Expression qualitative de politiques d'adaptation pour les composants Fractal

Franck Chauvel[*,**], Olivier Barais[*], Noël Plouzeau[*], Isabelle Borne[**], Jean-Marc Jézéquel[*]

[*] IRISA (INRIA & Université de Rennes 1)
Campus de Beaulieu
F-35042 RENNES (FRANCE)
prénom.nom@irisa.fr

[**] Laboratoire VALORIA (Université de Bretagne Sud)
Centre de Recherche Yves Coppens
Campus de Tohannic
56017 VANNES CEDEX
prénom.nom@univ-ubs.fr

Résumé. Les plates-formes d'exécution récentes telles que Fractal ou OpenCOM offrent de nombreuses facilités pour assurer la prise en compte de propriétés extra-fonctionnelles (introspection, sondes, chargement dynamique, etc). Cependant, l'intégration de politiques d'adaptation reste délicate car elle nécessite de corréler la configuration du système avec l'évolution de son environnement. Le travail présenté dans cet article propose une description qualitative des évolutions de l'environnement et une interprétation possible basée sur de la logique floue. L'article présente également une extension de la plate-forme Fractal implémentant les mécanismes nécessaires à l'exécution de ces politiques d'adaptation de haut niveau. L'approche est illustrée à l'aide d'un serveur HTTP qui modifie sa configuration (architecturale et locale) en fonction de plusieurs paramètres extra-fonctionnels tels que la charge du serveur et la dispersion des requêtes.

1 Introduction

Lors du développement de nouveaux systèmes logiciels, l'aspect fonctionnel n'est plus l'unique critère de satisfaction de l'utilisateur : les problématiques extra-fonctionnelles sont au cœur des efforts de recherche actuels. Il est, par exemple, nécessaire de maintenir un niveau correct pour des propriétés liées à la qualité de service (QoS pour "Quality of Service") telles que la fiabilité, la disponibilité, le temps de réponse, l'ergonomie, *etc*.

Pour maintenir la qualité de service d'un système, deux approches sont possibles. On peut d'une part, vérifier *a priori*, (lors de la conception) que les contraintes sur les propriétés de qualité de service ne sont pas violées à l'exécution. L'utilisation de techniques formelles comme les réseaux de Petri par exemple permet d'obtenir des prévisions sur les consommations de

ressources (mémoire, batterie, CPU, bande passante, etc) [Balsamo et al. (2004)] . On peut, d'autre part, s'assurer *a posteriori* de la qualité de service fournie (à l'exécution) en dotant les systèmes d'une capacité d'auto-adaptation à leur environnement pour assurer au mieux la QoS demandée.

Ainsi, les plates-formes à composants récentes permettant de construire des applications modulaires et reconfigurables telles Fractal [Bruneton et al. (2006)], OpenCOM [Coulson et al. (2004)] ou Spring [Johnson et al. (2005)] offrent les mécanismes nécessaires au support des adaptations (introspection, chargement dynamique). Cependant, elles n'intègrent pas directement la notion de politique d'adaptation. Deux raisons principales expliquent cela. D'une part, les problématiques extra fonctionnelles sont infinies et il faut alors les isoler pour en maîtriser la complexité. D'autre part, pour éviter toute « surspécification » il faut garder une description qualitative qui n'inclut pas de paramètres liés à la plate-forme d'exécution.

La contribution des travaux présentés dans cet article est de proposer un cadre pour la définition de politiques d'adaptation de haut niveau pour les plates-formes à composants. Les mécanismes nécessaires pour le support de ces politiques d'adaptation sont illustrés au travers d'une extension du modèle de composants Fractal et de son implémentation de référence, Julia. Les politiques d'adaptation utilisées sont qualitatives mais leur sémantique, basée sur la logique floue, permet d'inférer des valeurs quantitatives lors de l'exécution pour adapter l'architecture en fonction de son contexte d'exécution.

La suite de cet article est organisée de la façon suivante. La section 2 présente les motivations sur l'exemple d'un serveur HTTP et la section 3 introduit le formalisme utilisé pour modéliser les politiques d'adaptation. L'intégration dans la plate-forme Fractal est présentée dans la section 4 qui présente à la fois l'algorithme d'adaptation et son implémentation sous la forme d'un contrôleur Fractal. Une validation de l'approche, basée sur l'exemple du serveur HTTP, est présentée dans la section 5. Une sélection des travaux connexes est commentée dans la section 6. La section 7 conclut et présente les perspectives ouvertes par ces travaux.

2 Motivations

Pour démontrer la nécessité d'une description des aspects liés à l'adaptation, cette section présente brièvement un serveur HTTP ainsi que les exigences en termes de qualité et d'adaptation qui lui sont associées.

La figure 1 présente l'architecture proposée pour le serveur HTTP *Cherokee*. Il s'agit d'une variation de l'exemple *Comanche*[1] utilisé dans plusieurs communications [David et Ledoux (2006b)] autour de la plate-forme Fractal. Les requêtes HTTP sont lues sur le réseau par le composant *RequestReceiver* qui les transmet au composant *RequestHandler*. Pour traiter une requête, ce dernier peut, soit consulter le cache (composant *CacheHandler*), soit la transmettre au composant *RequestDispatcher* qui interroge alors une ferme de serveurs de fichiers pour résoudre la requête.

