Modélisation de la reconfiguration dynamique avec Focal

Virginia Aponte Vincent Benayoun Marianne Simonot

28 septembre 2008

Resumé

Les applications auto-adaptatives modifient leur comportement de façon dynamique en fonction des changements dans leur contexte d'exécution. Pour cela, elles ont besoin d'explorer leur structure interne (introspection) et de la modifier (reconfiguration dynamique). Nos travaux se situent dans le contexte de la programmation par composants qui se prête bien à la modélisation d'applications auto-adaptatives. Nous proposons un outillage formel permettant de spécifier des applications à base de composants avec introspection et reconfiguration dynamique ainsi que d'exprimer et de prouver des propriétés liées autant aux aspects métiers de l'application (propriétés *fonctionnelles*) qu'aux aspects de *contrôle* de l'application (i.e. concernant les opérations de liaison, de maintenance, etc.). Notre travail est basé sur le modèle de composants Fractal, modèle à la fois général, pleinement dynamique et réflexif, dont nous formalisons la spécification. Nous avons choisi l'atelier de développement formel Focal pour l'implantation de cette spécification formelle, ce qui fournit un cadre de spécification et de preuve de programmes avec reconfiguration dynamique.

1 Introduction

La variabilité de plus en plus grande des contextes d'exécution dans les systèmes informatiques a fait émerger le besoin d'applications auto-adaptatives, c'est à dire d'applications capables de répondre à ces changements en modifiant leur comportement de façon dynamique et autonome. La condition de possibilité de telles applications réside dans leur capacité à connaître leur organisation (introspection) et à y apporter des modifications (reconfiguration dynamique). Elles peuvent ainsi embarquer des programmes chargés de la mise en oeuvre dynamique de politiques d'adaptation.

Nous proposons un outillage formel réalisé avec l'atelier Focal [11] et permettant de spécifier des applications auto-adaptatives, d'exprimer et de prouver des propriétés liées tant aux aspects métiers de l'application (propriétés fonctionnelles), qu'aux aspects de reconfiguration (propriétés de contrôle). Notre travail est basé sur le modèle de composants [18] Fractal [9, 14]. Ce modèle a vocation à permettre la conception et la programmation de systèmes pleinement reconfigurables. Comme tout composant, un composant Fractal est une entité exécutable encapsulée qui exhibe les services qu'elle offre au moyen d'interfaces serveurs et les services dont elle dépend au moyen d'interfaces clientes. De ce point de vue, il vise à permettre la construction modulaire des applications. Mais un composant Fractal est aussi à même de fournir des informations sur

sa structure interne et des services pouvant agir dynamiquement sur celle-ci. Ces services d'introspection et de reconfiguration sont regroupés au sein d'interfaces de contrôle qui constituent l'API Fractal.

Notre travail utilise l'atelier de développement formel *Focal* [11], nous permettant ainsi de disposer d'un cadre de spécification et de preuves outillé pour les applications Fractal. De façon à tirer partie des points forts du langage de Focal, nous avons pris appui sur la notion d'*espèce* Focal, qui s'apparente à celle de module paramétré avec héritage multiple, pour réaliser un encodage superficiel de la notion de composant. Les composants et les ports seront assimilés à des espèces et on utilise l'héritage et la paramétrisation pour représenter les liens entre les interfaces et les composants. Ceci permet de rendre compte des aspects fonctionnels des composants Fractal.

Les capacités d'introspection et de reconfiguration des composants, par essence dynamique, ne peuvent être capturés de cette façon. Nous avons donc réalisé un encodage profond en Focal de la notion d'architecture d'un composant et des méthodes de contrôle permettant d'agir sur cette architecture, puis avons injecté cet encodage dans l'encodage superficiel. Nous obtenons ainsi un environnement de spécification et de preuves de propriétés pouvant mélanger les aspects métiers et architecturaux des composants ce qui, à notre connaissance, constitue une approche originale. L'encodage en Focal présenté ici est détaillé dans [5].

L'article est organisé comme suit : la première section présente le modèle Fractal. La deuxième section présente brièvement la spécification formelle de l'API Fractal de façon indépendante du langage Focal. Dans la troisième section, nous présentons l'atelier Focal. Dans la quatrième section, nous présentons l'encodage superficiel de la notion de composant dans Focal. Dans la cinquième section, nous présentons l'encodage profond de l'architecture dynamique des composants en Focal. Dans la sixième section, nous présentons l'injection du modèle d'architecture dans le modèle fonctionnel. Finalement, nous faisons le lien avec des travaux connexes avant de conclure.

