Une approche formelle de la reconfiguration dynamique*

Marianne Simonot — Maria-Virginia Aponte

Laboratoire CEDRIC
292 rue St. Martin – F-75141 Paris Cédex 03
{marianne.simonot, maria-virginia.aponte_garcia}@cnam.fr

* *Travail partiellement financé par le projet ANR SSIA 2005 REVE*

RÉSUMÉ. *Les applications auto-adaptatives adaptent leur comportement de façon dynamique et autonome par le biais d'opérations d'introspection, de recomposition, d'ajout et suppression de composants. Un des moyens de favoriser leur robustesse est de disposer d'un support formel permettant de modéliser ces applications, de spécifier les programmes d'adaptation, d'y exprimer des propriétés et de les vérifier. Nous proposons un cadre formel de spécification et de raisonnement sur des programmes avec reconfiguration dynamique, inspiré du modèle à composants Fractal, nommé FracL. Ce modèle possède un ensemble de primitives simple et complet permettant l'introspection et la reconfiguration au sein des systèmes à base de composants. Le cadre proposé est fondé sur une description axiomatique des primitives de Fractal en logique de premier ordre, et permet la spécification et la preuve des propriétés autant fonctionnelles que de contrôle dans ces systèmes. Notre modèle a été traduit dans l'atelier de spécification et de preuve Focal, ce qui permet à la fois d'en assurer la cohérence et de fournir un cadre outillé pour raisonner sur des applications concrètes.*

ABSTRACT. *Self-adapting software adapts its behavior in an autonomic way, by dynamically adding, suppressing and recomposing components, and by the use of computational reflection. One way to enforce software robustness while adding adaptative behavior is disposing of a formal support allowing these programs to be modeled, and their properties specified and verified. We propose FracL, a formal framework for specifying and reasoning about dynamic reconfiguration programs being written in a Fractal-like programming style. The Fractal component model supports a simple and extensible set of component reflection capabilities, enabling the definition of reconfiguration wherein component based systems. The proposed framework is founded on first order logic, and allows the specification and proof of properties concerning either functional concerns or control concerns. An encoding of our model using the Focal spec-*

ification framework, enabled us to prove its coherence and to obtain a mechanized framework for reasoning on concrete architectures.

MOTS-CLÉS : *composants, réconfiguration dynamique, Fractal, méthodes formelles, spécification des méthodes de contrôle, spécification et raisonnement sur les programmes.*

KEYWORDS: *components, dynamic reconfiguration, Fractal, formal methods, specifying and reasoning about programs.*

1. Introduction

Les systèmes informatiques actuels sont de plus en plus confrontés au besoin d'adaptation en réponse à des changements pouvant intervenir dans leur contexte d'exécution. Les applications *auto-adaptatives* sont une réponse à ce problème : conscientes de leur environnement et de leur structure, elles sont capables de changer leur comportement de façon dynamique et autonome. Pour exister, ce type d'application doit être capable durant l'exécution, de connaître son organisation (*introspection*), ainsi que d'y apporter des modifications (*reconfiguration dynamique*). Ces modifications pourront inclure le changement de paramètres de configuration, le remplacement, l'ajout ou la suppression de composants. Ces systèmes pourront ainsi embarquer des programmes chargés de la mise en oeuvre des politiques d'adaptation. Ils pourront, par exemple, recomposer dynamiquement une application afin de réduire le nombre de composants déployés face à une quantité de mémoire dégradée, ou encore, ajouter un composant pour répondre à une attaque.

La conception et la maintenance d'applications auto-adaptatives robustes est une tâche complexe. Un des moyens de maîtriser cette complexité réside dans le fait de disposer d'un support formel permettant de modéliser ces applications, de spécifier les programmes d'adaptation, d'y exprimer des propriétés et de les vérifier.

Dans cet article, nous proposons un cadre formel permettant de décrire et de raisonner sur des architectures reconfigurables et des programmes d'adaptation. Nous nous situons dans le contexte de la programmation par composants (Szyperski, 1997; McKinley *et al.*, 2004) qui se prête bien à la modélisation d'applications auto-adaptatives. Nous avons choisi de baser nos travaux sur le modèle de composants Fractal (Bruneton *et al.*, 2004) car il s'agit d'un modèle à la fois général, pleinement dynamique et réflexif. Fractal fournit au moyen d'une API les primitives permettant de mettre en oeuvre l'introspection et la reconfiguration dynamique.

Notre modélisation est fondée sur une définition en logique du premier ordre des notions de *composants* et de *systèmes à composants*. Celle-ci décrit un *état* (ensemble d'entités et de relations) et un *invariant d'état* (sous forme de prédicats). Nous spécifions ensuite un ensemble de primitives de reconfiguration et d'introspection, inspiré des *primitives de contrôle* de Fractal. Celles-ci sont vues comme des procédures qui consultent ou modifient l'état d'un système à composants et sont données sous la forme de couples de *pré-conditions* et *post-conditions*. L'ensemble de ces spécifications a été montré *cohérent* (i.e., préservant l'invariant du système). Nous terminons par la définition d'un langage minimal de *script* formé des primitives présentées, auxquelles s'ajoutent la séquence, le choix et l'itération. Ceci permet la définition de nouvelles méthodes d'introspection et de reconfiguration et forme le langage dans lequel les programmes d'adaptation peuvent être codés.

Ce cadre peut dès lors être utilisé pour décrire formellement des applications à base de composants, exprimer et vérifier des propriétés tant fonctionnelles qu'architecturales sur celles-ci. Il permet aussi de spécifier et de coder de façon incrémentale des programmes complexes de reconfiguration.

La formulation en logique de premier ordre présentée est indépendante de tout système de preuve. Ce travail a été néanmoins encodé dans l'atelier de preuve Focal. Ceci a permis de montrer la cohérence du modèle. Cela offre de plus le moyen de disposer d'un cadre outillé pour spécifier et raisonner sur des applications à composants concrètes.

La section 2 présente le modèle à composants Fractal. La section 3 présente la théorie définissant les notions de composant et de système à base de composants. La section 4 présente la spécification des primitives de reconfiguration et d'introspection et le résultat de cohérence. La section 5 donne un exemple de spécification d'une application à composant avec reconfiguration dynamique. La section 6 présente des exemples de spécifications de programmes d'adaptation. La section 7 présente quelques travaux connexes avant de conclure avec la section 8.

