Composition of Qualitative Adaptation Policies

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

Composition of Qualitative Adaptation Policies

Franck Chauvel[1,2], Olivier Barais[1], Isabelle Borne[2], Jean-Marc Jézéquel[1]

[1]Triskell Project, INRIA/IRISA
Université de Rennes 1, France
{chauvel, barais, jezequel}@irisa.fr

[2] Valoria, ALCC Group
Université de Bretagne Sud, France
{chauvel,borne}@ubs.fr

Abstract

In a highly dynamic environment, software systems requires a capacity of self-adaptation to fit the environment and the user needs evolution, which increases the software architecture complexity. Despite most current execution platforms include some facilities for handling dynamic adaptation, current design methodologies do not address this issue. One of the requirement for such a design process is to describe adaptation policies in a composable and qualitative fashion in order to cope with complexity. This paper introduces an approach for describing adaptation policies in a qualitative way while keeping the compositionality of adaptation policies. The basic example of a web server is used to illustrate how to specify and to compose two adaptations policies which handle respectively the use of a cache and the deployment of new data sources.

1 Introduction

In a highly dynamic environment, *self-adaptation* is a required property of software systems to maintain quality of service. However, although most recent middleware platforms support dynamic adaptations (such as Fractal [1] or OpenCOM [2]), current development methodologies do not address correctly the issue of how to exploit these low level facilities provided by middlewares.

The contribution of this work is to provide an approach to express *qualitative* and *composable* adaptation policies directly inspired from requirements. The description is based on simple rules whose imprecision is managed using fuzzy logic. The associated semantics, also based on fuzzy logic, preserves a way to compose these adaptation policies.

The rest of this paper is organised as follows. Section 2 presents a motivating example and illustrates why adaptation policies are complex to design with existing methodologies and Section 3 gives an overview of the approach. Section 4 defines the operational semantics of the adaptation policy composition. A numeric application is presented in Section 5. Finally Section 7 concludes and gives some future work.

2 Motivating Example

Let us consider a simple web server architecture that processes HTTP requests such as the Apache Web Server or the Microsoft IIS solution. One of the typical needs for such architectures is scalability. To improve it, we suggest to build a system which can adapt its own architecture to the load. Therefore, the system includes an optional *cache* component and a set of data servers as shown on Figure 1. The incoming requests are handled by the *Proxy* component which can use the *Cache* component to hold the request or transfer it to the *Load Balancer* component. The request is then transferred to a *Data server* and the answer is sent back to the *Proxy* component which delivers the related HTML page to the user.

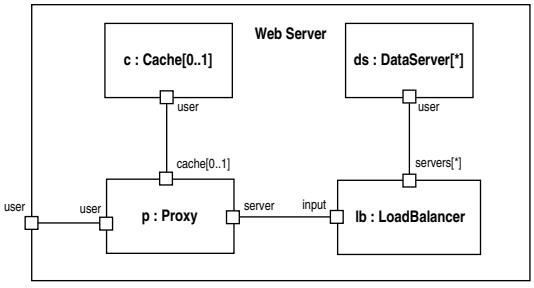

Figure 1. The web server architecture modelled as UML2.0 component diagram

The designer adds the following requirements about the adaptation of its web server's architecture:

1. The cache must be used only if the number of similar requests (so called the request deviation) is very high.
2. The amount of memory devoted to the cache component must be automatically adjusted to the load of web server.
3. The validity duration of the data put in the cache must be adjusted with respect to the load of the web server.
4. More data servers have to be deployed if the average load of the data servers is high.

On one hand, requirements 1 and 4 are related to some architectural reconfigurations since it is required to update the architecture by adding (or removing) components. On the other hand, requirements 2 and 3 are based on properties configuration (the data validity duration as a property of the *cache* component).

3 General Overview

In our approach, an *adaptation policy* is composed of a set of architectural reconfiguration rules and of a set of property configuration rules.

Fuzzy Properties In order to deal with adaptation in a *qualitative* way, designers need to specify what is the vocabulary related to a particular property, or more precisely to a particular domain. The load property is described using the vocabulary *low*, *medium* or *high*. Then the designer needs to explain where to measure the value of each property and how to impact on the value (when it is necessary). In the following code example, we use the keywords sensor and actuator to denote these two functions.

```
policy withCache
is
   property load : Real
      evolves in [0..100] as low medium high
      sensor is proxy.getLoad
   property size : Integer
      evolves in [0..2000] as small medium large
      sensor is cache.getMemoryUsed
      actuator is cache.setMemoryUsed
```