2 La programmation par composants en Fractal

La notion de *composant* [18] est apparue comme une extension de la notion d'objet visant à faciliter la conception d'applications réparties. L'objectif était d'offrir une séparation claire entre la partie réalisation d'un module logiciel et ses interfaces, c.a.d., les services qu'il fournit et ceux qu'il requiert. Chaque composant est une boîte noire, *exécutable*, *composable* et identifiable par les services qu'elle offre et requiert. Les applications sont dès lors construites par assemblage de composants, où les interactions sont clairement explicitées et pour lesquels la réutilisation est facilitée. La conséquence est la mise en lumière de l'*architecture logicielle de l'application*, rendant plus facile son analyse et son évolution.

Fractal [9, 8] est un modèle à composants simple et complet, objet de nombreuses extensions et de plusieurs implantations [10, 13]. Un composant Fractal est une entité exécutable [18], munie de *ports*[1] regroupant les services offerts et requis par le composant. Les ports appartiennent à deux catégories (*rôles*) : *serveurs* lorsqu'ils regroupent des services offerts par le composant ; *clients* s'ils regroupent des services requis. Un composant offrant plusieurs ports serveurs fournit autant de *vues* des services qu'il est en mesure d'exécuter. Les ports sont typés par 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

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

Comp `getFcItfOwner(`*Port p*`)`	le composant propriétaire d'un port *p*
`boolean isBound (`*Port p*`)`	teste si un port est lié
Port `getFcInterface(`*Pname n*`,` *Comp c*`)`	donne le port de ce nom dans le composant *c*
`void bindFc (`*Comp* `c,` *Pname* `n,` *PortS* p_s`)`	lie le port client de nom n du composant c

FIG. 1 – Quelques primitives de contrôle de Fractal

terme *port* pour les interfaces et *signature* pour leurs types.

Un port client de composant peut être ou non lié à un port serveur *compatible*[2] d'un autre composant ; l'absence de liaison pouvant provoquer des erreurs. Un port en Fractal possède un nom, qui est désigné afin de concrétiser une liaison. Ainsi, les ports d'un composant sont employés pour caractériser leurs dépendances et capacités logicielles, et pour contraindre et expliciter leurs liaisons éventuelles. Un composant a également un *statut* qui peut être *démarré* ou *stoppé*.

Fractal 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 membranes 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. La figure 1 montre quelques unes de ces primitives.

La prise en compte explicite des capacités de contrôle au niveau des composants est l'une des spécificités de Fractal. Une parmi ses capacités nous intéresse particulièrement : la *reconfiguration dynamique* des liaisons entre ports clients et ports serveurs. Ce type de modification étant pris en charge par les membranes des seuls composants concernés[3], cet aspect constitue un deuxième point remarquable de Fractal : le caractère *distribué* de la connaissance de l'architecture et de son contrôle.

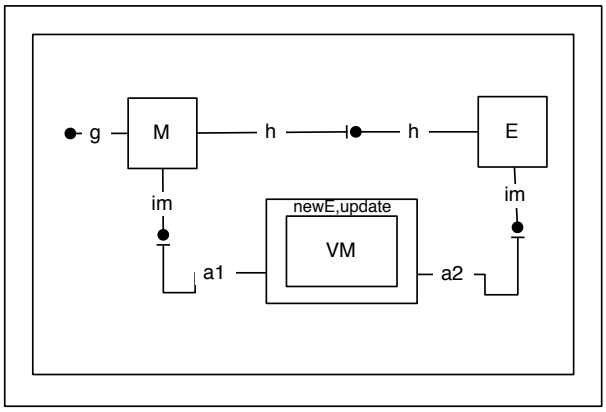
2.1 Exemple d'application avec reconfiguration dynamique

Nous présentons un exemple d'application à composants avec reconfiguration dynamique qui nous servira à illustrer les éléments de modélisation de ce travail. 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, en spécifiant son identifiant ainsi que ses versions minimale et maximale compatibles. L'architecture de l'application est illustrée dans la figure 2.

Nous simplifions le problème en supposant que l'architecture correspond à un assemblage de trois

[2] Par sous-typage, ou par une notion plus sophistiquée de raffinement syntaxique et sémantique.