2. Fractal

Fractal (Bruneton *et al.*, 2004; Bruneton *et al.*, 2002) est un modèle à composants (Szyperski, 1997) général qui repose sur le principe de la *séparation des préoccupations* : les services offerts par un composant sont distingués selon qu'ils sont d'ordre *fonctionnel*, c'est à dire prenant en charge les aspects métiers de l'application, ou d'ordre du *contrôle de l'application*, autrement dit, concernant ses capacités *réflexives* permettant la surveillance, l'inspection ou la reconfiguration des attributs et des assemblages. Ainsi, en Fractal, un composant est vu comme possédant deux parties : (a) un *contenu*, chargé d'implanter les capacités métiers du composant, possiblement formé d'un ensemble de sous-composants (on parle alors de composant *composite*) ; (b) une *membrane*, chargée de fournir les capacités de contrôle. Ces dernières sont spécifiées en Fractal sous forme d'API, avec des opérations telles que la liaison, l'ajout, la suppression de composants, ou des primitives pour la gestion du cycle de vie.

Un composant est une entité exécutable (Szyperski, 1997), pouvant recevoir des messages, qu'on appelle *services offerts* par le composant, ou invoquer des messages dits *services requis*, sur d'autres composants. Dans Fractal, ces différents services sont regroupés au sein de *ports*[1] offerts ou requis par le composant selon que le rôle spécifié pour celui-ci est *client* ou *serveur*. Les ports d'un composant sont typés 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 ne pas être liés à d'autres composants, l'absence de liaison pouvant provoquer des erreurs. Un composant a également un *statut* qui peut

1. Les ports sont nommés *interfaces*en Fractal.

être *démarré* ou *stoppé*.

3. Les entités de FracL

Nous présentons un modèle formel nommé FracL qui rend compte des aspects de contrôle de Fractal, autrement dit, des entités et opérations en rapport avec l'API Fractal. Nous ne modélisons, pour l'instant, qu'une version simplifiée de composants composites : les *boîtes* et ne prenons pas en charge la notion d'attribut de composant. FracL se situe au plus près de la spécification informelle donnée dans (Bruneton *et al.*, 2004), et respecte l'une des caractéristiques les plus remarquables de Fractal : il permet de modéliser un système à composants *décentralisé*. Décentralisé signifie ici que les opérations d'introspection et de reconfiguration dynamique ne sont pas effectuées par un moniteur central qui a 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.

Dans cette partie, nous définissons formellement les entités constituant une application à composants. Intuitivement, une telle application est formée de *boîtes*, de *composants*, de *ports* et de *signatures* dont les relations sont soumises à un ensemble de contraintes. Cet ensemble de contraintes constitue l'*invariant* du modèle. Chaque entité est définie par un ensemble muni de relations et contraint par un ensemble de propriétés. La figure 1 donne la liste des entités modélisées en FracL et 2 dépeint leur structure.

Sig	Les signatures
$Pname$	Les noms de ports
$Cname$	Les noms de composants
$Role = \{client, server\}$	Les catégories de ports
$Comp$	Les composants
Box	Les boîtes
$Status = \{stopped, started\}$	Les statuts des composants
$PortC$	L'ensemble des ports clients Les noms de ports
$PortS$	L'ensemble des ports clients

Figure 1. *Entités de FracL*

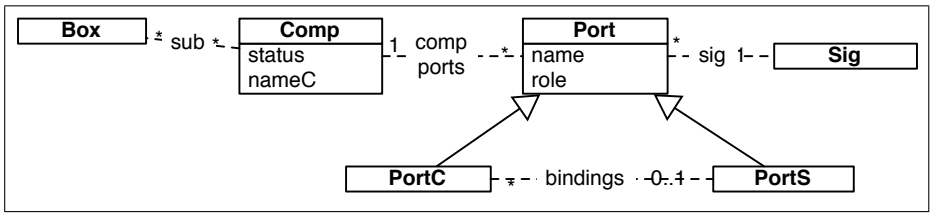

Figure 2. *Structure des entités de FracL*

3.1. *Signatures*

Elles correspondent aux *interfaces langage* de (Bruneton *et al.*, 2004), permettant de typer un port de composant. Il s'agit au minimum d'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, celle de raffinement (Abrial, 1996), ou encore de compatibilité entre contrats sémantiques (Beugnard *et al.*, 1999), etc. Il s'agit donc au moins d'une relation :

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

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 des 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 (Pierce, 2002). 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 réduite au sous-typage syntaxique, on aura que $\tau_1 <: \tau_2$ se traduit par $\tau_2 \sqsubseteq \tau_1$.

3.2. *Ports*

En termes de (Bruneton *et al.*, 2004), 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 :

$name \in Port \rightarrow Pname$	Un port a un nom unique	[1]
$role \in Port \rightarrow Role$	Un port est soit client soit serveur	[2]
$sig \in Port \rightarrow Sig$	Un port correspond à une unique signature	[3]
$comp \in Port \rightarrow Comp$	Un port appartient à un unique composant	[4]
$bindings \in PortC \nrightarrow PortS$	Un port client peut être lié à un seul port serveur	[5]

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

$PortC \equiv$	$dom(role \rhd \{client\})$	Ensemble de ports clients
$PortS \equiv$	$dom(role \rhd \{server\})$	Ensemble de ports serveurs

Les contraintes suivantes font partie de l'invariant du modèle :

– Compatibilité des liaisons entre ports :

$$\forall (p_c, p_s) \in bindings.sig(p_c) \sqsubseteq sig(p_s) \quad [6]$$

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

$$\forall p_1, p_2. \in Port.(p_1 \neq p_2 \wedge comp(p_1) = comp(p_2) \Rightarrow name(p_1) \neq name(p_2)) \quad [7]$$

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, nous autorisons la liaison entre un port client et un port serveur d'un même composant.

3.3. *Composants*

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

$nameC \in Comp \rightarrow Cname$	Un composant a un nom unique	[8]
$ports \in Comp \rightarrow \mathcal{P}(Port)$	Un composant a des ports	[9]
$status \in Comp \rightarrow Status$	Un composant est démarré ou stoppé	[10]

et est soumis aux contraintes suivantes :

– Les ports d'un composant ont ce composant comme propriétaire :

$$\forall c \in Comp. \forall p (p \in ports(c) \Leftrightarrow comp(p) = c) \quad [11]$$

– Si un composant est démarré, les liaisons de ses ports clients[2] doivent être effectives. Autrement dit, les émissions et réceptions d'appels de méthodes métiers doivent s'exécuter normalement.

$$\forall c \in Comp. (status(c) = started \Rightarrow ports(c) \cap PortC \subseteq dom(bindings)) \quad [12]$$

Définition 1 (clientPorts, serverPorts) *Nous définissons les fonctions qui associent respectivement à un composant l'ensemble de ses ports clients et serveurs :*

$$clientPorts(c) \equiv ports(c) \cap PortC$$
$$serverPorts(c) \equiv ports(c) \cap PortS$$

□

Définition 2 (portOfName) *Soit $portOfName$ défini par :*

$$portOfName \equiv \{((n, c), p) \mid c \in Comp \wedge n \in Pname \wedge p \in ports(c) \wedge n = name(p)\}$$

Il est clair que $portOfName$ est une fonction partielle de $Pname \times Comp$ vers $Port$
□

3.4. *Boites*

Une boîte $b \in Boite$ est un ensemble de composants pourvu d'une membrane, mais sans ports. Il constitue *un espace de noms*, au sens où il ne peut pas être traversé par des liaisons primitives. Une boîte est caractérisée par l'ensemble des composants qu'elle contient. L'ensemble des boîtes Box est muni de la fonction suivante :

$$sub \in Box \rightarrow \mathcal{P}(Comp) \qquad \text{Les sous-composants dans une boîte} \quad [13]$$

Définition 3 (super) *Nous définissons la fonction inverse de sub qui associe à un composant l'ensemble des boîtes dont il est un sous-composant :*

$$super(c) \equiv \quad \{b \in Box \mid c \in sub(b)\}$$

□

2. 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.

