ARA Sécurité, Systèmes embarqués & Intelligence ambiante

Projet REVE

safe Reuse of Embedded components in heterogeneous enVironmEnts

Document D2.3

Modélisation formelle du contrôle en Fractal

Auteur(s) : M. Simonot, M.-V. Aponte (CNAM-CEDRIC)

31/12/2007

Modélisation formelle du contrôle en Fractal

Marianne Simonot et Virginia Aponte

22 février 2008

1 Fractal

Fractal [4, 7] est un modèle à composants [16] extensible qui peut être utilisé avec plusieurs langages de programmation pour spécifier, implanter, configurer et déployer des applications à base de composants logiciels. Le modèle Fractal repose sur le principe de la *séparation des préoccupations :* l'idée étant de séparer les différents aspects d'une application dans des entités exécutables différentes, ainsi que dans différents morceaux de code. Ces aspects sont soit d'ordre *fonctionnel*, à savoir, d'implantation des services offerts par l'application, soit de l'ordre du *contrôle* de l'application, autrement dit, concernent les possibilités de configuration, de gestion du cycle de vie, de la sécurité ou autres services non fonctionnels de l'application.

Le principe de séparation des préoccupations est appliqué à la structure des composants : on peut voir un composant Fractal comme formé d'une partie *contenu* qui implante les services fonctionnels du composant, et d'une partie *contrôleur* chargée d'offrir les services de gestion des aspects non fonctionnels.

Le modèle Fractal ne fait aucun choix quant au paradigme ni quant au langage d'implantation des composants. Il est défini dans *The Fractal Component Model* [4], par un ensemble de spécifications *d'interfaces de contrôle* pouvant être implantées par les composants d'une application. Chaque *interface de contrôle* est spécifiée sous la forme d'une *interface Java*, accompagnée de commentaires en anglais. Ainsi, la description du modèle à composants Fractal est formée par l'*l'API Fractal* plus des commentaires. Cette forme de spécification a l'avantage de permettre à Fractal de rester indépendant du langage d'implantation des composants et lui ainsi donne toute sa généralité. Elle a en revanche l'inconvénient de rester relativement informelle.

2 Ce document

Dans ce document, nous présentons un modèle formel M_F qui rend compte des aspects de contrôle de Fractal, autrement dit, des entités et opérations en rapport avec l'API Fractal. Ces opérations agissent fondamentalement sur les liaisons entre composants et ont pour but de permettre leur re-configuration dynamique. Tout comment le modèle Fractal, nous ne reposons sur aucun choix de langage cible[1]. Nous rappelons que ce choix, loin d'être restrictif, nous permet de rester indépendants vis-à-vis du langage concret d'implantation des composants, tout comme l'est la définition de Fractal.

Plus précisément, nous ignorons le code fourni par l'utilisateur lors d'une spécification de composant (dans l'ADL[2]), ainsi que la création d'instances et le déploiement de composants. Enfin, notre modèle ne prend pas en compte pour l'instant la spécification de composants composites, ni celle d'une relation de compatibilité entre types de composants.

Dans [4] on distingue trois niveaux de fonctionnalités de contrôle pouvant être fournis pour la spécification et programmation de composants. On ne s'intéresse ici qu'au plus haut niveau de contrôle : le niveau de *configuration*. Les interfaces de contrôle manipulent des notions propres à Fractal, telles que les ensembles de services offerts ou requis d'un composant, leurs types, le statut d'un composant, etc. Ce sont ces notions et leurs relations que nous cherchons à modéliser.

Ce travail est un cahier des charges formel de Fractal. Nos spécifications établissent les contrats[3] des méthodes de contrôle de Fractal. A ce titre, comme tout contrat, il s'adresse au clients comme aux fournisseurs du contrat :

- Aux clients de l'API, c.a.d., à ceux qui écrivent des systèmes dans une des implantations de Fractal mettant en jeu des services de re-configuration dynamique, il fournit un cadre formel permettant d'exprimer et d'analyser des propriétés de leurs systèmes portant sur la re-configuration dynamique et l'introspection.

- Aux fournisseurs de l'API, c'est-à-dire, à ceux qui développent les implantations de Fractal, il fournit la sémantique précise des méthodes qu'ils doivent implanter et le moyen d'en prouver la correction.

Pour parvenir à nos fins, nous suivons le document *The Fractal Component Model* [4] au plus près, en le formalisant.

[1]En dehors des contraintes habituelles sur l'existence d'une relation de sous-typage entre types pour les interfaces des composants.

[2]Il s'agit d'un *langage de description d'architectures* utilisé dans Fractal pour spécifier les types des composants et leurs contraintes de composition [12, 7].

[3]Au sens de la programmation par contrats [3].

3 Qu'est ce qu'un composant Fractal ?

Un composant est une entité exécutable [16] pouvant offrir des services exécutables par le composant, ou en requérir en provenance d'autres composants. Un composant Fractal possède des points d'accès aux services qu'il fournit, appelés *interfaces serveur* et des points d'accès aux services qu'il requiert, appelés *interfaces client*. Chaque interface du composant regroupe une collection de services, requis ou offerts selon que le rôle spécifié pour l'interface est *client* ou *serveur*.

Toute invocation de service se fait nécessairement via une des interfaces du composant. Si le service est *offert par le composant*, l'invocation se fait via une de ses interfaces serveur. Elle correspond à l'exécution du code qui implante ce service dans ce même composant (qui est, rappelons-le, une entité exécutable). Si le service est *requis par le composant*, l'invocation se fait via une de ses interfaces client, et correspond à l'exécution du code qui implante ce service par le composant externe qui fourni ce service au composant appelant. Une interface en Fractal est donc un point d'accès pour l'exécution d'un service, soit du composant, s'il s'agit d'un service de l'une de ses interfaces offertes, soit d'un autre composant, s'il fait partie d'une interface requise.

Les interfaces d'un composant sont elles mêmes typées par ce que l'on nomme une *interface langage*, qui n'est autre chose que le type d'une collection de méthodes, par exemple, une interface dans un langage objet, une signature de module, etc. Afin d'éviter la surcharge de vocabulaire, nous adoptons le terme *port* pour les interfaces et *signature* pour les interfaces langage. Les ports d'un composant peuvent être ou non liés à d'autres composants, l'absence de liaison pouvant provoquer des erreurs. Un composant a également un *statut* qui peut être *démarré* ou *stoppé*.

Un composant Fractal possède aussi une *membrane* regroupant les services extra fonctionnels qu'il est susceptible d'offrir : arrêt, démarrage, nommage, liaison d'une de ses interfaces clientes, etc. La membrane fournit donc les services de contrôle permettant d'assurer les activités du composant liées à l'introspection et à la re-configuration dynamique. Les services offerts par la membrane se situent ainsi à un niveau méta par rapport aux services métiers. Malgré cela, Fractal regroupe les services offerts par la membrane dans des interfaces pré-définies qui se manipulent de la même façon que les interfaces métiers du composant.
Dans le travail que nous présentons, nous avons choisi de mettre en valeur l'aspect méta de la membrane. Nous ne modélisons pas les services métiers et les services de contrôle de façon homogène. Plus précisemment, notre travail donne un statut formel à la notion de port, mais les services de contrôle du composant ne seront pas regroupés dans des ports pré-définis de notre modèle.

