Composition et expression qualitative de politiques d'adaptation pour les composants Fractal

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

Composition et expression qualitative de politiques d'adaptation pour les composants Fractal

Franck Chauvel[3], Olivier Barais[1], Noël Plouzeau[1], Isabelle Borne[2], and Jean-Marc Jézéquel[1]

[1] IRISA (INRIA & Université de Rennes 1)
Campus de Beaulieu
F-35042 RENNES (FRANCE)
prenom.nom@irisa.fr
[2] Laboratoire VALORIA (Université de Bretagne Sud)
Centre de Recherche Yves Coppens
Campus de Tohannic
56017 VANNES CEDEX
prenom.nom@univ-ubs.fr
[3] National Laboratory of High Confidence Software Technologies
School of Electronics Engineering and Computer Science,
Peking University, Beijing, 100871, China
franck.chauvel@sei.pku.edu.cn

Abstract. 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 étend la plate-forme Fractal avec les mécanismes nécessaires à l'exécution de politiques d'adaptation de haut niveau. L'approche est illustrée à l'aide d'un serveur HTTP qui doit modifier sa configuration (architecturale et locale) en fonction de plusieurs paramètres extra-fonctionels.

1 Introduction

Lors du développement de nouveaux systèmes logiciels, l'aspect fonctionnel est de moins en moins 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 d'être capable 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 à 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 réseaux de Petri permet par exemple d'obtenir des prévisions sur les consommations de ressources (mémoire, batterie, CPU, bande passante, etc). On peut d'autre part, s'assurer, à 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 composant récentes permettant de construire des applications modulaires et reconfigurables telles Fractal ([3]), OpenCOM ([4]) ou Spring ([10]) 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 le nombre de propriétés extra-fonctionnelles est potentiellement infini. Il est alors impératif de pouvoir segmenter cet espace pour pouvoir en maîtriser la complexité. D'autre part, il nécessaire de conserver un niveau d'abstraction suffisant pour garder des spécifications exploitables.

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'adaptations sont illustrées au travers d'une extension du modèle de composants Fractal et de son implémentation de référence Julia. Les politiques d'adaptations 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 sémantique du moteur d'inférence présentée dans ce papier permet de préserver la capacité de composer ces politiques indépendanmment de l'ordre dans lequel elles ont été définies.

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'adaptations. La section 4 démontre la sémantique associée aux politiques d'adaptation permettant leur composition. L'intégration dans la plate-forme Fractal est présentée dans la section 5 qui présente à la fois l'algorithme d'adaptation et son implémentation sous forme de contrôleur Fractal. Une validation de l'approche, basée sur l'exemple du serveur HTTP, est présentée dans la section 6. Une sélection des travaux connexes est commentée dans la section 7. La section 8 conclue et présente les perspectives ouvertes par ces travaux.

2 Motivations

Pour démontrer la nécessité des politiques d'adaptation, cette section présente brièvement un serveur HTTP et 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*[4] utilisé dans plusieurs communications ([6]) 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 un ferme de serveurs de fichiers pour résoudre la requête.

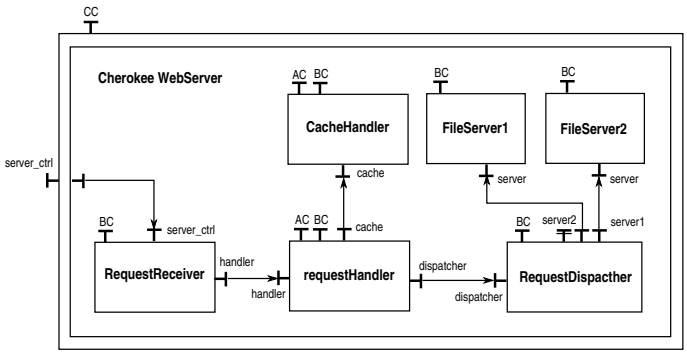

Fig. 1. Architecture du serveur HTTP Cherokee présentée à l'aide de la notation Fractal

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. 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.