Définition 4 (Prédicats sur les boîtes) *Quelques prédicats sur les composants et les boîtes :*

noBox(c) : *c n'appartient à aucune boîte.*

$$noBox(c) \equiv \quad super(c) = \emptyset$$

sameBox(c,d) : *c et d appartiennent à au moins une boîte commune.*

$$sameBox(c, d) \equiv \quad super(c) \cap super(d) \neq \emptyset$$

sameNSpace(c,d) : *c et d se trouvent dans un même espace de noms, c'est-à-dire, soit en dehors de toute boîte, soit dans une boîte commune.*

$$sameNSpace(c, d) \equiv \quad (noBox(c) \wedge noBox(d)) \vee sameBox(c, d)$$

□

Contraintes dans l'invariant :

– Deux composants appartenant à un même espace de noms ne peuvent avoir le même nom :

$$\forall c_1 c_2 \in Comp.(sameNSpace(c_1, c_2) \wedge c_1 \neq c_2) \Rightarrow nameC(c_1) \neq nameC(c_2) \quad [14]$$

– Les liaisons entre ports ne peuvent se faire que sur des ports de composants appartenant à une même boîte.

$$\forall (p_1, p_2) \in bindings.(sameNSpace(comp(p_1), comp(p_2)) \quad [15]$$

Définition 5 (Invariant de FracL) *Les contraintes [1] à [15] forment l'invariant de FracL.*
□

4. Primitives de contrôle dans FracL

Nous spécifions ici un ensemble de primitives de contrôle, inspiré des méthodes de l'API Fractal. Ces primitives sont spécifiées comme des fonctions qui modifient ou consultent l'*état du modèle FracL*, 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 de la forme[3] :

$$< Box, Comp, Port, PortC, PortS, Sig, Pname, Cname, Status, Role, name,$$
$$nameC, role, sig, comp, sub, bindings, ports, status >$$

3. Les primitives spécifiées portent sur cet état de manière implicite : elles ne prennent pas celui-ci en argument.

vérifiant les contraintes [1] à [15], à savoir, l'*invariant* de FracL.

La spécification de ces primitives fournit la sémantique axiomatique d'un noyau de langage de programmation pour la reconfiguration et l'introspection de composants. Nous avons gardé les noms issus de l'API Fractal.

4.1. *Primitives d'introspection*

Elles correspondent pour l'essentiel aux fonctions constitutives des entités du modèle. Nous distinguons les *accesseurs*, qui correspondent aux fonctions totales et qui se spécifient trivialement, des méthodes plus complexes et qui correspondent aux fonctions partielles. Ces dernières ont trait à l'accès aux ports par leur nom dans un composant. On peut ainsi accéder au port client de nom n dans un composant c (`getFcInterface`), mais aussi au port serveur éventuellement lié au port client de nom n (`lookupFc`). Les figures 3, 4, 5 et 6 correspondent aux spécifications des accesseurs et la figure 7 aux méthodes gérant l'accès par nom.

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

Figure 3. *Accesseur sur les signatures*

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

P(Pname) listFc(Comp c)
       pre:  true
       post: return name[ports(c) ∩ PortC]

Status getFcState (Comp c);
       pre:  true
       post: return status(c)

P(Box) getFcSuperComponents(Comp c)
       pre: true
       post: return super(c)
```

Figure 4. *Accesseurs sur les composants*

```
Pname getFcItfName(Port p);
  pre:  true
  post: return name(p)

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

Comp getFcItfOwner(Port p)
  pre:  true
  post: return comp(p)

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

boolean isBound (Port p)
   pre:   true
   post:  return   p ∈ dom(bindings)
```

Figure 5. *Accesseurs sur les ports*

```
P(Comp) getFcSubComponents(Box b)
       pre : true
       post : return sub(b)
```

Figure 6. *Accesseur sur les boites*

```
Port getFcInterface(Pname n, Comp c)
       pre:  (n,c) ∈ dom(portOfName)
       post: return portOfName(n,c)

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

Figure 7. *Accès par nom*

4.2. *Primitives de reconfiguration*

Ce sont des méthodes qui modifient l'état de l'application. Elles concernent les liaisons entre ports, l'état des composants et les sous-composants d'une boîte.

4.2.1. *Modification des liaisons*

`bindFc`(c, n, ps) : lie le port client de nom n du composant c au port serveur p_s. Échoue si le composant c n'a pas de port client de ce nom, ou s'il est déjà lié, ou si les composants ne sont pas dans le même espace de noms, ou s'il y a incompatibilité entre leur signature, ou encore , si c n'est pas dans l'état $stopped$.

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

La figure 8 correspond à la spécification de ces méthodes.

```
void bindFc (Comp c, Pname n, PortS p_s)
        pre : ∃p_c.
                  portOfName(n, c)=p_c        // n est le nom d'un port de c
                  ∧ p_c ∈ PortC                  // qui est un port client
                  ∧ p_c ∉ dom(bindings)       // qui n'est pas déjà lié
                  ∧ sig(p_c) ⊑ sig(p_s)       // et qui est compatible avec p_s
                  ∧ status(c)=stopped         // c est stoppé
                  ∧ sameNSpace(c, comp(p_s))
        post : bindings := bindings ∪ {(portOfName(n, c), p_s)}


void unbindFc (Comp c, Pname n)
        pre :∃p_c, p_s.
                  portOfName(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) ∈ bindings     // qui est lié
                  ∧ status(c)=stopped         // c est stoppé
        post: bindings :=
                 bindings − {(portOfName(n, c), bindings(portOfName(n, c)))}
```

Figure 8. *Modification des liaisons*

4.2.2. *Modification de l'état d'un composant*

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

`stopFc`(c) : arrête le composant c.