[3] Et non pas par un moniteur central chargé du contrôle de l'architecture de l'application




```
interface InstallMf{
  String getId();
  String getVersion();
  []TargetApp getTargets();
}

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

FIG. 2 – Architecture et interfaces pour la mise à jour d'une extension

uniques composants : `M`, le composant principal (p.e. *Firefox*), `E`, le composant qui représente une extension déjà installée, et `VM`, le composant qui représente 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 2) 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 deux méthodes de contrôle. La première, de type `E newE()`[4] 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. La seconde méthode de contrôle, `Update`, permet 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.

Dans notre exemple, les méthodes spécifiées par l'interface `InstallMf` correspondent à des services métier, alors que `newE` et `Update` sont des méthodes de contrôle. La modélisation que nous décrivons en section 2.2 autorise de spécifier et de raisonner sur ces deux sortes de méthodes. Par exemple, la spécification de la méthode `Update()` devra prendre en compte aussi bien l'état architectural de l'application et son évolution, que des contraintes de compatibilité fournies par les méthodes de `InstallMf`.

2.2 FracL : spécification formelle du modèle Fractal

Le modèle Fractal est présenté dans [9] par le biais d'une spécification informelle qui présente les concepts clés (composants, ports, interfaces, etc.) du modèle puis décrit les différentes interfaces de contrôle qui constituent l'API Fractal. Plusieurs implantations de ce modèle existent notamment *Julia*, une implantation en Java [7]. Notre premier travail a donc été de spécifier formellement cette API en définissant les entités essentielles du modèle Fractal et les liens qui les unissent, ce qui forme l'*invariant du modèle Fractal*, puis en définissant chaque méthode de l'API par la donnée d'une précondition et d'une post condition exprimant respectivement le *domaine de définition* et la *modification de l'état d'une application* produite par l'exécution de la méthode. Nous avons ainsi une spécification formelle du modèle Fractal indépendante

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

Sig	Signatures	Box	Boîtes
$Pname$	Noms de ports	$Comp$	Composants
$Cname$	Noms de composants	$PortC$	Ports clients
$Role = \{client, server\}$	Catégories de ports	$PortS$	Ports serveur
$Status = \{stopped, started\}$	Statut des composants		

FIG. 3 – Entités de FracL

$name \in Port \rightarrow Pname$	Unicité des noms de ports	(1)
$role \in Port \rightarrow Role$	Un port est client ou serveur	(2)
$sig \in Port \rightarrow Sig$	A un port correspond une seule signature	(3)
$comp \in Port \rightarrow Comp$	Un port appartient à un seul composant	(4)
$bindings \in PortC \nrightarrow PortS$	Un client peut être lié à un seul port serveur	(5)

où $PortC$ et $PortS$ sont définis par

$PortC = dom(role \triangleright \{client\})$	Ensemble de ports clients
$PortS = dom(role \triangleright \{serveur\})$	Ensemble de ports serveurs

Compatibilité des liaisons

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

Les ports d'un composant ont des noms distincts

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

FIG. 4 – Spécification de ports en FracL

de l'atelier Focal, nommée FracL. C'est cette spécification formelle qui fera l'objet d'un encodage profond en Focal. Nos travaux en l'état actuel ne modélisent qu'une version simplifiée des composants composites Fractal que nous appelons les *boîtes*. Notre notion de composant correspond au composant *primitif* de Fractal. L'intégralité de la spécification formelle peut se trouver en [3]. A titre d'illustration, nous en présentons ici un extrait concernant les ports et les composants.

Spécification des ports et composants : La figure 3 donne la liste des entités modélisées : les entités principales sont les composants et leurs ports ; les autres entités représentent leurs attributs. La figure 4 décrit les fonctions et les contraintes portant sur l'ensemble *Port*. Le point remarquable est le fait que les liaisons d'un composant sont portées (mémorisées) par les ports clients de ce composant. La figure 5 décrit les fonctions et les contraintes portant sur les composants. Un composant est caractérisé par son nom, son état (*stopped* ou *started*) et l'ensemble de ses ports.

Spécification des méthodes de L'API : Dans son API, Fractal fournit une méthode d'introspection pour chacune des caractéristiques des entités de FracL. Ainsi par exemple, un port peut exhiber son propriétaire (méthode `getFcItfOwner()`) et un composant peut connaître l'ensemble de ses ports (méthode `getFcInterfaces()`). La spécification de ces méthodes d'introspection est donc directe, tel qu'illustré par la figure 6. Les méthodes

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

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

Les ports d'un composants sont ceux dont il est propriétaire

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

Un composant démarré a tout ses ports clients liés.

$$\forall c \in Comp. (status(c) = started) \Rightarrow ports(c) \cap PortC \subseteq dom(bindings) \tag{11}$$

FIG. 5 – Spécification des composants en FracL

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

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

FIG. 6 – Spécification de deux méthodes d'introspection de l'API

de reconfiguration dynamique concernent essentiellement les liaisons entre ports et l'état des composants. A titre d'exemple, nous donnons dans la figure 7 la spécification de `bindFc(c,n,`p_s`)` qui lie le port client de nom n du composant c au port serveur p_s. Cette méthode est spécifié à l'aide des fonctions $sameNSpace(c,m)$, qui est vraie si deux composants c et m sont soit dans une même boîte, soit dans aucune boîte, et $portOfName(n,c)$, qui donne le port de nom n dans un composant c.

3 Focal

Focal est une plateforme de développement formel permettant de spécifier, implanter et prouver des applications. Il permet de développer des modules de code certifiés, au sens où le code est prouvé correct par rapport aux propriétés énoncées dans la spécification. La notion *d'espèce* Focal sert à structurer le code développé. Elle est formée par la définition plus ou moins abstraite d'un *type de données muni d'une représentation* éventuellement abstraite, d'opérations appelées *méthodes* et de *propriétés*. Une *méthode* peut être simplement déclarée, en fournissant sa signature, ou bien être complètement définie en fournissant le code de son implémentation. Dans ce dernier cas, la syntaxe utilisée est très proche de celle des expressions OCamL. Une *propriété* peut, elle aussi, être soit simplement déclarée, soit complètement prouvée. Dans le