4 Choix de modélisation

Notre ambition est d'élaborer une spécification formelle du modèle Fractal suffisamment souple pour permettre de multiples usages : preuve d'un programme Fractal avec re-configuration dynamique, étude de propriétés sémantiques du modèle Fractal, etc. Pour parvenir à cette fin, nous choisissons le formalisme le plus commun, à savoir la logique du premier ordre. Lorsqu'un usage particulier sera mis en oeuvre, la traduction dans le formalisme dédié ne posera ainsi pas de problème.

4.1 Approche par les modèles

Nous adoptons l'approche classique par les modèles [**?**], c'est à dire que nous allons définir :

1. Un langage du premier ordre L_F , c'est-à-dire, un ensemble de symboles de relation et de fonction d'arité fixée. Les relations d'arité un représentent les entités du modèle.

   Par exemple, notre modélisation de Fractal manipulera des entités comme les composants, ou encore les ports. Notre langage L_F possédera donc deux symboles de relation unaires $Comp$ et $Port$. Pour exprimer le fait que les ports appartiennent à un composant, L_F possédera une fonction binaire *compDe*. Les liaisons entre ports nécessitent une fonction $liaisonDe$.

2. Une théorie T_F sur le langage L_F : un ensemble de formules exprimant les liens entre les entités, relations et fonctions du système.

   Par exemple, nous exprimerons le fait que les composants et les ports sont des entités distinctes par la formule : $\forall x.Comp(x) \rightarrow \neg Port(x)$. Pour rendre la lecture des formules plus aisée, nous adoptons une écriture ensembliste en notant $Comp(x)$ de la façon suivante : $x \in Comp$. Ainsi, le fait qu'un port appartient à un unique composant, sera exprimé en imposant que la fonction binaire *compDe* soit une fonction totale des ports vers les composants : *compDe* $\in Port \rightarrow Comp$.

   La donnée d'un langage et d'une théorie sur ce langage définit *l'ensemble des états possibles du système* modélisé. Pour Fractal, nous obtiendrons ainsi l'ensemble de toutes les instances possibles de composants et de ports Fractal. C'est ce qui est communément appelé l'*invariant du système*.

   L'ensemble des états possibles du système que nous cherchons à modéliser est obtenu en considérant l'ensemble des structures sur L_F qui sont des modèles de T_F.

   Rappelons qu'une *structure* est la donnée de :
   - un domaine $\mathcal{D}$, c'est à dire un ensemble non vide pour chacune des entités du langage.
   - d'une interprétation des symboles de fonction et de relation dans $\mathcal{D}$.

Exemple : Extrait d'une structure sur L_F qui est un modèle de la théorie T_F.
- $\mathcal{D} = \{c1, c2, p1, p2, p3, \ldots\}$
- $Port = \{p1, p2, p3\}$
- $Comp = \{c1, c2\}$
- $CompDe = \{(p1, c1), (p2, c1), (p3, c2)\}$
- $liaisonDe = \emptyset$

3. Un ensemble d'*opérations qui modifient l'état du système*, c'est-à-dire, qui font passer d'une structure à une autre. Nous utiliserons les notations usuelles avec pré-conditions et post-conditions. Par exemple, la méthode $bindFc(p_c, p_s)$ qui permet de lier un port client p_c à un port serveur p_s aura pour post-condition : $liaisonDe := liaisonDe \cup \{(p_c, p_s)\}$. Ainsi, appliquée sur p_1 et p_3, elle associera à la structure précédente la nouvelle structure qui contient :
   - $\mathcal{D} = \{c_1, c_2, p_1, p_2, p_3, \ldots\}$
   - $Port = \{p_1, p_2, p_3\}$
   - $Comp = \{c_1, c_2\}$
   - $compDe = \{(p_1, c_1), (p_2, c_1), (p_3, c_2)\}$
   - $liaisonDe = \{(p_1, p_3)\}$

   Il faudra ensuite montrer que la structure engendrée par l'application de chacune de ces opérations sur un modèle de L_F est encore un modèle de L_F.

Les deux premiers points constituent ce que l'on appelle la *partie statique du système*. Le dernier point constitue la *partie dynamique du système*.

4.2 Notations

Nous adoptons les notations suivantes :

$x \in A$	$A(x)$(pour A symbole de relation unaire du langage)
$f \in A \rightarrow B$	f est une fonction totale de A vers B
$f \in A \nrightarrow B$	f est une fonction partielle de A vers B
$f \in A \twoheadrightarrow B$	f est une injection de A vers B

Pour alléger les définition, nous procédons de la façon suivante :

1. Nous donnerons d'abord la liste des entités du modèle $E1, \ldots En$
2. Nous introduirons ensuite la liste des symboles de fonction du langage accompagnés des contraintes qui s'apparentent au typage.
3. Nous donnerons enfin les contraintes supplémentaires.

Le langage L_F de notre modèle sera constitué d'un symbole de relation unaire pour chacune des entités de la liste, plus l'ensemble des symboles de fonctions présents au point 2.

La théorie sur L_F sera constituée d'un axiome exprimant le fait que les entités sont des ensembles disjoints, plus des contraintes de typage et des contraintes supplémentaires.

Nous noterons les substitutions d'une fonction f d'une structure F par une nouvelle fonction f', par $F' = F[f \mapsto f']$. Si une nouvelle paire (x, r) est à ajouter pour f dans F, nous noterons ceci par $F[f \mapsto f \cup (x, r)]$. Si dans une fonction f nous voulons changer la valeur de $f(x)$ par une nouvelle valeur r nous noterons : $f[x \mapsto r]$.

5 Un premier modèle décentralisé M_F

Le modèle M_F que nous présentons se situe au plus près de la spécification informelle donnée dans [4]. Il respecte l'une des caractéristiques les plus remarquables de Fractal : c'est un système à composant *décentralisé*. Nous entendons par cela, le fait que les opérations de reconfiguration dynamique ne sont pas effectuées par un moniteur central qui à la connaissance globale du système, mais par les composants eux mêmes. Ceci est rendu possible par le fait que chaque composant possède une connaissance partielle de son environnement.

