Une approche par aspects pour le développement de composants Fractal adaptatifs

Pierre-Charles DAVID [1] — **Thomas LEDOUX**

France Télécom, Recherche & Développement
28, chemin du vieux chêne
F-38243 MEYLAN
PierreCharles.David@francetelecom.com

Groupe OBASCO, EMN / INRIA, LINA
École des Mines de Nantes
4, rue Alfred Kastler
F-44307 NANTES CEDEX *3*
Thomas.Ledoux@emn.fr

RÉSUMÉ. *Les développeurs d'applications sont aujourd'hui confrontés à des contextes d'exécution de plus en plus variables, qui nécessitent la création d'applications capables de s'adapter de façon autonome aux évolutions de ces contextes. Dans cet article, nous montrons qu'une approche par aspects permet de construire des applications adaptatives dans lesquelles le code d'adaptation est modularisé aussi bien sur le plan spatial que temporel. Concrètement, nous proposons* SAFRAN, *une extension du modèle de composants Fractal permettant le développement de l'aspect d'adaptation sous la forme de politiques réactives. Celles-ci détectent les évolutions du contexte d'exécution et adaptent le programme de base en le reconfigurant.* SAFRAN *permet ainsi de développer l'aspect d'adaptation de façon modulaire et de le tisser dynamiquement dans les applications.*

ABSTRACT. *Today application developers have to deal with an increasingly variable execution context, requiring the creation of applications able to adapt themselves autonomously to the evolutions of this context. In this paper, we show how an aspect-oriented approach enables the development of self-adaptive applications where the adaptation code is well modularised, both spatially and temporally. Concretely, we propose* SAFRAN, *an extension of the Fractal component model for the development of the adaptation aspect as reactive adaptation policies. These*

1. Ce travail a été effectué dans le cadre de la thèse de l'auteur au sein du groupe OBASCO.

policies detect the evolutions of the execution context and adapt the base program by reconfiguring it. This way, SAFRAN *allows the development of the adaptation aspect in a modular way and its dynamic weaving into applications.*

MOTS-CLÉS : *composants, Fractal, reconfiguration dynamique, aspect d’adaptation*

KEYWORDS: *components, Fractal, dynamic reconfiguration, adaptation aspect*

1. Motivation & problématique

Les développeurs d'applications sont aujourd'hui confrontés à des contextes d'exécution de plus en plus variables. D'une part, on trouve une grande diversité de plates-formes couvrant un large spectre en terme de ressources disponibles (des systèmes embarqués aux grilles de calculs), ces machines hétérogènes étant de plus en plus reliées par le réseau et donc interdépendantes. D'autre part, le contexte d'exécution d'une application donnée évolue de plus en plus pendant son exécution (disponibilité des ressources matérielles et logicielles, nomadisme, etc.). Enfin, une même application peut être utilisée dans des situations qui nécessitent des modes de fonctionnement différents (par exemple un téléphone portable au bureau, en voiture, dans un lieu public…), et par des utilisateurs aux niveaux d'expertise et aux besoins spécifiques.

Cette situation rend le développement d'applications de plus en plus complexe, car il est souvent difficile de connaître au moment du développement les conditions précises dans lesquelles les applications seront utilisées, d'autant que ces conditions évoluent souvent de façon imprévisible en cours d'exécution. Plutôt que d'ignorer le contexte d'exécution ou de tenter de le masquer par des couches d'abstraction (intergiciel), nous pensons que les applications doivent au contraire être conscientes de leur environnement (*context-aware*) afin de pouvoir s'y adapter (Dey *et al.*, 2000). De telles *applications adaptatives* sont capables de s'adapter elles-mêmes, de façon autonome (Kephart, 2002), aux évolutions de leur contexte d'exécution afin non seulement de continuer à fonctionner, mais aussi de tirer le meilleur parti des nouvelles possibilités qui peuvent apparaître dynamiquement.

Le besoin de construire des applications qui s'adaptent à leur environnement n'est pas nouveau mais la situation actuelle rend cet objectif à la fois plus important et plus difficile à atteindre que jamais. Les techniques *ad hoc* utilisées généralement, dans lesquelles les décisions d'adaptation sont « câblées » dans l'application, ne sont pas suffisantes : le mélange du code métier et du code chargé d'adapter celui-ci implique une complexité accrue de l'application, qui rend son développement et sa maintenance difficile (Dowling *et al.*, 2001). De plus, il est le plus souvent impossible de prévoir pendant le développement les circonstances dans lesquelles une application sera utilisée, et encore moins les réactions appropriées. On aimerait pouvoir *modulariser* le code chargé de l'adaptation et *l'intégrer* dynamiquement dans le code métier afin de garantir un découplage spatio-temporel entre les deux types de code.

Dans cet article, nous proposons d'utiliser une approche par aspects pour modulariser le code d'adaptation lors du développement d'applications adaptatives. Les techniques de la programmation par aspects (Kiczales *et al.*, 1997, Filman *et al.*, 2005) nous offrent un cadre intéressant pour capturer l'essence de l'adaptation : développement séparé du code de l'adaptation dans un langage prenant en compte sa nature spécifique ; outils pour assembler (tisser) et désassembler dynamiquement le code métier et le code d'adaptation.

Notre proposition est fondée sur SAFRAN, un système pour le développement d'applications adaptatives basé sur le modèle de composants Fractal (Bruneton *et al.*, 2003). SAFRAN est constitué *(i)* d'un langage dédié permettant de programmer *l'aspect d'adaptation* sous la forme de politiques réactives, et *(ii)* du support d'exécution nécessaire au *tissage* et à l'exécution de ces politiques (aspects) dans les composants Fractal (programme de base).

La section 2 présente la problématique de l'adaptation des logiciels et montre comment cette préoccupation peut être – conceptuellement – considérée comme un aspect. La section 3 présente ensuite notre contribution, SAFRAN, en tant que système à aspect, en montrant comment cette similitude conceptuelle se traduit concrètement dans l'architecture de SAFRAN. Nous illustrons ensuite l'utilisation de SAFRAN sur un exemple simple (Section 4), puis présentons quelques travaux connexes (Section 5) avant de conclure (Section 6).

2. L'adaptation : de la préoccupation transverse à l'aspect

2.1. *L'adaptation : une préoccupation transverse*

D'après le dictionnaire, adapter consiste à « rendre un dispositif apte à assurer ses fonctions dans des conditions particulières ou nouvelles ». Cette définition suppose la présence d'un système (le dispositif), qui réalise un objectif particulier (ses fonctions). La façon dont le système réalise cet objectif dépend – au moins en partie – de l'environnement, du contexte dans lequel il se trouve. Lorsque le contexte est modifié, le système doit être ajusté pour qu'il soit à même de réaliser son objectif au mieux dans ces nouvelles conditions. Une adaptation est ainsi une *modification* d'un système, suite à la *décision* d'un ajustement en réponse à *un changement dans son contexte*.