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

3. La durée de validité des informations du cache doit évoluée 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 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 présentés dans la section précédente, les politiques d'adaptations sont décrites à l'aide de quatre élements:

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

Reconfiguration architecurales Les différentes reconfigurations architecturales qui sont utilisées dans une politique d'adaptation sont décrites sous la forme d'actions FScript. Chaque reconfiguration est nommée et pointe vers un fichier qui contient la description précise de la reconfiguration sous 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, chacune relié à un domaine spécifique. Chaque domaine inclut la description du vocabulaire spécifique à utiliser pour qualifier les propriétés qui lui sont associées. L'exemple suivant présente les domaines et les propriétés utilisés pour décrire l'adaptation du cache. La charge moyenne du serveur est décrite par une propriété *load* associée au domaine *AverageLoad* et peut donc être qualifiée 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 : Real
   sensor is getLoad on requestHandler

property requestDeviation : Real
   evolves in [0..100] as low medium large
   sensor is getRequestDeviation on requestHandler

property size : MemorySize
   evolves in [0..2000] as small medium large
   sensor is getMemoryUsed on cache
   actuator is setMemoryUsed on cache

property validityDuration : Duration
   evolves in [0..100] as short normal long
   sensor is getValidityDuration on cache
   actuator is setValidityDuration 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 inférieur à 50, 'medium' si elle est comprise entre 25 et 75 et 'high' si elle est supérieure à 50. (Voir la section 5.1.)
De plus, à chaque propriété sont associés un *sensor* et un *actuator* qui indiquent respectivement comment mesurer et/ou modifier cette propriété dans l'architecture. Pour une architecture Fractal, ces deux élements pointe vers des attributs de configuration d'un des composants impliqués (Fractal Attribute-Controller).

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. Dans l'exemple suivant, l'utilité de déployer le composant *Cache* est faible si les requêtes sont toutes différentes (propriété *requestDeviation*).

```
when requestDeviation is 'low'
if count($context::child/*::interface/CacheHandler[server(.)]/component) = 0
   then utility of addCache is very 'high'

when load is 'high'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) = 0
   then utility of addCache is 'medium' or 'high'

when load is 'low' and requestDeviation is 'high'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) > 0
   then utility of removeCache is 'high'
```

Règles de configuration 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 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 composants *Cache* avec la charge moyenne du serveur.

```
when proxy.load is 'low'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) > 0
   then cache.size is 'small'

when proxy.load is 'medium'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) > 0
   then cache.size is 'medium'

when proxy.load is 'high'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) > 0
   then cache.size is 'large'

end policy
```

Toutes les règles, qu'elles soient locales ou architecturales sont gardées. La contrainte associée est exprimée à l'aide du langage FPath qui permet de naviguer les architectures Fractal comme XPath permet de naviguer les documents XML ([5]). Le contexte d'exécution de la garde, c'est-à-dire le composant composite englobant la collaboration, est dénoté par *$context*.

4 Composition de politiques d'adaptation

Comme nous l'avons expliqué précédemment, la description d'une politique d'adaptation générale à un système est délicate à cause de la complexité de l'environnement et des différentes variations possibles de l'architecture. Pour pouvoir briser cette complexité, il faut pouvoir traiter les problèmes séparément, à l'aide de politiques d'adaptations distinctes. Toutefois, il faut alors être capable de composer facilement les politiques d'adaptation. Si l'on considère la composition comme l'union des règles et des propriétés qui constituent les politiques d'adaptation, l'ordre d'unification, dans les règles notamment, ne doit pas avoir d'impact sur l'interprétation de la politique résultante. La suite de cette section montre cela.

Soit Σ une structure telle que $\Sigma = \langle P, V, T, K \rangle$ où l'on considère les éléments suivants :

– P est un ensemble de propriétés
– V est une fonction partielle qui associe une valeur à chaque propriété telle que :

$$V = \{v : P \nrightarrow R\}$$

– T est l'ensemble des topologies possibles
– K est l'ensemble des intro-actions architecturales qui conservent la valeur des propriétés associées à chaque composant. Plus formellement on obtient :

$$\begin{aligned} K = \{k : P, V, T, K \to P, V, T, K | \\ \forall\, (t, v) \in T \times V, \forall\, p \in dom(v), \\ \text{let } (t', v') = k(t, v) \\ \text{in } v(p) = v'(p) \end{aligned}$$

Pour pouvoir définir plus formellement l'algorithme, considérons les éléments suivants :