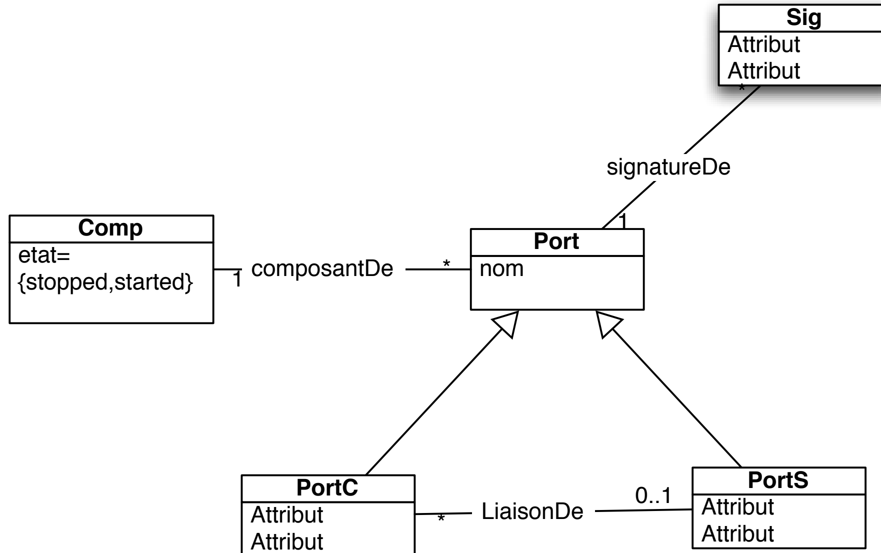
5.1 Partie statique de M_F

Nous décrivons la structure ainsi que les propriétés statiques des entités manipulées dans [4]. Chaque entité du modèle est définie par un ensemble muni de relations et contraint par un ensemble de propriétés.

5.1.1 Les entités de M_F

- Les signatures : Sig
- Les noms de ports : $NomP$
- Les ports : $Port$
- Les catégories de ports : $Role = \{client, serveur\}$
- Les composants : $Comp$
- Les statuts des composants : $Statut = \{stopped, started\}$
- L'ensemble des ports clients : $PortC$
- L'ensemble des ports serveurs : $PortS$

La figure suivante dépeint la structure des entités dans M_F :



5.1.2 Signatures

Nos signatures correspondent aux *interfaces langage* du document qui spécifie Fractal [4]. Une interface langage est au minimum un ensemble de signatures de méthodes. En fonction du langage cible, cela pourra être enrichi de spécifications. Nous aurons à définir la notion de compatibilité entre signatures. Cette notion pourra recouvrir la notion de sous-typage, mais pourquoi pas, aussi la notion de raffinement [2], de compatibilité entre contrats sémantiques [3], etc. Il s'agit donc au moins d'une relation :

$$\sqsubseteq \in Sig \longleftrightarrow Sig \qquad \text{reflexive, transitive et qui passe au contexte}$$

Les signatures sont donc représentées par n'importe quel ensemble Sig muni d'une opération $\sqsubseteq$ ainsi définie. La définition précise des signatures et de la relation $\sqsubseteq$ dépendra du choix du paradigme de programmation pour l'implantation des composants. Dans la plupart de cas, on devra exiger de ce langage l'existence d'au moins une notion syntaxique de type pour des collections de services et d'une relation de sous-typage associée, par exemple, la relation $<:$ du lambda-calcul simple avec sous-typage $\lambda_{<:}$ définie dans [14]. La relation $\tau_1 <: \tau_2$ est lue τ_1 *est un sous-type de* τ_2. De son coté, la relation de raffinement $a \sqsubseteq b$ est lue b *est un raffinement de* a. Lorsque la relation de raffinement $\sqsubseteq$ est reduite au sous-typage syntaxique, on aura que $\tau_1 <: \tau_2$ se traduit par $\tau_1 \sqsubseteq \tau_2$.

5.1.3 Ports

En termes de [4], ce sont les *interfaces des composants*, autrement dit, des points d'accès pour l'invocation des services. L'ensemble des ports $Port$ est muni des fonctions suivantes :

$$nomDe \in Port \rightarrow NomP \quad \text{Un port a un nom unique} \quad (1)$$
$$roleDe \in Port \rightarrow Role \quad \text{Un port est soit client soit serveur} \quad (2)$$
$$sigDe \in Port \rightarrow Sig \quad \text{Un port est un accès selon une unique signature} \quad (3)$$
$$compDe \in Port \rightarrow Comp \quad \text{Un port appartient à un unique composant} \quad (4)$$
$$liaisonDe \in PortC \twoheadrightarrow PortS \quad \text{Un port client peut être lié à un unique port serveur} \quad (5)$$

où $PortC$ et $PortS$ sont définis par :

$$PortC = \quad dom(roleDe \rhd \{client\}) \quad \text{Ensemble de ports clients}$$
$$PortS = \quad dom(roleDe \rhd \{serveur\}) \quad \text{Ensemble de ports serveurs}$$

Les contraintes supplémentaires doivent être vérifiées :

1. Compatibilité des liaisons entre ports :

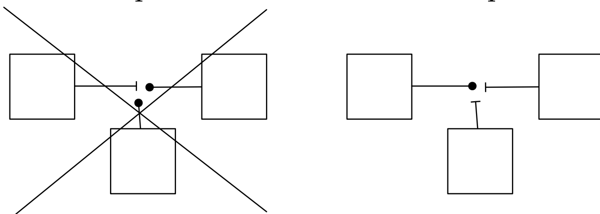
$$\forall (p_c, p_s) \in liaisonDe.sigDe(p_c) \sqsubseteq sigDe(p_s) \quad (6)$$

   A titre d'exemple, dans l'implantation Julia de Fractal, la relation $\sqsubseteq$ utilisée est le sous-typage Java.

2. Deux ports d'un même composant ne peuvent avoir le même nom :

$$\forall p_1 p_2 \in Port.(p_1 \neq p_2 \wedge compDe(p_1) = compDe(p_2)) \Rightarrow nomDe(p_1) \neq nomDe(p_2) \quad (7)$$

Remarque : (5) interdit qu'un port client soit lié à plusieurs ports serveurs, mais autorise que plusieurs ports clients soient liés à un même port serveur. De même, il est possible de lier un port client et un port serveur d'un même composant. Nous autorisons cette dernière configuration car il n'est dit nulle part dans [4] que cela ne doit pas être le cas. Les liaisons valides entre ports sont donc illustrées par le diagramme suivant :



5.1.4 Composants

Un composant est caractérisé par son état (*stopped* ou *started*) et l'ensemble de ses ports. Ainsi, l'ensemble des composants *Comp* est muni des fonctions suivantes :

$$portsDe \in Comp \rightarrow \mathcal{P}(Ports) \quad \text{Un composant a des ports} \tag{8}$$

$$statutDe \in Comp \rightarrow Statut \quad \text{Un composant est soit démarré soit stoppé} \tag{9}$$

Contraintes supplémentaires :

1. Les ports d'un composants sont les ports dont il est le propriétaire :

$$\forall c \in Comp. \forall p (p \in portsDe(c) \Leftrightarrow compDe(p) = c) \tag{10}$$

2. Contraintes sur le statut d'un composant :
   si un composant est démarré, les liaisons de ses ports clients[4] doivent être effectives. Autrement dit, les émissions et réception d'appels de méthodes métiers doivent s'exécuter normalement.

