introduit la notion de contrôleur qui sera reprise dans chacune des sections consacrées aux plates-formes existantes (voir respectivement les sections 3.2.1 et 3.2.2 sur Julia et AOKell).

3.1.1 Composants Fractal

La base du développement Fractal réside dans l'écriture de composants et de liaisons permettant aux composants de communiquer. Ces composants peuvent être typés. Finalement, le langage Fractal ADL constitue le vecteur privilégié pour la composition de composants.

Composants et liaisons

Un composant Fractal est une entité d'exécution qui possède une ou plusieurs interfaces. Une *interface* est un point d'accès au composant. Une interface implante un *type d'interface* qui spécifie les opérations supportées par l'interface. Il existe deux catégories d'interfaces : les interfaces *serveurs* — qui correspondent aux services fournis par le composant —, et les interfaces *clients* qui correspondent aux services requis par le composant.

Un composant Fractal est généralement composé de deux parties : une *membrane* — qui possède des interfaces fonctionnelles et des interfaces permettant l'introspection et la configuration (dynamique) du composant —, et un *contenu* qui est constitué d'un ensemble fini de *sous-composants*.

Les interfaces d'une membrane sont soit *externes*, soit *internes*. Les interfaces externes sont accessibles de l'extérieur du composant, alors que les interfaces internes sont accessibles par les sous-composants du composant. La membrane d'un composant est constituée d'un ensemble de *contrôleurs*. Les contrôleurs peuvent être considérés comme des méta-objets. Chaque contrôleur a un rôle particulier : par exemple, certains contrôleurs sont chargés de fournir une représentation causalement connectée de la structure d'un composant (en termes de sous-composants). D'autres contrôleurs permettent de contrôler le comportement d'un composant et/ou de ses sous-composants. Un contrôleur peut, par exemple, permettre de suspendre/reprendre l'exécution d'un composant.

Le modèle Fractal fournit deux mécanismes permettant de définir l'architecture d'une application : l'*imbrication* (à l'aide des composants composites) et la *liaison*. La liaison est ce qui permet aux composants Fractal de communiquer. Fractal définit deux types de liaisons : primitive et composite. Les liaisons *primitives* sont établies entre une interface client et une interface serveur de deux composants résidant dans le même espace d'adressage. Par exemple, une liaison primitive dans le langage C (resp. Java) est implantée à l'aide d'un pointeur (resp. référence). Les liaisons *composites* sont des chemins de communication arbitrairement complexes entre deux interfaces de composants. Les liaisons

composites sont constituées d'un ensemble de composants de liaison (e.g. *stub*, *skeleton*) reliés par des liaisons primitives.

Une caractéristique originale du modèle Fractal est qu'il permet de construire des composants *partagés*. Un composant partagé est un composant qui est inclus dans plusieurs composites. De façon paradoxale, les composants partagés sont utiles pour préserver l'encapsulation. En effet, il n'est pas nécessaire à un composant de bas niveau d'exporter une interface au niveau du composite qui l'encapsule pour accéder à une interface d'un composant partagé. De fait, les composants partagés sont particulièrement adaptés à la modélisation des ressources.

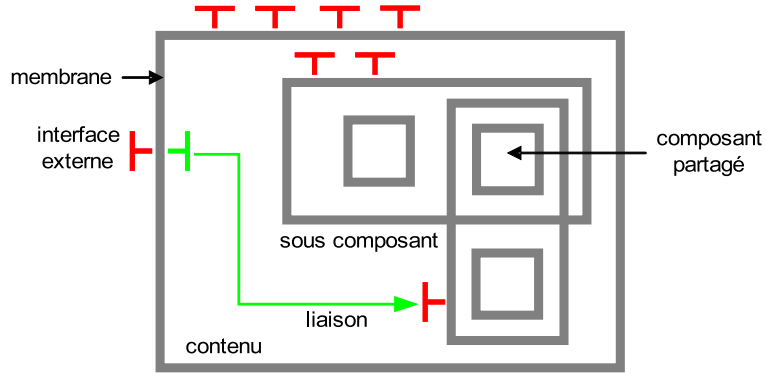

Fig. 3.1 – Exemple de composant Fractal.

La figure 3.1 représente un exemple de composant Fractal. Les composants sont représentés par des rectangles. Le tour gris du carré correspond à la membrane du composant. L'intérieur du carré correspond au contenu du composant du composant. Les interfaces sont représentées par des "T" (gris clair pour les interfaces clients ; gris foncé pour les interfaces serveurs). Notons que les interfaces internes permettent à un composite de contrôler l'exposition de ses interfaces externes à ses sous-composants. Les interfaces externes apparaissant au sommet des composants sont les interfaces de contrôle du composant. Les flèches représentent les liaisons entre composants. Enfin, nous avons représenté un composant partagé entre deux composites.

Système de types

Le modèle Fractal définit un système de types optionnel. Ce système de types autorise la description des opérations supportées par les différentes interfaces d'un composant. Il permet également de préciser le rôle de chacune des interfaces (i.e. client ou serveur), ainsi que sa cardinalité et sa contingence. La contingence d'une interface indique s'il est garanti que les opérations fournies ou requises d'une interface seront présentes ou non à l'exécution :

- les opérations d'une interface *obligatoire* sont toujours présentes. Pour une interface client, cela signifie que l'interface doit être liée pour que le composant s'exécute.
- les opérations d'une interface *optionnelle* ne sont pas nécessairement disponibles. Pour un composant serveur, cela peut signifier que l'interface interne complémentaire

n'est pas liée à un sous-composant implantant l'interface. Pour un composant client, cela signifie que le composant peut s'exécuter sans que son interface soit liée.

La cardinalité d'une interface de type T spécifie le nombre d'interfaces de type T que le composant peut avoir. Une cardinalité *singleton* signifie que le composant doit avoir une, et seulement une, interface de type T. Une cardinalité *collection* signifie que le composant peut avoir un nombre arbitraire d'interfaces du type T. Ces interfaces sont généralement créées de façon paresseuse à l'exécution. Fractal n'impose aucune contrainte sur la sémantique opérationnelle des interfaces de cardinalité *collection*. Il est ainsi possible de considérer les interfaces *collection* comme une collection d'interfaces dans laquelle chaque élément est traité comme une interface de cardinalité *singleton*, ou de considérer que l'invocation de méthode sur une interface *collection* provoque la diffusion du message à l'ensemble des composants liés à cette interface.

Le système de types permet également de décrire le type d'un composant comme l'ensemble des types de ses interfaces. Notons que le système de types définit une relation de sous-typage qui permet de vérifier des contraintes sur la substituabilité des composants.

Assemblage de composants

Le langage Fractal ADL permet de décrire, à l'aide d'une syntaxe XML, des assemblages de composants Fractal. Nous verrons à la section 3.3 que la DTD de ce langage n'est pas fixe, mais peut être étendue pour prendre en compte des propriétés extra-fonctionnelles.

La figure 3.3 donne un exemple de définition réalisée à l'aide de Fractal ADL. L'assemblage correspond est représenté figure 3.2. Le composant décrit est un composite dont le nom est `HelloWorld`. Ce composite possède une interface serveur, de nom `r` et de signature `java.lang.Runnable`. Par ailleurs, le composite encapsule deux composants : `Client` et `Server`. La définition du composant `Client` est intégrée à celle du composant `HelloWorld` : le composant a deux interfaces (`r` et `s`), sa classe d'implantation est `org.objectweb.julia.example.ClientImpl` et il possède une partie de contrôle de type `primitive`[1]. La définition du composant `Server` suit le même principe : une interface serveur `s` est définie et `org.objectweb.julia.example.ServerImpl` correspond à la classe d'implantation. Enfin, la description ADL mentionne deux liaisons : entre les interfaces `r` du composite et du `Client` et entre les interfaces `s` du `Client` et du `Server`.

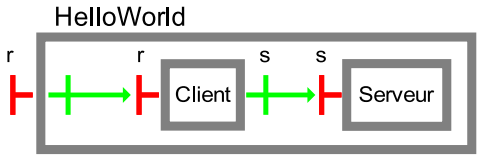

Fig. 3.2 – Assemblage Fractal correspondant à l'ADL de la figure 3.3.

La figure 3.4 fournit une implémentation des composants `Client` et `Server` mentionnés dans l'assemblage. La classe `ClientImpl` correspond au composant `Client` et fournit une

[1] Les deux mots utilisés pour décrire les parties contrôle des composants (`primitive` et `composite`) sont définis dans un fichier de configuration qui permet de spécifier l'ensemble des contrôleurs des différents composants.

```
<definition name="HelloWorld">
  <interface name="r" role="server" signature="java.lang.Runnable"/>
  <component name="Client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="org.objectweb.julia.example.ClientImpl"/>
    <controller desc="primitive"/>
  </component>
  <component name="Server">
    <interface name="s" role="server" signature="Service"/>
    <content class="org.objectweb.julia.example.ServerImpl"/>
    <controller desc="primitive"/>
  </component>
  <binding client="this.r" server="Client.r"/>
  <binding client="Client.s" server="Server.s"/>
  <controller desc="composite"/>
</definition>
```

Fig. 3.3 – Un exemple de définition ADL à l'aide du langage extensible Fractal ADL.

```
import org.objectweb.fractal.api.control.BindingController;

public class Client implements Runnable, BindingController {

  // Implementation de Runnable
  public void run() {
    service.print("Hello world!");
  }

  // Implementation de BindingController
  public String[] listFc() { return new String[] {"s"}; }
  public Object lookupFc( String cItf ) {
    if (cItf.equals("s")) { return service; }
    return null;
  }
  public void bindFc( String cItf, Object sItf ) {
    if (cItf.equals("s")) { service = (Service)sItf; }
  }
  public void unbindFc( String cItf ) {
    if (cItf.equals("s")) { service = null; }
  }
  private Service service;
}

public class ServerImpl implements Service {
  public void print( String msg) {
    System.err.println(msg);
  }
}
```

Fig. 3.4 – Implémentation des composants `Client` et `Server`.

implémentation pour l'interface `r` de type `java.lang.Runnable`. Par ailleurs, le composant `Client` manipule sa liaison avec le composant `Server` : il doit pour cela implémenter l'interface `BindingController` définie dans l'API Fractal. Le contrôleur de liaison notifiera cette implémentation de l'occurrence de toute opération concernant la gestion des liaisons de ce composant. Finalement, la classe `ServerImpl` implémente le composant `Server`.