La figure 9 donne la spécification de ces méthodes.

```
void startFc(Comp c)
        pre : clientPorts(c) ⊆ dom(bindings)  // toutes les liaisons sont faites
        post : status(c) := started

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

Figure 9. *Modification de l'état*

```
void addFcSubComponents(c, b)
  pre : c ∉ sub(b)                    // c n'est pas dans b
        ∧ ¬ bound(c)                  //  il n'est pas lié
        ∧ Status(c) = stopped         // il est stoppé
  post :
        sub(b) := sub(b) ∪ {c}

void removeFcSubComponents(c, b)
  pre : c ∈ sub(b)                    // c est dans b
        ∧ ¬ bound(c)                  // il n'est pas lié
        ∧ Status(c) := stopped        // il est stoppé

  post :
        sub(b) := sub(b) − {c}
```

Figure 10. *Modification des sous composants*

4.2.3. *Modification des sous composants*

`addFcSubComponents`(c, b) : ajoute le composant c dans la boîte b.

`removeFcSubComponents`(c, b) : enlève le composant c de la boîte b.
L'ajout et la suppression d'un composant (figure 10) d'une boîte requiert que celui-ci soit stoppé et non lié. Nous introduisons la définition suivante exprimant le fait qu'au moins un port (client ou serveur) de c est lié :

$$bound(c) \equiv (clientPorts(c) \cap dom(bindings)) \cup (serverPorts(c) \cap codom(bindings)) \neq \emptyset$$

4.3. *Cohérence du modèle*

La cohérence du modèle est établie en montrant que les spécifications des primitives conservent l'invariant. La preuve peut se trouver en (Simonot *et al.*, 2007). Cette preuve a aussi été faite dans l'atelier Focal. (Benayoun, 2008)

5. Modélisation d'un exemple de mise à jour

Nous illustrons FracL par la spécification d'une application à composants avec reconfiguration dynamique. Il s'agit d'une application semblable à *Mozilla*, pouvant mettre à jour ses modules d'extension (*plugins*) déjà installés. Dans ce type d'applications, les composants sont en général munis d'un *manifeste d'installation* au format XML, permettant, entre autres choses, de livrer les informations nécessaires à la gestion de la compatibilité des versions entre les composants. Dans ce manifeste sont décrits : (a) l'identifiant de l'application, (b) son numéro de version et, (c) l'ensemble de *spécifications des applications cibles*, avec qui le composant est compatible. Chaque *spécification d'application cible* décrit une application cible possible du module, en spécifiant l'identifiant de l'application cible ainsi que ses versions minimale et maximale compatibles.

D'un point de vue conceptuel, les informations de version permettent de surcharger la notion de correction des liaisons : pour qu'une extension puisse être liée à une application, il faut non seulement qu'elle fournisse un port compatible avec un port client de l'application, mais aussi que l'application soit une des cibles de l'extension, dans une version comprise dans l'intervalle spécifié dans le manifeste d'installation de l'extension.

L'architecture de l'application est illustrée dans la figure 11 .

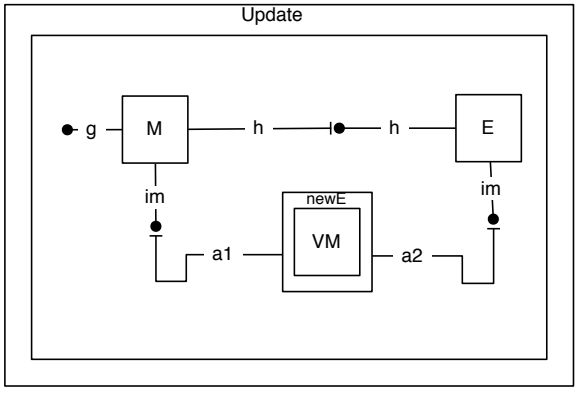

Figure 11. *Architecture pour la mise à jour d'une extension*

```
interface InstallMf{
  String getId();
  String getVersion();
  []TargetsApp getTargetsApp();
}