$$\forall c \in Comp.(statutDe(c) = started) \Rightarrow portsDe(c) \cap PortC \subseteq dom(liaisonDe) \tag{11}$$

Définitions :

1. Soit E l'ensemble des couples $((n, c), p)$ où p est le port de nom n dans le composant c défini par :

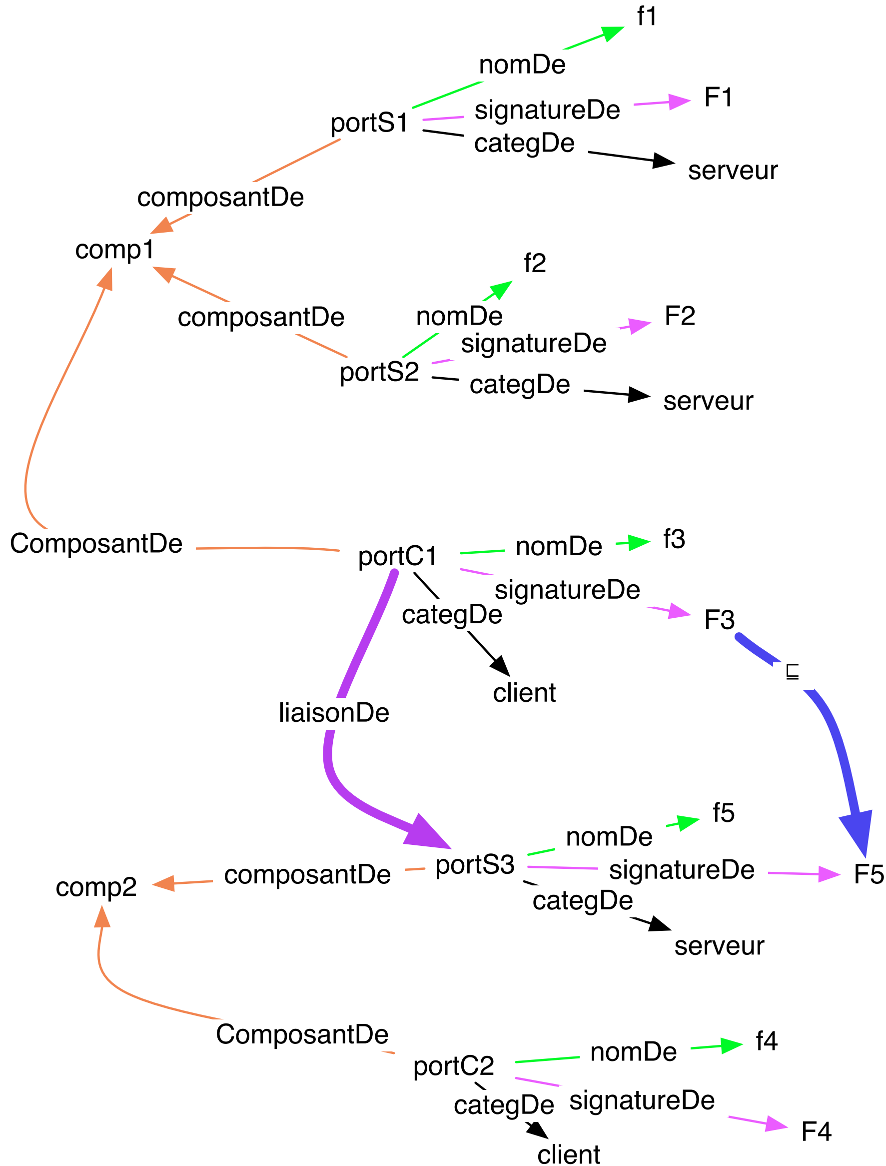
$$E \approx \{((n, c), p) \mid c \in Comp \wedge n \in NomP \\ \wedge\, p \in portsDe(c) \wedge n = nomDe(p)\}$$

   Il est facile de montrer que E est une fonction partielle de $NomP \times Comp$ vers $Port$. Il suffit en effet de montrer que E^{-1} est un sous ensemble du produit direct de *nomDe* et de *compDe* (grâce à (10)), qui est elle même une fonction totale injective (grâce a (7)). La fonction qui associe un port à un nom de port dans un composant donné, est définie par :

$$portAssocie \in NomP \times Comp \nrightarrow Port \approx E$$

La figure suivante donne un modèle M_F avec 2 composants. Il décrit un état possible d'un système avec 2 composants : l'état où il y a une liaison entre le port client de nom f3 du premier composant et le port serveur f5 du second. M_F :

[4] Cette contrainte est moins drastique en Fractal : elle ne concerne que les ports non optionnels du composant. Nous ne différencions pas pour l'instant les ports optionnels ou obligatoires.



5.2 Pourquoi M_F est décentralisé

Dans M_F, (7) est la seule contrainte globale à l'ensemble des composants. Intuitivement, la connaissance possédée par une instance d'une entité est l'ensemble des valeurs obtenues par application des fonctions sur lui même. Ainsi un composant c peut connaître ses ports : $portsDe(c)$, un port c peut connaître son propriétaire : $compDe(c)$, mais aussi l'ensemble des ports de son propriétaire : $portsDe(compDe(c))$. A partir d'un port donné, nous pouvons reconstruire une partie de l'état global du système. Ainsi, si nous nous plaçons dans une application composée d'instances de composants respectant les contraintes de M_F, nous pouvons reconstruire l'état global du système à partir des connaissances de chacune des instances. Ceci est vrai sauf pour la contrainte (7) : une instance de composant ne connaît que

les composants auxquels ses ports sont liés.
La connaissance possédée par un composant suffit à mettre en oeuvre la plupart des opérations de contrôle sur ce composant. Examinons par exemple l'opération consistant à lier un port client de nom `n` d'un composant `c` à un port serveur : Le rôle et l'état de la liaison du port de nom `n` est connue par `p` Dès lors, M_F pourra être implanté par une application objet où la plupart des méthodes de contrôle seront des méthodes fournies par les composants ou par les ports eux-mêmes. Les seules opérations où cela est impossible sont celles concernant la création de composants et la gestion des noms. Ces aspects devront être pris en charge au niveau global. A ce niveau global figurera aussi évidemment la gestion des types de composants (en Fractal, on crée des composants d'un certain type), mais nous ne prenons pas pour l'instant cela en charge dans M_F.

5.3 Partie dynamique de M_F

Nous spécifions ici les méthodes de contrôle de l'API Fractal, interface par interface. Ces méthodes sont spécifiées comme des fonctions qui modifient ou consultent l'état du modèle M_F, en donnant une pré-condition et une post-condition. Formellement, cela signifie que ces méthodes portent sur un modèle quelconque de notre théorie, c'est-à-dire sur une structure quelconque F :

$$F = < Comp, Port, PortC, PortS, Sig, NomP, Statut, Role,\\ nomDe, roleDe, sigDe, compDe, liaisonDe, portsDe, statutDe >$$

vérifiant les contraintes (1) à (11).