– A comme l'ensemble des politiques d'adaptation où chaque politique d'adaptation est un ensemble de règles d'adaptation tel que :

$$A = \{\, \langle R_P, R_A \rangle \,\}$$

où R_P est l'ensemble des règles configuratives R_A est l'ensemble des règles architecturales. On peut donc définir l'opérateur de composition entre deux politiques d'adaptation comme une union des ensembles de propriétés (en admettant un renommage) et une union des ensembles de règles. Formellement, on obtient :

$$\begin{aligned} \forall\, (f, g) \in A^2 \\ f \otimes g = \langle R_{P_F} \cup R_{P_G}, R_{A_F} \cup R_{A_G} \rangle \end{aligned}$$

– U est une fonction qui associe une utilité aux intro-actions architecturales

$$U = \{\, u : K \to \mathbb{R} \,\}$$

– *control* est une fonction qui représente l'algorithme de contrôle flou. Elle calcule une nouvelle valeur pour chaque propriété de l'architecture qu'elle a reçue en paramètre et calcule également l'utilité de chaque intro-action architecturale.

$$control : P, V, T, K, A \to V, U$$

– *select* est une fonction qui sélectionne l'intro-action architecturale la plus utile. Elle retourne l'intro-action la plus utile s'il existe un maximum unique parmi les utilités associées aux intro-actions. Si plusieurs intro-actions ont la même utilité, alors elle utilise une fonction σ (commutative) pour sélectionner l'une des intro-actions ayant une valeur d'utilité maximale.

$$\begin{aligned} select \;:\; & U \to K \\ select(u) = & \begin{cases} k \in \max\limits_{k_i \in dom(u)} \big(u(k_i)\big) \;\Big|\; \max\limits_{k_i \in dom(u)} \big(u(k_i)\big) \Big| = 1 \\ \sigma\Big(\max\limits_{k_i \in dom(u)} \big(u(k_i)\big)\Big) \;\Big|\; \max\limits_{k_i \in dom(u)} \big(u(k_i)\big) \Big| > 1 \end{cases} \end{aligned}$$

où σ est une fonction commutative qui sélectionne un élément parmi n. Elle rend alors la fonction *select* commutative également.

A l'aide de ces éléments notre algorithme peut se modéliser comme suit :

$$\begin{aligned} apply &: P, V, T, K, A \rightarrow P, V, T, K \\ apply(p, v, t, k, a) &= \text{let } (v', u) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(u) \\ &\qquad\quad \text{in } k_0(p, v', t, k) \end{aligned}$$

On peut alors démontrer que l'ordre parmi les règles n'a pas d'importance et que l'opérateur de composition est bien commutatif. Formellement, cela se traduit par l'équation suivante :

$$\begin{aligned} &\forall (p, v, t, k) \in (P \times V \times T \times K), \forall\, (f, g) \in A^2 \\ &\quad apply(p, v, t, k, \langle f, g \rangle) = apply(p, v, t, k, \langle g, f \rangle) \end{aligned}$$

Considérons les trois cas suivants:

– Si $(f, g) \in (R_A)^2$ alors,

$$\begin{aligned} apply(p, v, t, k, \langle f, g \rangle) &= \text{let } (v', \{u_f, u_g\}) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(\{u_f, u_g\}) \\ &\qquad\quad \text{in } k_0(p, v', t, k) \end{aligned}$$

$$\begin{aligned} apply(p, v, t, k, \langle g, f \rangle) &= \text{let } (v', \{u_g, u_f\}) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(\{u_g, u_f\}) \\ &\qquad\quad \text{in } k_0(p, v', t, k) \end{aligned}$$

Dans ces deux cas précis, k_0 prend la même valeur puisque l'opération *select* est commutative.

– Si $(f, g) \in (R_P \times R_A)$ alors,

$$\begin{aligned} apply(p, v, t, k, \langle f, g \rangle) &= \text{let } (v_f, u_g) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(u_g) \\ &\qquad\quad \text{in } k_0(p, v_f, t, k) \end{aligned}$$

$$\begin{aligned} apply(p, v, t, k, \langle g, f \rangle) &= \text{let } (v_f, u_g) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\qquad \text{let } k_0 \text{ be } select(u_g) \\ &\qquad\quad \text{in } k_0(p, v_f, t, k) \end{aligned}$$

Dans ces deux cas, l'opération *select* est utilisée sur des ensembles à un seul élément et k_0 prend donc la valeur de cet unique élément.