interface TargetApp{
  String getId();
  String getMinVersion();
  String getMaxVersion();
}
```

Figure 12. *Interfaces pour la mise à jour*

Nous simplifions le problème en supposant que l'architecture correspond à un assemblage (composant composite) de trois uniques composants : `M`, le composant principal (p.e. *Firefox*), `E`, le composant qui représente une extension déjà installée, et `VM`, le composant qui répresente le gestionnaire des versions pour ces deux composants.

Les composants `M` et `E` ont chacun un port serveur de nom im et de signature `InstallMf` (figure 12) livrant les informations de version du manifeste d'installation. `E` et `M` ont un port de nom h et de signature `H`, respectivement serveur et client. `M` a de plus un port serveur g de signature `G`. Ils correspondent aux services métiers que nous ne spécifions pas ici.

Le composant `VM` a deux ports clients a_1 et a_2 de signature `InstallMf`. Son rôle est de fournir une méthode de contrôle de type `E newE()`[4] qui retourne un composant de même type que celui lié à son port client de nom a_2, et compatible (au sens des contraintes d'installation) avec le composant lié à son port client de nom a_1. Cette méthode représente l'accès au serveur de mise à jour.

Le composite `Main` possède une méthode de contrôle `Upadte` permettant de mettre à jour l'extension. Cette méthode cherche un composant de même identifiant que celui du composant `E`, et d'une version postérieure et compatible avec le module `M`. Si elle le trouve, elle doit débrancher `E` et mettre le composant trouvé à sa place.

5.1. *Modélisation des entités de l'exemple*

Nous allons étendre FracL de façon à rendre compte de l'architecture de notre exemple. Nous allons ainsi définir des sous ensembles de $Port$, $Comp$, Sig et Box.

4. Pour simplifier, nous utilisons dans cet exemple le même nom pour désigner le nom d'un composant et son type.

Par exemple, nous définirons l'ensemble `VM` comme le sous ensemble de $Comp$ contenant des composants ayant deux ports clients de signature`InstallMf`. La nouvelle architecture est illustrée dans la figure 13.

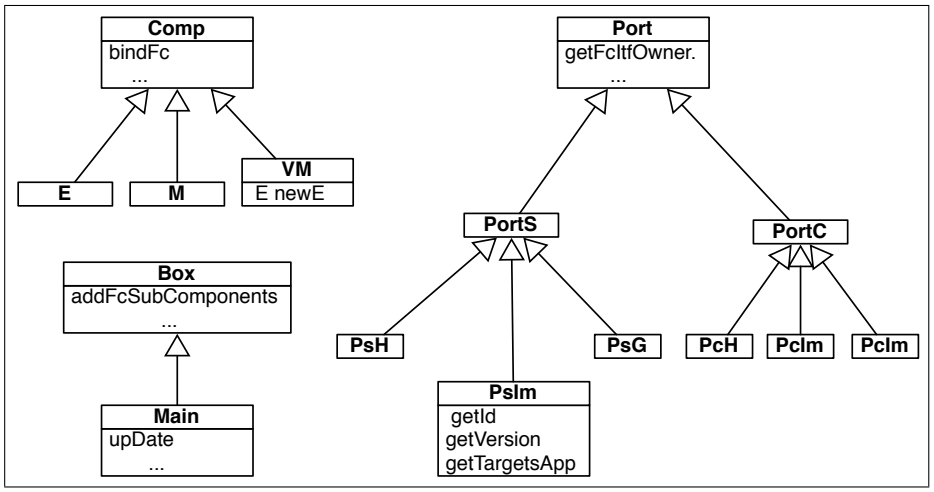

Figure 13. *Extension de Fracl pour la mise à jour*

5.1.1. *Spécification des signatures et des ports*

Afin de spécifier les signatures associées aux interfaces de l'utilisateur, nous ajoutons trois nouvelles constantes dans Sig. Nous définissons également un ensemble pour caractériser chaque type de port de l'application selon son rôle (client ou serveur) et sa signature. L'ensemble de ces définitions est donnée dans la figure 14.

$G, H, \mathrm{InstallMf}, \mathit{TargetApp} \in \mathit{Sig}$	constantes dans Sig
$\mathrm{PcH} = \{p \in \mathit{PortC} \mid \mathit{sig}(p) = H\}$	ports clients de type H
$\mathrm{PsH} = \{p \in \mathit{PortS} \mid \mathit{sig}(p) = H\}$	ports serveurs de type H
$\mathrm{PsG} = \{p \in \mathit{PortS} \mid \mathit{sig}(p) = G\}$	ports serveurs de type G
$\mathrm{PcIm} = \{p \in \mathit{PortC} \mid \mathit{sig}(p) = \mathrm{InstallMf}\}$	ports clients de type $\mathrm{InstallMf}$
$\mathrm{PsIm} = \{p \in \mathit{PortS} \mid \mathit{sig}(p) = \mathrm{InstallMf}\}$	ports serveurs de type $\mathrm{InstallMf}$

Figure 14. *Signatures et types de ports pour la mise à jour*

	$h, a_1, a_2, g, i_m \in Pname$	constantes
$E =$	$\{c \in Comp \mid \exists p_1, p_2 (ports(c) = \{p_1, p_2\} \wedge$	composant E
	$name(p_1) = h \wedge p_1 \in \text{PsH} \wedge name(p_2) = i_m \wedge p_2 \in \text{PsIm}\}$	
$M =$	$\{c \in Comp \mid \exists p_1, p_2, p_3 (ports(c) = \{p_1, p_2, p_3\} \wedge$	composant M
	$name(p_1) = i_m \wedge p1 \in \text{PsIm} \wedge name(p_2) = g \wedge p_2 \in \text{PsG} \wedge$	
	$name(p_3) = h \wedge p_3 \in \text{PcH}\}$	
$VM =$	$\{c \in Comp \mid \exists p_1, p_2 (ports(c) = \{p_1, p_2\} \wedge$	composant VM
	$name(p_1) = a_1 \wedge p_1 \in \text{PcIm} \wedge name(p_2) = a_2 \wedge p_2 \in \text{PcIm}\}$	
Main $=$	$\{b \in Box \mid \exists m, e, v_m . (sub(b) = \{m, e, v_m\} \wedge$	composant Main
	$m \in M \wedge e \in E \wedge v_m \in VM\}$	

Figure 15. *Types des composants pour la mise à jour*

5.1.2. *Spécification des composants*

Nous procédons de manière analogue pour les composants : nous ajoutons des constantes nouvelles dans $Pname$ pour les noms de ports, puis nous définissons des ensembles E, M, et VM pour décrire la structures des composants `E`, `M` et `VM` : à savoir les noms, types et rôles de leurs ports respectifs. Le composite `Main` est défini par une boîte contenant exactement trois composants respectivement de types M, E et VM. Ces définitions sont données dans la table 15.

Cette architecture n'est qu'une simplification extrême de l'exemple. Dans une configuration plus réaliste, notre composite, pour réaliser son opération de mise à jour, aurait pu créer une instance de composant de gestion de version `VM`, le lier à l'extension dont on veut une mise à jour ainsi qu'à sa cible, faire la mise à jour puis détruire ce composant. Dans ce cas, le composite principal ne pourrait être défini comme un ensemble fixe de 3 composants.

5.2. *Spécification des méthodes*

Dans FracL il est possible de spécifier et de raisonner sur les méthodes tant fonctionnelles que de contrôle. Une méthode fonctionnelle ne nécessite aucune connaissance de la structure du composant (pas d'introspection) et ne peut agir sur cette structure. Ainsi, la spécification des méthodes fonctionnelles peut se faire par le biais d'une signature munie d'un invariant exprimant une vue partielle de l'état fonctionnel du

composant plus des spécifications de chacune des méthodes de l'interface. Ces méthodes ne nécessitant pas de vision globale du composant auquel elles appartiennent, nous avons choisi de les inclure dans notre modèle au niveau des ports serveurs correspondant à la signature dont elles sont issues.

De leur côté, les méthodes de contrôle modifient l'état l'architectural du composant. Nous les décrirons comme des opérations agissant sur leurs composants cibles. Dans notre exemple, les méthodes spécifiées par l'interface `InstallMf` sont fonctionnelles, alors que `newE` et `Update` sont des méthodes de contrôle. Les premières seront décrites sur les ports de signature `InstallMf` c'est à dire sur PsIm. alors que les deux autres, porteront sur les composants de type `VM` et `Main`. Par ailleurs, E, M et VM sont des sous-ensembles de $Comp$: leurs éléments héritent donc des méthodes de contrôles générales (primitives). Ceci correspond à une extension des méthodes sur les ports et les composants de FracL telle qu'illustré par la figure 13.

Nous spécifions les deux méthodes de contrôle de l'exemple : (a) la méthode `newE()` du composant `VM`, qui retourne un composant possible pour la mise à jour, et (b) la méthode `Update` du composite `Main`, qui effectue la mise à jour. Nous utilisons la propriété $\mathit{Compatible}(e_1, e)$, qui exprime le fait que 2 ports de signature `InstallMft` ont le même `Id` et que le premier est d'une version postérieure ou égale au second, ainsi que $\mathit{Target}(e_1, e_m)$, qui exprime le fait que e_m est une des applications cibles de e_1 dans une version comprise dans l'intervalle donné par e_1.

$$\begin{aligned}\mathit{Compatible}(e_1, e) =& \mathrm{getId}(e_1) = \mathrm{getId}(e) \wedge \mathrm{getVersion}(e_1) \geq \mathrm{getVersion}(e)\\ \mathit{Target}(e_1, e_m) =& \exists t_a \in \mathrm{getTargetsApp}(e_1).(\mathrm{getId}(t_a) = \mathrm{getId}(e_m) \wedge\\ & \mathrm{getMinVersion}(t_a) \leq \mathrm{getVersion}(e_m) \leq \mathrm{getMaxVersion}(t_a))\end{aligned}$$

5.2.1. *La méthode newE*

Cette méthode retourne un composant pour la mise à jour du composant lié au port client a_2, et compatible avec le composant lié au port client a_1, s'il en existe ou bien retourne le composant lié à a_2.