Cette courte description informelle constitue notre point de départ pour modéliser l'adaptation dans le monde du logiciel. L'adaptation d'un logiciel sera représentée par un programme chargé de *(i)* l'observation de l'environnement dans lequel évolue le logiciel, *(ii)* la prise de décision des ajustements à apporter au logiciel suite à la détection d'un changement significatif dans l'environnement, et enfin *(iii)* l'application des modifications choisies pour adapter le logiciel.

Avec l'émergence de l'informatique nomade, les nouvelles applications doivent être capables de s'adapter elle-mêmes de façon autonome (Kephart, 2002) aux divers contextes d'exécution auxquels elles seront confrontées, et aux évolutions dynamiques de ceux-ci. Il devient ainsi nécessaire d'embarquer le programme chargé de l'adaptation dans les applications elles-même. On nommera *logiciel adaptatif* un logiciel qui s'adapte automatiquement et de façon autonome.

D'un point de vue génie logiciel, la mise en œuvre de l'adaptation pose plusieurs problèmes. Tout d'abord, l'intégration du code chargé de l'adaptation dans l'application même augmente sa complexité : le code métier se retrouve « pollué » par des préoccupations non-fonctionnelles comme l'observation de l'environnement et le choix

de la réaction appropriée. De plus, les composants (au sens large) développés de cette manière sont faiblement réutilisables : ils ne sont capables de fonctionner que dans les situations anticipées pendant leur développement, voire uniquement dans une application spécifique. En effet, dès qu'une application atteint une certaine taille, les adaptations de ces différentes parties doivent être coordonnées pour rester globalement cohérentes, ce qui augmente le couplage entre ces différentes parties, par rapport à ce que le code métier seul nécessite. On aimerait donc un couplage plus lâche et plus dynamique entre le code métier et le code chargé de l'adaptation.

L'adaptation apparaît donc comme une préoccupation transverse au code métier que l'on aimerait modulariser afin de garantir une meilleure réutilisation et maintenabilité du code métier. Les techniques de la programmation par aspects (Filman *et al.*, 2005) vont nous offrir un cadre de solution à cette problématique de modularisation.

2.2. *Vers un aspect d'adaptation*

Un aspect est un module qui regroupe et intègre des couples $(coupe, action)$ qui ensemble implémentent une préoccupation, en modifiant la sémantique du programme de base (i.e. métier) de façon transparente pour le programmeur du code de base (Filman *et al.*, 2000). Les actions sont des fragments de code à exécuter à certains points du programme appelés *points de jonction*, qui peuvent correspondre soit à des éléments du code source du logiciel (point d'entrée d'une méthode par exemple (Kiczales *et al.*, 2001)), soit à des événements se produisant lors de l'exécution (appel de méthode (Douence *et al.*, 2002)). Ces points de jonction sont groupés en ensembles appelés coupes. Le programme de base et les aspects sont ensuite *tissés* en un tout cohérent, soit statiquement (Kiczales *et al.*, 2001), soit dynamiquement (Douence *et al.*, 2002, Pawlak *et al.*, 2001).

Pour « aspectiser » (i.e. déterminer les couples $(coupe, action)$) notre préoccupation transverse adaptation, notre point de départ a été de s'intéresser au *processus d'adaptation*. Celui-ci est dynamique et réactif : un changement significatif (i.e. un événement) s'est produit dans l'environnement, il faut prendre une décision d'adaptation et la réaliser. Le caractère événementiel de l'adaptation possède un point commun avec la vision EAOP (Event-Based AOP) de la programmation par aspects. (Douence *et al.*, 2002) ont montré avec EAOP que les points de jonction de la *coupe* pouvaient être vus comme des *événements* relatifs à l'exécution du programme (invocation de méthode, création d'objet…). Dans cette approche, le rôle du tisseur est de coordonner / synchroniser l'exécution du programme de base et des actions des aspects par rapport à l'occurrence de ces événements. Ces événements *endogènes* (liés à l'exécution du programme lui-même) ne sont pas suffisants pour déclencher l'adaptation dans le cadre plus général des applications sensibles à leur contexte (*context-aware*). Une application adaptative doit en effet être capable de réagir aux évolutions de son contexte d'exécution comme par exemple l'apparition d'un nouveau périphérique ou la chute brutale de la bande passante disponible. Les événements qui correspondent aux changements significatifs du contexte, seront nommés événements *exogènes*. Mal-

gré leur origine différente (le contexte plutôt que l'application elle-même), nous pensons que ces événements peuvent aussi être considérés comme des points de jonction, puisqu'ils déclenchent l'exécution des actions d'adaptation. Notre proposition pour les *coupes* de l'aspect d'adaptation est donc l'extension du domaine des points de jonction à l'ensemble du contexte de l'application. Les événements *endogènes* correspondent ainsi aux points de jonction traditionnels, alors que les événements *exogènes* (liés au contexte d'exécution) constituent un nouveau type de points de jonction qui permet de réagir aux évolutions du contexte, enrichissant par là même le pouvoir d'expression des aspects.

Concernant les *actions* déclenchées par la détection d'un point de jonction (i.e. un événement exogène ou endogène s'est produit), elles vont tout simplement correspondre aux modifications à mettre en oeuvre pour adapter le programme de base. Notons que contrairement à AspectJ (Kiczales *et al.*, 2001) qui est un langage d'aspects généraliste, nos actions seront programmées dans un langage dédié à l'adaptation et aux types de modification alors supportés (reconfiguration, spécialisation...).

Concernant le *tissage*, le choix de s'intéresser à l'adaptation dans des domaines ouverts et non figés (e.g. informatique mobile), nous amène à penser que le tissage de l'aspect d'adaptation doit pouvoir être dynamique. Si le programme de base était tissé statiquement avec l'aspect d'adaptation, le code métier deviendrait inutilisable dans des contextes qui n'ont pas été prévus a priori. Nous allons donc privilégier le tissage dynamique permettant un découplage spatio-temporel entre métier et adaptation.

3. Un aspect d'adaptation dans SAFRAN

SAFRAN[1] (David, 2005) est une extension du modèle de composants Fractal (Bruneton *et al.*, 2003) pour la création d'applications adaptatives. Un des points clefs de notre conception est de considérer l'adaptation comme un *aspect*.

En repartant du schéma générique d'un aspect d'adaptation décrit dans 2, l'adaptation dans SAFRAN est caractérisé par :

– un *programme de base* correspondant à une architecture de composants Fractal ;

– des *coupes* correspondant à la notification d'événements endogènes (invocations de messages sur interfaces Fractal, modification des connexions...) ou exogènes (grâce à un framework pour le développement d'applications sensibles au contexte) ;

– des *actions* volontairement restreintes à la reconfiguration de l'architecture (attributs de configuration et structure de l'application) afin de pouvoir offrir des garanties de cohérence ;

– et enfin l'*aspect* d'adaptation lui-même, intégrant coupes et actions, et représenté par des politiques d'adaptation modulaires tissées dynamiquement grâce à un nouveau contrôleur Fractal.