3.1.2 Contrôleurs

Le modèle de composants Fractal n'impose la présence d'aucun contrôleur dans la membrane d'un composant. Il permet, au contraire, de créer des formes arbitrairement complexes de membranes implantant diverses sémantiques de contrôle. La spécification Fractal [Bruneton et al. 2003] définit un certain nombre de niveaux de contrôle. En l'absence de contrôle, un composant Fractal est une boîte noire qui ne permet ni introspection, ni intercession. Les composants ainsi construits sont comparables aux objets instanciés dans les langages à objets comme Java. L'intérêt de ces composants réside dans le fait qu'ils permettent d'intégrer facilement des logiciels patrimoniaux.

Au niveau de contrôle suivant, un composant Fractal fournit une interface `Component`, similaire à l'interface `IUnknown` du modèle COM [Rogerson 1997]. Cette interface donne accès aux interfaces externes (clients ou serveurs) du composant. Chaque interface a un nom qui permet de la distinguer des autres interfaces du composant.

Au niveau de contrôle supérieur, un composant Fractal possède des interfaces réifiant sa structure interne et permettant de contrôler son exécution. La spécification Fractal définit différents contrôleurs :

- le **contrôleur d'attributs** pour configurer les attributs d'un composant.
- le **contrôleur de liaisons** pour créer/rompre une liaison primitive entre deux interfaces de composants.
- le **contrôleur de contenu** pour ajouter/retrancher des sous-composants au contenu d'un composant composite.
- le **contrôleur de cycle de vie** pour contrôler les principales phases comportementales d'un composant. Par exemple, les méthodes de base fournies par un tel contrôleur permettent de démarrer et stopper l'exécution du composant.

Au delà de cet ensemble prédéfini par les spécifications, les développeurs peuvent implémenter leurs propres contrôleurs pour étendre ou spécialiser les capacités réflexives de leurs composants.

3.2 Plates-formes

Fractal est un modèle de composant indépendant des langages de programmation. Plusieurs plates-formes sont ainsi disponibles dans différents langages de programmation. Julia (voir section 3.2.1) l'implémentation de référence de Fractal, a été développée pour le langage Java. Une deuxième plate-forme Java, AOKell, développée plus récemment est présentée en section 3.2.2. Par rapport à Julia, AOKell apporte une mise sous forme de composants des membranes. La section 3.2.3 présente un aperçu des autres plates-formes existantes.

3.2.1 Julia

Julia est l'implémentation de référence du modèle de composant Fractal. Sa première version remonte à juin 2002. Julia est disponible sous licence *open source* LGPL sur le site du projet Fractal[2].

Julia est un canevas logiciel écrit en Java qui permet de programmer les membranes des composants. Il fournit un ensemble de contrôleurs que l'utilisateur peut assembler. Par ailleurs, Julia fournit des mécanismes d'optimisation qui permettent d'obtenir un continuum allant de configurations entièrement statiques et très efficaces à des configurations dynamiquement reconfigurables et moins performantes. Le développeur d'application peut ainsi choisir l'équilibre performance/dynamicité dont il a besoin. Enfin, notons que Julia s'exécute sur toute JVM, y compris celles qui ne fournissent ni chargeur de classe, ni API de réflexivité.

Principales structures de données

Un composant Fractal est formé de plusieurs objets Java que l'on peut séparer en trois groupes (figure 3.5) :

- Les objets qui implémentent le contenu du composant. Ces objets n'ont pas été représentés sur la figure. Ils peuvent être des sous-composants (dans le cas de composants composites) ou des objets Java (pour les composants primitifs).
- Les objets qui implémentent la partie de contrôle du composant (représentés en gris). Ces objets peuvent être séparés en deux groupes : les objets implémentant les interfaces de contrôle et des intercepteurs optionnels qui interceptent les appels de méthodes entrants et sortants. Les fonctions de contrôle n'étant généralement pas indépendantes, les contrôleurs et les intercepteurs possèdent généralement des références les uns vers les autres.
- Les objets qui référencent les interfaces du composant (en blanc). Ces objets sont le seul moyen pour un composant de posséder des références vers un autre composant.

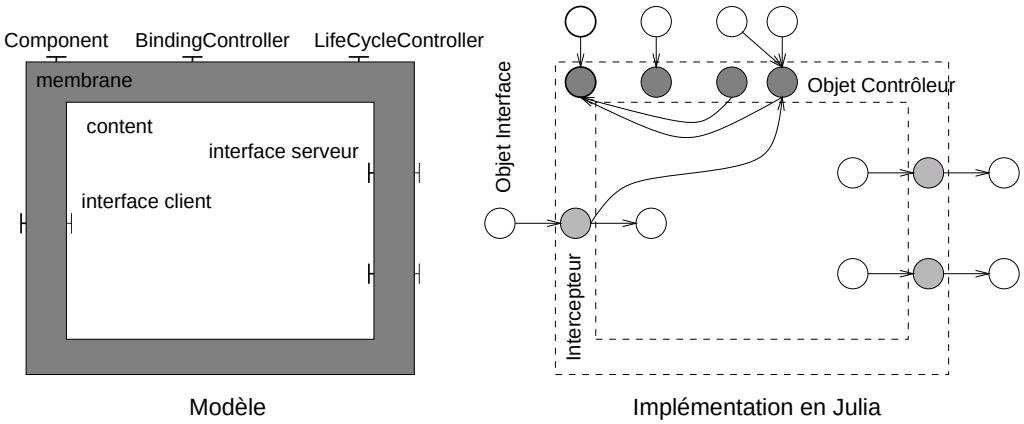

Fig. 3.5 – Un composant Fractal et son implantation Julia.

[2] `fractal.objectweb.org`

La mise en place de ces différents objets est effectuée par des fabriques de composants. Celles-ci fournissent une méthode de création qui prend en paramètres la description des parties fonctionnelle et de contrôle du composant.

Développement des contrôleurs

Fractal étant un modèle de composants extensible, il est nécessaire de pouvoir construire facilement diverses formes de contrôleurs et diverses sémantiques de contrôle. Par exemple, si l'on considère l'interface de contrôle de liaisons (`BindingController`), il est nécessaire de fournir différentes implantations de cette interface qui diffèrent par les vérifications qu'elles font lors de la création/destruction d'une liaison : interaction avec le contrôleur de cycle de vie pour vérifier qu'un composant est stoppé, vérification que les types d'interface sont compatibles quand un système de types est utilisé, vérification que les composants liés sont parents d'un même composite quand les contrôleurs de contenu sont utilisés, etc.

Il n'est pas envisageable d'utiliser l'héritage de classe pour fournir ces différentes implantations. En effet cela conduirait à une explosion combinatoire du nombre de classes nécessaires. Supposons que l'on souhaite effectuer des vérifications concernant le système de types, le cycle de vie et le contrôleur de contenu. Il existe $2^3 = 8$ combinaisons possibles de ces différentes vérifications. De fait, pour implanter toutes les combinaisons possibles, il serait nécessaire de fournir huit classes, ce qui engendrerait de nombreuses duplications de code.

La solution adoptée dans Julia est l'utilisation de *classes mixin* [Bracha and Cook 1990] : une classe *mixin* est une classe dont la super-classe est spécifiée de manière abstraite en indiquant les champs et méthodes que cette super-classe doit posséder. La classe *mixin* peut s'appliquer (c'est-à-dire surcharger et ajouter des méthodes) à toute classe qui possède les caractéristiques de cette super-classe. Ces classes *mixin* sont appliquées au chargement à l'aide de l'outil ASM [ASM 2002]. Dans Julia, les classes *mixin* sont des classes abstraites développées avec certaines conventions. En l'occurrence, elles ne nécessitent pas l'utilisation d'un compilateur Java modifié ou d'un pré-processeur comme c'est le cas des classes *mixins* développées à l'aide d'extensions du langage Java. Par exemple la classe *mixin* JAM [Ancona et al. 2000] illustrée sur la partie gauche de la figure 3.6 s'écrit en Julia en pur Java (partie droite). Le mot clé `inherited` en JAM est équivalent au préfixe `_super_` utilisé dans Julia. Il permet de spécifier les membres qui doivent être présents dans la classe de base pour que le *mixin* lui soit appliqué. De façon plus précise, le préfixe `_super_` spécifie les méthodes qui sont surchargées par le *mixin*. Les méthodes qui sont requises mais pas surchargées sont spécifiées à l'aide du préfixe `_this_`.

L'application de la classe *mixin* `A` à la classe `Base` décrite sur la partie gauche de la figure 3.7 donne la classe `C55d992cb_0` représentée sur la partie droite de la figure 3.7.

Développement des intercepteurs

Julia donne la possibilité de développer des *intercepteurs* dont le rôle est d'intercepter les appels de méthode entrant et/ou sortant des interfaces d'un composant. Les intercepteurs doivent implémenter les interfaces interceptées. Cependant, il est inconcevable de

```
mixin Compteur {
   inherited public void m ();
   public int count;
   public void m () {
      ++count;
      super.m();
   }
}
```

```
abstract class Compteur {
   abstract void _super_m ();
   public int count;
   public void m () {
      ++count;
      _super_m();
   }
}
```

Fig. 3.6 – Ecriture d'une classe mixin en JAM et en Julia.

```
abstract class Base {
   public void m () {
      System.out.println("m");
    }
}
```

```
public class C55d992cb_0 implements
                                Generated {
   // from Base
   private void m$0 () {
      System.out.println("m");
   }
   // from A
   public int count;
   public void m () {
      ++count;
      m$0();
   }
}
```

Fig. 3.7 – Application d'une classe mixin.

développer, pour un aspect de contrôle donné, autant d'intercepteurs qu'il y a d'interfaces à intercepter dans l'application. En conséquence, Julia fournit un outil, appelé *générateur d'intercepteurs*, qui permet de générer dynamiquement le code de ces intercepteurs. Cette génération est effectuée à partir d'informations fournies par le développeur comme par exemple les blocs de code à exécuter avant et après l'interception.