Cette architecture est dotée d'un cache (*CacheHandler*) et d'un contrôleur de charge (composant *RequestDispatcher* pour pouvoir maîtriser le temps de réponse et le garder aussi court que possible. Pour pouvoir supporter d'éventuelles montées en charge, les exigences suivantes ont été définies :

[1] Accessible dans le tutorial Fractal, à l'adresse : `http://fractal.objectweb.org/tutorial/index.html`

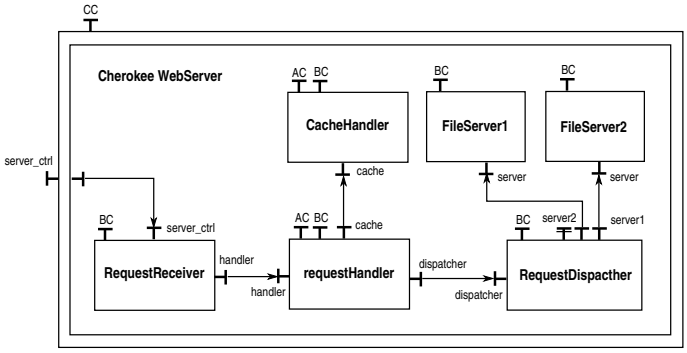

FIG. 1 – *Architecture du serveur HTTP Cherokee présentée à l'aide de la notation Fractal*

1. Le composant *CacheHandler* ne doit être déployé que si le nombre de requêtes HTTP similaires (l'indice de dispersion) est élevé.
2. La quantité mémoire allouée pour le fonctionnement du composant *CacheHandler* doit évoluer en fonction de la charge globale du serveur.
3. La durée de validité des informations du cache doit évoluer également en fonction de la charge globale du serveur (nombre de requêtes par unité de temps).
4. Le nombre de serveurs de données déployés doit être corrélé avec la charge globale du serveur

Plusieurs caractéristiques de ces exigences doivent être soulignées. D'une part, ces exigences peuvent être classifiées en deux catégories : les règles locales, et les règles architecturales. Les règles locales concernent la configuration d'un composant particulier. C'est le cas de l'exigence 2 qui ne concerne que la configuration du cache. Les règles architecturales, par opposition, concernent la configuration de la collaboration entre les composants. Les exigences 1 et 4 en sont de bons exemples, puisque qu'elles nécessitent de modifier l'agencement des composants et des connecteurs (appelés *bindings* dans la terminologie Fractal).

D'autre part, ces exigences sont décrites de manière qualitative, c'est-à-dire à l'aide d'un vocabulaire spécifique à chaque type de propriété. Dans la première exigence par exemple, il est question d'un indice de dispersion « élevé », ce qui est purement qualitatif car aucune valeur précise n'est fournie pour quantifier le terme « élevé ».

3 Modélisation des politiques d'adaptation

Pour prendre en compte les exigences que nous avons présentées dans la section précédente, les adaptations sont décrites à l'aide de politiques qui définissent les règles d'adaptation à appliquer pour un problème donné. Ces règles ne régissent que la configuration d'une collaboration entre plusieurs composants, et leur interprétation est donc à la charge du composant qui englobe la collaboration : un composant composite ou une membrane dans la terminologie Fractal.

Cette section présente le langage utilisé pour décrire les politiques d'adaptation. Celles-ci sont voulues qualitatives et n'incluent donc pas de seuils « durs » exprimés par des valeurs numériques. Elles sont construites sous la forme de règles gardées, dont la garde est exprimée à l'aide du langage FPath qui permet de naviguer dans les architectures Fractal comme XPath permet de naviguer dans les documents XML [David et Ledoux (2006a)]. Le contexte d'exécution de la garde, c'est-à-dire le composant composite englobant la collaboration, est dénoté par *$context*.

Les politiques d'adaptation sont décrites à l'aide de quatre ensemble :

- un ensemble de reconfigurations architecturales
- un ensemble de propriétés et les domaines de valeur qui leur sont associés
- un ensemble de règles pour les configurations locales
- un ensemble de règles pour les reconfigurations architecturales

Reconfigurations architecturales Les différentes reconfigurations architecturales qui sont utilisées dans une politique d'adaptation sont décrites sous la forme d'actions FScript [David et Ledoux (2006a)]. Chaque reconfiguration est nommée et pointe vers un fichier qui contient la description précise de la reconfiguration sous la forme d'actions FScript.

```
policy withCache
is
   reconfiguration addCache is 'addCache.fscript'
   reconfiguration removeCache is 'removeCache.fscript'
```

Propriétés L'environnement du système est capturé par un ensemble de propriétés reliées à un domaine spécifique. Chaque domaine inclut la description du vocabulaire spécifique pour qualifier les propriétés qui lui sont associées. L'exemple suivant présente les domaines et les propriétés utilisés pour décrire les règles d'adaptation du cache. La charge moyenne du serveur est décrite par une propriété « *load* » associée au domaine *AverageLoad*. Ce domaine peut être qualifié en utilisant les termes « *low* », « *medium* » et « *high* ». Pour plus de concision, la syntaxe que nous proposons offre également la possibilité de décrire une propriété et le domaine qui lui correspond d'une seule traite.

```
domain AverageLoad : Real
    evolves in [0, 100] as 'low' 'medium' 'high'

property load : AverageLoad
    sensor is getAverageLoad on handler

property requestDensity : Real
    evolves in [0, 100] as 'low' 'medium' 'high'
    sensor is getAverageRequestDeviation on handler

property size : MemorySize
    evolves in [0, 25] as 'small' 'medium' 'large'
    sensor is getSizeMemory on cache
    actuator is setSizeMemory on cache

property validityDuration : Duration
    evolves in [30000, 120000] as 'short' 'normal' 'long'
    sensor is getDataValidityDuration on cache
    actuator is setDataValidityDuration on cache
```

Pour chaque domaine, la sémantique des termes qui lui sont associés est inférée à partir du nombre de termes et du domaine de définition. Par exemple, la charge du serveur, qui évolue entre 0 et 100 sera « *low* » si elle est inférieure à 50, « *medium* » si elle est comprise entre 25 et 75 et « *high* » si elle est supérieure à 50 (Voir la section 4.1).