1. Self-Adaptive FRActal compoNents

Le reste de cette section va présenter en détail la mise en œuvre de chacun de ces points.

3.1. *Programme de base : composants Fractal*

Fractal (Bruneton *et al.*, 2003) est un modèle de composants développé par France Télécom R&D et distribué dans le cadre du consortium ObjectWeb. Nous avons choisi de baser nos travaux sur Fractal car il s'agit d'un modèle simple mais complet, à la fois dynamique et extensible.

Une application Fractal est vue comme un assemblage de composants, chacun constitué de deux parties : un *contrôleur* et son *contenu*. Ces composants communiquent deux à deux de façon synchrone en s'envoyant des messages. Les types de message qu'un composant est capable d'envoyer ou de recevoir sont matérialisés par des *interfaces* portées par le contrôleur du composant et visibles de l'extérieur. Le rôle du contrôleur d'un composant est double : *(i)* interpréter directement certains messages, reçus par ses interfaces dites *de contrôle* et qui permettent d'introspecter et de reconfigurer le composant, et *(ii)* déléguer les autres messages – entrants ou sortants – à leur destinataire final, après les avoir éventuellement manipulés. Le contenu d'un composant peut être constitué soit d'autres composants, auquel cas on parle de *composant composite*, soit d'un objet (au sens large) du langage de programmation sous-jacent, auquel cas on parle de *composant primitif*.

Le caractère dynamique des composants Fractal permet de reconfigurer facilement l'architecture d'une application en cours d'exécution. Les interfaces de contrôle standard fournies par les composants Fractal permettent ainsi de créer de nouveaux composants, de modifier le contenu des composites en y ajoutant ou en en retirant des sous-composants, et de créer ou supprimer des connexions entre interfaces.

L'autre caractéristique de Fractal qui rend ce modèle particulièrement approprié pour la construction d'applications adaptables est sa réflexivité. Par rapport à des modèles de composants comme ArchJava (Aldrich *et al.*, 2002), qui supportent les reconfigurations dynamiques uniquement si elles sont codées « en dur », la réflexivité permet la découverte et la modification de la structure des composants de façon non-anticipée. Ce point est essentiel pour la création d'applications adaptatives (Redmond *et al.*, 2002) car la plupart des adaptations ne sont par définition pas connues au moment de la construction initiale du logiciel.

Au-delà des avantages inhérents à la programmation par composants, les caractéristiques spécifiques de Fractal en font un candidat idéal pour créer des applications adaptables, premier pas vers des applications adaptatives autonomes.

3.2. *Actions de reconfiguration avec FScript*

Nous avons réalisé un langage dédié (van Deursen *et al.*, 2000) appelé FScript permettant de programmer les reconfigurations de composants Fractal déclenchées suite à la notification d'un événement endogène ou exogène. FScript est un langage procédural simple qui donne accès à toutes les opérations supportées par les composants Fractal : création de composants, introspection de l'architecture, et reconfiguration de cette dernière en manipulant le contenu des composites et les connexions entre interfaces. Les caractéristiques principales de FScript sont *(i)* une notation spécifique pour naviguer facilement dans l'architecture Fractal du programme de base, et *(ii)* la garantie que les reconfigurations laissent toujours l'application dans un état consistant. Bien qu'il ait été conçu dans le cadre de SAFRAN, FScript peut être utilisé individuellement pour programmer des reconfigurations de composants Fractal en dehors du cadre de SAFRAN.

3.2.1. *Notation FPath*

FScript utilise une syntaxe spécifique, FPath (inspirée du langage XPath (World Wide Web Consortium, 1999)), pour *naviguer* facilement dans une architecture Fractal sans la modifier et *sélectionner* les éléments (composants, interfaces, attributs) répondant à certains critères. Le langage repose sur la modélisation des composants Fractal sous la forme d'un graphe orienté dont les nœuds représentent les composants, leurs interfaces et attributs, et dont les arcs sont annotés par des *labels* qui dénotent le type de relation entre deux nœuds (interface, sous-composant, connexion...). En plus des types d'expressions habituelles (arithmétique, combinateurs booléens et comparaisons, etc.), FPath ajoute la possibilité d'exprimer des *chemins relatifs* (à un composant de départ). Un chemin consiste en une suite de pas, chacun constitué de trois éléments : `axe::test[predicat]`. À chaque pas, un ensemble de nœuds de départ est converti en un nouvel ensemble en suivant les arcs du graphe identifié par l'axe, puis en filtrant le résultat grâce au *test* sur le nom du nœud et aux *prédicats* optionnels. Plus précisément, l'algorithme d'évaluation d'un pas est le suivant :

P1. [Initialisation] $result \leftarrow \emptyset$.

P2. [Sélection] Sélectionne tous les nœuds connectés à l'un des nœuds courants par un arc dont le label correspond à l'axe : $result \leftarrow \cup\{n : c \xrightarrow{axis} n, c \in current\}$.

P3. [Test] Si le test est un identifiant (plutôt que `*`), retire de $result$ les nœuds dont le nom ne correspond pas : $result \leftarrow \{n \in result : name(n) = test\}$.

P4. [Filtrage] Ne conserve que les éléments pour lesquels tous les prédicats sont vrais : $result \leftarrow \{x \in result : pred_1(x) \wedge \cdots \wedge pred_n(x)\}$.

P5. [Fin] L'algorithme se termine et renvoie $result$.

Pour un chemin constitué de plusieurs pas, cet algorithme est répété avec le résultat du pas précédent comme ensemble courant pour le pas suivant.

FPath offre un ensemble d'axes pour naviguer dans les architectures Fractal, en sélectionnant les interfaces d'un composant (axe `interface`), ses attributs de confi-

guration (`attribute`), ses sous-composants directs (`child`) ou parents[2] (`parent`), et en suivant une connexion à partir d'une interface (`binding`). Il est aussi possible de sélectionner en une seule fois tous les sous-composant (resp. parents) directs et indirects d'un composant avec l'axe `descendant` (resp. `ancestor`), qui est la fermeture transitive de `child` (resp. `parent`).

Par exemple, l'expression FPath `child::server/attribute::cacheEnabled` sélectionne dans un premier temps les sous-composants (axe `child`) nommés `server` (test sur le nom du nœud) du composant initial pour finalement retourner son attribut de configuration nommé `cacheEnabled`. De même, l'expression `count(interface::*[required(.) and not(bound(.))]) > 0` renvoie *vrai* si et seulement si le composant initial possède des interfaces requises qui ne sont pas encore connectées (dans un prédicat, le point « . » correspond au nœud courant sur lequel le prédicat est évalué).