– Si $(f, g) \in (R_P)^2$ alors,

$$apply(p, v, t, k, \langle f, g \rangle) = \text{let } (\{v_f, v_g\}, \varnothing) \text{ be } control(p, v, t, k, a)$$
$$\text{in}$$
$$\text{let } k_0 \text{ be } select(\varnothing)$$
$$\text{in } k_0(p, v', t, k)$$

$$apply(p, v, t, k, \langle g, f \rangle) = \text{let } (\{v_f, v_g\}, \varnothing) \text{ be } control(p, v, t, k, a)$$
$$\text{in}$$
$$\text{let } k_0 \text{ be } select(\varnothing)$$
$$\text{in } k_0(p, v', t, k)$$

Nous considérons ici que f et g ne portent pas sur les mêmes propriétés. Dans ce cas particulier, la fonction *control* produirait une seule valeur pour cette propriété (une tendance moyenne) et l'ordre n'a donc pas non plus d'influence ici.

5 Implémentation pour la plate-forme Fractal

5.1 Du qualitatif vers le quantitatif

Pour pouvoir interpréter les politiques d'adaptations 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 ([17]), 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 définit dans le vocabulaire d'un domaine particulier est associé à un ensemble flou.

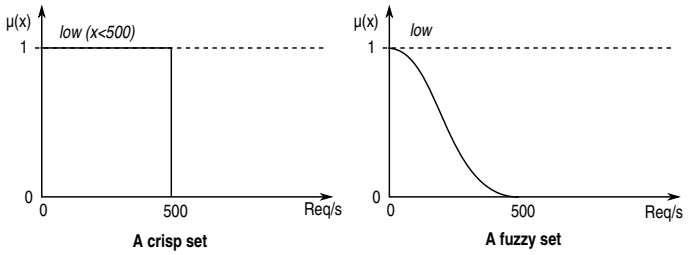

Fig. 2. Défintion d'un ensemble flou et de l'ensemble classique équivalent

En plus des opérateurs de base que sont l'union et l'intersection, la logique floue introduit également des opérateurs unaires (appelés *modificateurs*) 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*.

Cette notion d'ensemble flou a été réutilisée dans le domaine du contrôle flou ([14]) pour permettre de décrire des règles de contrôle de la façon la plus intuitive possible. 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 ⇒ cacheSize is *medium*
2. load is low ⇒ cacheSize is *low*

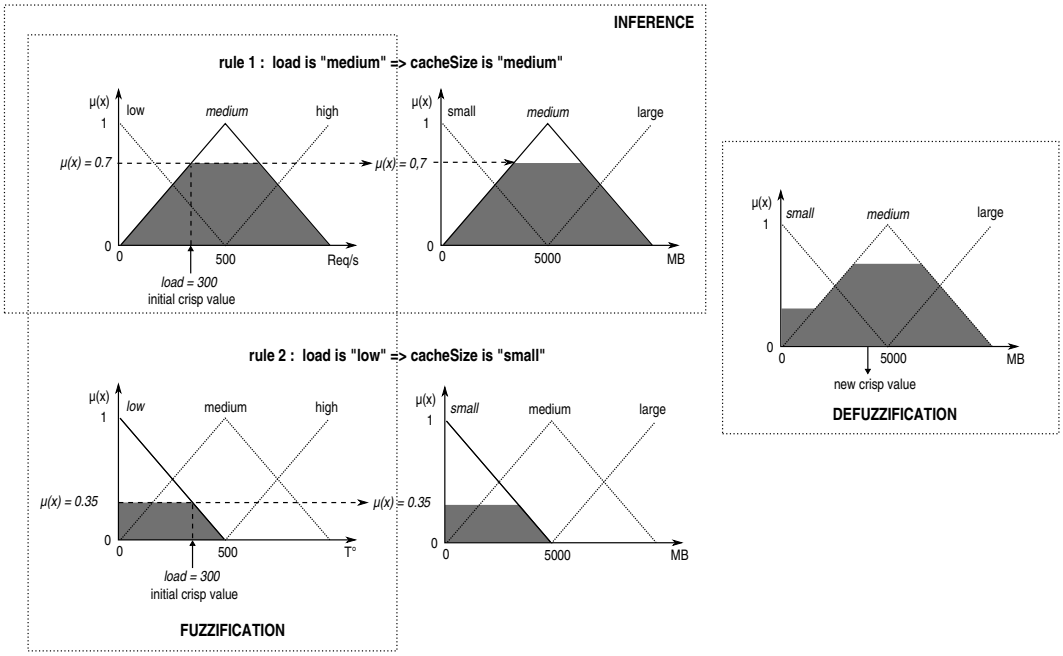

Fig. 3. Procédé d'évaluation des règles floues

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 permet donc de déduire 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.