The local configuration rules allow the designer to specify the update of some local properties. For instance, in the example of the web-server, the size of the cache component must be automatically adjusted to the average load of the proxy. We assume that the platform provides an expression language which entails the verification of architectural invariants such as(*cache.isDeployed*). The behaviour can be expressed using the following three local configuration rules:

```
   when load is low
      if not cache.isDeployed
      then size is small
   when load is medium
      if not cache.isDeployed
      then size is medium
   when load is high
      if not cache.isDeployed
      then size is large
```

The architectural reconfiguration rules enable the description of the evolution of the structure of the architecture, in terms of component, connectors, and ports. For instance, in the example of the web server, two *architectural actions* can be defined to add or remove the cache component to the architecture. We assume these actions are not included into the adaptation policies, but are described separately. For instance, the Fractal platform provides a specific language named FScript which is devoted to express such actions.

```
   when requestDeviation is low
   if cache.isDeployed
      then utility of addCache is very high
   when load is high
      if cache.isDeployed
      then utility of addCache is medium or high
   when load is low  and requestDeviation is large
      if not cache.isDeployed
      then utility of removeCache is high
end policy
```

Each architectural reconfiguration rule describes the utility of performing a specific action under a particular context. These rules will enable the selection of the most appropriate architectural action when performing the adaptation.

4 Semantics

The interpretation of such fuzzy rules is performed in three steps: we fuzzify a crisp value (ie. a real value) to get a fuzzy value (such load is 'high'), then we use fuzzy inference to propagate fuzzy values on rule and finally we use defuzzification to get another crisp value.

Fuzzification A crisp value representing the current load is first measured. The membership degree of this particular value is calculated with respected to the membership function associated with the term "medium" defined for the load and used in the antecedent of the rule 1. In Figure 2, according to rule 1, the load is measured at 300 requests per second and one can say that the load is "medium" at 70% since the membership of medium is $\mu(300) = 0.7$.

Inference The fuzzy value 70% is then used as the membership degree of the "medium" term used in the outcome of the rule 1.The particular membership degree is preserved during the fuzzy inference. Figure 2,

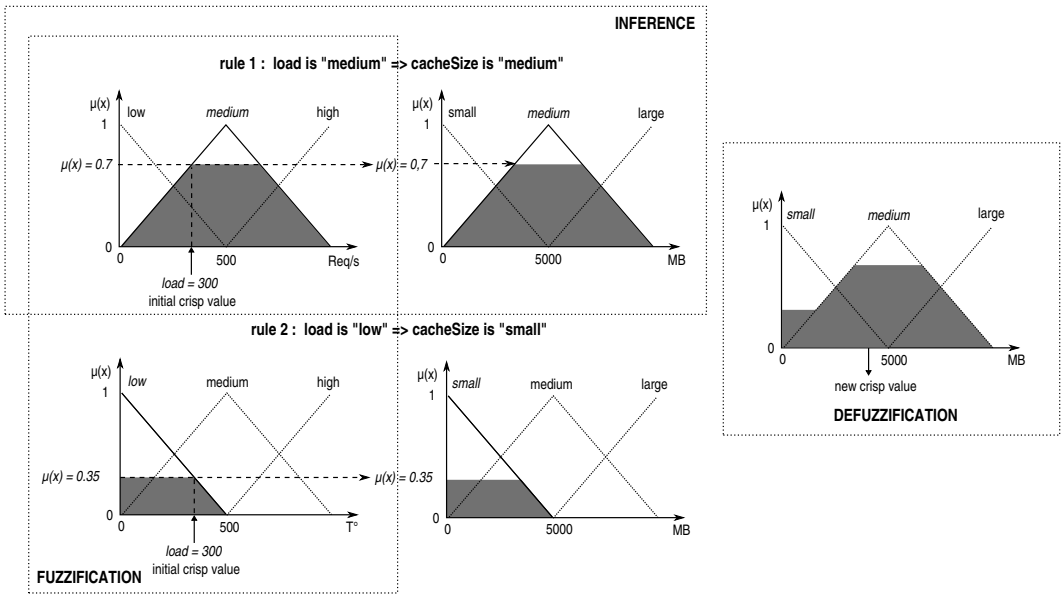

Figure 2. The Fuzzy Control process: fuzzification, inference and defuzzification

the rule 1 says that, since the load is "medium" at 70%, then the cache size will be "medium" as 70% as well.

Defuzzification When the two previous steps are performed on each rule, the result is a set of fuzzy values for each properties. For instance, in Figure 2, after processing the two rules, the result is a pair of fuzzy values which says that the cache size is "medium" at 70% and "small" at 35%. To get a crisp value related to the cache size, the defuzzification process computes the centroid of the total area depicted in gray on the figure.