```
E newE(VM v)
  pre:
      portOfName(a2, v) ∈ dom(bindings) ∧
      portOfName(a1, v) ∈ dom(bindings) ∧
      Target(bindings(portOfName(a2, v)),
             bindings(portOfName(a1, v)))
  post: returns e1 such that
        (e1 ∈ E ∧ noBox(e1) ∧ ¬bound(e1)
        ∃ ps . ps = bindings(portOfName(a2, v)) ∧
        ∃ pm . pm = bindings(portOfName(a1, v)) ∧
        Compatible(portOfName(im, e1), ps) ∧ Target(portOfName(im, e1), pm))
```

5.2.2. *La méthode de mise à jour update de Main*

On peut maintenant spécifier l'opération de mise à jour update du composite Main.

void update (Main main)
pre : $\exists\ m, e, v_m\ .\ ($
$m \in sub(\text{main}) \wedge m \in M \wedge e \in sub(\text{main}) \wedge e \in E \wedge$
$v_m \in sub(\text{main}) \wedge v_m \in VM \wedge$
$bindings(portOfName(h, m)) = portOfName(h, e)) \wedge$
$bindings(portOfName(a_1, v_m)) = portOfName(i_m, m)) \wedge$
$bindings(portOfName(a_2, v_m)) = portOfName(i_m, e))$

post : $\exists e_1 \in E, e \in E, m \in M, v_m \in VM.$
$(sub(\text{main}) := sub(main) - \{e\} \cup \{e_1\} \wedge$
$sub(\text{main}) = \{e_1, m, v_m\} \wedge$
$bindings(portOfName(h, m)) = portOfName(h, e_1) \wedge$
$bindings(portOfName(a_1, v_m)) = portOfName(i_m, m) \wedge$
$bindings(portOfName(a_2, v_m)) = portOfName(i_m, e_1) \wedge$
$Compatible(e1, e) \wedge Target(e1, e_m))$

Nous avons, à l'aide de notre modèle, spécifié une application capable d'effectuer dynamiquement une mise à jour de son extension. Nous avons exprimé le comportement attendu de l'application en ce qui concerne son aspect touchant à la reconfiguration dynamique. Cette spécification est purement abstraite : elle ne dit pas comment l'application fournie réalise cette fonctionnalité, mais ce qu'elle doit réaliser. Autrement dit, les propriétés que doit vérifier l'opération de mise à jour forment la spécification de cette opération. Pour coder cette opération, il suffira de spécifier des fonctions qui cette fois, n'utiliseront que des appels sur les primitives de contrôle de FracL.

6. Construire des méthodes de contrôle sûres

Dans cette partie nous donnons des exemples de nouvelles méthodes de contrôle pouvant être définies dans le cadre de FracL. Une méthode de contrôle consulte ou modifie l'état architectural d'un composant ou d'une boîte. Nous avons décrit jusqu'ici : (a) une définition formelle des notions de composants et de boîtes ; (b) une définition (en termes de pré-conditions et post-conditions) de certaines primitives de contrôle. Nous prenons le terme de primitives au sens fort : nous ne nous intéressons pas à en fournir du code. Nous faisons l'hypothèse de l'existence d'un code (par exemple, celui fourni par les implantations de Fractal) qui se comporte comme nous l'avons prescrit dans notre modèle. Nous sommes fidèles en ceci à la notion de *contrat* (Beugnard *et al.*, 1999). En tant que contrat, FracL engage le programmeur d'un langage de programmation à composants, à fournir un code respectant ses spécifications. Il serait théoriquement envisageable de vérifier formellement cet aspect, bien que cela ne soit pas notre axe de travail privilégié. C'est en ce sens que nous avons donné

une sémantique axiomatique d'un ensemble de primitives pour la reconfiguration et l'introspection.

Pour aller au delà, et en particulier pour spécifier et raisonner sur des nouvelles méthodes d'introspection et reconfiguration, nous avons ajouté à FracL un langage de script. Celui-ci est formé de l'appel aux primitives de FracL, de la séquence, du choix et de l'itération. Il est ainsi possible de définir de nouvelles méthodes de reconfiguration qui correspondent aux actions d'un programme d'adaptation.

Il suffit à cet effet de donner :

1) Une spécification (*pré - post*) des ces méthodes

2) un code dans ce langage de script pour ces méthodes.

Une preuve dans une logique de Hoare usuelle peut être faite pour montrer que le programme réalise sa spécification. On dispose alors de nouveaux programmes de contrôle, dont le comportement est spécifié formellement et prouvé, et qui peuvent à leur tour être utilisés pour construire des programmes plus sophistiqués.

6.1. *Quelques méthodes utilitaires d'introspection*

Nous illustrons cet aspect en donnant deux exemples de méthodes simples d'introspection. Les primitives de FracL permettent d'accéder directement aux ports serveurs liés aux ports clients d'un composant c. Il est possible de réaliser l'opération duale afin d'accéder aux ports clients[5] qui sont liés à des ports serveurs de c. Cependant, cette opération n'existe pas de façon primitive. Nous montrons ici comment spécifier et programmer cette opération. Pour cela, on doit accéder à la boîte contenant c. Sachant que l'invariant de FracL spécifie que les liaisons sont faites uniquement entre composants d'un même espace de noms, il suffit de collecter tous les ports clients de composants de même boîte que c et qui sont liés à des ports serveurs de c. Nous définissons une méthode `siblingTo`(c), qui retourne l'ensemble des composants ayant une boîte en commun avec c, et une méthode `bindingsTo`(c), qui retourne l'ensemble des ports clients liés à des ports serveurs de c. La spécification et le code pour ces méthodes sont donnés dans les figures 16 et 17.

Les preuves de correction de ces programmes, que nous ne détaillons pas ici, se font en logique de Hoare. Elles nécessitent de raisonner sur des boucles, et donc de produire un invariant de boucle. Il faut également remarquer que la pré-condition de `bindingsTo` est nécessaire pour la démonstration : nos primitives ne permettent pas de retrouver les ports serveurs liés à un port client d'un composant qui n'est dans aucune boite.

5. Possiblement dans plusieurs composants.