De plus, un *sensor* et/ou un *actuator* sont associés à chaque propriété. Ils indiquent respectivement quel service de l'architecture utiliser pour mesurer et/ou modifier cette propriété. Dans une architecture Fractal par exemple, ces deux élements pointent vers des attributs de configuration d'un des composants impliqués (accessibles via le Fractal Attribute-Controller). Certaines propriétés n'ont cependant pas d'actuator car il n'est pas possible pour le système d'agir directement sur leur valeur. La charge du serveur est indépendante du serveur dans notre exemple, et la propriété load n'a donc pas d'actuator associé.

Règles de reconfiguration architecturale Les différentes reconfigurations architecturales possibles au sein d'une architecture sont capturées à l'aide d'actions architecturales. Ces actions peuvent être utilisées dans des règles de reconfiguration architecturale. Chacune de ces règles met en relation le contexte d'exécution du système et l'utilité de déclencher une action architecturale particulière. Dans l'exemple suivant, l'utilité de déployer le composant *CacheHandler* est faible si les requêtes sont toutes différentes (propriété *requestDensity*).

```
# Adding the cache
when requestDensity is 'high' or 'medium'
    if size($context/child::cache) == 0
    then utility of addCache is 'high'

when requestDensity is 'low'
    if size($context/child::cache) == 0
    then utility of addCache is 'low'

# Removing the cache
when requestDensity is 'high'
    if size($context/child::cache) > 0
    then utility of removeCache is 'low'

when requestDensity is 'medium'
    if size($context/child::cache) > 0
    then utility of removeCache is 'medium'

when requestDensity is 'low'
    if size($context/child::cache) > 0
    then utility of removeCache is 'high'
```

Règles de reconfiguration locale Les reconfigurations locales, c'est-à-dire les reconfigurations qui n'impactent que les propriétés locales d'un composant, sont décrites à l'aide de règles qui spécifient l'évolution voulue de la configuration locale. L'exemple suivant présente les règles nécessaires pour corréler la quantité de mémoire allouée pour le composant *Cache* avec la charge moyenne du serveur.

```
# Update the size of the cache
when requestDensity is 'low'
    if size($context/child::cache) > 0
    then size is 'small'

when requestDensity is 'medium'
    if size($context/child::cache) > 0
    then size is 'medium'

when requestDensity is 'high'
    if size($context/child::cache) > 0
    then size is 'large'

# Update the data validity duration
when load is 'low'
```

```
    if size($context/child::cache) > 0
    then validityDuration is 'short'

when load is 'medium'
    if size($context/child::cache) > 0
    then validityDuration is 'normal'

when load is 'high'
    if size($context/child::cache) > 0
    then validityDuration is 'long'

end policy
```

4 Implémentation pour la plate-forme Fractal

4.1 Du qualitatif vers le quantitatif

Pour pouvoir interpréter les politiques d'adaptation telles que nous les avons modélisées dans la section précédente, il faut pouvoir les interpréter avec l'imprécision qu'elles comportent. Pour cela, nous proposons d'utiliser la théorie des ensembles flous et ses applications en théorie du contrôle pour inférer des valeurs réelles à partir de descriptions qualitatives.

En logique floue [Zadeh (1993)], les variables appartiennent à des ensembles flous. La particularité de ces derniers, est que l'appartenance d'une variable à un ensemble n'est pas stricte (comme pour les ensembles classiques) mais graduelle. Un ensemble flou est donc principalement défini par sa fonction d'appartenance (généralement dénommé $\mu(x)$) et une variable peut donc appartenir à un ensemble à 76% par exemple. La figure 2 propose une modélisation du terme « low » utilisé pour décrire la charge du serveur. Dans notre approche, chaque terme défini dans le vocabulaire d'un domaine particulier est associé à un ensemble flou.

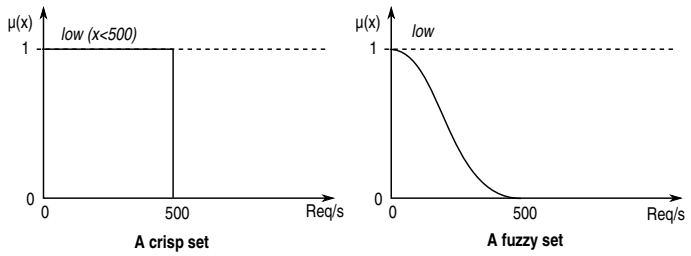

FIG. 2 – *Défintion d'un ensemble flou et de l'ensemble classique équivalent*

En plus des opérateurs de base que sont l'union et l'intersection, la logique floue introduit également des opérateurs unaires (appelés *modifiers* en anglais) qui permettent de construire des expressions plus précises, telles que « *load is high or slightly medium* ». Les principaux opérateurs sont « *not* », « *very* », « *moderately* » et « *slightly* ». Ces opérateurs sont en fait des fonctions qui seront composées avec les fonctions d'appartenance des ensembles. Par exemple, « *very* » est défini par $\mu(x) = x^2$ et a donc pour effet d'affiner les pics d'appartenance.

Cette notion d'ensemble flou a été réutilisée dans le domaine du contrôle flou [Pedrycz (1993)] pour permettre de décrire des règles de contrôle de façon intuitive. Les règles peuvent

être évaluées en trois étapes distinctes, respectivement : la fuzzification, l'inférence floue, et la défuzzification.

Considérons par exemple les deux règles suivantes pour illustrer l'évaluation de règles qualitatives. Ces deux règles sont également utilisées dans la figure 3.