Optimisations

Julia offre deux mécanismes d'optimisation, intra et inter composants. Le premier mécanisme permet de réduire l'empreinte mémoire d'un composant en fusionnant une partie de ses objets de contrôle. Pour ce faire, Julia fournit un outil utilisant ASM [ASM 2002] et imposant certaines contraintes sur les objets de contrôle fusionnés : par exemple, deux objets fusionnés ne peuvent pas implémenter la même interface.

Le second mécanisme d'optimisation a pour fonction d'optimiser les chaînes de liaison entre composants : il permet de court-circuiter les parties contrôle des composites qui n'ont pas d'intercepteurs. Comme nous l'avons expliqué au paragraphe 3.2.1, chaque interface serveur de composant est représentée par un objet qui contient une référence vers un objet implantant réellement l'interface. Le principe du mécanisme de court-circuitage est représenté sur la figure 3.8 : un composant primitif est relié à un composite exportant l'interface d'un composant primitif qu'il encapsule. En conséquence, il existe deux objets de référencement d'interface (r_1 et r_2). Seuls les appels du primitif sont interceptés (objet i_1).

En conséquence, Julia court-circuite l'objet r_2 et r_1 référence directement i_1.

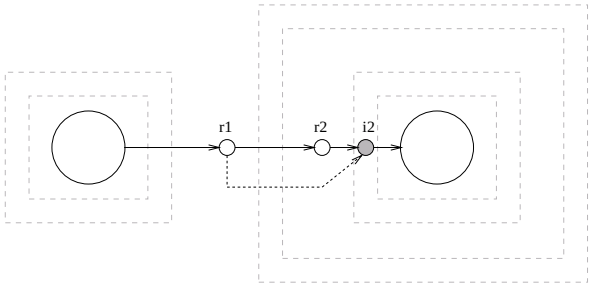

Fig. 3.8 – Optimisation des chaînes de liaison.

3.2.2 AOKell

Comme Julia, le canevas logiciel AOKell [Seinturier et al. 2006] est une implémentation complète des spécifications Fractal. Le respect de l'API Fractal permet ainsi d'exécuter telle quelle, avec AOKell, des applications conçues pour Julia ou vice-versa. AOKell est disponible sous licence *open source* LGPL sur le site du projet Fractal[3]. Le développement de AOKell a débuté en décembre 2004.

Le canevas logiciel AOKell diffère de Julia sur deux points : l'intégration des fonctions de contrôle dans les composants est réalisée à l'aide d'aspects et les contrôleurs sont implémentés eux-mêmes sous forme de composants. Par rapport à Julia qui utilise un mécanisme de *mixin* [Bracha and Cook 1990] et de la génération de *bytecode* à la volée avec ASM, l'objectif d'AOKell est de simplifier et de réduire le temps de développement de nouveaux contrôleurs et de nouvelles membranes.

La suite de cette section présente le principe de mise sous forme de composants des membranes et la façon dont AOKell utilise les aspects.

Membranes componentisées

La membrane d'un composant Fractal est composée d'un ensemble de contrôleurs. Chaque contrôleur est dédié à une tâche précise : gestion des liaisons, du cycle de vie, etc. Loin d'être complètement autonomes, ces contrôleurs collaborent entre eux afin de remplir la fonction qui leur est assignée. Par exemple, lors du démarrage d'un composite, son contenu doit être visité afin de démarrer récursivement tous les sous-composants[4]. De ce fait, le contrôleur de cycle de vie dépend du contrôleur de contenu. Plusieurs autres dépendances de ce type peuvent être exhibées entre contrôleurs.

Jusqu'à présent ces dépendances étaient implémentées sous la forme de références stockées dans le code des contrôleurs. L'idée d'AOKell est d'appliquer à la conception de la membrane de contrôle le principe qui a été appliqué aux applications : extraire du code les schémas de dépendances et programmer celui-ci sous forme de composants. En "spécifiant

[3] `fractal.objectweb.org`

[4] Notons que cela ne constitue pas une obligation formelle. Il est tout à fait possible de concevoir une fonction de contrôle pour laquelle le démarrage n'implique pas cette récursion.

contractuellement les interfaces" [Szyperski 2002] de ces composants de contrôle, AOKell espère favoriser leur réutilisation, clarifier l'architecture de la membrane et faciliter le développement de nouveaux composants de contrôle et de nouvelles membranes. Cette approche devrait également permettre d'apporter plus de flexibilité aux applications Fractal en permettant d'adapter plus facilement leur contrôle à des contextes d'exécutions variés ayant des caractéristiques différentes en terme de gestion des ressources (mémoire, activité, etc.).

Ainsi, AOKell est un canevas logiciel dans lequel les concepts de composant, d'interface et de liaison sont utilisés pour concevoir le niveau applicatif et le niveau de contrôle. Une membrane AOKell est un assemblage exportant des interfaces de contrôle et contenant un nombre quelconque de sous-composants. Chacun d'eux implémente une fonctionnalité de contrôle particulière. Comme expliqué précédemment, chaque composant de contrôle est aussi associé à un aspect qui intègre cette logique de contrôle dans les composants de niveau applicatif. La figure 3.9 présente le schéma de principe de cette solution. Par soucis de clarté, la membrane de contrôle du troisième composant applicatif a été omise.

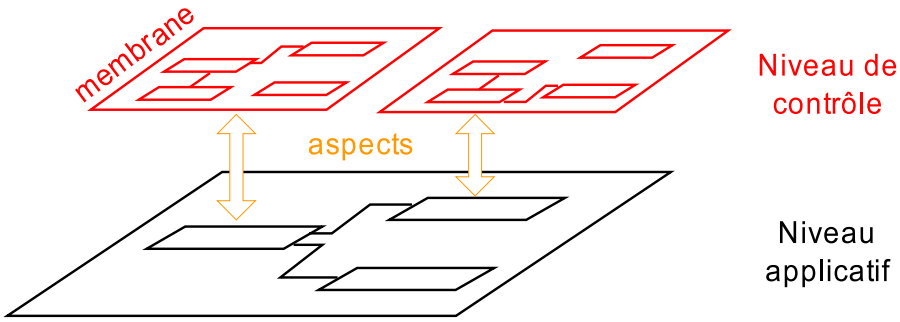

Fig. 3.9 – Les niveaux de composant du canevas logiciel AOKell.

La membrane la plus courante dans les applications Fractal est celle associée aux composants primitifs. L'architecture de cette membrane est illustrée figure 3.10. Cette membrane fournit cinq contrôleurs pour gérer le cycle de vie (`LC`), les liaisons (`BC`), le nommage (`NC`), les références vers les composants parents (`SC`) et les caractéristiques communes à tout composant Fractal (`Comp`).

L'architecture présentée figure 3.10 illustre le fait que la fonction de contrôle des composants primitifs n'est pas réalisée simplement par cinq contrôleurs isolés, mais est le résultat de leur coopération. Comparée à une approche purement objet, l'implémentation des membranes sous la forme d'un assemblage permet de décrire explicitement les dépendances entre contrôleurs. Elle permet également d'aboutir à des architectures logicielles de contrôle plus explicites, plus évolutives et plus maintenables. De nouvelles membranes peuvent être développées en étendant les existantes, ou en en développement des nouvelles.

Le code Fractal ADL suivant fournit la description de la membrane de la figure 3.10. La DTD permettant d'écrire cet assemblage et l'API Java permettant de le manipuler sont exactement les mêmes que ceux utilisés pour les composants Fractal applicatifs.

```
<definition name="org.objectweb.fractal.aokell.lib.membrane.primitive.Primitive">
```

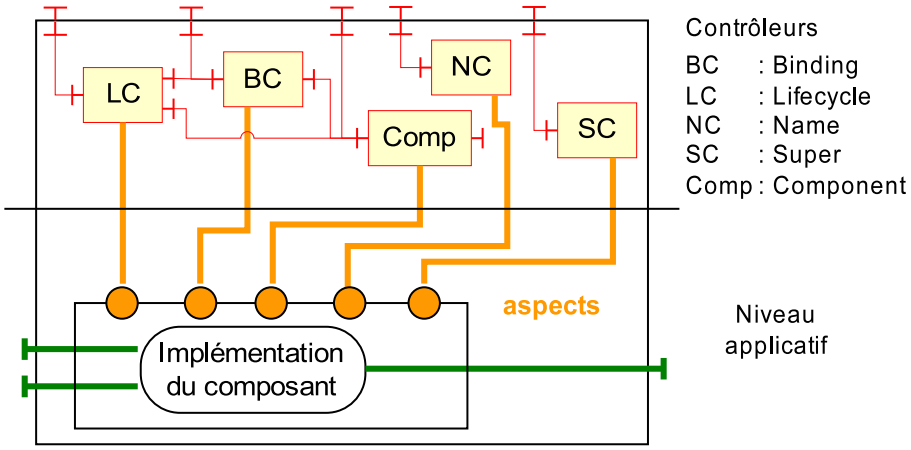

Fig. 3.10 – Membrane de contrôle pour les composants primitifs.

```
  <!-- Composants de contrôle inclus dans une membrane primitive -->
  <component name="Comp" definition="ComponentController"/>
  <component name="NC" definition="NameController"/>
  <component name="LC" definition="NonCompositeLifeCycleController"/>
  <component name="BC" definition="PrimitiveBindingController"/>
  <component name="SC" definition="SuperController"/>

  <!-- Export des interfaces de contrôle -->
  <binding client="this.//component" server="Comp.//component"/>
  <binding client="this.//name-controller" server="NC.//name-controller"/>
  <binding client="this.//lifecycle-controller" server="LC.//lifecycle-controller"/>
  <binding client="this.//binding-controller" server="BC.//binding-controller"/>
  <binding client="this.//super-controller" server="SC.//super-controller"/>

  <!-- Liaisons entre composants de contrôle -->
  <binding client="BC.//component" server="Comp.//component"/>
  <binding client="LC.//binding-controller" server="//BC.binding-controller"/>
  <binding client="LC.//component" server="Comp.//component"/>

  <controller desc="mComposite"/>
</definition>
```

L'apport de l'ingénierie du contrôle sous forme de composants a été expérimenté en ré-implémentant le canevas logiciel Dream [Leclercq et al. 2005b] avec AOKell. Ce canevas logiciel est présenté à la section 3.4.1.