```
P(Comp) siblingTo(Comp c)
    pre:  true
    post: return {c' | super(c') ∩ super(c) ≠ ∅}

P(PortC) bindingsTo(Comp c);
    pre:  ¬(noBox(c))
    post: return bindings^-1[{serverPorts(c)}]
```

Figure 16. *Spécifications pour* `siblingTo` *et* `bindingsTo`(c)

```
P(Comp) siblingTo(Comp c)
   comps = ∅;
   bbs = getFcSuperComponents(c);
   for (b :bbs){
      comps = comps ∪ getFcSubComponents(b);
   }
   return comps;
}

P(PortC) bindingsTo(Comp c)
   pps = ∅;
   ccs =siblingTo(c);
   for(d : ccs){
      for(p : getFcInterfaces(d){
         if (getRole(p) == client and isBound(p)
              and lookupFc(d,getFcItfName(p))==s
              and getFcItfOwner(s) == c){
                   pps = pps ∪ {p};
         }
       }
   } return pps;
}
```

Figure 17. *Programmes pour* `siblingTo` *et* `bindingsTo`(c)

6.2. *Vers des programmes plus sophistiqués*

L'approche incrémentale permet de définir des méthodes sophistiquées. Nous travaillons par exemple à l'élaboration de programmes effectuant la substitution d'un composant par un autre. Ces programmes se caractérisent par une double complexité :

– une complexité conceptuelle : la substitution repose sur la notion de compatibilité ($\sqsubseteq$) entre composants. On peut par exemple faire correspondre cette opération à une relation de *sous-typage en profondeur*(Pierce, 2002) entre les types des deux

composants. On autorisera le remplacement de c par c' s'ils ont le même nombre et noms de ports, de mêmes rôles et avec des types compatibles. Mais on peut aussi faire correspondre cette opération à une relation de sous-typage *en largeur*, où l'on ajoute la possibilité d'avoir plus de ports serveurs dans c' et moins de ports clients dans c.

– une complexité au niveau des programmes : parcours du graphe de l'architecture, choix du port remplaçant, respect des pré-conditions concernant l'état et les liaisons des composants ...

Disposer d'un ensemble de programmes de substitution, aux comportements clairement définis car spécifiés formellement et dont la cohérence est prouvée, devrait faciliter grandement l'écriture de programme de reconfiguration comme par exemple celui de notre mise à jour de modules d'extensions.

7. Travaux connexes

7.1. *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 composants s'intéressent à la notion d'interface et à celle de compatibilité entre interfaces 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 (Lanoix *et al.*, 2007; Kouchnarenko *et al.*, 2006), où l'architecture des composants est conçue à l'aide de diagrammes UML, puis la méthode B 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 (Mouakher *et al.*, 2006) permettant d'adapter 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 l'aspect contrôle des composants. Ils ne peuvent donc exprimer de propriétés concernant la reconfiguration dynamique ou l'introspection. C'est en cela que notre approche est fondamentalement différente.

7.2. *Analyse formelle des architectures*

Kmelia (Attiogbé *et al.*, 2006; André *et al.*, 2006) est un modèle de spécification de composants outillé permettant de modéliser des architectures logicielles et leurs propriétés. L' idée des auteurs est d'enrichir suffisamment les interfaces des composants afin de pouvoir vérifier des propriétés sur leur assemblage dans une architecture don-

née (correction, fiabilité, sûreté, etc.). Dans ce système, les services sont des notions de première classe : ce sont les services, et non pas les ports, qui sont requis ou offerts et sont dotés d'interfaces spécifiques. Ceci constitue une différence importante avec notre modèle dans lequel les citoyens de première classe sont les composants et les ports, fidèlement au modèle Fractal. La seconde différence est que dans *Kmelia*, les services sont aussi dotés d'un comportement qui est un système de transition étiqueté étendu (*eLTS*, *extended Label Transition System*), ce qui n'est pas le cas dans notre modèle. En conclusion, il nous semble que la différence tient essentiellement au fait qu'un des objectifs premiers de Kmelia est l'étude fine de la composabilité de services (interopérabilité statique et interopérabilité dynamique par étude de l'interaction entre services), là où notre modélisation vise plutôt l'étude approfondie des changements dynamiques de configuration d'une application à composants. Ces deux approches ont en commun un caractère mixte permettant de raisonner à la fois sur les propriétés fonctionnelles et architecturales d'une application.

7.3. *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 (Warren *et al.*, 2006) sont présentés des travaux proches du nôtre. Les auteurs ont pour objectif de rendre possible des vérifications automatiques à l'exécution d'applications à base de composants reconfigurables telles qu'elles existent dans le *framework* OpenRec (Hillman *et al.*, 2004). Pour permettre l'expression et la vérification de contraintes d'architecture dans *OpenRec*, ils ont écrit un modèle formel qui utilise ALLOY (Jackson, 2002; AA, n.d.). Dans ce modèle, l'architecture d'une application et les contraintes d'architecture sont exprimables. Ils ont ensuite intégré l'analyseur ALLOY à OpenRec de façon à permettre des vérifications automatiques à l'exécution.

Notre modélisation est proche : tout comme ALLOY, nous utilisons la logique du premier ordre et notre modèle pourrait d'ailleurs très aisément se traduire en ALLOY[6]. Mais ils ne modélisent que les aspects des composants concernant la liaison entre composants. Notre spécification est plus détaillée puisqu'elle définit des concepts comme la compatibilité entre signatures, permet de parler de l'espace de nommage , de l'état d'un composant ou encore du typage. Nous avons donc une approche plus intégrée, 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.

Dans (Chatley *et al.*, 2003), 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 ont conçu une spécification formelle en ALLOY. Leur

6. Ceci est en cours de réalisation.

spécification est, là encore, proche de la nôtre : un composant est un ensemble de fiches (*pegs*, ou ports serveurs) et de *trous* (ports 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 nôtre : 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 évidemment nombre d'ingrédients présents dans notre spécification de `bindfc` (compatibilité de liaisons, etc.). Il serait sans doute intéressant de voir comment ce *framework* peut être codé en Fractal, et d'examiner sa correction grâce à notre modèle.

7.4. *Safran*

Notre façon d'aborder la reconfiguration dynamique est conceptuellement très proche de celle des auteurs de *Safran*. *Safran* (David *et al.*, 2006a) est une extension de Fractal permettant le développement d'applications adaptatives. Il est constitué d'un langage dédié pour programmer des politiques d'adaptation ainsi que d'un mécanisme permettant d'attacher ou de détacher dynamiquement ces politiques aux composants de base Fractal. Une politique d'adaptation est un ensemble de règles réactives de la forme :

when <event> **if** <condition> **do** <action>