1. load is medium $\Rightarrow$ cacheSize is $medium$
2. load is low $\Rightarrow$ cacheSize is low

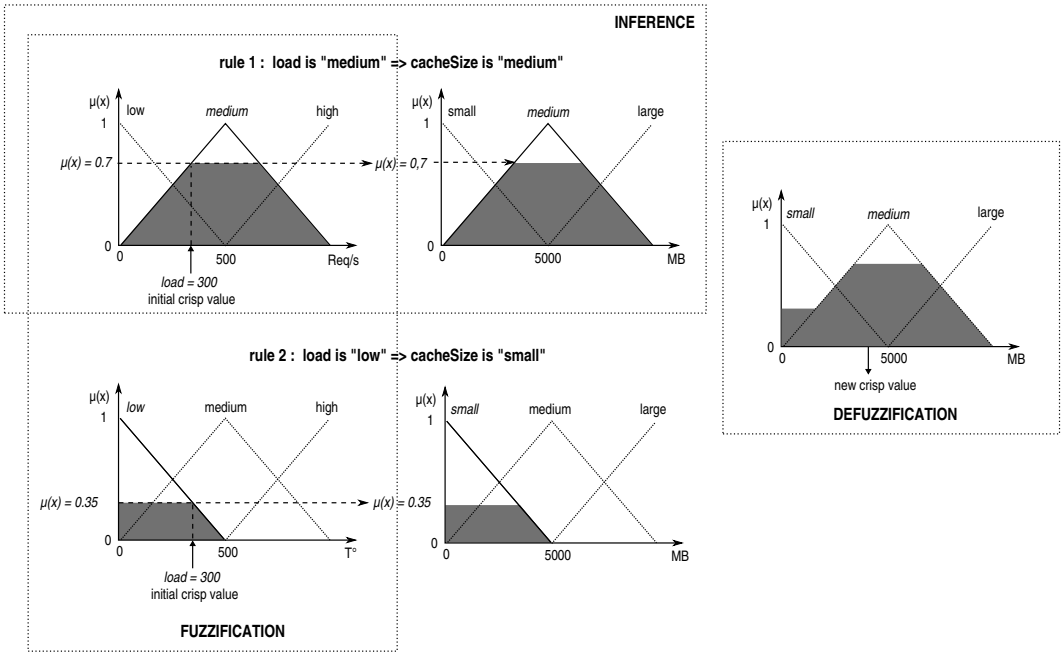

FIG. 3 – *Procédé d'évaluation des règles floues*

La fuzzification Dans le cas des règles floues, il s'agit de mesurer le degré d'appartenance d'une variable à un ensemble flou. On s'intéresse à la partie gauche des règles : pour la règle 1 par exemple, on va calculer le degré d'appartenance de la propriété « *load* » à l'ensemble représenté par le terme « *medium* ». La figure 3 explique cela de façon graphique : la charge réelle est mesurée à 300 req/s et la fonction d'appartenance indique que cette charge est donc » *medium* » à 70%.

L'inférence floue Cette notion consiste à considérer que le degré d'appartenance mesuré dans la partie gauche d'une règle peut être réutilisé tel quel dans sa partie droite. Par exemple, puisque la fuzzification de la règle 1 a établi que la charge est médium à 70%, le principe d'inférence floue indique donc que la taille du cache sera « *medium* » à 70% également.

La défuzzification Une fois que la fuzzification et l'inférence floue ont été appliquées sur toutes les règles, chaque propriété peut prendre plusieurs valeurs floues (ou degré d'appartenance). Pour les deux règles utilisées dans la figure 3, la taille du cache est donc

« *medium* » à 70% (règle 1) et « *low* » à 35% (règle 2). Pour calculer la valeur réelle correspondante (une quantité de mémoire), on agrège les deux aires correspondantes, et on calcule la projection du centre de gravité de l'aire résultante sur l'axe correspondant. On trouve donc 4238MB dans l'exemple de la figure.

Ce processus de calcul implique un comportement global de consensus. En effet, si deux règles sont contradictoires, le système calcule le barycentre entre les deux extrêmes d'une dimension et donc produire une valeur moyenne. L'une des bonnes propriétés des ces règles de contrôle flou est que l'interprétation qui en est faite n'implique pas d'ordre entre les règles. Cependant, du fait de l'effet de consensus, il est alors nécessaire de tester le comportement d'une composition de politiques d'adaptation pour détecter d'éventuelles contradictions.

4.2 L'algorithme d'adaptation

L'algorithme utilisé pour appliquer les politiques d'adaptation est basé sur le processus d'évaluation des règles floues présenté dans la section précédente. Appliqué aux différentes règles d'une politique d'adaptation, ce processus permet de calculer à la fois une nouvelle valeur pour chacune des propriétés impactées par les règles mais également l'utilité de chaque action architecturale.

Le principe de l'algorithme est donc le suivant :

1. L'ensemble des règles de configuration locale est évalué, ce qui produit une nouvelle valeur pour chaque propriété impactée par au moins une règle.
2. L'indice d'utilité de chaque action architecturale est calculé à l'aide du même procédé, mais appliqué aux règles de reconfiguration architecturale (l'utilité d'une règle est considérée comme un propriété à part entière dont le domaine, défini par défaut, est une valeur comprise entre 0 et 100).
3. L'action qui a l'indice d'utilité le plus élevé est déclenchée mais elle n'est exécutée que si l'indice dépasse un seuil d'utilité spécifié par l'utilisateur ; ce qui permet de conserver le système dans une certaine stabilité.