3.2.2. *Actions FScript*

FScript permet de définir des *actions de reconfiguration*, en combinant expressions FPath, structures de contrôles simples et manipulation de variables (référencées par `$nomVar`). Toutes les opérations de reconfiguration dynamique supportées par Fractal sont disponibles pour les programmes FScript sous la forme d'actions prédéfinies, dites primitives, y compris les actions `attach()` et `detach()` introduites par SAFRAN et qui permettent à une action d'adaptation d'attacher ou de détacher des politiques SAFRAN aux composants Fractal. L'exemple suivant montre la définition d'une action de reconfiguration FScript qui pourrait être utilisée pour adapter un composant SAFRAN.

```
// Change la politique de remplacement d'un cache.
action select-strategy(cache, strat) = {
  // Obtient l'interface client du cache vers sa stratégie
  itf := $cache/interface::strategy;
   // Est-elle déjà connectée à une interface serveur ?
  if (bound($itf)) {
    // La déconnecte et stoppe le composant maintenant inutilisé.
    previous := $itf/binding::*;
    unbind($itf);
    stop($previous/component::*);
  }
  // Connecte l'interface client du cache à l'interface
  // serveur appropriée de $strat
  bind($itf, $strat/interface::replacement-strategy);
  // S'assure que le composant $strat est bien démarré.
  start($strat);
}
```

2. Fractal supporte le partage de composants, un composant peut donc avoir plusieurs parents.

Cette action permet de modifier la stratégie de remplacement utilisée par un composant cache en modifiant la connexion qui relie le composant cache au composant stratégie. Elle utilise pour cela des expressions FPath pour naviguer dans la structure de l'application, et des actions primitives qui correspondent aux opérations supportées par Fractal (`bind()`, `stop()`...). Notons qu'avec FScript, il n'est pas nécessaire de stopper explicitement un composant avant de le reconfigurer. Si une opération particulier (par exemple `bind()` ci-dessus) nécessite que le composant cible soit arrêté, l'interprète FScript le stoppera automatiquement si nécessaire. À la fin d'une reconfiguration, l'interprète redémarrera automatiquement tous les composants qu'il a stoppé de lui-même, après avoir vérifié que cela ne génèrera pas d'erreur (i.e. toutes leurs interfaces requises sont bien connectées).

Si l'action ci-dessus est relativement spécifique à une application particulière, FScript permet aussi de programmer des actions plus génériques (remplacement d'un composant par un autre par exemple) réutilisables dans de nombreuses applications (motifs architecturaux).

3.2.3. *Garanties offertes par FScript*

La conception et l'implémentation de FScript garantissent la consistance des reconfigurations. Ces reconfigurations FScript étant destinées à adapter des applications en cours d'exécution, nous devons garantir que les reconfigurations ne risquent pas de rendre l'application inutilisable. Pour cela, nous avons choisi un certain nombre de critères de consistance, en particulier l'*intégrité transactionnelle* (atomicité, consistance de l'état final, isolation) et la *terminaison en temps borné* des reconfigurations. La validation de ces critères est garantie d'une part par la structure même du langage, dont le pouvoir d'expression est volontairement limité, et d'autre part par son implémentation. Plus précisément :

– La définition d'actions directement ou indirectement récursives est interdite, et la seule structure du contrôle disponible pour itérer, une boucle `for each`, itère sur le résultat d'un expression FPath, qui renvoie toujours un ensemble fini de nœuds. Ces contraintes garantissent la *terminaison* des actions en temps fini mais non borné.

– Au cours de l'exécution d'une reconfiguration, l'interpréteur du langage conserve un journal complet des toutes les actions primitives effectuées, avec toutes les informations nécessaires pour les annuler / inverser. Dès qu'une erreur se produit, l'interpréteur utilise ce journal pour annuler (roll-back) la reconfiguration en cours et revenir à l'état initial. Étant donné que toutes les opérations primitives de Fractal sont elles-mêmes atomiques et réversibles, ceci garantie l'*atomicité* des reconfigurations FScript.

– À la fin d'une reconfiguration, l'interprète vérifie que la configuration en cours est cohérente, c'est-à-dire que toutes les interfaces clientes obligatoires sont correctement connectées à une interface serveur compatible et que tous les composants qui ont été stoppés temporairement pendant la reconfiguration peuvent être redémarrés. Si cela n'est pas le cas, l'interpréteur annule la reconfiguration en cours et revient à l'état initial, assurant ainsi la cohérence de l'application.

– Enfin, l'*isolation* des reconfigurations est pour l'instant garantie en les sérialisant au niveau global : une reconfiguration peut s'exécuter en parallèle avec un ou plusieurs threads applicatifs, mais une seule reconfiguration peut être en cours à un moment donné. Bien qu'elle fonctionne, cette approche interdit l'exécution concurrente de reconfigurations, même lorsqu'elles ne se chevauchent pas, et pourrait être améliorée dans l'avenir.

3.3. *Coupes et points de jonctions*

Nous décrivons maintenant les points de jonctions supportés par SAFRAN pour déclencher l'exécution des adaptations. Suivant l'approche proposée par (Douence *et al.*, 2002), nous considérons ces points de jonctions comme correspondant à l'occurrence d'*événements*. Si traditionnellement les points de jonction ne prennent en compte que l'exécution du programme de base lui-même, notre problématique nous a conduit à étendre le domaine considéré. Ainsi, en plus des événements *endogènes* correspondant au programme de base, SAFRAN supporte la détection d'événements *exogènes*, qui correspondent aux évolutions du contexte d'exécution[3].

Quelle que soit leur origine, toutes les occurrences d'événements dans SAFRAN sont représentées par des ensembles de propriétés, certaines étant communes, et d'autres spécifiques au type d'événement. Les propriétés communes à tous les événements sont : le `type` de l'événement, sous la forme d'une chaîne de caractères, la `source` de l'événement, qui peut être soit un composant Fractal soit un élément du contexte d'exécution (voir plus loin), et `timestamp`, l'instant où l'événement s'est produit.

La spécification des événements à détecter se fait grâce à un *descripteur d'événement*, dont la syntaxe exacte varie selon le type d'événement mais a toujours la forme générale `type-événement(paramètres)`. Ainsi, le descripteur `changed(sys://storage/memory#free)` permet de désigner les variations de la quantité de mémoire disponible.

3.3.1. *Événements endogènes*

Les événements endogènes correspondent à des points d'exécution dans le programme de base, qui dans notre cas est un ensemble de composants Fractal.