5.2 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. Appliquer 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 locales 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, définit 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 cependant 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
```

5.3 Un contrôleur dédié à l'adaptation

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

La plate-forme Fractal ([3]) offre un support d'exécution pour les architectures hiérarchiques de composants. Elle offre également un mécanisme d'extension appelé *contrôleur* : les connexions au sein 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 constitutifs du contrôleur d'adaptation[5]. Les classes grisées sont celles issues du module de logique floue. Ses fonctionnalités sont les suivantes:

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 constuire des politiques d'adaptation, les composer, et les utiliser pour adapter un composant.

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

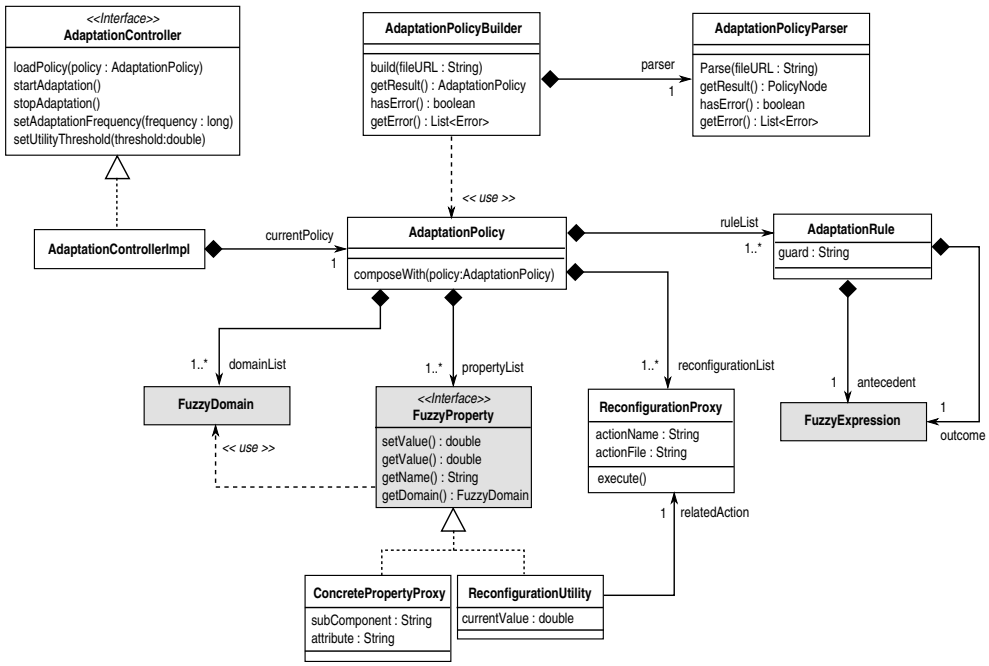

Fig. 4. Conception détaillée du contrôleur d'adaptation pour Fractal

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'adapate 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 milliseconde entre deux appels succesifs à l'algorithme d'adaptation.

setUtilityThreshold permet de fixé 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.

Les reconfigurations architecturales qui sont impliquées dans une politique d'adaptation doivent être décrites dans des fichiers séparés à l'aide d'actions FScript ([5]). 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 cache dans l'architecture. Il s'agit du contenu du fichier *"addCache.fscript"* utilisé dans la politique d'adaptation qui gère le cache.