Dans la description suivante, le procédé d'évaluation des règles a été encapsulé dans la méthode « *control* » de la classe « *Controller* ».

```
operation adaptation(dataRules : Set<Rule>, architecturalRules : Set<Rule>) is
do
  var controller : Controller init Controller.new
  var newValues : Table<FuzzyProperty, Value> init Table<FuzzyProperty, Value>

  controller.control(dataRules, newValues)
  newValues.each{t:Entry | t.getKey().set(t.getValue())}

  var newActionUtilities : Table<ActionUtility, Value> init Table<ActionUtility, Value>

  controller.control(architecturalRules, newActionUtilities);

  var selectedAction : ActionUtility init
        newValues.select{ e:Entry | newValues.notexists{t:Entry | st.getValue() > e.
             getValue()} }.getKey()

  if (maxUtility > UsefulnessBound) then
     rule.action.execute()
  end
end
```

4.3 Un contrôleur dédié à l'adaptation

Les politiques d'adaptation que nous avons présentées jusqu'ici sont relatives à une collaboration entre plusieurs composants, encapsulée dans un composant composite. C'est donc ce même composant composite qui est responsable à la fois des connections entre ses sous composants et de leur bonne configuration.

La plate-forme Fractal [Bruneton et al. (2006)] offre un support d'exécution pour les architectures hiérarchiques de composants. Elle offre également un mécanisme d'extension appelé *Controller* : les connexions d'un composant composite sont gérées par exemple par le *Binding-Controller* et son contenu par le *Content-Controller*. L'exécution des politiques d'adaptation est donc assurée par un nouveau type de contrôleur : le *Adaptation-Controller*. La figure 4 présente la conception détaillée des principaux éléments du contrôleur d'adaptation[2]. Les classes grisées sont celles issues du module de logique floue. Ses fonctionnalités sont les suivantes :

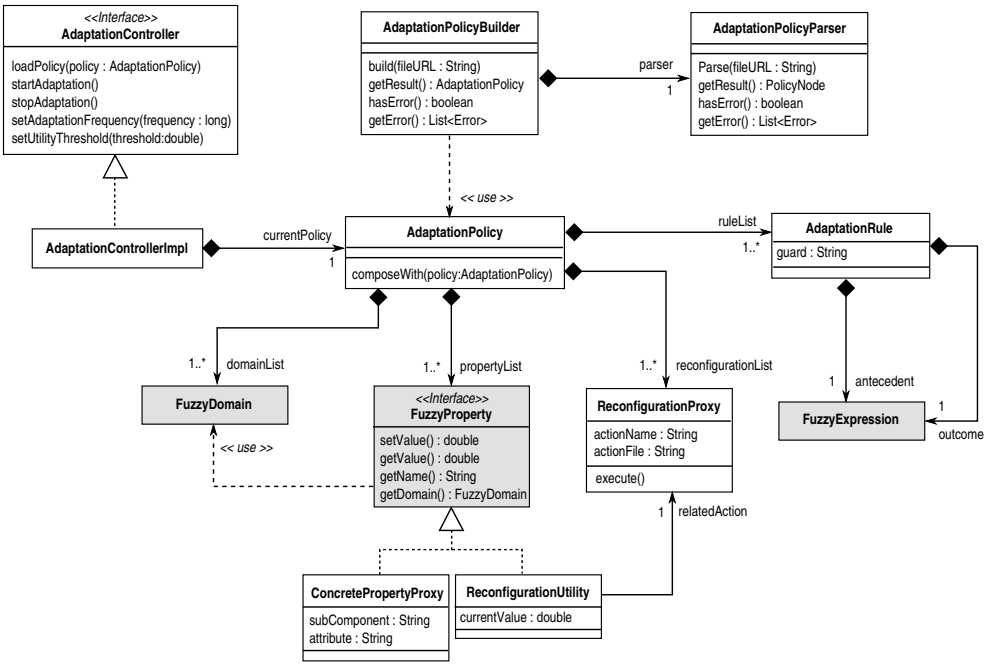

FIG. 4 – *Conception détaillée du contrôleur d'adaptation pour Fractal*

loadAdaptationPolicy permet de charger une nouvelle politique d'adaptation. Le framework offre les outils nécessaires (Parser et Builder) pour instancier et composer des politiques d'adaptation à partir de fichiers textes. A l'aide de la syntaxe présentée dans la section 3, on peut ainsi construire des politiques d'adaptation, les composer, et les utiliser pour adapter un composant.

[2]Les sources sont disponibles à l'adresse `http://www.irisa.fr/triskell/perso_pro/obarais/pmwiki.php?n=Research.Fuzzy`

startAdaptation permet de démarrer le processus de contrôle (évaluation des règles d'adaptation) et ainsi le processus d'adaptation. A intervalle de temps régulier, le contrôleur d'adaptation applique l'algorithme présenté dans la section précédente.

stopAdaptation permet de stopper le processus de contrôle. Le composant continu à s'exécuter, mais n'adapte plus sa configuration aux changements de son environnement.

setAdaptationFrequency permet de modifier la fréquence à laquelle l'algorithme d'adaptation est appliqué. Il s'agit d'un délai exprimé en millisecondes entre deux appels successifs à l'algorithme d'adaptation.

setUtilityThreshold permet de fixer le seuil d'utilité au delà duquel une reconfiguration architecturale est déclenchée. Ce paramètre, utilisé directement dans l'algorithme d'adaptation, influe sur la stabilité du système.

Plusieurs stratégies pour le déclenchement des adaptations sont possibles. Du fait des reconfigurations locales liées à des données, il est utile de recalculer les adaptations à intervalle de temps réguliers. Cependant, d'autres stratégies sont possibles comme de déclencher l'adaptation à la suite d'une violation de contrat par exemple.