Les trois premiers types d'événements correspondent à l'invocation de messages des interfaces Fractal : `message-received` indique la réception d'un message (avant son exécution), `message-returned` indique qu'un message s'est terminé correctement et `message-failed` indique au contraire qu'un message s'est

3. Dans l'avenir, nous envisageons d'enrichir ce système de coupes pour supporter des coupes plus complexes, et en particulier des coupes hybrides faisant intervenir à la fois des événements endogènes et exogènes, qui permettraient de mieux coordonner l'exécution des adaptations avec celle du programme de base.

terminé par une erreur (exception). Les descripteurs de ces trois types d'événements utilisent les mêmes paramètres, sous la forme d'expressions FPath, pour indiquer quelles interfaces et quelles méthodes doivent être observées, par exemple `message-received($c/interface::logger)` détecte les invocations de n'importe quelle méthode de l'interface `logger` du composant désigné par la variable `$c`, alors que l'expression `message-failed($c/interface::*)` détecte les erreurs survenues sur n'importe quelle interface de service de ce même composant.

Les autres types d'événements endogènes correspondent aux reconfigurations possibles des composants Fractal : création de composants (`component-created`), démarrage et arrêt (`component-{started,stopped}`), paramètrage (`parameter-changed`), modification du contenu (`subcomponent-{added,removed}`) et enfin manipulation des connexions (`binding-{created,destroyed}`). Les descripteurs de chacun de ces types de composants prennent des arguments afin de spécifier plus ou moins finement les composants, interfaces ou attributs à observer. Ainsi, le descripteur `component-started($c/child::*)` permet de détecter le démarrage de n'importe quel sous-composant direct de `$c`. Afin d'éviter les problèmes de déclenchements de règles en cascade, ces événements structurels ne sont pas générés lors des reconfigurations qui sont déclenchées par une règle.

L'implémentation de ces événements repose sur l'instrumentation des contrôleurs Fractal associés, par exemple `lifecycle-controller` pour les événements `component-{started,stopped}`. Cette instrumentation est très simple dans Julia, l'implémentation de référence de Fractal. L'ensemble des types d'événements endogènes peut donc être facilement étendu pour supporter de nouvelles extensions du modèle Fractal.

La portée des événements endogènes est globale, c'est-à-dire que n'importe quelle règle d'adaptation, quel que soit composant auquel elle est associée, peut détecter les événements issus de n'importe quel autre composant de l'application à partir du moment où celui-ci est désigné par le descripteur d'événement. Bien que cela rende le système très puissant, cela le rend aussi difficile à implémenter de façon efficace, puisque sans notion de localité n'importe quel événement dans l'application peut avoir un impact sur n'importe quel autre composant. Des travaux sont en cours pour permettre un contrôle plus fin et explicite de cette notion de portée d'événements, à la fois pour permettre une implémentation plus efficace et pour faciliter le raisonnement sur le comportement des règles d'adaptation.

3.3.2. *Événements exogènes*

La détection des événements exogènes par SAFRAN nécessite de réfier le contexte d'exécution de l'application, habituellement implicite. Nous utilisons pour cela WildCAT (David *et al.*, 2005), un système que nous avons conçu pour faciliter la création d'applications sensibles au contexte (CAT = *Context-Awareness Toolkit*) (Dey *et al.*, 2000). WildCAT est utilisé par SAFRAN pour observer le contexte d'exécution et notifier les événements exogènes qui doivent déclencher l'exécution de l'aspect d'adap-

tation. Tout comme FScript, WildCAT a été conçu pour être utilisable en dehors de SAFRAN.

WildCAT modélise le contexte d'exécution par un ensemble de *domaines contextuels*, qui représentent chacun un aspect spécifique du contexte, par exemple les ressources matérielles, le réseau (topologie, performances…), le contexte géo-physique (position géographique, température ambiante…), etc. Chacun de ces domaines contextuels est modélisé sous la forme d'une arborescence de *ressources* nommées, chacune d'entre elle étant à son tour décrite par des *attributs* (de simples paires $(nom, valeur)$). Au niveau de l'implémentation, chaque domaine contextuel est indépendant, et peut utiliser les techniques les plus appropriées à son domaine en terme de performances ; il lui suffit de se conformer à une interface simple qui expose sa structure arborescente à l'extérieur. WildCAT fournit une implémentation par défaut générique et configurable, ainsi que quelques sondes (mémoire, processeurs, réseau…), mais est avant tout un *framework* pour la création de domaines contextuels. La syntaxe utilisée pour désigner les ressources et attributs est inspirée de celle des URI : `domaine://chemin/vers/ressource#attribut` (la partie `#attribut` étant optionnelle). Par exemple, `sys://storage/drives/hdc#removable` indique si le disque `hdc` est amovible.

De la même manière que le contexte d'une application change pendant son exécution, le modèle de ce contexte fournit par WildCAT évolue dynamiquement : les valeurs des attributs peuvent changer, attributs et ressources peuvent apparaître et disparaître à tout moment. Toutes les modifications du modèle WildCAT se traduisent par la génération d'*événements exogènes*. Les différents types d'événements exogènes supportés par SAFRAN sont les suivants :

`changed(expression)` : détecte toute modification de la valeur de l'expression, qui peut référencer n'importe quel attribut ou ressource du contexte, par exemple `changed(geo://location/logical#room)` pour être notifié lorsque l'utilisateur change de pièce (localisation logique).

`realized(condition)` : détecte l'occurrence d'une condition booléenne, comme par exemple `realized(sys://storage/memory#free > 2* sys://storage/swap#used)`. Il s'agit d'un cas particulier de `changed` qui ne détecte que les changements de *faux* à *vrai*. Dans les deux cas, WildCAT recalcule automatiquement les valeurs des expressions à chaque fois qu'une de leurs dépendances est modifiés.

`appears(chemin)` **et** `disappears(chemin)` : détectent l'apparition et la disparition d'une ressource ou d'un attribut dans le contexte. Le chemin utilisé peut utiliser un caractère *joker* « `*` » comme dernier élément. Par exemple `appears( sys://devices/input/*)` détecte l'apparition d'un nouveau périphérique d'entrée, quel qu'il soit.

3.4. *Politiques d'adaptation : l'aspect d'adaptation SAFRAN*

3.4.1. *Structure de l'aspect d'adaptation*

Conformément à la nature réactive du processus d'adaptation, les politiques d'adaptation de SAFRAN sont structurées sous la forme d'ensembles de *règles réactives*[4] de la forme

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

Une règle d'adaptation indique que *lorsque* un événement correspondant à l'expression `<event>` se produit, *si* l'expression `<condition>` est vraie, *alors* la reconfiguration `<action>` est exécutée, adaptant ainsi le système à la nouvelle situation résultant de l'occurrence de l'événement.