We propose an algorithm based on fuzzy control which computes the relevant value for each properties but also select the most relevant architectural reconfigurations.

```
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 |
          t.getValue() > e.getValue()} }.getKey()

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

5 Numerical Application

Our web server example involves two different adaptation policies. The first one handles the use of the cache component and has been described in Section 3. The other one handles the adaptation policy that manages the numbers of data servers. The system must deploy a data server when the server load is *high* and the request deviation is *high* but remove servers when the load is *low* and request deviation is *low* as well.

During the simulation, the load of the web server increases until it reaches its maximum values, and then decrease to a *low* value. In the same time, the request deviation is evolving for a *low* to *high* values. The results presented by those two figures explain the behaviour of each adaptation policy with respect to this context changes.

Figure 3 shows that the cache component is really used when the load is *'medium or high'* and the request deviation is *'low'*. We assume that the values of the properties of the cache are undefined when the cache component is not deployed. It shows also that the size of the cache increases with respect to the request deviation. For instance, between the simulation steps 100 and 300, the size of the cache increases since the re-

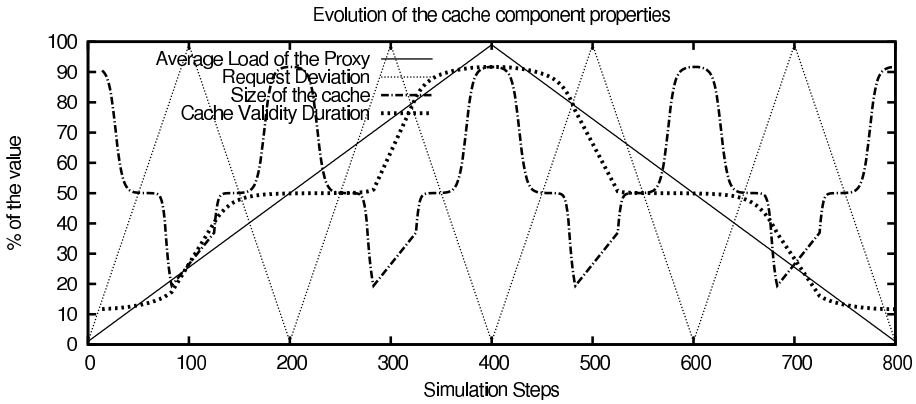

Figure 3. Evolution of the web server architecture and its properties

quest deviation is decreasing and then decreases when the request deviation is increasing again. Moreover, Figure 3 shows that when the load is *'low'*, the validity duration of the cache is *'low'* (simulation steps from 100 to 300) and *'high'* when the load is *'high'* (simulation step from 300 to 500). Finally, Figure 3 shows that servers are deployed when the load is becoming *'high'*. Between simulation steps 300 and 400, the cache component has already been deployed, the only architectural rule available is *addDataServer*. That's why data servers are added in a series. Then, between simulation steps 700 and 800 the data server are removed since the load is becoming *'low'*.

6 Tools

We developed an extension of the Fractal platform [1] to support the execution and the composition of our adaptation policies at run-time. Fractal, as many reflexive component models provides many facilities to work on component based architectures. The FScript language enables the description of imperative architecture reconfigurations based on high level primitives (such as create, delete, connect, disconnect) and thus it is perfectly suitable for the architectural actions that we want to described. Fractal also provides FPath an expression language which can be used to check architectural thus encapsulated into a specific Fractal Controller with handle adaptation in a composite component with respect to some given adaptation policies. This engine allows developers to set up various technical parameters such as the frequency of the control loop or the utility threshold used to trigger reconfigurations. This tools, containing the implementation of the semantics, the parsers for the adaptation policy language and the extension of the Fractal Component is available here [1].

[1] http://www.irisa.fr/triskell/perso_pro/obarais/pmwiki.php?n=Research.Cherokee

7 Conclusion

In a highly dynamic context, the complexity of the environment and its low level implications make the early design of adaptation a very hard task. The issue of the complexity is tackled by a systematic use of the principle of separation of concerns. Several composable adaptation policies are thus defined, each one associated to a specific dimension of the environment. The low level implications can be handled using an abstract description where the environment is described in a qualitative fashion.

The contribution of the work presented here is to provide qualitative description of adaptation policies that preserve their composability. The qualitative is build over fuzzy expression that enable the use of adhoc vocabularies such as *'small'*, *'large'* and of modifiers such as *'very'*, *'slightly'* for instance. The semantics of composition is based on fuzzy logic and enable the composition of adaptation policies which contain both architectural and property reconfigurations.

To go further on the early design of adaptation policies is to compute systematically the best set of rules for each adaptation policies. This kind of optimization already exists in classical fuzzy control theory where the optimization of the rule set is performed using neural networks or genetics algorithm.

References

[1] Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. *Software – Practice and Experience (SP&E)*, 36(11-12):1257–1284, September 2006. special issue on "Experiences with Auto-adaptive and Reconfigurable Systems".

[2] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software, 2004.