Les reconfigurations architecturales qui sont impliquées dans une politique d'adaptation doivent être décrites dans des fichiers séparés par des actions FScript [David et Ledoux (2006a)] qui seront chargées à l'initialisation du contrôleur. FScript permet de décrire des procédures de reconfiguration dans les architectures Fractal. Il permet par exemple d'exprimer des connexions d'interfaces, des ajouts ou des suppressions de composants, etc. L'exemple suivant permet d'ajouter le composant *CacheHandler* dans l'architecture. Il s'agit du contenu du fichier « *addCache.fscript* » utilisé dans la politique d'adaptation qui gère le cache.

```
action addCache(root)
{
        newCache = new("fr.irisa.triskell.cherokee.cacheHandler.CacheHandler");
        set-name($newCache, "cache");
        add($root, $newCache);
        bind($root/child::handler/interface::cache, $newCache/interface::cache);
        start($newCache);
}
```

5 Évaluation et discussions sur l'approche

Cette section présente une première validation de l'algorithme présenté dans les sections précédentes. Cette validation est basée sur une application numérique de l'algorithme car le contrôleur Fractal interprétant les politiques d'adaptation n'est pas encore complètement implémenté. Nous proposons, cependant, d'observer le comportement de l'algorithme au travers d'une simulation, c'est-à-dire dans un environnement où les actions architecturales ne sont pas réellement exécutées. Dans cette simulation, des bouchons de test sont créés pour chaque propriété observée et/ou modifiée par l'algorithme. Nous ne présentons donc que le comportement théorique de l'algorithme : il s'agit donc bien d'une simulation numérique.

Cette simulation présente le comportement de l'algorithme d'adaptation utilisé dans le contexte du serveur HTTP. Deux politiques d'adaptation sont utilisées : l'une assure les exigences liées à la bonne utilisation du cache et a été présentée dans la section 3 alors que la seconde prend en compte les exigences liées au nombre de serveurs de données déployés. Cette seconde politique est la suivante :

```
policy FileServerManagement
is
   reconfiguration addFileServer is "reconfiguration/addFileServer.fscript"
   reconfiguration removeFileServer is "reconfiguration/removeFileServer.fscript"

   property load : AverageLoad
      evolves in [0, 400] as 'low' 'medium' 'high'
      sensor is getAverageLoad on handler

   # Add Data Server
   when load is 'high'
      if size($context/child::dispatcher/interface::*[collection(.)][bound(.)]) <= 10
      then utility of addFileServer is 'high'

   when load is 'medium'
      if size($context/child::dispatcher/interface::*[collection(.)][bound(.)]) <= 10
      then utility of addFileServer is 'low'

   when load is 'low'
      if size($context/child::dispatcher/interface::*[collection(.)][bound(.)]) <= 10
      then utility of addFileServer is 'low'

   # Remove Data Server
   when load is 'low'
      if size($context/child::dispatcher/interface::*[collection(.)][bound(.)]) > 1
      then utility of removeFileServer is 'high'

   when load is 'medium'
      if size($context/child::dispatcher/interface::*[collection(.)][bound(.)]) > 1
      then utility of removeFileServer is 'medium'

   when load is 'high'
      if size($context/child::dispatcher/interface::*[collection(.)][bound(.)]) > 1
      then utility of removeFileServer is 'low'

end policy
```

Les résultats obtenus lors de la simulation sont présentés par la figure 5. Les deux graphiques présentent respectivement l'évolution des propriétés impactées par les deux politiques d'adaptation, à savoir, celles relatives à la gestion du cache et le nombre de serveurs de données déployés. Pour plus de lisibilité et de concision, les valeurs mesurées ont été ramenées à un pourcentage (dans l'intervalle où elles sont définies) en ordonnée. Les étapes de simulation (temps discret) sont présentées en abscisse. Cinq propriétés on été mesurées :

- La charge du serveur, c'est-à-dire, le nombre de requêtes reçues par le serveur (exprimé en nombre de requêtes par seconde).
- La dispersion des requêtes HTTP. Il s'agit d'un indice mesurant (indépendamment de la charge) la dispersion des requêtes.
- La taille du cache, c'est-à-dire, le nombre de pages maximal que le cache peut contenir.
- La durée de validité des pages contenues dans le cache. Il s'agit d'une durée en milli secondes pendant laquelle une page peut rester dans le cache.
- Le nombre de serveurs de données déployés.

La simulation présentée ici décrit le comportement adaptatif de notre serveur HTTP dans un contexte où la charge du serveur (en req/s) augmente jusqu'à une valeur élevée, puis décroît jusqu'à une valeur dite faible. Dans le même temps, la dispersion des requêtes évolue plusieurs fois entre des valeurs « faible » et « élevée ». La simulation montre ainsi les différentes combinaisons possibles entre la charge du serveur et la dispersion des requêtes.

Pour ce qui concerne la gestion des propriétés relatives à la gestion du cache, on peut noter que le composant cache n'est activé que lorsque la charge et la dispersion sont élevées. De plus,

