Modelling Adaptation Policies for Self-Adaptive Component Architectures

Franck Chauvel, Olivier Barais

Modelling Adaptation Policies for Self-Adaptive Component Architectures

Franck Chauvel [1,2] and Olivier Barais [2]
[1] VALORIA, Université de Bretagne Sud
Campus de Tohannic, BP 573, 56017 Vannes Cedex, France
[2] IRISA, Université de Rennes 1
Campus de Beaulieu, F-35042 Rennes, France
Email: {prenom.nom}@irisa.fr

Abstract. In most of software systems, designers try to include some self-adaptation facilities to increase the reliability of their systems. However, despite of the new methods and technologies in software engineering such as CBSE, or AOP, it is still difficult to talk about adaptation since adaptation policies might impact the architecture, the configuration data, and some extra-functional features as well. We suggest in this paper a rule-based approach to model adaptation policies that enables the description of both architectural and functional adaptation and to relate them with extra-functional properties.

1 Introduction

Self-adaptations facilities are more and more used to fulfil some extra-functional requirements. In embedded systems, designers often use self-adaptation in order to maximize the reliability by adapting the system with respect to the available resources. However self-adaptation procedures are mostly designed and hard-coded in the same time when the initial design choices do not match with the extra-functional requirements. Two main reasons can explain this failure in the development process. On one hand, adaptation policies involved the description of both high level and low level extra-functional properties (such as reliability and memory consumption). For instance, it might be interesting to adjust the system in respect with the amount of available memory in order to adjust the reliability. These extra-functional properties are not easily handled on the design level and it makes difficult the expression of adaptation policies that are based on them. On the other hand, adaptation policies can impact both the architecture and the data configuration of the system. For example, in order to maintain the reliability of a web server, more data servers can be deployed, or the cache management policy can be changed to a more efficient one.

The contribution of our work is to enable a precise description of adaptation policies where both the architecture of the system and the configuration of particular components are impacted to involve extra-functional properties. An adaptation policy is thus described at two levels: at the architectural level using some imperative actions and at the functional level where the modifications of the configuration of a component are described using a rule-based approach.

The remainder of this paper starts with the introduction of a web server example to motivate the modelling of adaptation policies. Then, Section 3 shows how software architectures are modelled in order to enables the description of adaptation policies. Section 4 focuses on this particular point Finally, after having presented the main related works in Section 5, we sum up our contribution before discussing some future works in the conclusion.

2 Motivating Example

Let's 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 need of people who design such architectures, is to make it scalable. From the architectural point of view, it means that the architecture needs to self-adapt with respect to the load of the web server.

Because of the reliability requirements, we suggest a first solution of architecture that included a 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 solve 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 deliver the related HTML page to the user.

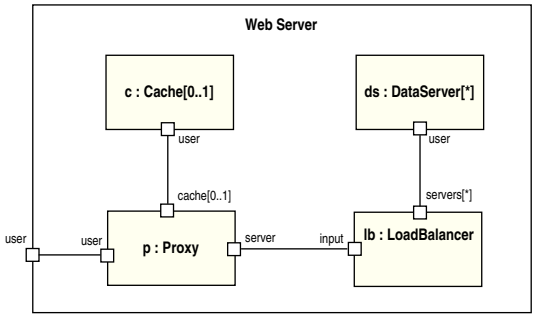

Fig. 1. The web server architecture modelled as UML2.0 component diagram

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 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 server have to be deployed if the average load of the data servers is high.
5. The algorithm used to perform the load balancing must be changed according to load of the web server.

Here, requirements 1 and 4 are related to some architectural adaptations since it is required to update the architecture by adding (or removing) components. The others requirements are based on reconfiguration if we consider for instance that the data validity duration is a part of the configuration of the *cache* component. The approach presented in the following section allows designers to model adaptation policies which correspond to these requirements.

3 Modelling Component Architectures

This section presents briefly the component model used to described component-based architectures. In this component model, a *component* is an entity which interacts with its environment (other components) through well defined connection points named *ports*. A component is also an instance of a *component class* which defines the various features included in the component.

3.1 Modelling Primitive Components

A primitive component is a "basic" component: one which does not contain any other component. A component might interact with its environment in two ways: it can provide or require some services. Provided and required services are grouped into *interfaces* which are used to describe the different ports of a component. Figure 2 (left part) models the component class *Proxy* introduced in the web server example. This component class defines three ports, namely the *user* port, the *cache* port and the data port. Each port requires or provides interfaces.