Dans le système SAFRAN, les politiques d'adaptation associées dynamiquement aux composants Fractal adaptatifs sont constituées de séquences (ordonnées) de règles d'adaptation :

```
policy example = {
  rule { when <event1> if <cond1> do <action1> }
  rule { when <event2> if <cond2> do <action2> }
  rule { when <event3> if <cond3> do <action3> }
  ...
}
```

Une telle politique englobe en un tout cohérent[5] un ensemble de règles qui implémentent un aspect d'adaptation pour un composant donné. Puisqu'une politique n'adapte qu'un composant individuel (même si plusieurs composants peuvent chacun être affectés par une instance distincte d'une même politique) SAFRAN n'offre pas de mécanisme spécifique pour gérer les aspects d'adaptation qui sont transverses à plusieurs composants. Introduire un tel mécanisme n'est pas nécessaire car en combinant ceux déjà existant il est possible d'implémenter de tels aspects transverses : il suffit pour cela d'utiliser dans un premier temps le support du partage de composants offert pas Fractal pour regrouper dans un « lieu » unique (un composite) l'ensemble – éventuellement dynamique – des composants qui doivent être affectés. Il devient alors très simple d'écrire une action FScript qui attache à tous ces composants (ou en détache) la ou les politiques nécessaires, et ce de façon atomique, grâce aux propritétés existantes de FScript.

---

4. Ce type de règles est similaire à ce que l'on peut trouver dans le domaine des bases de données actives (Collet, 1996) sous la dénomination de règles ECA : *Événement, Condition, Action*.
5. SAFRAN ne fournissant pas actuellement d'outils d'analyse du comportement des règles, la cohérence d'une politique, c'est-à-dire de l'interaction des règles entre elles, est actuellement à la charge de son développeur.

3.4.2. *Tissage et exécution de l'aspect d'adaptation*

SAFRAN introduit une extension au modèle Fractal qui permet d'associer (tisser) dynamiquement des politiques d'adaptation aux composants métiers d'une application. Comme toutes les extensions du modèle Fractal, celle-ci se traduit par la définition d'une nouvelle interface de contrôle, `AdaptationController` (cf. figure 1). C'est ce contrôleur qui implémente le tissage des politiques d'adaptation, et rend ainsi les composants *adaptatifs* : là où un composant Fractal standard permet à un acteur externe de le reconfigurer (et donc de l'adapter), un composant SAFRAN intègre grâce à cette interface le code d'adaptation lui-même et devient ainsi adaptatif, c'est-à-dire autonome et acteur de sa propre adaptation.

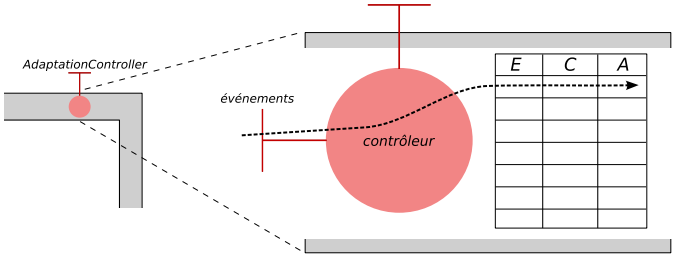

Figure 1. *Intégration des politiques dans le contrôleur d'un composant adaptatif.*

Lorsqu'une politique est attachée à un composant, le contrôleur d'adaptation de ce dernier l'analyse et, en fonction des points de jonction mentionnés dans les règles, instrumente le composant cible pour générer les événements endogène appropriés et s'enregistre auprès de WildCAT afin d'être notifié de l'occurrence des événements exogènes. Après cette phase d'initialisation, lorsque le contrôleur reçoit un événement, qu'il soit endogène ou exogène, il détermine la réaction appropriée en fonction de l'ensemble courant de politiques et de règles du composant cible, puis exécute cette réaction afin d'adapter le composant aux nouvelles circonstances. Ce schéma d'exécution correspond à la nature réactive du processus d'adaptation, en en reprenant les trois phases : observation, décision, action.

4. Exemple

L'application exemple que nous avons choisie pour illustrer l'utilisation de SAFRAN est un petit serveur web nommé Comanche, développé par É. Bruneton dans le cadre d'un tutoriel sur la programmation avec Fractal. Comanche, se voulant extrêmement simple, n'intègre pas de mécanisme pour mettre en cache le contenu des fichiers qu'il lit. Afin d'améliorer ses performances, nous décidons donc d'introduire un nouveau composant adaptatif qui ajoute cette fonctionnalité.

Les performances du cache dépendent essentiellement de la quantité de mémoire qu'il est autorisé à utiliser pour stocker le contenu des fichiers lus précédemment. Si

cette quantité est trop faible, le système ne tirera pas pleinement partie de la présence d'un cache. Si elle est trop importante, les performances risquent d'être encore moins bonnes, le système d'exploitation devant avoir recours à de la mémoire virtuelle sur disque, beaucoup plus lente (« *trashing* »). La quantité de mémoire allouée au composant cache doit donc être déterminée en fonction de la quantité de mémoire disponible sur le système hôte, qui varie au cours du temps. Si par exemple un utilisateur démarre une application gourmande en mémoire alors que le cache utilise une grande partie de la mémoire système, le système d'exploitation devra déplacer une partie de la mémoire utilisée par le cache sur le disque dur pour pouvoir fournir la mémoire nécessaire au nouveau programme.

Notre politique ou aspect d'adaptation va donc être chargé d'adapter dynamiquement la quantité de mémoire allouée au composant cache afin de garantir de bonnes performances en toutes circonstances. L'introduction d'un cache de fichiers dans Comanche se fait très simplement, puisqu'il suffit de modifier l'architecture décrite avec l'ADL Fractal (fichiers `.fractal`), après avoir bien sûr implémenté le composant cache lui-même (cf. figure 2). Lorsqu'il reçoit une requête, le cache renvoie directement le contenu du fichier demandé s'il est en mémoire, et sinon délègue la requête au composant `file-handler`, qui lit le contenu du fichier sur le disque. Le cache ajoute alors ces données à son tampon, puis les renvoie à l'appelant comme résultat de la requête. S'il y a besoin de faire de la place pour stocker le nouveau fichier, le cache utilise une politique de remplacement (LFU) pour évincer un fichier plus ancien.