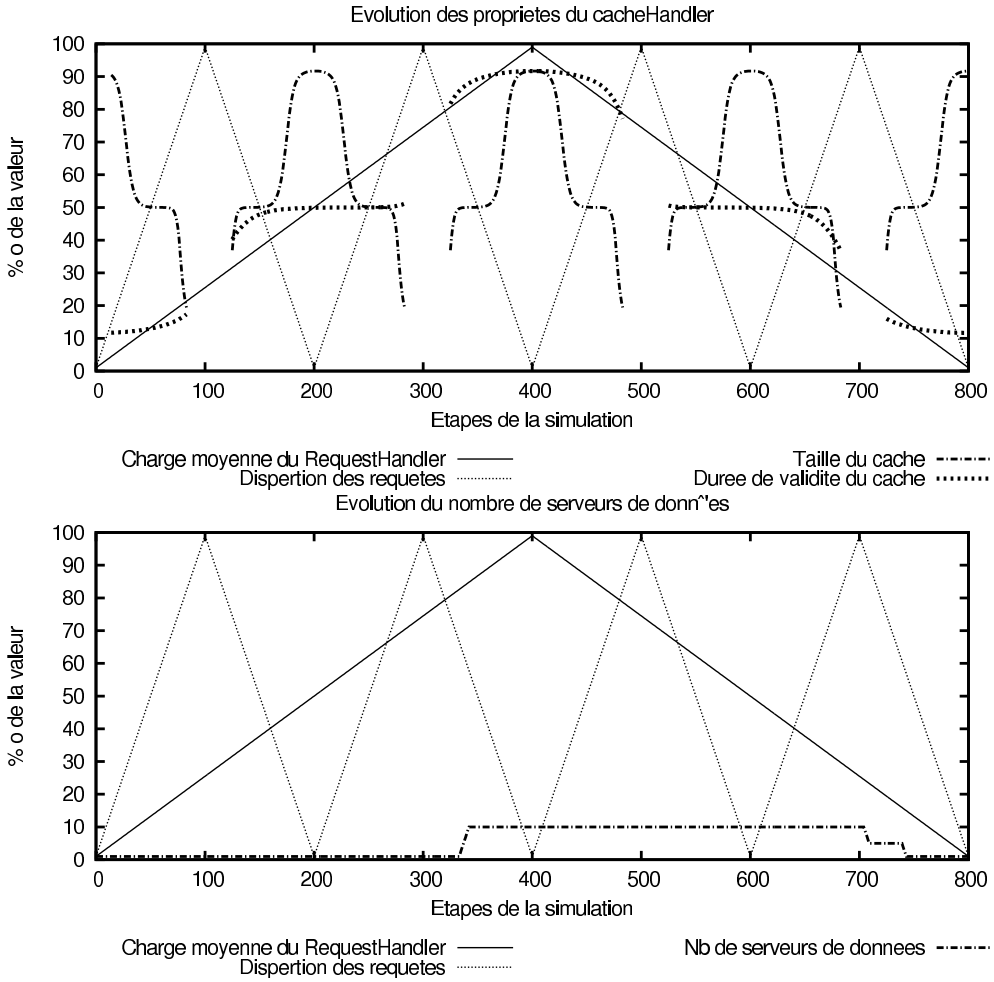

FIG. 5 – *Évolution des configurations locales des composants impliqués dans la réalisation du serveur web*

la taille du cache évolue correctement en fonction de ces deux paramètres. Par exemple, entre les étapes 100 et 300 de la simulation, la taille du cache augmente en fonction de la dispersion des requêtes. Il en est de même pour la durée de validité des informations qui évolue en fonction de la charge du serveur.

Le comportement de la seconde politique d'adaptation montre que l'ajout de serveurs de données est bien corrélé à l'évolution de la charge du serveur.

6 Travaux connexes

De nombreux ADLs (Architectures Description Languages) ont été décrits dans la littérature [Medvidovic et Taylor (2000)]. La plupart de ces travaux ne considère les architectures logicielles que sous un angle statique. Cependant, des travaux récents tels que [Bradbury et al. (2004)] soulignent l'intérêt des architectures dynamiques.

Wright et plus spécialement Dynamic Wright [Allen et al. (1998)] est un autre ADL qui prend en charge les reconfigurations dynamiques. Dans ces travaux, l'objectif est de faire des vérifications sur les reconfigurations architecturales. Dans Wright, le comportement des composants est décrit à l'aide de processus communiquants, ce qui permet de faire des vérifications de cohérence par exemple. Cependant, Wright ne prend pas en compte les reconfigurations locales comme le permet notre approche

AADL [SAE (2004)] est l'un des premiers langages de description d'architectures à avoir pris en compte la qualité de service. AADL permet de décrire différents *modes* d'une architecture, c'est-à-dire les différentes configurations architecturales dans lesquelles un système peut évoluer. Cependant, AADL ne permet de décrire la dynamique qui régit les changements de modes.

Dans un cadre plus formel, l'outil π-ADL [Oquendo (2004); Verjus et al. (2006)] permet de décrire les architectures logicielles à l'aide d'une famille de langages formels. π-ADL permet de faire des vérifications de validité et de conformance sur les architectures, mais n'offre pas une syntaxe de haut niveau proche des exigences, comme le permet notre approche basée sur la logique floue.

Parmi les autres outils existants, Rainbow [Garlan et al. (2004)] présente un outil qui intégre une gestion des politiques d'adaptation. Il permet au concepteur d'application de concevoir des stratégies d'adaptation qui sont déclenchées par la violation de contraintes. Si l'objectif de Rainbow est similaire au nôtre, notre approche se caractérise par un niveau d'abstraction élevé dû à l'utilisation de la logique floue qui permet de conserver une description qualitative et non quantitative du contexte du système.

Dubus et Merle (2006) proposent d'enrichir l'ADL de Fractal avec des règles de reconfiguration basées également sur la notion d'évenement-condition-action où les actions modifient l'architecture et où les évènements sont issus de l'environnement. Notre approche diffère par l'utilisation de la logique floue qui permet d'exploiter une description qualitative de l'environnement. Dans la même veine, Grine et al. (2005) proposent une approche similaire à la nôtre dans la mesure où les adaptations sont représentées sous la forme de règle "evénement-condition-action". Les travaux en question ici sont implémentés également sur la plate-forme Fractal, mais n'offrent pas une description qualitative de l'environnement.

Les travaux présentés dans [Conan et al. (2007)] proposent une architecture à base de composants pour l'aquisition de données provenant de l'environnoment d'exécution. Cette ap-

proche pourrait être utilisée ici pour gérer de manière efficace l'accès aux mesures faites sur l'environnement.

7 Conclusion

Dans un environnement hautement dynamique, les systèmes ont besoin de s'adapter aux évolutions de ce dernier pour pouvoir assurer un niveau de qualité maximum. Pour cela, la plupart des plates-formes d'exécution récentes offrent les mécanismes nécessaires à l'exécution d'adaptations dynamiques tels que la réflexivité ou le chargement dynamique. Cependant, ces mêmes plate-formes n'intègrent pas encore une description abstraite de politiques d'adaptation.