L'action est un programme de reconfiguration exprimé dans un langage dédié nommé *FScript* (David *et al.*, 2006b). Ce langage est un langage procédural simple pouvant utiliser des expressions permettant de naviguer dans l'architecture Fractal à l'image de *XPath*. Dans ces deux approches, l'accent est mis sur la possibilité de définir des programmes d'adaptation de façon sûre. Nos deux langages de scripts sont proches. *Fscript* devrait facilement pouvoir se traduire dans notre langage, étant donné que nous avons accès au graphe de l'architecture sur lequel *FPath* est basé. Ceci nous permettrait d'hériter de l'élégance de ce langage. *FScript* offre une garantie de terminaison des programmes grâce à la limitation du langage. Dans la mesure où nous sommes dans un contexte formel, nous ne sommes pas obligés de nous limiter à un langage assurant la terminaison comme c'est le cas de *FScript*. Dans notre modèle, nous pouvons énoncer et prouver la terminaison des programmes écrits dans le langage de script. Dans le même ordre d'idées, *Safran* implante un mécanisme permettant de défaire à l'exécution les reconfigurations lorsqu'une incohérence est détectée. Dans notre modèle, il est possible de prouver la cohérence d'une reconfiguration, ainsi que toute autre propriété concernant l'architecture de l'application. Ces deux approches sont donc conceptuellement très proches et complémentaires.

8. Conclusion et perspectives

Nous avons présenté un ensemble de primitives pour la spécification, la programmation et la preuve de propriétés sur des applications à base de composants avec capacité d'introspection et de reconfiguration dynamique. En attachant les spécifications des méthodes fonctionnelles aux ports de composants et celles des méthodes de contrôle aux composants eux-mêmes, notre approche autorise l'expression de programmes et des spécifications qui mélangent aspects fonctionnel et de contrôle. Ceci nous permet de raisonner finement sur un large spectre de programmes pour l'adaptation, tout en évitant la confusion entre ces deux types de préoccupations. Nous avons montré la cohérence de notre ensemble de primitives (Simonot *et al.*, 2007) puis encodé notre approche dans l'atelier de spécification et de preuve Focal (Benayoun, 2008) ce qui nous a permis, outre de vérifier mécaniquement la cohérence, de spécifier et prouver des propriétés sur l'exemple détaillé en 5.

Les perspectives de ce travail concernent l'extension de la modélisation proprement dite, l'extension du langage de scripts, et l'expérimentation sur des exemples concrets d'adaptation. Les extensions du modèle visées à court terme incluent la prise en compte des ports optionnels et des attributs reconfigurables des composants, ainsi que l'élaboration de notions plus complètes de composites, de type pour les ports et pour les composants, et la modélisation des notions de compatibilité associées. Notre langage de script, fondé sur l'utilisation des primitives très minimalistes inspirées de Fractal, tout en étant complet, peut donner lieu à du code d'assez bas niveau et relativement difficile à lire. Deux pistes sont à explorer ici. D'une part, nous travaillons actuellement à l'écriture de bibliothèques de méthodes certifiées sophistiquées pouvant constituer des utilitaires pour l'adaptation, tels que l'utilitaire de remplacement d'un composant brièvement décrit dans la partie 6.2. Une deuxième piste à explorer consiste à chercher à bénéficier de l'élégance et de la concision du langage de navigation sur l'architecture de composants FPath, en traduisant automatiquement les expressions FPath vers notre langage de script. Ceci permettrait de faciliter grandement l'expression des programmes et des spécifications de FracL, tout en ajoutant un moyen supplémentaire aux vérifications proposées par FPath.

Enfin, concernant les expérimentations concrètes, nous projetons de travailler sur la spécification d'une portion de l'architecture à base de composants Quinna (Tournier *et al.*, 2005) pour la gestion dynamique de la qualité de services. Notre approche semble particulièrement intéressante pour ce cas d'étude dans la mesure où le traitement de la qualité de service fait souvent appel à la prise en compte simultanée d'aspects métiers et de contrôle au sein des applications.

9. Bibliographie

AA, « The Alloy Analyzer », , http ://alloy.mit.edu, n.d.

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

André P., Ardourel G., Attiogbé C., « Spécification d'architectures logicielles en Kmelia : hiérarchie de connexion et composition », *1ère Conférence Francophone sur les Architectures Logicielles*, Hermès Sciences Publications - Lavoisier, p. 101-118, 2006.

Attiogbé C., André P., Ardourel G., « Checking Component Composability », *5th International Symposium on Software Composition (ETAPS/SC'06)*, vol. 4089 of *Lecture Notes in Computer Science*, Springer Verlag, 2006.

Benayoun V., « Composants Fractal avec reconfiguration dynamique : formalisation avec l'atelier Focal », , http ://reve.futurs.inria.fr/, 2008.

Beugnard A., Jézequel J.-M., Plouzeau N., Watkins D., « Making Components Contract Aware », *Computer*, vol. 32, n° 7, p. 38-45, 1999.

Bruneton E., Coupaye T., Stefani J., « The Fractal Component Model », , http : fractal.objectweb.org/specification/index.html, 2004.

Bruneton E., Coupaye T., Stefani J.-B., « Recursive and Dynamic Software Composition with Sharing », *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga (Spain), 2002.

Chatley R., Eisenbach S., Magee J., « Modelling a framework for plugins », *Specification and verification of component-based systems, September 2003*, 2003.

David P.-C., Ledoux T., « An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components », *Software Composition*, p. 82-97, 2006a.

David P., Ledoux T., « Safe Dynamic Reconfigurations of Fractal Architectures with FScript », *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, 2006b.

Hillman J., Warren I., « An Open Framework for Dynamic Reconfiguration », *icse*, vol. 0, p. 594-603, 2004.

Jackson D., « Alloy : a lightweight object modelling notation », *ACM Transactions on Software Engineering and Methodology*, vol. 11, n° 2, p. 256-290, 2002.

Kouchnarenko O., Lanoix A., « How to Refine and to Exploit a Refinement of Component-based Systems », *in* I. Virbitskaite, A. Voronkov (eds), *PSI 2006, Pesrpectives of System Informatics, 6th Int. Andrei Ershov Memorial Conf.*, vol. 4378 of *LNCS*, Springer, Novosibirsk, Akademgorodok, Russia, p. 297-309, June, 2006.

Lanoix A., Hatebur D., Heisel M., Souquières J., « Enhancing Dependability of Component-Based Systems », *Reliable Software Technologies – Ada Europe 2007*, Springer-Verlag, p. 41-54, 2007.

McKinley P. K., Sadjadi S. M., Kasten E. P., Cheng B. H. C., « Composing Adaptive Software », *Computer*, vol. 37, n° 7, p. 56-64, 2004.

Mouakher I., Lanoix A., Souquières J., « Component Adaptation : Specification and Verification », *11th International Workshop on Component Oriented Programming (WCOP 2006), In Conjunction of ECOOP 2006*, 2006.

Pierce B. C. (ed.), *Types and Programming Languages*, MIT Press, 2002.

Simonot M., Aponte M., Modélisation formelle du contrôle en Fractal, Technical Report n° 1563 - http ://reve.futurs.inria.fr/, CNAM-CEDRIC, 2007.

Szyperski C., *Component Software : Beyond Object-Oriented Programming*, Addison-Wesley Professional, December, 1997.

Tournier J.-C., Olive V., Babau J.-P., « Qinna, an Component-Based QoS Architecture », *8th SIGSOFT symposium on CBSE*, Saint-Louis, USA, June, 2005.

Warren I., Sun J., Krishnamohan S., Weerasinghe T., « An Automated Formal Approach to Managing Dynamic Reconfiguration », *ASE*, p. 37-46, 2006.