```
action addCache(server) =
{
  cacheComponent = new("fr.irisa.triskell.cherokee.CacheHandler.CacheHandler");
  requestHandler = $server/child::requestHandler;
  add(server, cache);
  bind($requestHandler/interface::cache, $cache/interface::cache);
  start($cache);
}
```

6 Evaluation 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 l'utilisation du

contrôleur d'adaptation, dans une application réelle n'est pas encore complétement implémentée. Nous proposons, cependant, d'observer le comportement de l'algorithme au travers d'une simulation, c'est-à-dire dans un environement où les actions architecturales ne sont pas réellement exécutées.

Cette simulation présente le comportement de l'algorithme d'adaptation utilisé dans le contexte du serveur web. 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 DataServerManagement
is
   reconfiguration addFileServer is ''addFileServer.fscript''
   reconfiguration removeFileServer is ''removeFileServer.fscript''

   property load : AverageLoad
      evolves in [0..100] as low medium high
      sensor is getLoad on requestHandler

   when load is high and requestDeviation is high
      if count($context::child/*::interface/FileServer[server(.)]/component) < 10
      then utility of addFileServer is high

   when load is low and requestDeviation is low
      if count($context::child/*::interface/FileServer[server(.)]/component) > 1
      then utility of removeFileServer  is medium or high

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, celle relative à la gestion du cache et celle relative à la gestion du nombre de serveurs de données déployés.

La simulation présentée ici décrit le comportement adaptatif de notre serveur web dans un contexte où la charge du serveur (en req/s) augmente jusqu'à une valeur élevée, puis décroit jusqu'à une valeur dite faible. Dans le même temps la dispertion des requêtes évolue plusieurs fois entre des valeurs "faible" et "elévée". La simulation montre ainsi les différentes combinaisons possibles entre la charge du serveur et la dispertion 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 est élevée et que la dispertion des requêtes l'est également. De plus, 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 dispertion 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.

7 Travaux connexes

De nombreux ADLs (Architectures Description Languages) ont été décrits dans la littérature ([11]). La plupart de ces travaux ne considère les architectures logicielles que sous un angle statique. Cependant, des travaux récents tels que ([2]) soulignent l'intérêt des architectures dynamiques.

Wright et plus spécialement Dynamic Wright [1] 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 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 ([15]) 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.

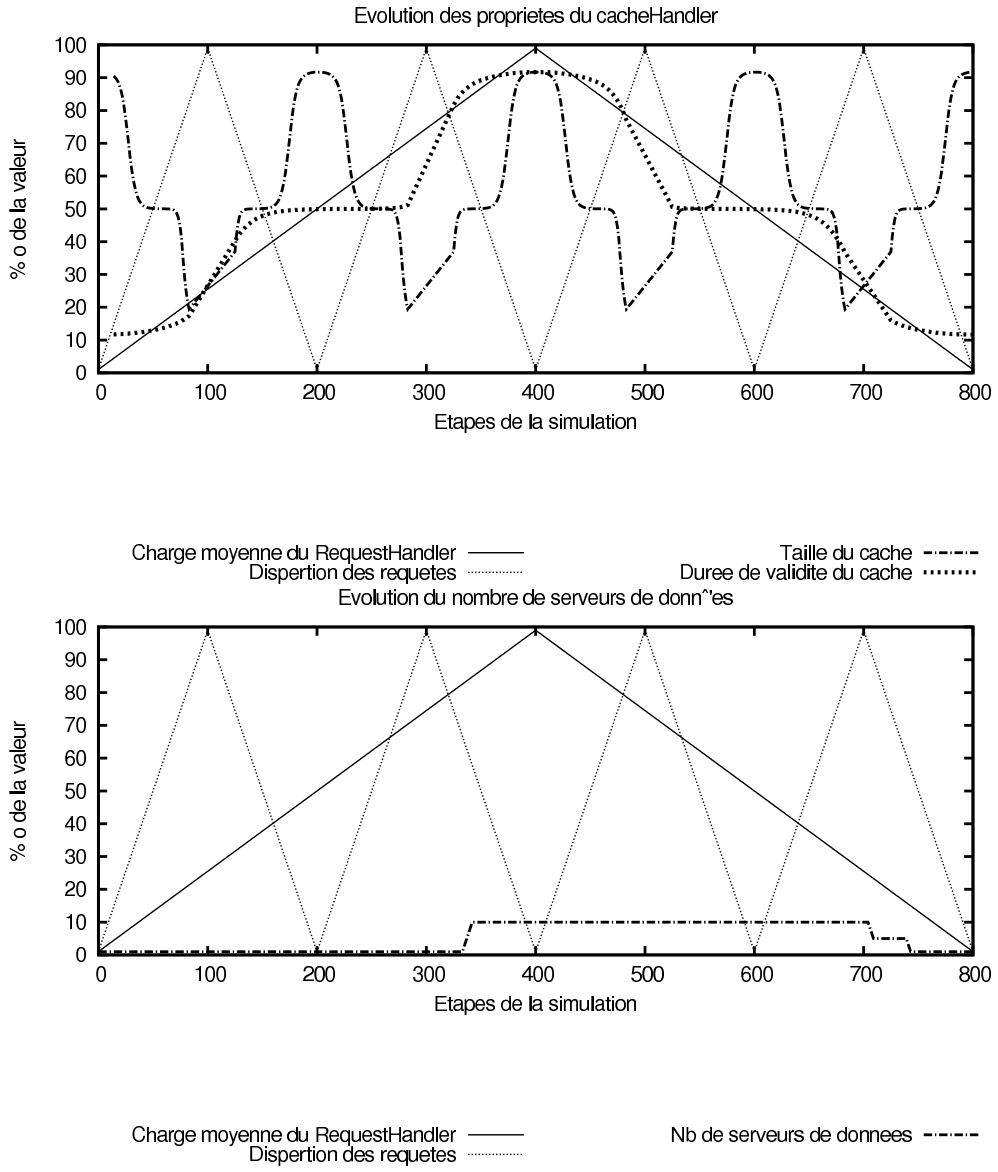

Fig. 5. Evolution des configurations locales des composants impliqués dans la réalisation du serveur web

Dans un cadre plus formel, l'outil π-ADL ([12, 16]) permet de décrire les architectures logicelles 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 ([8]) 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 notre, 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.