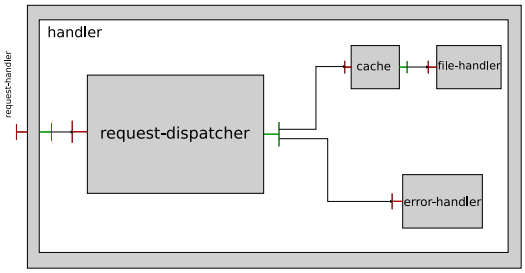

Figure 2. *Introduction d'un cache de fichiers dans Comanche.*

Le composant cache expose deux paramètres de configuration accessibles par son interface `attribute-controller`. Le premier, `currentSize`, est accessible en lecture seule et indique la quantité de mémoire actuellement utilisée par le cache. Le second, accessible en lecture et écriture, se nomme `maximumSize` et indique la quantité de mémoire maximale utilisable. Comme nous l'avons indiqué plus haut, la valeur de ce paramètre doit être assujettie à la quantité de mémoire disponible sur le système pour offrir de bonnes performances en toutes circonstances. Notre politique d'adaptation doit donc réagir aux évolutions de cette quantité de mémoire libre. WildCAT fourni en standard une sonde observant la mémoire, et la quantité de mémoire libre est accessible par l'attribut `sys://storage/memory#free`.

Nous avons maintenant toutes les informations nécessaires pour écrire la politique d'adaptation :

```
action disable-cache(handler) = {
  dispatcher := $handler/child::request-dispatcher;
  handlerItf := $dispatcher/interface::handler;
  if (name($handlerItf/binding::*/component::*) = 'cache') {
    unbind($handlerItf);
    file-handler := $handler/child::file-handler;
    bind($handlerItf, $file-handler/interface::request-handler);
  }
}

action enable-cache(handler) = {
  dispatcher := $handler/child::request-dispatcher;
  handlerItf := $dispatcher/interface::handler;
  if (name($handlerItf/binding::*/component::*) != 'cache') {
    unbind($handlerItf);
    file-handler := $handler/child::file-handler;
    cache := $handler/child::cache;
    bind($handlerItf, $cache/interface::request-handler);
    bind($cache/interface::handler, $file-handler/interface::request-handler);
  }
}

policy adaptive-cache = {
  rule {
    when realized(sys://storage/memory#free < 10*1024)
    do {
      to-free := 10*1024 - sys://storage/memory#free;
      cache := $target/child::cache;
      size := $cache/attribute::currentSize - $to-free;
      if ($size < 500) then {
        set-value($cache/attribute::maximumSize, 0);
        disable-cache($target);
      } else {
        set-value($cache/attribute::maximumSize, $size);
      }
    }
  }

  rule {
    when mem:changed(sys://storage/memory#free)
    if (sys://storage/memory#free >= 10*1024)
    do {
      enable-cache($target);
      cache := $target/child::cache;
      size := 0.8 * ($mem.new-value + $cache/attribute::currentSize);
      max := sys://storage/memory#used - $cache/attribute::currentSize + $size;
```

```
      if ($max < sys://storage/memory@total - 10*1024) {
        set-value($cache/attribute::maximumSize, $size);
      }
    }
  }
}
```

Ce fichier `adaptive-cache.policy` commence par définir deux actions FScript qui seront ensuite utilisées dans le corps de la politique. La première, `disable-cache`, désactive le cache en le déconnectant complètement, et la seconde, `enable-cache`, le ré-introduit dans le flot d'exécution des composants.

La politique d'adaptation elle-même est constituée de deux règles. La première est déclenchée si la quantité totale de mémoire disponible sur le système passe en dessous de 10 Mo. Lorsque cela se produit, l'action de reconfiguration tente de libérer de la mémoire en réduisant la taille maximale du cache, ou en le désactivant complètement en dessous d'une certaine taille minimale (ici 500 Ko). La seconde règle se déclenche lorsque la quantité de mémoire libre varie[6] mais est au dessus de 10 Mo. Dans ce cas, la reconfiguration ajuste la taille allouée au cache à 80% de la quantité de mémoire utilisable, mais uniquement si cela laisse suffisamment de mémoire libre globalement.

L'aspect d'adaptation développé dans cet exemple illustre *(i)* une coupe basée sur deux types d'événements exogènes (`realized` et `changed`) ; *(ii)* deux types d'action de reconfiguration : un paramétrage et la manipulation de connexions entre composants du programme de base. Non seulement la reconfiguration est dynamique mais grâce au tissage dynamique, `adaptive-cache.policy` peut évoluer à l'exécution permettant ainsi le développement de système ouvert.

5. Travaux connexes

Ces dernières années de nombreux travaux ont essayé de rendre les logiciels plus adaptables, en particulier afin de prendre en compte l'émergence de l'informatique nomade et le besoin d'applications autonomes (Kephart, 2002). L'approche la plus prometteuse semble être l'utilisation de modèles de composants dynamiques et extensibles, qui permettent d'intégrer des services non-fonctionnels de façon adaptée aux besoins de l'application (Vadet *et al.*, 2001), et surtout permettent la reconfiguration dynamique de l'application elle-même (McKinley *et al.*, 2004). Certains travaux, comme ACEEL (Cherfour *et al.*, 2003b) ou K-Components (Dowling *et al.*, 2001) sont basés sur des modèles de composants *ad hoc*, qui imposent au programmeur de l'application de base une structure particulière, pas toujours très naturelle. D'autres utilisent des modèles existants mais sont souvent limités à un domaine d'application

6. En pratique, un tel événement n'est pas généré à chaque fois que la mémoire libre change, mais à chaque fois qu'un tel changement est détecté par WildCAT. La granularité temporelle des mesures (et donc les performances) dépendent donc de la configuration de WildCAT.

spécifique : PLASMA (Layaïda *et al.,* 2005) par exemple repose sur Fractal mais se limite au traitement des flux multimédia.

Concernant l'aspect d'adaptation lui-même, (Cilia *et al.,* 2003) ont montré les liens qui existent entre la programmation par aspects et les règles réactives issues des bases de données actives, en particulier dans le cadre de la construction d'applications autonomes. En effet, pour s'adapter à leur environnement ces applications doivent être réactives, et les principes de la programmation par aspect permettent d'introduire cette réactivité de façon non-invasive vis-à-vis du programme de base. Cependant, les auteurs ne présentent que des concepts là où SAFRAN propose une implémentation concrète.

Notons aussi l'existence de FAC (Pessemier *et al.,* 2004) et de Fractal AOP (Fakih *et al.,* 2005), deux extensions du modèle Fractal pour la programmation par aspect, dans lesquelles les aspects sont représentés par des composants Fractal. Bien que SAFRAN soit inspiré par AOP, notre démarche est assez différente de celle de ces extensions. L'objectif de SAFRAN est de permettre la programmation d'applications adaptatives, et la programmation par aspect n'est qu'une technique utilisée pour structurer et présenter le système. La différence entre FAC et Fractal AOP d'une part et SAFRAN d'autre part est essentiellement la même qu'entre une langage généraliste, puissant mais générique et un langage dédié, plus limité mais mieux adapté à une tâche particulière.