```
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 ∧ p_c ∉ dom(bindings)          // qui est un port client non lié
          ∧ sig(p_c) ⊑ sig(p_s) ∧ status(c)=stopped
          ∧ sameNSpace(c,comp(p_s))
    post : bindings := bindings ∪ {(portOfName(n,c), p_s)}
```

FIG. 7 – Spécification de la méthode `bind` de reconfiguration de l'API

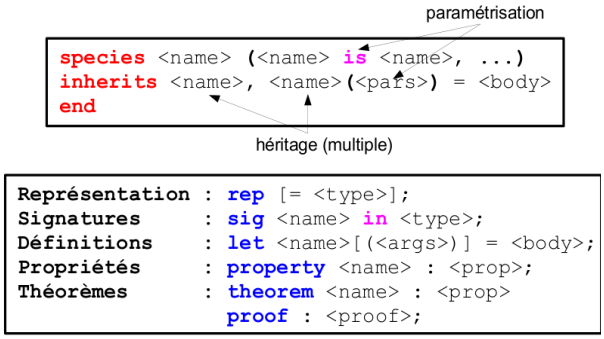


FIG. 8 – Syntaxe Focal

premier cas, il suffit d'énoncer la propriété, dans un langage propre à Focal et basé sur la logique du premier ordre. Dans l'autre cas, on accompagne l'énoncé de la propriété avec une preuve de celle-ci. La syntaxe utilisée pour les preuves est celle de Zenon [6], un prouveur automatique capable également de générer des preuves Coq.

Deux caractéristiques du langage Focal nous intéressent particulièrement : (a) *l'héritage multiple* : une espèce peut hériter d'autres espèces précédemment définies. Dans ce cas, l'espèce fille hérite de toutes les méthodes (définies ou simplement déclarées) et de toutes les propriétés (prouvées ou simplement énoncées) de l'espèce mère. On peut dans une espèce fille donner l'implémentation d'une méthode simplement déclarée dans l'espèce mère et donner la preuve d'une propriété simplement énoncée. (b)*la paramétrisation* : il est possible de paramétrer une espèce par d'autres espèces précédemment définies. Cela permet de manipuler de façon abstraite des objets à travers l'interface des espèces données en paramètre. La figure 8 décrit la syntaxe des espèces, et des expressions pouvant apparaître dans leurs corps.

4 Modèle modulaire paramétrique

En nous appuyant sur les espèces Focal, proches de la notion de module paramétrique et supportant par ailleurs l'héritage multiple, nous avons réalisé un encodage superficiel de la notion de composant qui permet de rendre compte de leur aspect fonctionnel. En effet, dans cet encodage, la spécification d'un composant est *paramétrée* par les ports clients spécifiant les services métier dont le composant dépend, et elle est *étendue* par les ports serveurs spécifiant les services qu'il implante. Nous représentons les signatures de port par des espèces abstraites Focal qui jouent le rôle des *interfaces langage* du modèle Fractal. Par exemple, les signatures des ports de l'exemple présenté en 2.1, figure 2 sont représentées par les espèces de la figure 9. On remarquera, qu'il est possible de spécifier les fonctions de façon plus précise sous forme de pré-condition/post-condition, mais ceci n'est pas nécessaire dans cet exemple simplifié.

Nous représentons les composants également par des espèces Focal. Une espèce Focal modélisant un composant *hérite* des espèces modélisant ses ports serveurs et *est paramétrée* par les espèces modélisant ses ports clients. Ainsi, les composants M, E et VM de notre exemple sont représentés en figure 9. Le composant M est paramétré par l'espèce H, ce qui traduit qu'il a un port client de signature H. Il hérite des espèces G et InstallMf, ce qui exprime le fait qu'il possède deux ports serveurs de signature G et InstallMf. Les composants E et VM sont spécifiés de le même manière. En outre, on a la possibilité de donner à l'intérieur