Intégration des contrôleurs à l'aide d'aspects

Les modèles de composants comme EJB ou CCM fournissent des environnements dans lesquels les composants sont hébergés par des conteneurs fournissant des services techniques. Par exemple, les spécifications EJB définissent des services de sécurité, persistance, transaction et de gestion de cycle de vie. La plupart du temps, cet ensemble de services est fermé et codé en dur dans les conteneurs. Une exception notable est le serveur J2EE

JBoss [Fleury and Reverbel 2003] dans lequel les services peuvent être accédés via des aspects définis à l'aide du canevas logiciel JBoss AOP [Burke 2003]. De nouveaux services peuvent être définis qui seront alors accédés à l'aide de leurs aspects associés.

L'idée générale illustrée par le serveur JBoss est que les aspects, tout en fournissant un mécanisme pour modulariser les fonctionnalités transverses, permettent également d'intégrer de façon harmonieuse de nouveaux services dans les applications. Cela illustre également une des bonnes pratiques de la programmation orientée aspect : il est conseillé de ne pas implémenter directement les fonctionnalités transverses dans les aspects, mais d'y implémenter simplement la logique d'intégration (i.e. comment la fonctionnalité interagit avec le reste de l'application) et de déléguer la réalisation concrète de la fonctionnalité à un ou plusieurs objets externes. On obtient ainsi une séparation des préoccupations quasi optimale entre la logique d'intégration et celle du service à intégrer.

Cette pratique est mise en œuvre dans AOKell : chaque contrôleur est associé à un aspect chargé de l'intégration de la logique du contrôleur dans le composant. La logique d'intégration repose sur deux mécanismes : l'injection de code et la modification de comportement. Le premier mécanisme est connu dans AspectJ, sous la dénomination de déclaration inter-type (en anglais ITD pour *Inter-Type Declaration*). Avec ce mécanisme, les aspects peuvent déclarer des éléments de code (méthodes ou attributs) qui étendent la définition de classes existantes (d'où le terme ITD car les aspects sont des types qui déclarent des éléments pour le compte d'autres types, i.e. des classes). Dans le cas d'AOKell, les méthodes des interfaces de contrôle sont injectées dans les classes implémentant les composants[5]. Le code injecté est constitué d'une souche qui délègue le traitement à l'objet implémentant le contrôleur.

Le second mécanisme, la modification de comportement, correspond en AspectJ à la notion de code dit *advice*. Ainsi, la définition d'un aspect est constituée de coupes et de code advice. Les coupes sélectionnent un ensemble de points de jonction qui sont des points dans le flot d'exécution du programme autour desquels l'aspect doit être appliqué. Les blocs de code advice sont alors exécutés autour de ces points. Le code advice est utilisé dans AOKell pour intercepter les appels et les exécutions des opérations des composants. Par exemple, le contrôleur de cycle de vie peut rejeter des invocations tant qu'un composant n'a pas été démarré.

Avec les mécanismes d'injection de code et de modification de comportement, les aspects intègrent de nouvelles fonctionnalités dans les composants et contrôlent leur exécution. La réalisation concrète de la logique de contrôle est déléguée par l'aspect au contrôleur implémenté sous la forme d'un objet.

3.2.3 Autres plates-formes

Au delà de Julia et de AOKell, plusieurs autres plates-formes de développement Fractal existent. On peut citer en particulier, THINK pour le langage C, ProActive pour le langage Java, FracTalk pour Smalltalk, Plasma pour C++ et FractNet pour les langages de la plate-forme .NET.

THINK [Fassino et al. 2002, Fassino 2001] est une implémentation de Fractal pour

[5] Ce comportement par défaut peut être modifié afin, par exemple, de conserver des classes libres de toute injection.

le langage C. Elle vise plus particulièrement le développement de noyaux de systèmes qu'ils soient conventionnels ou embarqués. THINK est associé à KORTEX qui est une librairie de composants système pour la gestion de la mémoire, des activités (processus et processus légers) et de leur ordonnancement, des systèmes de fichiers ou des contrôleurs matériels (IDE, Ethernet, carte graphique, etc.). Les architectures matérielles les plus courantes comme celles à base de processeurs PowerPC, ARM ou x86 sont supportées. Le développement d'applications avec THINK passe par l'utilisation d'un langage pour la définition des interfaces (IDL) des composants et un langage pour la description des assemblages de composants (ADL). Dans le cadre des travaux en cours sur THINK v3, l'ADL est en cours de convergence vers Fractal ADL (voir section 3.3).

ProActive [Baude et al. 2003] est une implémentation de Fractal pour le langage Java. Elle vise plus particulièrement le développement de composants pour les applications s'exécutant sur les grilles de calcul. ProActive est basée sur la notion d'objet/composant actif. Chaque composant est muni d'un ensemble de *threads* qui lui sont propres et qui gèrent l'activité de ce composant. Une deuxième caractéristique originale des composants Fractal/ProActive réside dans leur sémantique de communication : un mécanisme d'invocation de méthode asynchrone avec futur est utilisé. Cela permet d'obtenir de façon transparente des interactions non bloquantes dans lesquelles les composants client poursuivent leurs traitements pendant l'exécution des services invoqués.

Finalement, trois autres implémentations du modèle Fractal existent : FracTalk [Bouraqadi 2005] pour Smalltalk, Plasma [Layaida et al. 2004] pour C++ et FractNet [Escoffier and Donsez 2005] pour les langages de la plate-forme .NET. Cette dernière est basée sur AOKell.

3.3 Fractal ADL

Fractal fournit un langage de description d'architecture (ADL) dont la caractéristique principale est d'être extensible. La motivation pour une telle extensibilité est double. D'une part, le modèle de composants étant lui-même extensible, il est possible d'associer un nombre arbitraire de contrôleurs aux composants. Supposons, par exemple, qu'un contrôleur de journalisation (`LoggerController`) soit ajouté à un composant. Il est nécessaire que l'ADL puisse être étendu facilement pour prendre en compte ce nouveau contrôleur, c'est-à-dire pour que le déployeur d'application puisse spécifier, via l'ADL, le nom du système de journalisation ainsi que son niveau (e.g. *debug*, *warning*, *error*). La seconde motivation réside dans le fait qu'il existe de multiples usages qui peuvent être faits d'une définition ADL : déploiement, vérification, analyse, etc.

Fractal ADL est constitué de deux parties : un langage basé sur XML et une usine qui permet de traiter les définitions faites à l'aide du langage. Nous présentons ces deux éléments dans les deux sections suivantes. La troisième section décrit le procédé d'extension de l'ADL.

3.3.1 Le langage extensible

Le langage ADL de Fractal est basé sur le langage XML. Contrairement aux autres ADL qui fixent l'ensemble des propriétés (implantation, liaisons, attributs, localisation,

etc.) qui doivent être décrites pour chaque composant, l'ADL Fractal n'impose rien. Il est constitué d'un ensemble (extensible) de modules permettant la description de divers aspects de l'application. Chaque module — à l'exception du module de base — s'applique à un ou plusieurs autres modules, c'est-à-dire rajoute un ensemble d'éléments et d'attributs XML à ces modules. Le module de base définit l'élément XML qui doit être utilisé pour démarrer la description de tout composant. Cet élément, appelé `definition`, a un attribut obligatoire, appelé `name`, qui spécifie le nom du composant décrit.

Différents types de modules peuvent être définis. Un exemple typique de module est le module *containment* qui s'applique au module de base en permettant d'exprimer des relations de contenance entre composants. Ce module définit un élément XML `component` qui peut être ajouté en sous-élément d'un élément `definition` ou de lui-même pour spécifier les sous-composants d'un composant. Notons que l'élément `component` a un attribut obligatoire `name` qui permet de spécifier le nom du sous-composant. Fractal ADL définit actuellement trois autres modules qui s'appliquent soit au module de base, soit au module `containment` pour spécifier l'architecture de l'application : le module `interface` permet de décrire les interfaces d'un composant ; le module `implementation` permet de décrire l'implantation des composant primitifs ; le module `controller` permet la description de la partie contrôle des composants.

Les modules ne servent pas uniquement à décrire les aspects architecturaux de l'application. Par exemple, Fractal ADL fournit des modules permettant d'exprimer des relations de référencement et d'héritage entre descriptions ADL. Le rôle principal de ces modules est de faciliter l'écriture de définition ADL[6]. Le module de référencement s'applique au module `containment` en ajoutant un attribut `definition` à l'élément `component` pour référencer une définition de composant. Le module d'héritage s'applique au module de base. Il permet à une définition d'en étendre une autre (via un attribut `extends`). Notons que l'héritage proposé par Fractal ADL est multiple.

3.3.2 L'usine extensible

L'usine permet de traiter les définitions écrites à l'aide du langage extensible Fractal ADL. Elle est constituée d'un ensemble de composants Fractal qui peuvent être assemblés pour traiter les différents modules de Fractal ADL décrits précédemment. Ces composants sont représentés sur la figure 3.11.