6. Conclusion et perspectives

Dans cet article, nous avons montré comment les principes de la programmation par aspects pouvaient être utilisés pour faciliter la construction d'applications adaptatives. Sur le plan conceptuel, nous avons montré que l'adaptation peut être considérée comme une préoccupation transverse et qu'il était possible de transposer les concepts classiques de la programmation par aspects (programme de base, coupes, actions et tisseur) dans ce cas particulier pour modéliser l'aspect d'adaptation (Tableau 1). Sur le plan pratique, nous avons ensuite décrit SAFRAN, une extension du modèle Fractal qui concrétise cette approche et permet le développement modulaire de politiques d'adaptation réactives, qui sont tissées dynamiquement dans les composants Fractal d'une application. SAFRAN étend la notion de points de jonction pour intégrer (grâce à WildCAT) les événements exogènes liés au contexte, et est capable de reconfigurer dynamiquement – et de façon sûre – les applications Fractal grâce au langage dédié FScript.

Dans un futur proche, nous avons l'intention d'étendre les principes de SAFRAN pour permettre l'adaptation d'applications distribuées. Cela implique dans un premier temps d'enrichir FScript pour supporter par exemple la migration de composants ou la manipulation de connexions distribuées. De nouveaux domaines contextuels WildCAT devront être implémentés pour pouvoir partager les informations entre les différents nœuds d'une application. Enfin, l'aspect d'adaptation lui-même pourrait être étendu

AOP	SAFRAN
Programme de base	Composants Fractal reconfigurables
Aspects	Politiques d'adaptation réactives
Points de jonction	Événements endogènes et exogènes
« Advices »	Action de reconfiguration FScript
Tisseur	Contrôleur d'adaptation

Tableau 1. *Correspondance entre les concepts d'AOP et les mécanismes de* SAFRAN.

pour permettre à des composants distants de s'adapter de façon coordonnée (Cherfour *et al.*, 2003a).

7. Bibliographie

Aldrich J., Chambers C., Notkin D., « Architectural Reasoning in ArchJava », *Proceedings of ECOOP'2002*, AITO, Malaga, Spain, June, 2002.

Bruneton r., Coupaye T., Stéfani J.-B., The Fractal Component Model, Technical report, The ObjectWeb Consortium, September, 2003. version 2.0.

Cherfour D., André F., « Auto-adaptation de composants ACEEL coopérants », *CFSE'3, Conférence Française sur les Systèmes d'Exploitation*, La Colle sur Loup, France, October, 2003a.

Cherfour D., André F., « Développement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif ACEEL », *LMO 2003*, Hermès, Vannes, February, 2003b.

Cilia M., Haupt M., Mezini M., Buchmann A., « The Convergence of AOP and Active Databases : Towards Reactive Middleware », *Proceedings of the Intl Conference on Generative Programming and Component Engineering (GPCE'03)*, vol. 2830 of *Lecture Notes in Computer Science*, Springer-Verlag, Erfurt, Germany, p. 169-188, September, 2003.

Collet C., Bases de Données actives : des systèmes relationnels aux systèmes à objets., Mémoire pour l'obtention du diplôme d'Habilitation à diriger des recherches n° RR 965-I-LSR 4, LSR-IMAG, Grenoble, France, October, 1996.

David P.-C., Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation, Phd thesis, Université de Nantes / École des Mines de Nantes, July, 2005.

David P.-C., Ledoux T., « WildCAT : a generic framework for context-aware applications », *Proceeding of MPAC'05, the 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing*, Grenoble, France, November, 2005.

Dey A. K., Abowd G. D., « Towards a Better Understanding of Context and Context-Awareness », *Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, The Hague, The Netherlands, April, 2000. Also GVU Technical Report GIT-GVU-99-22.

Douence R., Fradet P., Südholt M., « A Framework for the Detection and Resolution of Aspect Interactions », *in* , D. S. Batory, , C. Consel, , W. Taha (eds), *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, vol. 2487

of *Lecture Notes in Computer Science*, Springer-Verlag, Pittsburgh, PA, USA, p. 173-188, October, 2002.

Dowling J., Cahill V., « The K-Component Architecture Meta-Model for Self-Adaptive Software », *in* , A. Yonezawa, , S. Matsuoka (eds), *Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan*, vol. 2192 of *LNCS*, AITO, Springer-Verlag, p. 81-88, September, 2001.

Fakih H., Bouraqadi N., « Les aspects et les composants logiciels, étude de cas avec le modèle de composant Fractal », *L'Objet*, 2005.

Filman R. E., Elrad T., Clarke S., Akşit M., *Aspect-Oriented Software Development*, Addison-Wesley, Boston, 2005.

Filman R. E., Friedman D. P., « Aspect-Oriented Programming is Quantification and Obliviousness », *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, October, 2000. Minneapolis.

Kephart J., « A Vision of Autonomic Computing », *in* , R. P. Gabriel (ed.), *Onward ! proceedings from an OOPSLA 2002 track*, ACM, Seattle, WA, USA, p. 13-36, November, 2002.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., « An overview of AspectJ », *in* , J. L. Knudsen (ed.), *ECOOP 2001*, vol. 2072 of *LNCS*, Springer-Verlag, p. 327-353, 2001.

Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C. V., Loingtier J.-M., Irwin J., « Aspect-Oriented Programming », *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, vol. 1241 of *LNCS*, Springer-Verlag, June, 1997.

Layaïda O., Hagimont D., « Composition et Reconfiguration Hiérarchiques pour des Services Multimédia Auto-Adaptables », *4ième Conférence Française sur les Systèmes d'Exploitation (CFSE-4)*, Chapitre Français de l'ACM-SIGOPS, Le Croisic, France, April, 2005.

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

Pawlak R., Seinturier L., Duchien L., Florin G., « JAC : A Flexible and Efficient Solution for Aspect-Oriented Programming in Java », *Reflection 2001*, vol. 2192 of *LNCS*, Springer-Verlag, p. 1-24, September, 2001.

Pessemier N., Seinturier L., Duchien L., Barais O., « Une extension de Fractal pour l'AOP », *Première Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA'04)*, Paris, September, 2004.

Redmond B., Cahill V., « Supporting Unanticipated Dynamic Adaptation of Application Behaviour », *Proceedings of ECOOP 2002*, vol. 2374 of *Lecture Notes in Computer Science*, Springer-Verlag, Malaga, Spain, p. 205-230, May, 2002.

Vadet M., Merle P., « Les conteneurs ouverts dans les plates-formes à composants », *in* , L. Philippe, , L. Seinturier (eds), *Actes des Journées Composants 2001*, Besançon, France, p. 33-46, October, 2001.

van Deursen A., Klint P., Visser J., « Domain-Specific Languages : An Annotated Bibliography. », *ACM SIGPLAN Notices*, vol. 35, n° 6, p. 26-36, June, 2000.

World Wide Web Consortium, « XML Path Language (XPath) Version 1.0 », W3C Recommendation, November, 1999. `http ://www.w3.org/TR/xpath`.