```
species InstallMf =
  sig getId in self → string;
  sig getVersion in self → string;
  sig getTargets in self → list(targetApp);
end
species G = end

species M(h is H) inherits G, InstallMf= end
species E inherits H, InstallMf= end
```

```
species TargetApp =
  sig getId in self → string;
  sig getMinVersion in self → string;
  sig getMaxVersion in self → string;
end
species H = end

species VM(a1 is InstallMf,
           a2 is InstallMf) = end
```

FIG. 9 – Encodage superficiel de signatures et composants pour la mise à jour

d'une espèce composant des implémentations de fonctions déclarées dans ses interfaces serveurs. On peut aussi exprimer et prouver des propriétés sur elles.

Le modèle paramétrique, étant calqué sur la notion de module paramétré, les dépendances entre les composants sont résolues à la compilation, ce qui ne laisse pas de place à la reconfiguration dynamique. Nous présentons donc dans la section suivante un modèle d'architecture de composants réalisé par un encodage en profondeur de manière à exprimer des propriétés d'introspection et de reconfiguration dynamique.

5 Modèle d'architecture

Ce modèle est basé sur la spécification formelle brièvement décrite dans la section 2.2, qui modélise les notions de port, de composant ainsi que les méthodes de contrôle pour l'introspection et la reconfiguration dynamique. Nous exprimons donc en Focal les propriétés invariantes que doit vérifier toute architecture de composants, ainsi que les méthodes de contrôle de l'API, et enfin nous prouvons la cohérence du modèle. Nous définissons trois espèces : *espèce Port* : qui modélise les ports et leurs liaisons ; *espèce Composant* : formée par l'ensemble de ses ports, et *espèce Architecture* : modélisant l'état global du système de composants. On manipulera cette architecture dans le modèle modulaire pour explorer et modifier l'état du système. Chacune de ces trois espèces [5] possède la structure suivante : (a) déclaration des méthodes d'introspection (accesseurs de l'état), (b) déclaration des méthodes de contrôle (reconfiguration dynamique), (c) spécification des invariants, (d) expression des propriétés de conservation des invariants.

5.1 Espèce Port

Nous présentons ici les grandes lignes de l'espèce `Port`. Tout d'abord sont définies les méthodes d'introspection qui permettent d'accéder à l'état d'un port. Cet état comprend l'adresse du port dans l'architecture, son nom, son rôle (client ou serveur), sa signature ainsi que l'adresse du composant auquel il appartient. La fonction `linkedTo` permet d'obtenir l'adresse de l'éventuel port serveur auquel un port client est lié.

```
species abstract_port( p_address ..., p_name ..., p_role ..., p_signature ... )
inherits setoid =
```

[5] Ces espèces sont purement abstraites. On ne définit pas de champs pour chacun des attributs mais uniquement des accesseurs (fonction d'introspection). On ne fournit pas non plus de code pour les fonctions.

```
  sig addressOf in self → p_address;
  sig nameOf in self → p_name;
  sig roleOf in self → p_role;
  sig signatureOf in self → p_signature;
  sig linkedTo in self → partiel(p_address);
  sig componentOf in self → c_address;
```

On déclare ensuite les méthodes de contrôle, qui modifient l'état d'un port, accompagnées de leur spécification. Par exemple, la fonction `bindTo` qui prend en argument un port client `pc` et l'adresse d'un port serveur `ps` et retourne une nouvelle instance du port client `pc2` telle que `linkedTo(pc2) = ps`.

```
  sig bindTo in self → p_address → self;
  property bindTo_spec : ... ;
  sig unbind in self → self;
  property unbind_spec : ... ;
```

Sont exprimées ensuite les propriétés invariantes. Dans l'espèce `Port`, le seul invariant est qu'un port serveur ne peut pas être lié (ce sont les ports clients qu'on lie aux ports serveurs et non l'inverse). Il est donné par : [6] :

```
  letprop server_not_linked(x in self) =
    !is_server(x) → not !is_linked(x);
```

Pour finir, on exprime les propriétés de conservation des invariants. La propriété `server_not_linked` est vraie pour tout port créé (`server_not_linked__create`), et si la propriété est vraie pour un port client, alors elle est toujours vraie après qu'on lui ait appliqué la fonction `bindTo` (`server_not_linked__bindTo`).