Nous adoptons un mode de spécification offensif : la pré-condition établit le domaine de définition de la méthode, c'est-à-dire, le domaine pour lequel n'est pas déclenché d'exception. Un programme implantant la spécification ne doit pas prendre en charge la vérification de la pré-condition. Il doit en revanche, sur le domaine spécifié par la pré-condition, produire un résultat conforme à la post-condition.

5.3.1 Interface Component

Cette interface regroupe des méthodes portant sur les composants. En plus de porter sur F, elles ont comme argument un composant, c'est-à-dire un élément de *Comp*.

`getFcInterfaces(c)` retourne les ports du composant c

```
P(Port) getFcInterfaces(Comp c);
        pre:  true
        post: return PortsDe(c)
```

`getFcInterface(n,c)` retourne le port de nom n dans le composant c et lève une exception s'il n'y a pas de port de nom n dans c.

```
Port getFcInterface(NomP n, Comp c)
            pre:  (n,c) ∈ dom(portAssocie)
            post: return portAssocie(n,c)
```

`getFcType(c)` retourne le type du composant c, notion qui n'est pour l'instant pas modelisée dans M_F. Nous ne pouvons donc pas spécifier cette méthode.

5.3.2 Interface Type

Cette interface propose une seule méthode permettant de tester si un type est un sous type d'un autre. Dans notre modèle, le seul type que nous ayons est celui des ports (sa signature). Nous pouvons donc partiellement spécifier cette méthode comme une méthode sur les signatures.

```
boolean isFcSubTypeOf(Sig S1, Sig S2);
            pre:  true
            post: return S2 ⊑ S1
```

5.3.3 Interface Interface

Cette interface permet l'introspection des ports d'un composant. Les méthodes qui la composent sont des opérations sur les ports.

`getFcItfName(p)` retourne le nom du port p dans le composant auquel ce port appartient

```
NomP getFcItfName(Port p);
            pre:  true
            post: return NomDe(p)
```

`getFcItfType(p)` retourne la signature du port p

```
Sig getFcItfType(Port p);
            pre:  true
            post: return sigDe(p)
```

`getFcItfOwner(p)` retourne le composant auquel ce port appartient

```
Comp getFcItfOwner(Port p)
            pre:  true
            post: return compDe(p)
```

`isFcInternalItf` n'est pas traitée dans la mesure où M_F ne prend pas en compte les composant composites.

5.3.4 Interface BindingController

Cette interface fournit les méthodes permettant de modifier les liaisons entre composants. Nous les modélisons comme des méthodes sur les composants.

`listFc(Comp c)` retourne les noms des ports clients du composant c

```
𝒫(NomP) listFc(Comp c);
        pre:  true
        post: return nomDe[portsDe(c) ∩ PortC]
```

`lookUpFc(c,n)` retourne le port serveur auquel le port client de nom n, et appartenant au composant c est éventuellement lié

```
PortS lookupFc(Comp c, NomP n);
      pre:  ∃ p.
                 portAssocie(n,c) = p        // n est le nom d'un port de c
                 ∧ p ∈ PortC                 // qui est un port client
                 ∧ p ∈ dom(liaisonDe)  // lié a un autre port
      post: return liaisonDe(portAssocie(n,c))
```

`bindFc(c,n,ps)` : lie le port client de nom n du composant c au port serveur s. Échoue si le composant c n'a pas de port client de nom n, ou s'il est déjà lié, ou encore s'il y a incompatibilité entre les signatures, ou finalement , si c n'est pas dans l'état *stopped*.********* (a vérifier avec Lionel).

```
void bindFc (Comp c, NomP n,  PortS p_s)
      pre : ∃p_c.
               portAssocie(n,c)=p_c         // n est le nom d'un port  de c
               ∧p_c ∈ PortC                  // qui est un port client
               ∧p_c ∉ dom(liaisonDe)     // qui  n'est pas déja lié
               ∧ sigDe(p_c) ⊑ sigDe(p_s)  // et qui est compatible avec p_s
               ∧ statutDe(c)=stopped          // c est stoppé

      post : liaisonDe = liaisonDe ∪ {(portAssocie(n,c),p_s)}
```

`unbindFc(c,n)` : délie le port client de nom n du composant c. Échoue si c n'a pas de port client de nom n, ou s'il n'est pas lié (à vérifier), ou encore si c n'est pas dans l'état stopped. (a vérifier avec Lionel).

```
void unbindFc (Comp c, NomP n)
     pre :∃p_c, p_s.
               portAssocie(n,c)=p_c          // n est le nom d'un port  de c
               ∧p_c ∈ PortC                   // qui est un port client
               ∧(p_c, p_s) ∈ liaisonDe  // qui est  lié
```

```
            ∧ statutDe(c)=stopped          // c est stoppé
    post:liaisonDe =
          liaisonDe − {(portAssocie(n,c),liaisonDe(portAssocie(n,c)))}
```

Remarques :
- Dans [4], *lookUpFc*(c, n) retourne *null* si le port client de c n'est pas lié à un port serveur, et lève une exception si le composant c n'a pas de port client de ce nom là. Pour l'instant, nous ne différencions pas *null* et exception.

5.3.5 Interface LifeCycleController

Cette interface regroupe les méthodes permettant de modifier le statut du composant. Ces méthodes sont nécessaires pour faire de la re-configuration dynamique car il n'est pas concevable par exemple d'enlever une liaison à un composant si des appels de méthodes du composant "à remplacer" sont en cours d'exécution. Ces méthodes sont volontairement sous-spécifiées dans Fractal. Leur spécification formelle l'est tout autant. Ces méthodes portent sur les composants

`getFcState(c)` retourne le l'état du composant c

```
 Statut getFcState (Comp c);
        pre:    true
        post:   return statutDe(c)
```

`startFc(c)` démarre le composant c. N'est possible que si toutes ses liaisons clientes sont faites.

```
void startFc(Comp c);
        pre  : portsDe(c) ∩ PortC ⊆ dom(liaisonDe)  // toutes les liaisons sont faites
        post : statutDe(c) = started
```

`stopFc(c)` arrête le composant c.

```
void stopFc(Comp c);
          pre: true
          post : StatutDe(c) = stopped
```

5.3.6 Méthodes supplémentaires

Les méthodes précédemment définies ne sont appelables que dans un état vérifiant la pré-condition. Il faut donc avoir accès à des méthodes permettant de tester les pré-conditions. Cela correspond aux accesseurs, ainsi qu'aux fonctions booléennes permettant de tester l'appartenance au domaine de la fonction partielle *liaisonDe*. L'API Fractal définit la plupart d'entre elles. Nous ajoutons les trois méthodes manquantes : `existePort, estLie` qui

testent respectivement qu'un port de nom n existe dans un composant c, et qu'un port est lié, et la méthode `getRole` qui retourne le rôle d'un port.

```
Role getRole (Port p);
    pre:   true
    post:  return roleDe(p)

boolean estLie (Port p);
    pre:   true
    post:  return p ∈ dom(liaisonDe)

Role existePort (Comp c, NomP n);
    pre:   true
    post:  return (n,c) ∈ dom(portAssocie)
```

6 Cohérence de M_F

Nous allons montrer que les opérations définies dans la partie dynamique de M_F établissent l'invariant. Ceci nous permettra d'assurer la cohérence du modèle. La plupart des méthodes de contrôle de l'API Fractal sont des fonctions et ne modifient pas l'état. Il n'y a donc aucune obligation de preuve les concernant. Les seules méthodes nécessitant une preuve sont `bindFc()`, `unBindFc()`, `startFc()` et `stopFc()`.