Le composant loader analyse les définitions ADL et construit un arbre abstrait correspondant (AST pour *Abstract Syntax Tree*). L'AST implante deux API distinctes : une API *générique* similaire à celle de DOM [W3C-DOM 2005] qui permet de naviguer dans l'arbre ; une API typée qui varie suivant les modules qui sont utilisés dans Fractal ADL. Par exemple, si le module `interface` est utilisé, l'API typée contient des méthodes permettant de récupérer les informations sur les interfaces de composants (nom, signature, rôle, etc.). Le composant *loader* est un composite encapsulant une chaîne de composants primitifs (figure 3.12). Le composant le plus à droite dans la chaîne (*basic loader*) est responsable de la création des AST à partir de définitions ADL. Les autres composants effectuent des vérifications et des transformations sur les AST. Chaque composant correspond à un

[6]Notons cependant que le module de référencement est nécessaire pour la description des composants partagés.

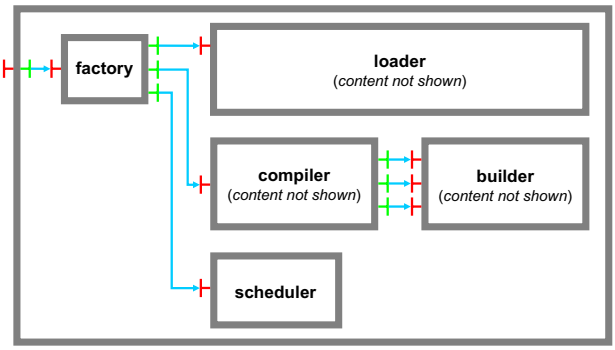

Fig. 3.11 – Architecture de l'usine Fractal ADL.

module de l'ADL. Par exemple, le composant *binding loader* vérifie les liaisons déclarées dans l'AST ; le composant *attribute loader* vérifie les attributs, etc.

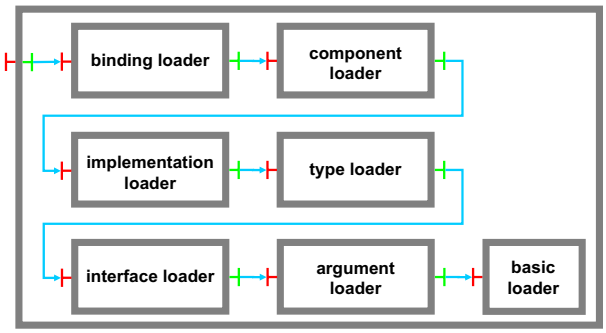

Fig. 3.12 – Architecture du composant *loader*.

Le composant compiler utilise l'AST pour définir un ensemble de tâches à exécuter (e.g. création de composants, établissement de liaisons). Le composant *compiler* est un composite encapsulant un ensemble de composants *compiler* primitifs (figure 3.13). Chaque *compiler* primitif produit des tâches correspondant à un ou plusieurs modules ADL. Par exemple, le composant *binding compiler* produit des tâches de création de liaisons. Les tâches produites par un *compiler* primitif peuvent dépendre des tâches produites par un autre *compiler* primitif ce qui impose un ordre dans l'exécution des *compiler*. Par exemple, le *compiler* primitif qui produit les tâches de création des composants doit être exécuté avant les *compilers* en charge des liaisons et des attributs.

Le composant builder définit un comportement concret pour les tâches créées par le *compiler*. Par exemple, un comportement concret d'une tâche de création de composant peut être d'instancier le composant à partir d'une classe Java. Le composant *builder* est un composite qui encapsule plusieurs *builders* primitifs (figure 3.13). Chaque canevas logiciel peut définir ses composants *builder*. Par exemple, Julia fournit actuellement quatre composants *builder* qui permettent respectivement de créer des composants avec l'API Java, l'API Fractal ou de produire du code source permettant d'instancier les composants à partir de ces deux API.

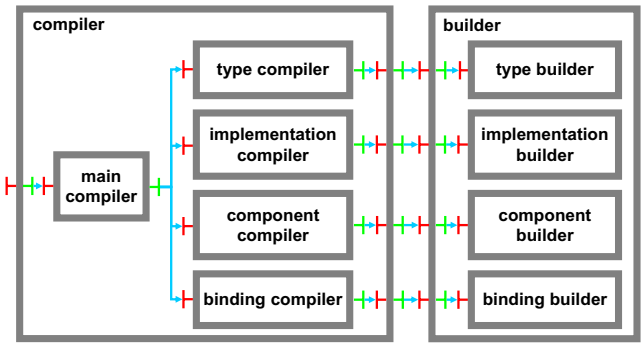

Fig. 3.13 – Architecture des composants *compiler* et *builder*.

3.3.3 Extension de Fractal ADL

Cette section décrit brièvement la démarche à suivre pour étendre Fractal ADL. Supposons que le développeur de l'application ait défini une interface de contrôle `LoggerController` permettant de configurer le nom et le niveau de journalisation de *loggers* associés aux composants (voir figure 3.14). Nous montrons tout d'abord comment le langage ADL peut être étendu pour autoriser la description de ces deux paramètres dans la description ADL de l'application. Nous décrivons ensuite les composants qui doivent être rajoutés à l'usine pour que les *loggers* soient configurés lors du déploiement de l'application.

```
public interface LoggerController {
   String getLoggerName ();
   void setLoggerName (String logger);
   int getLogLevel ();
   void setLogLevel (int level);
}
```

Fig. 3.14 – L'interface `LoggerController`.

L'extension du langage consiste à créer un module *logger* qui s'applique aux modules de base `containment` en permettant de rajouter un élément `logger` dont la syntaxe est la suivante :

```
<logger name="logger" level="DEBUG"/>
```

Le développeur du module doit effectuer le travail suivant : (1) définition des interfaces qui seront implantées par l'arbre abstrait lorsque le fichier XML aura été analysé par le *basic loader*; (2) écriture de "fragments" de DTD spécifiant que le module `logger` permet d'ajouter un élément XML `logger` aux éléments `definition` et `component`.

L'extension de l'usine requiert l'ajout de deux composants : un *compiler* et un *builder*[7]. Le composant *compiler* doit créer des tâches qui configureront le nom et le niveau de

[7]Le module `logger` ne nécessitant pas de vérifications sémantiques, il n'est pas utile de modifier le composant *loader*. Notons, néanmoins, qu'il serait possible de développer un loader vérifiant que les noms et niveaux de journalisation déclarés dans la description ADL ne sont pas nuls.

journalisation des *loggers* déclarés dans la description ADL. L'implantation du *compiler* est simple : il crée une tâche pour chaque *logger* à configurer et ajoute une dépendance de cette tâche vers la tâche de création du composant auquel le *logger* appartient. Concernant le composant *builder*, il est uniquement nécessaire de récupérer le logger associé au composant et de l'initialiser à l'aide des informations contenues dans l'arbre abstrait.

3.4 Bibliothèques de composants

Cette section présente quelques bibliothèques majeures existant à ce jour pour le développement d'applications à base de composants Fractal.

3.4.1 Dream

Dans cette section, nous présentons Dream [Quéma 2005], une bibliothèque de composants dédiées à la construction d'intergiciels orientés messages dynamiquement configurables plus ou moins complexes : de simples files de messages distribuées à des systèmes publication/abonnement complexes. Nous décrivons les éléments principaux du canevas Dream et illustrons son utilisation par la ré-ingénierie de Joram, une implantation *open source* de la spécification JMS. Nous montrons que la version réalisée à l'aide de Dream a des performances comparables et offre un gain de configurabilité significatif.

Motivations L'utilisation d'intergiciels orientés messages (MOM pour *Message-Oriented Middleware*) est reconnue comme un moyen efficace pour construire des applications distribuées constituées d'entités faiblement couplées [Banavar et al. 1999]. Plusieurs MOM ont été développés ces dix dernières années [Blakeley et al. 1995, msmq 2002, joram 2002, Strom et al. 1998, van Renesse et al. 2003]. Les travaux de recherche se sont concentrés sur le support de propriétés non fonctionnelles variées : ordonnancement des messages, fiabilité, sécurité, etc. En revanche, la configurabilité des MOM a fait l'objet de moins de travaux. D'un point de vue fonctionnel, les MOM existants implantent un modèle de communication figé : publication/abonnement, événement/réaction, files de messages, etc. D'un point de vue non fonctionnel, les MOM existants fournissent souvent les mêmes propriétés non fonctionnelles pour tous les échanges de messages. Cela réduit leurs performances et rend difficile leur utilisation pour des équipements aux ressources restreintes.

Pour faire face à ces limitations, la bibliothèque Dream propose de construire des architectures modulaires à base de composants pouvant être assemblés statiquement ou dynamiquement. Dream se démarque des travaux existants par le fait qu'il ne cible pas uniquement les intergiciels synchrones et qu'il intègre des primitives de gestion de ressources permettant d'améliorer les performances des intergiciels construits.

Architecture d'un composant Dream Les composants Dream sont des composants Fractal ayant deux caractéristiques : la présence d'interfaces d'entrée/sortie de messages et la possibilité de manipuler les ressources rencontrées dans les intergiciels de communication : messages et activités.

- *Les interfaces d'entrée et sortie de messages.* Le canevas Dream définit deux interfaces permettant aux composants de s'échanger des messages. Une interface d'entrée

(input) permet à un composant de recevoir un message. Une interface de sortie (output) permet à un composant d'émettre un message. Les messages sont toujours transmis des sorties vers les entrées (figure 3.15 (a)). On distingue néanmoins deux modes de transfert : *push* et *pull*. Dans le mode *push*, l'échange de message est initié par la sortie qui est une interface client `Push` liée au composant auquel le message est destiné (figure 3.15 (b)). Dans le mode pull, l'échange de message est initié par l'entrée qui est une interface client `Pull` liée au composant émetteur du message (figure 3.15 (c)).

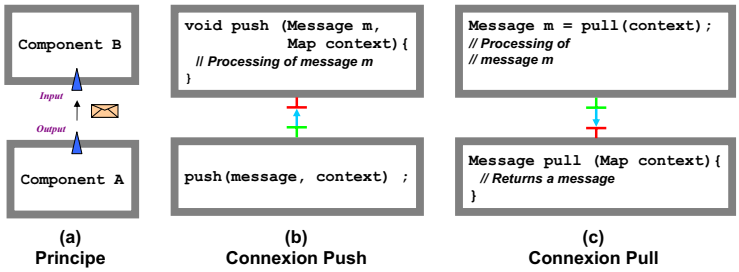

Fig. 3.15 – Les interfaces d'entrée et de sortie de messages.