```
  property server_not_linked__create : all ... in ...,
    !server_not_linked(!create(n,r,s,c));
  property server_not_linked__bindTo : all pc ps in self,
    !is_client(pc) → !server_not_linked(pc) → !server_not_linked(!bindTo(pc,ps));
end
```

5.2 Espèce Composant

L'espèce `Composant` est l'espèce principale en ce qui concerne la gestion de la dynamique du modèle et la conservation des invariants par les opérations de liaison des ports, de démarrage et d'arrêt des composants. Cette espèce devant manipuler des ports ainsi que des ensembles de ports, nous lui donnons en paramètre `P` et `P_set`. Puis, de la même manière que pour l'espèce `Port`, nous définissons en premier lieu les méthodes d'introspection. La figure 10 montre le squelette de cette espèce avec les méthodes d'introspection.

C'est à l'intérieur de cette espèce que sont vérifiées la plupart des invariants de l'architecture. Chaque composant maintient la cohérence de son environnement local, ce qui permet de montrer par la suite des propriétés globales au système. Le listing 1 montre un exemple de spécification de ces propriétés. Il s'agit de la contrainte (11) de l'invariant présenté en 2.2, et qui exprime qu'un composant ne peut se trouver dans l'état *started* que si tous ses ports clients sont liés chacun à un port serveur. Ceci peut être spécifié de la

[6] `!is_server(x)` teste si `!roleDe(x)=server` et `!is_linked(x)` teste si `x` a une image par `!linkedTo`. Ces deux fonctions booléenne sont elles aussi définies dans l'espèce `Port`

```
species abstract_component( P is abstract_port, P_set ... ) inherits setoid =
  sig addressOf in self → c_address; (* adresse du composant *)
  sig statusOf in self → c_status; (* statut du composant *)
  sig portsOf in self → P_set; (* ensemble des ports du composant *)
    (* portOfName renvoie le port de nom n du composant si il existe *)
  sig portOfName in self → p_name → partiel(P);
```

FIG. 10 – Squelette de l'espèce `Composant` avec les méthodes d'introspection

manière suivante : pour tout port p1, si le composant c1 est démarré et que p1 est un port client appartenant aux ports de c1, alors p1 doit être lié [7].

Listing 1 – Contrainte sur les composants démarrés

```
  letprop component_started(c1 in self) = all p1 in P,
       !is_started(c1)
    → P!is_client(p1)
    → P_set!member(p1, !portsOf(c1))
    → P!is_linked(p1);
```

5.3 Espèce Architecture

L'espèce Architecture modélise un espace d'adressage de composants. Chaque composant et chaque port y est accessible par son adresse. La fonction `componentAt` (resp. `portAt`) prend en argument une architecture ainsi qu'une adresse de composant (resp. de port) et retourne le composant (resp. port) se trouvant à cette adresse. On a ainsi accès aux méthodes d'introspection et de reconfiguration dynamique sur les composants et ports de l'architecture.

Listing 2 – Espèce Architecture

```
species abstract_architecture ( C is abstract_component, P is abstract_port, ...)
inherits setoid =
  sig componentAt in self → c_address → partiel(C);
  sig portAt in self → p_address → partiel(P);
  sig new in self;                                    (* new is an empty memory *)
  (* createComp returns the new state and the address of the created component *)
  sig createComp in self → list(p_type) → (self * c_address);
```

Les fonctions `new` et `createComp` permettent respectivement de créer une architecture vide et de créer un nouveau composant dans une architecture existante. Nous illustrerons plus loin leur utilisation. Les composants et les ports effectuent les opérations de reconfiguration à leur niveau [8]. Chaque fonction de reconfiguration de l'espèce `Architecture` appelle la fonction correspondante dans l'espèce `Composant` (resp. `Port`) et reconstruit l'état global à partir des reconfigurations locales. C'est le cas par exemple, de la fonction `bind` dans l'espèce `Architecture` ayant la signature suivante :

[7] La notation `P!is_linked(p1)` représente l'application sur l'instance `p1` de la fonction booléenne `is_linked` définie dans l'espèce `P`

[8] Les méthodes de reconfiguration dynamique définies dans les espèces `Composant` et `Port` prennent un composant (resp. port) en paramètre et renvoient une nouvelle instance de composant (resp. port).

```
  sig bind in self → c_address → p_name → p_address → self;
```

Son code remplace un composant dans l'ensemble des composants de l'architecture par le résultat de l'appel de la fonction `bind` de l'espèce `Composant`.