6.1 Cohérence de `bindFc()`

Soit F une structures quelconque :

$$F = < Comp, Port, PortC, PortS, Sig, NomP, Statut, Role, \\ nomDe, roleDe, sigDe, compDe, liaisonDe, portsDe, statutDe >$$

vérifiant les contraintes (1) à (11). Soit $c \in Comp$ un composant, $n \in NomP$ un nom de port et $p_s \in PortS$ vérifiant la pré-condition de `bindFc(c,n,`p_s`)`, c'est-à-dire :

$$(n, c) \in dom(portAssocie) \tag{12}$$

$$portAssocie(n, c) \in PortC \tag{13}$$

$$portAssocie(n, c) \notin dom(liaisonDe) \tag{14}$$

$$sigDe(portAssocie(n, c)) \sqsubseteq sigDe(p_s) \tag{15}$$

$$statutDe(c) = stopped \tag{16}$$

Montrons que la nouvelle structure $F' = F[liaisonDe \mapsto liaisonDe \cup \{(portAssocie(n, c), p_s)\}]$ obtenue par substitution de $liaisonDe$ par $liaisonDe \cup \{(portAssocie(n, c), p_s)$ dans F, vérifie les contraintes (1) à (11).

Le respect des contraintes (1) à (4), ainsi que (7) à (10) est trivial puisque ces contraintes ne portent pas sur $liaisonDe$, seule fonction dont la valeur dans F' est différente de la valeur dans F. Il reste donc à examiner les contraintes 5,6 et 11.

Contrainte (5)

Montrons que $liaisonDe \cup \{(portAssocie(n,c), p_s)\}] \in PortC \twoheadrightarrow portS$. Comme F vérifie (5), il reste à montrer que :

- $portAssocie(n,c) \notin dom(liaisonDe)$ qui est donné par (14)
- $portAssocie(n,c) \in PortC$ qui est donné par (13)
- $p_s \in PortS$ qui est donné par hypothèse.

Contrainte (6)

Montrons que

$$\forall (p_c, p'_s) \in liaisonDe \cup \{(portAssocie(n,c), p_s)\}.sigDe(p_c) \sqsubseteq sigDe(p'_s)$$

Pour tous les couples de $liaisonDe$ autres que $(portAssocie(n,c), p_s)$, ceci est assuré par le fait que F vérifie (6). Pour $(portAssocie(n,c), p_s)$ ceci est donné par (15).

Contrainte (11)

Montrons que

$$\forall c' \in Comp.(statutDe(c') = started) \Rightarrow portsDe(c') \cap PortC \subseteq dom(liaisonDe \cup \{(portAssocie(n,c), p_s)\})$$

Pour tous les composants autres que c, ceci est assuré par le fait que F vérifie (11). Pour c, ceci est assuré trivialement car (16) rend la prémisse fausse.

Cohérence de `unBindFc()`

Soit $c \in Comp$ un composant, $n \in NomP$ un nom de port et $p_s \in PortS$ un port vérifiant la pré-condition de `unBindFc(c,n)`, c'est-à-dire :

$$(n,c) \in dom(portAssocie) \quad (17)$$

$$portAssocie(n,c) \in PortC \quad (18)$$

$$p_s = liaisonDe(portAssocie(c,n)) \quad (19)$$

$$statutDe(c) = stopped \quad (20)$$

Contrainte (5)

Montrons que $(liaisonDe - \{(portAssocie(n,c), p_s)\}) \in PortC \twoheadrightarrow portS$. Ceci est assuré par le fait que F vérifie (5).

Contrainte (6)

Montrons que $\forall (p_c, p'_s) \in liaisonDe - \{(portAssocie(n,c), p_s)\}.sigDe(p_c) \sqsubseteq sigDe(p'_s)$. Ceci est assuré par le fait que F vérifie (6).

Contrainte (11)

Montrons que

$$\forall c' \in Comp.(statutDe(c') = started) \Rightarrow portsDe(c') \cap PortC \subseteq dom(liaisonDe - \{(portAssocie(n, c), p_s)\})$$

Pour tous les composants autres que c, ceci est assuré par le fait que F vérifie (11). Pour c, ceci est assuré trivialement car (20) rend la prémisse fausse.

6.2 Cohérence de `startFc()`

Les seules contraintes à examiner sont (9) et (11). Soit $c \in Comp$ un composant vérifiant la pré-condition de `startFc(c)`, c'est-à-dire :

$$portsDe(c) \cap PortC \subseteq dom(liaisonDe) \tag{21}$$

Contrainte (9)

Montrons que $statutDe[c \mapsto started] \in Comp \rightarrow Statut$. Ceci est assuré par le fait que F vérifie (9) et que $c \in Comp$ et $started \in Statut$.

Contrainte (11)

Montrons que

$$\forall c' \in Comp.(statutDe[c \mapsto started](c') = started) \Rightarrow portsDe(c') \cap PortC \subseteq dom(liaisonDe)$$

Pour tous les composants autres que c, ceci est assuré par le fait que F vérifie (11). Pour c ceci est assuré par (21).

6.3 Cohérence de `stopFc()`

Contrainte (9)

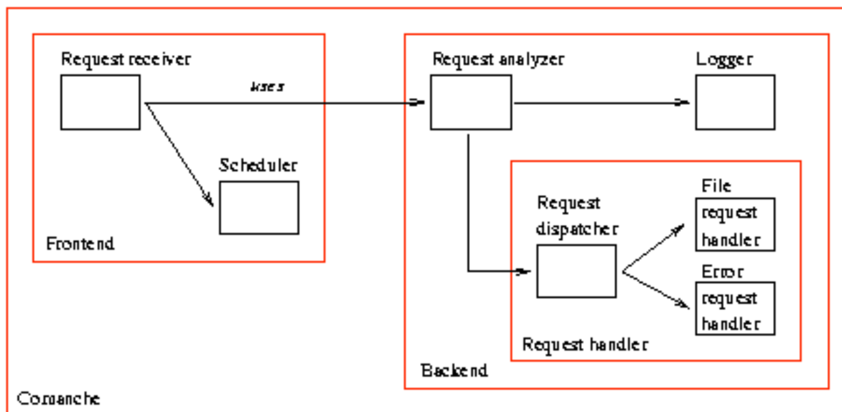
Est montrée par un raisonnement analogue à celui employé pour montrer la cohérence de `startFc`.