[7] proposent d'enrichir l'ADL de Fractal avec des règles de reconfigurations basée également sur la notion d'évenement-condition-action ou les actions modifie l'architecture et les évènements sont issus de l'environnement. Notre appoche diffère par l'utilisation de la logique floue qui permet d'exploiter une description qualitative de l'environnement.Dans la même veine, [9] propose une approche similaire à la notre dans la mesure ou 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'offre pas une description qualitative de l'environnement.

8 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 reflexivité ou le chargement dynamique. Cependant, ces mêmes plate-formes n'intègrent pas encore une description abstraite de politiques d'adaptations.

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 proche des exigences que pourraient exprimer un concepteur de systèmes adaptatifs.

La solution que nous proposons décrit les politiques d'adaptations 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.

References

1. Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, 1382:21–??, 1998.
2. Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM Press.

3. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. 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, 2006.
4. G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software, 2004.
5. Philippe David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, 2006.
6. Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In Welf Löwe and Mario Südholt, editors, *Software Composition, 5th International Symposium, SC 2006, Vienna, Austria, March 25-26, 2006, Revised Papers*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006.
7. Jérémy Dubus and Philippe Merle. Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués. In Oussalah et al. [13], pages 13–29.
8. David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.
9. Hocine Grine, Thierry Delot, and Sylvain Lecomte. Adaptive query processing in mobile environment. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–8, New York, NY, USA, 2005. ACM Press.
10. Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, and Dmitriy Kopylenko. *Professional Java Development with the Spring Framework*. Wrox Press Ltd., Birmingham, UK, UK, 2005.
11. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.
12. Flavio Oquendo. Formally Describing Dynamic Software Architectures with π-ADL. *World Scientific and Engineering Transactions on Systems*, 3(8):673–679, October 2004.
13. Mourad Chabane Oussalah, Flávio Oquendo, Dalila Tamzalit, and Tahar Khammaci, editors. *1er Conférence francophone sur les Architectures Logicielles (CAL 2006), 4-6 September 2006, Nantes, France*. Hermes Science, 2006.
14. W. Pedrycz. *Fuzzy Control and Fuzzy Systems*. Wiley, New York, USA, second edition, 1993.
15. As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n° AS5506, November 2004.
16. Hervé Verjus, Sorana Cîmpan, Ilham Alloui, and Flávio Oquendo. Gestion des architectures évolutives dans archware. In Oussalah et al. [13], pages 41–57.
17. L. A. Zadeh. Fuzzy sets. In D. Dubois, H. Prade, and R. R. Yager, editors, *Readings in Fuzzy Sets for Intelligent Systems*, pages 27–64. Kaufmann, San Mateo, CA, 1993.