- *Les gestionnaires de messages*. Les messages sont gérés par des composants partagés, appelés gestionnaires de messages (*message managers*). Ces composants permettent de créer, détruire et dupliquer des messages[8]. Leur but est de gérer les ressources mémoires consommées par les MOM. Pour ce faire, ils utilisent des réserves (*pools*) de messages permettant de réduire le nombre d'allocations d'objets.
- *Les gestionnaires d'activités*. Dream distingue deux sortes de composants : les *composants actifs* et les *composants passifs*. Les composants actifs définissent des tâches à exécuter. Ces tâches permettent au composant de posséder son propre flot d'exécution. Au contraire, les composants passifs ne peuvent effectuer d'appels sur leurs interfaces clients que dans une tâche d'un composant appelant une de leurs interfaces serveurs. Les tâches d'un composant actif sont accessibles par l'intermédiaire d'un contrôleur spécifique, appelé contrôleur de tâches (*task controller*). Pour qu'une tâche soit exécutée, il faut qu'elle soit enregistrée auprès d'un gestionnaire d'activités (*activity manager*). Les gestionnaires d'activités sont des composants partagés qui encapsulent des tâches et des ordonnanceurs (*schedulers*). Les ordonnanceurs sont en charge d'associer des tâches de haut niveau à des tâches de bas niveau. Les tâches de plus haut niveau sont les tâches applicatives (i.e. enregistrées par le MOM). Les tâches de plus bas niveau encapsulent des *threads* Java. Ces concepts sont représentés sur la figure 3.16. Les composants A et B ont enregistré trois tâches qui sont ordonnancées par un ordonnanceur FIFO. Celui-ci utilise pour cela deux tâches de bas niveau.

La bibliothèque de composants La bibliothèque de composants Dream contient des composants implantant les fonctions que l'on trouve de façon commune dans les différents

[8] Les messages sont des objets Java encapsulant des *chunks* et des sous-messages. Chaque *chunk* est également un objet Java implantant des accesseurs (*getter*) et des mutateurs (*setter*).

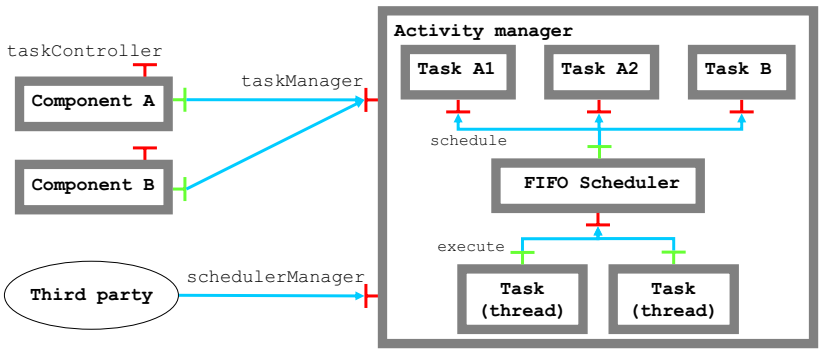

Fig. 3.16 – Exemple de gestionnaire de tâches.

MOM. Elle contient également des composants spécifiques développés pour des personnalités particulières de MOM. Par manque de place, nous nous contentons de décrire les composants formant le cœur de la bibliothèque.

- *Les files de messages* servent à stocker les messages. Elles ont une entrée — qui est utilisée par les autres composants pour stocker des messages —, et une sortie — que la file utilise pour délivrer les messages. Les files diffèrent par la manière dont les messages sont triés (FIFO, LIFO, ordre causal, etc.), leur comportement dans les différents états : file pleine (bloque vs. détruit des messages), file vide, etc.
- *Les transformateurs* sont des composants avec une entrée et une sortie. Chaque message reçu sur l'entrée est transformé, puis délivré sur la sortie. Un exemple typique de transformation consiste à rajouter une adresse IP à un message.
- *Les pompes* sont des composants avec une entrée *pull* et une sortie *push*. Les pompes ont une activité qui consiste à récupérer un message sur l'entrée et le délivrer sur la sortie.
- *Les routeurs* ont une entrée et plusieurs sorties. Le rôle d'un routeur est de router les messages reçus en entrée sur une ou plusieurs sorties. Le routage peut se faire sur le contenu du message, sa destination, etc.
- *Les agrégateurs/désagrégateurs*. Les agrégateurs sont des composants avec une ou plusieurs entrées et une sortie. Leur rôle est de délivrer sur la sortie un agrégat des messages reçus en entrée. Les désagrégateurs implémentent le comportement inverse des agrégateurs.
- *Les canaux* permettent l'échange de messages entre différents espaces d'adressages. Un canal est un composant composite distribué qui encapsule, au minimum, deux composants : un canal sortant (*channel out*) — qui permet d'envoyer des messages vers un autre espace d'adressage —, et un canal entrant (*channel in*) — qui permet de recevoir des messages en provenance d'autres espaces d'adressage.

Ré-ingénierie de JORAM Cette section présente une expérimentations qui a été réalisée à l'aide de Dream : la ré-ingénierie de la plate-forme ScalAgent qui supporte l'exécution de JORAM [joram 2002]. Nous commençons par décrire la plate-forme ScalAgent existante, puis nous présentons son implémentation à l'aide de Dream.

– *La plate-forme ScalAgent* [Quéma et al. 2004] permet le déploiement et l'exécution d'agents. Les agents sont des objets réactifs qui se comportent conformément au modèle "événement → réaction" [Agha 1986]. Les agents sont persistants et ont des réactions atomiques. La création, l'exécution et les communications des agents sont prises en charge par un MOM. Celui-ci est constitué d'un ensemble de serveurs d'agents organisés en bus. Comme on peut le voir sur la figure 3.17, chaque serveur d'agents est constitué de trois éléments architecturaux : l'*engine* est en charge de la création et de l'exécution des agents. Il effectue, en boucle, un ensemble d'instructions consistant à prendre un message dans le *conduit* et faire réagir l'agent destinataire. Il garantit la persistance des agents et l'atomicité de leurs réactions. Le *conduit* route les messages de l'*engine* vers les *networks*. Les *networks* assurent la transmission fiable et causalement ordonnée des messages.

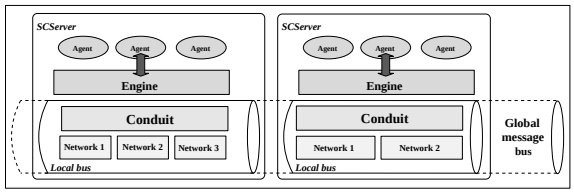

Fig. 3.17 – Deux serveurs d'agents interconnectés

– *Ré-ingénierie de l'intergiciel ScalAgent avec Dream*
L'intergiciel ScalAgent a été ré-ingénieré à l'aide de Dream (figure 3.18). Ses principales structures (*networks*, *engine* et *conduit*) ont été préservées de manière à faciliter la comparaison. L'*engine* est un composite qui comprend deux parties. La première partie traite des messages Dream. Elle est constituée d'une file de messages — qui stocke les messages entrants — et d'un composite (`AtomicityProtocol`) qui garantit l'atomicité de la réaction des agents. La seconde partie correspond au composite `Repository`. Celui-ci est en charge de la création et de l'exécution des agents. La figure représente également deux *networks* typiques. Les deux sont des composites qui encapsulent un canal entrant (`TCPChannelIn`), un canal sortant (`TCPChannelOut`), et un transformateur (`DestinationResolver`) en charge d'ajouter l'adresse IP et le numéro de port du destinataire du message. Le *network* 1 encapsule deux composants supplémentaires : le `CausalSorter` garantit l'ordonnancement causal des messages échangés. La file de messages permet de découpler les flots d'exécution de l'*engine* et du *network*. Enfin, le *conduit* est implémenté par un routeur dont l'algorithme de routage est basé sur l'identifiant de l'*engine* auquel le message est destiné.
– *Gain en configurabilité*. L'implémentation Dream de la plate-forme ScalAgent apporte de nombreux bénéfices en termes de configurabilité : il est aisé de changer les propriétés non fonctionnelles fournies par le MOM (e.g. atomicité, ordre causal). Par ailleurs, il est possible de modifier le nombre de composants actifs s'exécutant dans un serveur d'agents. L'architecture que nous avons présentée sur la figure 3.18 fait intervenir trois composants actifs pour un serveur d'agents avec un seul *network*. Il est possible d'obtenir une implémentation mono-threadée en supprimant les files de messages de l'*engine* et du *network*. Enfin, il est possible d'adapter la plate-forme à des environnements contraints. Ainsi, [Leclercq et al. 2005a] présente une modifica-

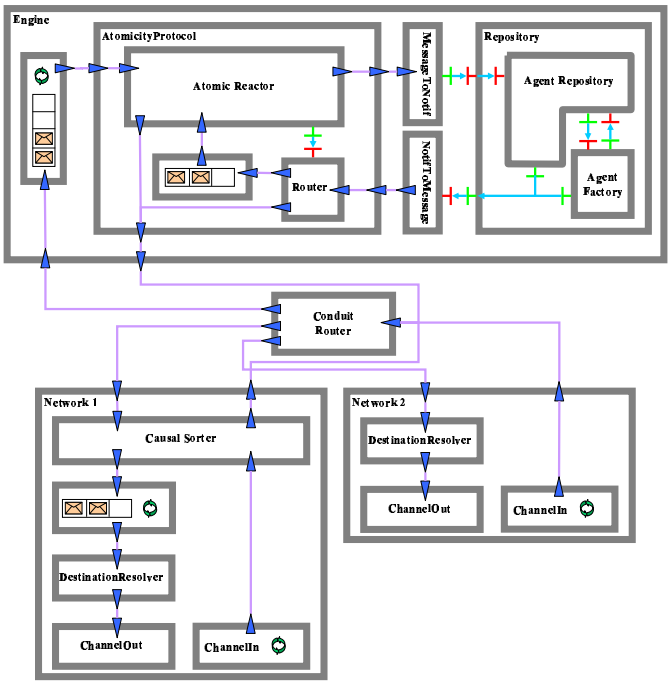

Fig. 3.18 – Architecture Dream d'un serveur d'agents

tion du serveur d'agents permettant son déploiement sur un équipement mobile.