```
  let bind(m, c_adr, n, ps_adr) =
    let old_c = !getComponentAt(m, c_adr) in
    let new_c = C!bind(old_c, n, ps_adr) in
    C_set!replace(old_c, new_c, !componentsOf(m));
```

6 Modèle intégré

La dernière étape dans la mise en place de notre modélisation a été de réunir les deux modèles précédemment exposés en un seul dans le but d'obtenir un environnement de spécification et de preuve ayant la capacité d'exprimer des propriétés mélangeant les aspects métiers et architecturaux des applications. Nous avons pour cela injecté le modèle d'architecture dans le modèle modulaire paramétrique dans le but d'ajouter à ce dernier les capacités d'expression qu'il lui manquait, à savoir celles concernant l'introspection et la reconfiguration dynamique.

Le modèle d'architecture que nous avons mis au point a la propriété d'être totalement générique car il ne manipule les composants et les ports que de manière abstraite et les méthodes de contrôle qu'il modélise sont les primitives de contrôle communes à tout système de composant. Ainsi il nous a été possible d'encapsuler les entités architecturales de composants et de ports dans l'espèce Architecture qui servira d'interface pour toute manipulation de ceux-ci.

6.1 Injection du modèle d'architecture dans le modèle modulaire

Nous définissons une espèce `fun_Component` de laquelle tous les composants fonctionnels hériteront et qui s'occupera de la gestion générique entre les composants fonctionnels et l'architecture.

Ainsi, dans l'espèce `fun_Component` nous définissons une fonction `getAddress` qui permettra d'obtenir pour un composant fonctionnel donné l'adresse de son représentant dans l'architecture et une fonction `create` qui effectuera la création simultanée du composant fonctionnel et de son représentant abstrait dans l'architecture en leur affectant la même adresse. Cette fonction prend en paramètre l'architecture courante et retourne le composant créé ainsi que l'architecture contenant le nouveau composant.

```
species fun_Component =
  sig getAddress in self → component_address;
  sig create in architecture → (architecture * self);
end
```

6.2 Méthodologie de modélisation d'une application à base de composants

Tout d'abord, il faut modéliser l'aspect fonctionnel des composants de l'application de la manière décrite en section 4 en ajoutant l'espèce `fun_Component` comme espèce mère de tous les composants et en implémentant ses méthodes. Par exemple, le composant `E` sera représenté ainsi :

```
species M (h is H) inherits fun_Component, G, InstallMf =
  let getAddress (c) = ...;
  let create (archi) = (architecture!createComp(archi,[('h', port!client);
                                                       ('g', port!server);
                                                       ('im',port!server)]),...);
end
```

On ajoute également dans l'espèce composant (ou dans ses ports serveurs) les fonctions utilisant l'introspection et la reconfiguration dynamique. Ces fonctions prendront en argument l'architecture courante ainsi que chacun des composants fonctionnels qu'elles manipulent pour renvoyer leur valeur de retour accompagnée de l'architecture éventuellement modifiée. Par exemple la fonction `Update` manipule les composants M et E et retourne un nouveau E qui a remplacé l'ancien dans l'architecture. Elle pourra être déclarée dans le composant VM par la signature suivante :

```
species VM (a1 is InstallMf, a2 is InstallMf) inherits fun_Component =
  ...
  sig Update in architecture → M → E → (architecture * E);
  ...
end
```

On part ensuite d'une architecture vide dans laquelle on crée chacun des composants un par un grâce à la fonction `create` de chaque espèce de composant. Puis on effectue les liaisons initiales.

```
let archi0 = architecture!new in
let (archi1, m) = M!create(archi0) in
let (archi2, e) = E!create(archi1) in
let (archi3, vm) = VM!create(archi2) in
(* on lie le port client h du composant m au port serveur h du composant e *)
let archi4 = architecture!bind(archi3, M!getAddress(m), 'h',
                               architecture!portAt(archi3, (E!getAddress(e),'h'))) in
...
let archi_initiale = ...;
```

6.3 Exemple d'expression de propriété mixte

Au sein des composants, on pourra exprimer des propriétés mélangeant les aspects fonctionnel et architectural comme on peut le voir dans l'expression de la spécification simplifiée de la méthode `Update`[9] montrée en figure 11.

[9] `#portOfName(c,n)` retourne le port de nom `n` du composant à l'adresse `c` et est définie en utilisant les fonctions d'introspection du modèle d'architecture `#portOfName(a,c,n) = component!portOfName(architecture!componentAt(a,c), n)`