Contrainte (11)

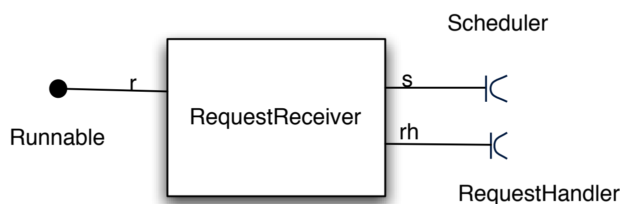
S'obtient par un raisonnement analogue à celui employé pour `bindFc`.

7 Extension du modèle par la spécification des composants utilisateurs

Dans cette partie nous examinons comment exploiter notre modélisation des opérations de contrôle afin de raissoner sur les composants définis par l'utilisateur. Nous illustrons cette partie par un exemple de serveur HTTP très simplifié extrait de [5]. Ce serveur est composé de deux grand modules : un *frontend* chargé de receptionner les requêtes, et un *backend* chargé de les traiter.



Nous modélisons les contraintes sur les ports d'un seul composant de l'application : `RequestReceiver`. Ce composant fournit un port de signature `Runnable` avec une méthode `main`, et utilise les services de deux ports, typés respectivement par les signatures `Scheduler` et `RequestHandler`. La première regroupe les services pour la création d'un *thread* associé à chaque requête, et le deuxième, ceux nécessaires à son traitement.



La spécification d'un système à base de composants se fait en Fractal en trois grandes étapes : (a) *spécification des types de composants* du système, (b) *spécification des composants du système*, donné par le nom de chaque composant, son type, et le code qui l'implante dans un langage de programmation concret, et (c) *instantiation et déploiement* des composants du système. Puisque nous ne rendons pas compte du contenu exécutable des composants

(autre que celui ayant une incidence sur leurs liens), nous ne traiterons pas l'étape (c). Pour l'étape (b), nous ignorons le code associé à l'implantation d'un composant.

En Fractal on peut spécifier les applications à base de composants via un *langage de description d'architecture* ou ADL [7, 12] permettant de décrire la forme des composants de l'application (*types des composants*) et les composants eux mêmes. Un *type de composant* dans l'ADL correspond à une collection de spécifications de ports du composant, décrivant le nom du port, sa signature, et son rôle. L'ADL donné dans [5] est basé sur la syntaxe XML. Voici par exemple, la spécification de deux types de composants : `SchedulerType` qui décrit les composants fournissant une interface `Scheduler`, et le type de composant `ReceiverType` décrivant les composants `RequestReceiver`. Elles utilisent les interfaces Java spécifiées au préalable par l'utilisateur : `Scheduler`, `Runnable` et `RequestHandler`.

```
<definition name="SchedulerType">
  <interface name="s" signature="Scheduler" role="server"/>
</definition>

<definition name="ReceiverType">
  <interface name="r" signature="Runnable" role="server"/>
  <interface name="rh" signature="RequestHandler" role="client"/>
  <interface name="s" signature="Scheduler" role="client"/>
</definition>
```

Le type de composant `SchedulerType` spécifie un unique port serveur, sans port requis. Le type de composant `ReceiverType` spécifie deux ports requis et un port offert. Dans l'ADL on peut également spécifier qu'un composant particulier, disons `rR`, appartient à un type de composant declaré par l'utilisateur :

```
<definition name="rR" extends="ReceiverType">
  <content class="comanche.RequestReceiver"/>
</definition>
```

Cette spécification indique, d'une part que le composant `rR` appartient au type `ReceiverType`, et d'autre part que le code qui implante ce composant se trouve dans une classe de nom `comanche.RequestReceiver`. Comme nous l'avons déjà indiqué, ne faisant pas de choix quant au langage cible, nous ignorons le code contenu dans cette declaration.

Les notions de *type* de port, ou de composant, ne sont pas présentes en tant qu'entités structurées dans M_F. Cependant, les principales caractéristiques du type d'un port[5] y

[5]Nous ne rendons pas compte pour l'instant de la notion de *cardinalité* d'un port, ni de sa *contingence*. Cette dernière est considérée dans M_F comme *obligatoire* pour tout port spécifié.

figurent : nom, signature et rôle. Nous pouvons donc aisemment définir l'ensemble des composants correspondants à un type de composant Fractal donné. Par exemple, l'ensemble des composants ayant pour type de composant le type `ShedulerType`, est donné par l'ensemble des composants ayant un port serveur de nom s, et de signature `Scheduler`. Formellement, il suffit d'enrichir le langage L_F des constantes `s` et `Scheduler` et d'ajouter les contraintes suivantes :

$$s \in NomP$$
$$Scheduler \in Sig$$

Puis d'ajouter des axiomes imposant l'existence d'un sous ensemble de *Comp* constitué des composants ayant pour seul port un port serveur de nom s et de signature `Scheduler` :

$$\begin{array}{rcl} \exists E_s & \subseteq & Comp. \\ & & \forall c \in E_s, \forall p \in portsDe(c).nomDe(p) = s \wedge sigDe(p) = Scheduler \\ & \wedge & roleDe(p) = serveur \wedge \exists! p \in portsDe(c) \end{array}$$

Bien que nous ne donnions pas ici d'algorithme, ces contraintes sont facilement extractibles de manière automatique à partir des spécifications ADL de l'utilisateur. Dans leur formulation actuelle il s'agit de contraintes assez fortes, qui imposent un typage exact pour la signature de chaque port, au lieu d'un sous-typage en profondeur. Une formulation plus souple ne devrait pas poser de problème particulier, mais va au délà de l'objectif de cet article. Cependant, il n'est pas clair pour nous que des vérifications de sous-typage soient éffectivement prises en charge sur les spécifications ADL de l'utilisateur, auquel cas, une spécification minimaliste comme celle-ci suffirait à rendre compte des déclarations de l'ADL, qu'il faudrait ensuite enrichir d'une définition de la relation $\sqsubseteq$ qui inclut le sous-typage.

8 Etat de l'art

8.1 Analyse formelle de l'aspect fonctionnel des composants

L'approche par composants, en ajoutant à la classique notion d'interface serveur, celle d'interface cliente, donne un statut de première classe à la notion de dépendance et de composition entre entités exécutables. Ainsi, de nombreux travaux dans le domaine de l'approche par composant s'interessent à la notion d'interface et à celle de compatibilité entre interface fournies et requises. Il s'agit alors de spécifier formellement les interfaces fonctionnelles des composant et de donner un statut formel à la notion de compatibilité entre interfaces, afin de permettre la vérification de la correction d'une composition de composants. C'est le cas par exemple des travaux de l'equipe Dédale du Loria, dirigée par J. Souquières. Dans leurs travaux [11], [10], l'architecture des composants est concue à l'aide de diagrammes UML,

puis la méthode B [2] est utilisée pour vérifier que les interfaces serveurs sont des raffinements des interfaces clientes auxquelles on veut les connecter. Ils s'attachent aussi à concevoir des *adaptateurs* [13] permettant de modifier les deux spécifications lorsqu'elles ne sont pas dans une stricte relation de raffinement.