– *Comparaison des performances.* Des mesures de performances ont montré que les performances obtenues par la ré-ingénierie à l'aide de Dream étaient tout à fait comparables à celles obtenues par la plate-forme ScalAgent. Par ailleurs, ces mesures ont montré qu'il était possible d'avoir des gains de performances significatifs en configurant le MOM de façon adéquat. Par exemple, la réduction du nombre de composants actifs peut permettre des améliorations de performances de l'ordre de 15% pour certaines applications.

3.4.2 Autres bibliothèques

Plusieurs autres bibliothèques de composants Fractal ont été développées. Il s'agit dans la plupart des cas de fournir des briques de base pour la construction d'intergiciels.

Parmi les bibliothèques existantes nous pouvons citer :

– Le projet Perseus [Chassande-Barrioz and Dechamboux 2003] définit une bibliothèque de composants Fractal pour la construction de services de persistance de données. Il fournit des composants de base pour la gestion de cache, de réserves (*pools*), de politiques de concurrence (mutex, lecteurs/écrivain FIFO, lecteurs/écrivain avec priorité aux écrivains, optimiste), de journalisation et de gestion de dépendances. Les composants Perseus sont utilisés notamment pour construire Speedo [Chassande-Barrioz 2003] qui est une implémentation des spécifications JDO [JDO 2002] de Sun.

– CLIF [Dillenseger 2004] est un canevas logiciel pour l'injection de charge. Il consti-

tue un des éléments du projet ObjectWeb JMOB pour la mesure des performances des intergiciels. CLIF permet de déployer, contrôler et superviser des injecteurs de charge. Il permet également de gérer des sondes pour mesurer l'état de ressources systèmes telles que le CPU, la mémoire ou tout autre type de ressources logicielle ou matérielle. Chaque sonde est représentée par un composant Fractal. L'implémentation de ce composant repose sur des mécanismes systèmes externes à Fractal comme par exemple l'utilisation de `/proc` sous Unix. Des composants sont également disponibles pour décrire les scénarios de test de charge, la collecte, le stockage et l'analyse du résultat des tests.

– Le projet GoTM [Rouvoy 2004] est une bibliothèque de composants Fractal pour la construction de moniteurs transactionnels. Il permet de construire des moniteurs transactionnels conformes à différents standards (par exemple JTS, OTS ou WS-AT) et supportant différentes formes de protocoles de validation (2PC, 2PC presume commit, 2PC presume abort, etc.).

3.5 Comparaison

De nombreux modèles de composants ont été proposés ces dix dernières années. Dans le domaine des intergiciels, les propositions peuvent être classées selon qu'ils sont issus d'initiatives industrielles, de la communauté du logiciel libre ou d'équipes de recherche académiques.

Initiatives industrielles La première catégorie comprend les modèles issus d'initiatives industrielles comme EJB (voir chapitre **??**), COM+/.NET, CCM ou OSGi (voir chapitre **??**). Les caractéristiques de ces modèles varient, allant de composants avec des propriétés d'introspection (COM+), à des composants avec liaisons et cycle de vie (OSGi et CCM). Le modèle Fractal est, quant à lui, entièrement réflexif et introspectable, autorise des liaisons selon différentes sémantiques de communication et fournit un modèle hiérarchique autorisant le partage. Par ailleurs, EJB, CCM et COM+/.NET sont tous les trois accompagnés d'un modèle figé de services techniques offerts par des conteneurs aux composants. Par contraste, Fractal est basé sur un modèle ouvert, dans lequel les services techniques sont entièrement programmables via la notion de contrôleur.

Plus récemment, sous l'impulsion d'IBM, l'initiative SCA [SCA 2005] a défini un modèle de composants pour des architectures orientées services. Le projet Tuscany [Tuscany 2006] fournit une implémentation en Java et en C++ de ces spécifications. SCA propose la notion de liaison pour l'assemblage et de module pour la création de hiérarchies de composants. SCA n'impose pas une forme prédéterminée de liaison, mais autorise l'utilisation de différentes technologies, comme SOAP, JMS ou IIOP pour mettre en œuvre ces liaisons. De même, Fractal autorise différents types de liaisons et n'impose pas de technologie particulière.

Initiatives du monde du logiciel libre Dans la catégorie des modèles de composants issus d'initiatives de type logiciel libre, nous pouvons notamment citer Avalon [Avalon] qui est un modèle de composants général, Kilim [ObjectWeb 2004], Pico [PicoContainer 2004] et Hivemind [Hivemind 2004] qui ciblent la configuration de logiciel, Spring [Spring 2004],

Carbon [Carbon 2004] et Plexus [Plexus 2004] qui ciblent les conteneurs de composants de type EJB. De manière générale, ces modèles sont moins ouverts et extensibles que ne l'est Fractal.

Initiatives académiques Plusieurs modèles de composants ont également été proposés par des équipes de recherche académiques. Sans être exhaustif, on peut citer ArchJava [Aldrich et al. 2002], FuseJ [Suvée et al. 2005], K-Component [Dowling and Cahill 2001], OpenCOM v1 [Clarke et al. 2001] et v2 [Coulson et al. 2004].

OpenCOM est certainement le modèle le plus proche de Fractal. Il cible les systèmes devant être reconfigurés dynamiquement et en particulier les systèmes d'exploitation, les intergiciels, les PDA et les systèmes embarqués. Au niveau applicatif, les composants OpenCOM fournissent des interfaces et requièrent des réceptacles. L'architecture d'une application OpenCOM est introspectable et peut être modifiée dynamiquement.

Depuis la version 2, OpenCOM fournit les quatre notions suivantes : capsule, caplet, loader et binder. Une capsule est l'entité qui contient et gère les composants applicatifs. Une caplet est une partie d'une capsule qui contient un sous-système de l'application. Les binders et les loaders sont des entités de première classe qui offrent différentes sémantiques de chargement et de liaison pour les composants. Les caplets, les loaders et les binders sont eux-mêmes des composants.

Comparé à Fractal, les capsules et les caplets sont similaires aux composants composites. Les binders et les loaders sont quant à eux comparables aux contrôleurs. Cependant les contrôleurs Fractal ne sont pas limités à ces deux types de propriétés extra-fonctionnelles et peuvent englober d'autres services techniques. Par ailleurs, si comme dans Fractal/AOKell les binders et les loaders sont aussi des composants, Fractal/AOKell va au-delà et permet de réifier complètement l'architecture de contrôle sous la forme d'un assemblage de composants.

3.6 Conclusion

Ce chapitre a présenté le système de composants Fractal, les principales plates-formes le mettant en œuvre, le langage de description d'architecture et les bibliothèques permettant de développer des systèmes à base de composants Fractal.

La section 3.1 est consacrée à la présentation du modèle de composant Fractal. C'est un modèle hiérarchique au sens où les composants peuvent être soit primitif, soit composite et contenir des sous-composants (primitifs ou composites). Deux parties sont mises en avant dans un composant Fractal : le contenu et la membrane. Cette dernière fournit un niveau méta de contrôle et de supervision du contenu. Elle est composée d'entités élémentaires, les contrôleurs, qui implémentent des interfaces dites de contrôle. Le modèle de composant Fractal est ouvert au sens où il ne présuppose pas un ensemble fini et figé de contrôleurs : de nouveaux contrôleurs peuvent être ajoutés par les développeurs en fonction des besoins. De même, la granularité des contrôleurs est quelconque, allant d'un simple service de gestion de noms à des services plus complexes de gestion de persistance ou de transaction. Un composant Fractal est une entité logicielle qui possède des interfaces fournies (dite serveur) et/ou des interfaces requises (dites client). Le concept de liaison permet de définir

des chemins de communication entre interfaces clientes et serveurs. La granularité des liaisons est également quelconque allant d'une simple référence dans l'espace d'adressage courant à des liaisons plus complexes mettant en œuvre des mécanismes de communication distante.

Le modèle Fractal est indépendant des langages de programmation. La section 3.2 a présenté deux plates-formes, Julia et AOKell, mettant en œuvre ce modèle pour le langage Java. D'autres plates-formes existent pour les langages Smalltalk, C, C++ et les langages de la plate-forme .NET. Julia est la plate-forme de référence du modèle Fractal. Le développement des contrôleurs se fait à l'aide d'un système de classes *mixin*. AOKell a été développé par la suite et apporte une approche à base de composants de contrôle pour le développement des membranes. Celles-ci sont des assemblages de composants de contrôle qui gèrent et administrent les composants du niveau de base. Les composants de contrôle sont eux-mêmes des composants Fractal qui sont contrôlés de façon ad-hoc. Finalement, AOKell utilise des techniques issues de la programmation orientée aspect pour l'intégration des niveaux de base et de contrôle.

L'API Fractal permet de construire d'assembler des composants élémentaires afin de construire des applications complètes. Le langage de description d'architecture Fractal ADL (voir section 3.3) permet de décrire ces architectures de façon plus concise qu'avec une simple API. Fractal ADL est un langage ouvert basé sur XML. Contrairement à nombre d'ADL existants, la DTD de Fractal ADL n'est pas figée et peut être étendue avec de nouvelles balises permettant d'associer des caractéristiques supplémentaires aux composants. Un mécanisme d'extension permet de définir les traitements à exécuter lorsque ces nouvelles balises sont rencontrées. Notons enfin que l'outil Fractal ADL d'analyse et d'interprétation d'un assemblage est lui même une application Fractal (donc un assemblage de composant Fractal).