La contribution de cet article est de proposer un mécanisme d'exécution de politiques d'adaptation pour la plate-forme Fractal. Les politiques d'adaptation sont interprétées à l'aide d'un moteur qui prend en charge l'interprétation de règles floues. Ces règles floues, par leur imprécision, restent très proches des exigences que pourraient exprimer un concepteur de systèmes adaptatifs.

La solution que nous proposons décrit les politiques d'adaptation sous la forme d'un ensemble de règles qualitatives qui peuvent soit impacter la configuration architecturale (en terme de composants et de connecteurs) soit impacter la configuration locale d'un composant, en modifiant la valeur d'un des paramètres de sa configuration. Le moteur d'adaptation est implémenté sous la forme d'un contrôleur Fractal et peut donc être réutilisé facilement dans d'autres contexte sur la plate-forme Fratal.

Plusieurs extensions peuvent être envisagées à cette approche. La solution proposée permet une conception précoce des politiques d'adaptation. De plus, la logique floue laisse la possibilité d'influer sur plusieurs paramètres liés à l'interprétation des politiques d'adaptation, comme la définition des fonctions d'appartenance associées au vocabulaire des domaines ciblés. Nous envisageons l'utilisation de techniques issues du monde de l'intelligence artificielle pour optimiser ces différents paramètres dans la cadre de simulations de l'évolution de l'architecture et de son environnement. L'utilisation de réseaux de neurones, ou d'un algorithme génétique sont de bons moyens pour mettre au point un ensemble de règles d'adaptation particulièrement efficace dans une situation donnée.

Références

Allen, R., R. Douence, et D. Garlan (1998). Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science 1382*, 21–36.

Balsamo, S., A. D. Marco, P. Inverardi, et M. Simeoni (2004). Model-based performance prediction in software development : A survey. *IEEE Trans. Softw. Eng. 30*(5), 295–310.

Bradbury, J. S., J. R. Cordy, J. Dingel, et M. Wermelinger (2004). A survey of self-management in dynamic software architecture specifications. In *WOSS '04 : Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, New York, NY, USA, pp. 28–33. ACM Press.

Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma, et J.-B. Stefani (2006). The Fractal Component Model and its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems 36*(11–12), 1257–1284.

Conan, D., R. Rouvoy, et L. Seinturier (2007). Scalable processing of context information with cosmos. In *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'07)*, Volume 4531 of *Lecture Notes in Computer Science*, pp. 210–224. Springer.

Coulson, G., G. Blair, P. Grace, A. Joolia, K. Lee, et J. Ueyama (2004). A component model for building systems software.

David, P. et T. Ledoux (2006a). Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France.

David, P.-C. et T. Ledoux (2006b). An aspect-oriented approach for developing self-adaptive fractal components. In W. Löwe et M. Südholt (Eds.), *Software Composition, 5th International Symposium, SC 2006, Vienna, Austria, March 25-26, 2006, Revised Papers*, Volume 4089 of *Lecture Notes in Computer Science*, pp. 82–97. Springer.

Dubus, J. et P. Merle (2006). Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués. In M. C. Oussalah, F. Oquendo, D. Tamzalit, et T. Khammaci (Eds.), *1er Conférence francophone sur les Architectures Logicielles (CAL 2006), 4-6 September 2006, Nantes, France*, pp. 13–29. Hermes Science.

Garlan, D., S.-W. Cheng, A.-C. Huang, B. Schmerl, et P. Steenkiste (2004). Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer 37*(10), 46–54.

Grine, H., T. Delot, et S. Lecomte (2005). Adaptive query processing in mobile environment. In *MPAC '05 : Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, New York, NY, USA, pp. 1–8. ACM Press.

Johnson, R., J. Hoeller, A. Arendsen, T. Risberg, et D. Kopylenko (2005). *Professional Java Development with the Spring Framework*. Birmingham, UK, UK : Wrox Press Ltd.

Medvidovic, N. et R. N. Taylor (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng. 26*(1), 70–93.

Oquendo, F. (2004). Formally Describing Dynamic Software Architectures with π-ADL. *World Scientific and Engineering Transactions on Systems 3*(8), 673–679.

Pedrycz, W. (1993). *Fuzzy Control and Fuzzy Systems* (Second ed.). New York, USA : Wiley.

SAE, A.-. E. C. S. C. (2004). Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506.

Verjus, H., S. Cîmpan, I. Alloui, et F. Oquendo (2006). Gestion des architectures évolutives dans archware. In M. C. Oussalah, F. Oquendo, D. Tamzalit, et T. Khammaci (Eds.), *1er Conférence francophone sur les Architectures Logicielles (CAL 2006), 4-6 September 2006, Nantes, France*, pp. 41–57. Hermes Science.

Zadeh, L. A. (1993). Fuzzy sets. In D. Dubois, H. Prade, et R. R. Yager (Eds.), *Readings in Fuzzy Sets for Intelligent Systems*, pp. 27–64. San Mateo, CA : Kaufmann.

Adaptations qualitatives pour Fractal

Summary

Most of runtime platforms such as Fractal or OpenCOM provide various facilities to manage extra-functional properties (reflexivity, ressources sensors, dynamic loading, etc). However, besides these facilities it is still not possible to describe and interpret directly some high level adaptation policies which correlate the interal configuration of the system with the environment's state. Teh contribution of this paper is provides a extension of the Fractal platform which enable the direct execution of high-level adaptation policies. Our approach is illustrated thanks to a web server where two adaptation policies are deployed: one for handling the use of a cache, and other for handling the number of deployed data servers.