Nos travaux ont en commun l'approche formelle : nous adoptons la même démarche par les modèles. Mais ces travaux ne s'intéressent qu'à l'aspect fonctionnel des composants et ne prennent pas en charge leurs préoccupations de contrôle. Ils ne peuvent donc exprimer de propriétés concernant la reconfiguration dynamique ou l'introspection. C'est en cela que notre approche est fondamentalement différente.

8.2 Analyse formelle de la reconfiguration dynamique

Les travaux s'attachant à prendre en charge formellement l'aspect contrôle des composants dans une approche par les modèles sont à notre connaissance plus rares. Dans [17] sont présentés des travaux proches du nôtre. Les auteurs ont pout objectif de rendre possible des vérifications automatiques à l'exécution d'applications à base de composants reconfigurables tel qu'elles existent dans le framework *OpenRec* [8]. Pour permettre l'expression et la vérification de contraintes d'architecture dans *OpenRec*, ils donnent un modèle formel qui utilise *ALLOY* [9] [1], et qui permet l'expression de l'architecture d'une application ainsi que des contraintes sur celle-ci. De plus, un analyseur Alloy est integré à OpenRec de façon à permettre des vérifications automatiques à l'éxecution.

Notre modélisation est proche : tout comme ALLOY, nous utilisons la logique du premier ordre et d'ailleurs, notre modèle pourrai ttrès aisemment se traduire en ALLOY[6]. Mais alors que dans [1]on ne modélise que les aspects concernant la liaison entre composants, notre spécification est plus détaillée puisqu'elle inclut des notions comme la compatibilité entre signatures, elle permet de parler de l'espace de nommage, de l'état d'un composant ou encore de son typage. Nous avons donc une approche plus integrée, visant à terme la prise en charge du composant dans tous ses aspects.

La seconde différence est l'aspect décentralisé de notre modèle. Leur modélisation rend compte d'un système où la gestion du contrôle est entièrement assurée par un moniteur global. Le nôtre, fidèlement à la philosophie de Fractal, rend compte d'un système où ces services sont fournis par les composants de manière "distribuée". Ainsi, nos travaux peuvent être vus comme un raffinement du leur : raffinement parce que nous ajoutons du détail concernant les composants, et raffinement parce qu'il peut sans doute être montré que notre modèle est une implantation du leur.

Dans [6], le but des auteurs est d'implanter un framework pour le développement d'applications extensibles par plugins. Avant de développer le système proprement dit, ils en

[6]Ceci est en cours de réalisation.

ont conçu une spécification formelle en ALLOY. Leur spécification est, là encore, proche de la nôtre : un composant est un ensemble de fiches, (peg) (port serveur), et de trous, (port clients). Comme dans les travaux cités précédemment, leur notion de liaison est globale : une liaison relie des fiches d'un certain composant à des trous d'un autre composant. La dynamique de leur modèle est plus pauvre que la notre : elle définit le contrat de l'opération que doit fournir leur framework : la possibilité d'ajouter un plugin dans un des trous d'un composant. Nous y retrouvons evidemment nombre d'ingrédients présents dans notre spécification de `bindfc` (par exemple, la compatibilité de liaisons). Il serait sans dout intéressant de voir comment ce framework pourrait se coder en Fractal, et d'examiner sa correction gràce à notre modèle.

9 Travaux ultérieurs

Le moèle M_F ne possède pas de notion structurée de type de port (signature) ni de type de composant. Or, ces notions sont nécessaires si l'on veut modéliser certaines opérations d'instropection de l'API qui portent sur ces entités. Il faudra en particulier modéliser une notion assez originale en Fractal, qui est celle de cardinalité d'un port : un port peut être *singleton*, et cela correspond à ce qui est modélisé dans M_F, ou peut être une *collection*, pouvant alors être potentiellement lié à plusieurs autres ports. De même, nous devrons modéliser des attributs sur les ports : un port peut être obligatoire ou optionnel. Ces notions introduisent des difficultés dans la définition de la relation de sous-typage dont nous pensons, au moins pour certaines, qu'elles pourront être traitées par une modélisation des types de composants sous forme d'enregistrements de ports, et basé sur l'algèbre d'enregistrements décrite dans [15].

D'autres notions plus ou moins complexes sont à ajouter dans notre modèle. Celles de composant composite, de sous-composants et de contenu d'un composant ne devrient pas poser de problème particulier. Il faudra ensuite modéliser le contenu exécutable d'un composant et les méthodes de création de la fabrique de composants.

Références

[1] The Alloy Analyzer. http ://alloy.mit.edu.

[2] J.-R. Abrial. *The B-book : assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.

[3] A. Beugnard, J-M. Jézequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999.

[4] E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal component model, 2004.

[5] E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal tutorial, 2004.

[6] Robert Chatley, Susan Eisenbach, and Jeff Magee. Modelling a framework for plugins. In *Specification and verification of component-based systems, September 2003*, 2003.

[7] T. Coupaye, V. Quéma, L. Seinturier, and J.-B. Stefani. Intergiciel et construction d'applications réparties, 2007.

[8] Jamie Hillman and Ian Warren. An open framework for dynamic reconfiguration. *icse*, 0 :594–603, 2004.

[9] Daniel Jackson. Alloy : a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2) :256–290, 2002.

[10] O. Kouchnarenko and A. Lanoix. How to refine and to exploit a refinement of component-based systems. In I. Virbitskaite and A. Voronkov, editors, *PSI 2006, Pesrpectives of System Informatics, 6th Int. Andrei Ershov Memorial Conf.*, volume 4378 of *LNCS*, pages 297–309, Novosibirsk, Akademgorodok, Russia, June 2006. Springer.

[11] Arnaud Lanoix, Denis Hatebur, Maritta Heisel, and Jeanine Souquières. Enhancing dependability of component-based systems. In *Reliable Software Technologies – Ada Europe 2007*, pages 41–54. Springer-Verlag, 2007.

[12] Matthieu Leclercq, Ali Erdem Ozcan, Vivien Quema, and Jean-Bernard Stefani. Supporting heterogeneous architecture descriptions in an extensible toolset. In *ICSE '07 : Proceedings of the 29th International Conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Ines Mouakher, Arnaud Lanoix, and Jeanine Souquières. Component adaptation : Specification and verification. In *11th International Workshop on Component Oriented Programming (WCOP 2006), In Conjunction of ECOOP 2006*, 2006.

[14] Benjamin C. Pierce, editor. *Types and Programming Languages.* MIT Press, 2002.

[15] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1993.

[16] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming.* Addison-Wesley Professional, December 1997.

[17] Ian Warren, Jing Sun, Sanjev Krishnamohan, and Thiranjith Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *ASE*, pages 37–46, 2006.