Plusieurs bibliothèques de composants sont disponibles pour faciliter la tâche des développeurs Fractal. Nous en avons présentées quelques unes dans la section 3.4, dont Dream, qui permet de développer des intergiciels. Nous aurions pu également mentionner les outils qui existent autour de Fractal, comme Fractal GUI qui permet de concevoir graphiquement une architecture de composants et de générer des squelettes de code, Fractal Explorer [Merle et al. 2004] qui est une console graphique d'administration d'applications Fractal, Fractal RMI qui permet de construire des assemblages de composants distribués communicants via un mécanisme d'invocation de méthodes à distance et Fractal JMX pour l'administration de composants à l'aide de la technologie JMX.

Après une première version stable diffusée en juillet 2002, la version 2 des spécifications Fractal parue en septembre 2003 a permis d'en consolider l'assise. Fractal est maintenant une spécification stable et mature grâce à laquelle de nombreux systèmes et applications ont pu être développés. Par ailleurs, de nombreuses activités de recherche sont conduites autour de Fractal. Sans être exhaustif, nous pouvons citer les travaux sur les approches formelles et les calculs de composants, la vérification de comportements, la sécurité et l'isolation des composants, les systèmes autonomiques, l'unification des styles de développement à base de composants et d'aspects, la gestion de la qualité de service ou les composants pour les grilles de calcul. Au delà de leur intérêt propre, ces travaux devraient également servir de terreau pour compléter Fractal sur plusieurs points qui sont actuellement peu pris en compte comme le paquetage (*packaging*), le déploiement, la sémantique des interfaces collection ou les modèles de composants pour les architectures orientées service.

Bibliographie

[Agha 1986] Agha, G. A. (1986). Actors : A Model of Concurrent Computation in Distributed Systems. In *The MIT Press, ISBN 0-262-01092-5*, Cambridge, MA.

[Aldrich et al. 2002] Aldrich, J., Chambers, C., and Notkin, D. (2002). ArchJava : Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 187–197. ACM Press.

[Ancona et al. 2000] Ancona, D., Lagorio, G., and Zucca, E. (2000). A Smooth Extension of Java with Mixins. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00)*, pages 154–178, Sophia Antipolis and Cannes, France.

[ASM 2002] ASM (2002). ASM : a Java Byte-Code Manipulation Framework. The ObjectWeb Consortium, `http ://www.objectweb.org/asm/`.

[Avalon] Avalon. The Apache Avalon Project. http ://avalon.apache.org.

[Banavar et al. 1999] Banavar, G., Chandra, T., Strom, R., and Sturman, D. (1999). A Case for Message Oriented Middleware. In *13th Int. Symp. on Distributed Computing (DISC), LNCS 1693*.

[Baude et al. 2003] Baude, F., Caromel, D., and Morel, M. (2003). From distributed objects to hierarchical grid components. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'03)*.

[Blakeley et al. 1995] Blakeley, B., Harris, H., and Lewis, J. (1995). *Messaging and Queueing Using the MQI : Concepts and Analysis, Design and Development*. McGraw-Hill.

[Bouraqadi 2005] Bouraqadi, N. (2005). *FracTalk : Fractal Components in Smalltalk*. `csl.ensm-douai.fr/FracTalk`.

[Bracha and Cook 1990] Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming : Systems, Languages and Applications (ECOOP/OOPSLA'90)*, volume 25 of *SIGPLAN Notices*, pages 303–311. ACM Press.

[Bruneton et al. 2003] Bruneton, E., Coupaye, T., and Stefani, J. (2003). The Fractal Component Model. Technical report, Specification v2, ObjectWeb Consortium, http ://www.object.org/fractal.

[Burke 2003] Burke, B. (2003). It's the aspects. Java's Developer's Journal. `www.sys-con.com/story/?storyid=38104&DE=1`.

[Carbon 2004] Carbon (2004). *The Carbon Project*. `carbon.sourceforge.net`.

[Chassande-Barrioz 2003] Chassande-Barrioz, S. (2003). *The Speedo Project*. ObjectWeb. `speedo.objectweb.org`.

[Chassande-Barrioz and Dechamboux 2003] Chassande-Barrioz, S. and Dechamboux, P. (2003). *The Perseus Project*. ObjectWeb. `perseus.objectweb.org`.

[Clarke et al. 2001] Clarke, M., Blair, G., Coulson, G., and Parlavantzas, N. (2001). An efficient component model for the construction of adaptive middleware. In *Proceedings of Middleware'01*.

[Coulson et al. 2004] Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., and Uyema, J. (2004). A component model for building systems software. In *Proceedings of the IASTED Software Engineering and Applications (SEA'04)*.

[Dillenseger 2004] Dillenseger, B. (2004). *The CLIF Project*. ObjectWeb. `clif.objectweb.org`.

[Dowling and Cahill 2001] Dowling, J. and Cahill, V. (2001). The K-Component architecture meta-model for self-adaptative software. In *Proceedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 81–88. Springer-Verlag.

[Dumant et al. 1998] Dumant, B., Horn, F., Dang Tran, F., and Stefani, J.-B. (1998). Jonathan : an open distributed processing environment in Java. In *Proceedings of Middleware'98*. `jonathan.objectweb.org`.

[Escoffier and Donsez 2005] Escoffier, C. and Donsez, D. (2005). FractNet : An implementation of the Fractal component model for .NET. In *2ème Journée Francophone sur Développement de Logiciels par Aspects (JFDLPA'05)*. `www-adele.imag.fr/fractnet/`.

[Fassino et al. 2002] Fassino, J.-P., Stefani, J.-B., Lawall, J., and Muller, G. (2002). Think : A software framework for component-based operating system kernels. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86.

[Fassino 2001] Fassino, J.-Ph. (2001). THINK *: vers une architecture de systèmes flexibles*. PhD thesis, École Nationale Supérieure des Télécommunications, Paris.

[Fleury and Reverbel 2003] Fleury, M. and Reverbel, F. (2003). The JBoss extensible server. In *Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03)*, volume 2672 of *Lecture Notes in Computer Science*, pages 344–373. Springer-Verlag.

[Hivemind 2004] Hivemind (2004). *The Hivemind Project*. `jakarta.apache.org/hivemind`.

[JDO 2002] JDO (2002). *Java Data Objects*. Sun Microsystems. `java.sun.com/products/jdo/`.

[joram 2002] joram (2002). JORAM. ObjectWeb, `http ://www.objectweb.org/joram/`.

[Layaida et al. 2004] Layaida, O., Atallah, S. B., and Hagimont, D. (2004). A framework for dynamically configurable and reconfigurable network-based multimedia adaptations. *Journal of Internet Technology*, 5(4) :57–66. Special Issue on Real Time Media Delivery over the Internet.

[Leclercq et al. 2005a] Leclercq, M., Quéma, V., and Stefani, J.-B. (2005a). DREAM : un canevas logiciel à composants pour la construction d'intergiciels orientés messages dynamiquement configurables. In *4ème Conférence Francophone autour des Composants Logiciels (avec CFSE-RENPAR 2005)*, Le Croisic, France.

[Leclercq et al. 2005b] Leclercq, M., Quema, V., and Stefani, J.-B. (2005b). DREAM : a component framework for the construction of resource-aware, configurable middleware. *IEEE Distributed Systems Online*, 6(9).

[Merle et al. 2004] Merle, P., Moroy, J., Rouvoy, R., and Contreras, C. (2004). *Fractal Explorer*. ObjectWeb. `fractal.objectweb.org/tutorials/explorer/index.html`.

[msmq 2002] msmq (2002). Microsoft Message Queuing (MSMQ). Microsoft, http ://www.microsoft.com/msmq/.

[ObjectWeb 2004] ObjectWeb (2004). *The Kilim Project*. `kilim.objectweb.org`.

[PicoContainer 2004] PicoContainer (2004). *The PicoContainer Project*. `www.picocontainer.org`.

[Plexus 2004] Plexus (2004). *The Plexus Project*. `plexus.codehaus.org`.

[Quéma et al. 2004] Quéma, V., Balter, R., Bellissard, L., Féliot, D., Freyssinet, A., and Lacourte, S. (2004). Scalagent : une plate-forme à composants pour applications asynchrones. *Technique et Science Informatiques*, 23(2).

[Quéma 2005] Quéma, V. (2005). Vers l'exogiciel - une approche de la construction d'infrastructures logicielles radicalement configurables. Thèse de Doctorat de l'Institut National Polytechnique de Grenoble.

[Rogerson 1997] Rogerson, D. (1997). *Inside COM*. Microsoft Press, Redmond, USA.

[Rouvoy 2004] Rouvoy, R. (2004). *The GoTM Project*. ObjectWeb. `gotm.objectweb.org`.

[SCA 2005] SCA (2005). *Service Component Architecture Assembly Model Specification*. `www-128.ibm.com/developerworks/library/specification/ws-sca/`.

[Seinturier et al. 2006] Seinturier, L., Pessemier, N., Duchien, L., and Coupaye, T. (2006). A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153. Springer.

[Spring 2004] Spring (2004). *The Spring Framework*. `www.springframework.org`.

[Strom et al. 1998] Strom, R., Banavar, G., Chandra, T., Kaplan, M., Miller, K., Mukherjee, B., Sturman, D., and Ward, M. (1998). Gryphon : An Information Flow Based Approach to Message Brokering. In *Proceedings of ISSRE'98*.

[Suvée et al. 2005] Suvée, D., Vanderperren, W., and Jonckers, V. (2005). FuseJ : An architectural description language for unifying aspects and components. In *Workshop Software-engineering Properties of Languages and Aspect Technologies (SPLAT) at AOSD'05*. `ssel.vub.ac.be/Members/dsuvee/papers/splatsuvee2.pdf`.

[Szyperski 2002] Szyperski, C. (2002). *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley. 2nd ed., 589 pp.

[Tuscany 2006] Tuscany (2006). *The Tuscany Project*. `incubator.apache.org/projects/tuscany.html`.

[van Renesse et al. 2003] van Renesse, R., Birman, K., and Vogels, W. (2003). Astrolabe : A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2).

[W3C-DOM 2005] W3C-DOM (2005). The Document Object Model (DOM), W3C Consortium. `http ://www.w3.org/DOM`.