```
property Update_specification :
  all archi1 archi2 in architecture,
  all e1 e2 in E,
  all m in M,
  (* m et e sont lies par leur port h *)
  port!linkedTo(#portOfName(archi1, M!getAddress(m), 'h'))
  = #portOfName(archi1, E!getAddress(e), 'h')
  → (archi2, e2) = Update(archi1, m, e1)
  → E!getVersion(e1) < E!getVersion(e2);
```

FIG. 11 – Propriété mixte : spécification de la méthode `Update` pour la mise à jour

7 Travaux connexes

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. C'est ce que nous appelons *l'aspect fonctionnel des composants*. De nombreux travaux se consacrent à la spécification formelle et à la vérification de ces aspects fonctionnels. C'est le cas par exemple des travaux [17, 16], qui font un encodage faible en B de la notion d'interface permettant d'utiliser le raffinement pour vérifier la compatibilité entre interfaces. Notre approche se veut plus générale puisqu'elle vise à prendre en charge à la fois les aspects architecturaux et fonctionnels des composants.

Kmelia [4, 2] est un modèle de spécification de composants outillé permettant de modéliser des architectures logicielles et leurs propriétés. Nous avons donc en commun la prise en charge unifiée des composants. Cependant, dans ce système, les notions de première classe sont les services, là où nous privilégions les ports. La seconde différence est que 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.

Plusieurs travaux se consacrent spécifiquement à l'élaboration de modèles formels de la reconfiguration dynamique des composants. On peut citer [19] et [12] qui ont en commun l'utilisation d'ALLOY [15, 1]. Nos modèles de l'architecture sont voisins en ce qui concerne les aspects touchant 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 la prise en charge du composant dans tous ses aspects. La seconde différence est l'aspect décentralisé de notre modèle. Leurs modélisations rendent compte d'un système où la gestion du contrôle est 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. Finalement, l'utilisation d'ALLOY permet une analyse par la recherche de modèles ou de contre modèles, là où celle de Focal permet de faire des preuves.

8 Conclusion

Nous avons présenté dans cet article, une formalisation du modèle à composants Fractal ainsi que son encodage en Focal.

Ce travail fournit un cadre formel permettant d'exprimer et d'analyser des propriétés de systèmes à base

de composants Fractal. Il fournit aussi une sémantique formelle d'une partie du modèle Fractal qui jusque là n'était défini que par une spécification informelle. Il offre de plus le moyen de disposer d'un cadre outillé pour spécifier et raisonner sur des applications à composants concrètes.

La diversité des préoccupations prise en compte par notre cadre formel autorise l'expression et l'analyse des propriétés portant autant sur des aspects métiers d'une application que sur sa reconfiguration dynamique et son introspection. Notre travail constitue dans ce sens un cadre de raisonnement unifié où l'on retrouve les concepts et les capacités de preuve jusque là traités séparément. Cela nous permet le traitement de propriétés mixtes, difficilement spécifiables dans des cadres formels plus spécialisés. Il s'agit à notre connaissance de la formalisation à vocation la plus complète qui existe actuellement pour Fractal.

Les perspectives de ce travail concernent l'extension de la modélisation proprement dite, et celle de l'encodage associé ainsi que l'expérimentation sur des exemples concrets. Les extensions du modèle et de l'encodage 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.

Références

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

[2] Pascal André, Gilles Ardourel, and Christian Attiogbé. Spécification d'architectures logicielles en Kmelia : hiérarchie de connexion et composition. In *1ère Conférence Francophone sur les Architectures Logicielles*, pages 101–118. Hermès Sciences Publications - Lavoisier, 2006.

[3] M.V. Aponte and M. Simonot. Une approche formelle de la reconfiguration dynamique. Technical Report 1590, CNAM-CEDRIC, 2008.

[4] Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition (ETAPS/SC'06)*, volume 4089 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.

[5] V. Benayoun. Composants fractal avec reconfiguration dynamique : formalisation avec l'atelier focal. http ://reve.futurs.inria.fr/, 2008.

[6] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In Nachum Dershowitz and Andrei Voronkov, editors, *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.

[7] E. Bruneton. Julia tutorial. http ://fractal.objectweb.org/tutorials/julia/index.html, 2002.

[8] E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga (Spain), 2002.

[9] E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal component model. http : fractal.objectweb.org/specification/index.html, 2004.

[10] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2003)*, Edinburgh, Scotland, May 2004.

[11] T. Hardin C. Dubois and V. Viguié Donzeau-Gouge. Focal : an environment for developing certified components. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programing*. 2006.

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

[13] OW2 Consortium. Cecilia tutorial. http ://fractal.objectweb.org/cecilia-examples-site/current/helloworld/index.html, 2008.

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

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

[16] 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.

[17] 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.

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

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