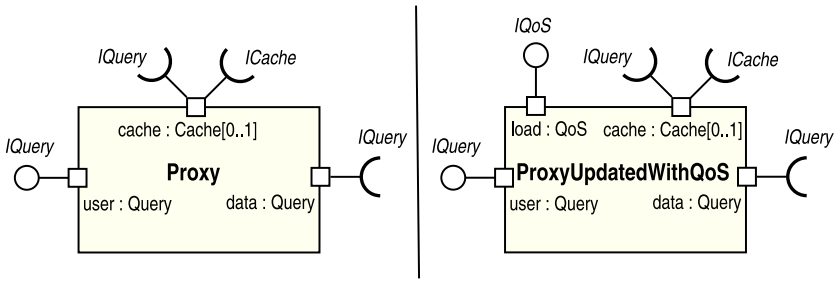

Fig. 2. The *Proxy* and *Proxy Updated* primitive components

3.2 Modelling Extra-Functional Properties

In order to include extra-functional properties in the architecture design our approach claims to reify in the architecture the sensor and the actuator (if they exist) related to these properties. For instance, to include the memory consumption of a component, we need to add a service which measures the memory consumption and another service which frees memory such as a garbage collector service.

In the example of the web server, since the designer needs to adapt the architecture with respect to the average load of the proxy, he has to define a *sensor* which is able to measure the average load. This *sensor* is thus reified as a service that can be added on any port of this component. Figure 2 (right part) shows the *Proxy* component class where a new port devoted to the quality of service purpose has been added. The service which measures the average load is defined in the interface named *IQoS*.

3.3 Modelling Component Collaboration

Primitive components can be put together to build more complex systems. A simple collaboration between two components is realized thanks to a connector that links one port of each of the two components. Connectors enables the description of complex collaborations which might be encapsulated into a *composite component*. For instance, the architecture shown by Figure 1 can be seen as a collaboration between a proxy component, a load balancer, a cache, and a collection of data servers.

In Self-adaptive architectures, the structure of the collaboration might change and the composite component has to handle this modification in order to keep the collaboration in a consistent state. To enable this, the interaction between the composite component and every sub components involved in the collaboration has to be described as all the others possible interactions: that is to say by using a specific port. In Figure 2 (right part) the port *load* will be used by a composite component which contains a proxy component to configure it.

4 Modelling Adaptation Policies

As shown in the motivating example, both architectural adaptation and reconfiguration might be needed. This section describes how to model this two types of adaptation in the component model previously described.

4.1 Architectural Adaptation

The first requirement of the motivation example sets that the cache component should be used only if the number of similar requests rises above a specific threshold. In our approach we suggest to add a sensor which measures the number of similar requests and to define an adaptation policy which adds (or removes) the cache component from the architecture with respect to the sensor's measurement. Since composite components are the entities which know all the participants of a collaboration, we suggest to make them responsible for the adaptation policy.

In order to describe such architectural adaptations we suggest to use an imperative language based on a set of architectural actions. These architectural actions are:

- **Create/Remove component instance** As we defined both component types and instances, we need to instantiate component types to get new instance and to remove old ones.
- **Create/Remove port instance** we defined also port types (with multiplicity) and so users need to create new port instances to support a new customer connection for instance.
- **Create/Remove slot instance** we also defined slot into composite components. Composite component types do not directly contain sub-component type, but a reference to another component type (called a *slot*) on which one can specify some multiplicity. So, we need a notation to create (or remove) slot instances into composite components.

– **Fill slot instance** During the life of a composite component, the contained-slot might be updated with a more efficient component. So we need to be able to fill slots.
– **Connect/disconnect port from connector** The most important aspects in component architecture is the composability of component that is reified by the notion of *connector*. So we need to connect (or disconnect) port instances form connectors.

Figure 3 shows the two operations required to describe the addition and the removal of the cache component. The *addCache* operation creates a new instance of the cache component class, put it into the devoted slot of the collaboration, and connects the two ad-hoc ports. The *RemoveCache* operation just breaks the connector that links the cache to the proxy component.

```
operation addCache() is
do
   self.cache := Cache.new
   connect(self.proxy.cache, self.cache.user)
end

operation removeCache() is
do
   disconnect(self.proxy, self.cache.user)
   self.cache.setEmpty
end
```

Fig. 3. An architectural adaptation described using architectural primitives

4.2 Modelling Functional Data Reconfiguration

The requirements 2 and 3 of the motivating example are related to component data configuration. In these two examples, it is required to update the configuration of the cache component with respect to the load of the web server. Since this kind of reconfiguration is mostly expressed as rules, we suggest to use a rule-based notation to stay as closed as possible of the requirements set by the designers.

The rules that are used to describe data reconfiguration are based on facts. Each rule links the state of a sensor to the state of an actuator. The left part of Figure 4 shows the two rules required to model the data reconfiguration of the cache component with respect to the load of the proxy component. The first rule set up that when the average load of the proxy component is “high” then the size of the cache is “big” and the validity duration is “long”. The other rule set up that when the average load of the proxy component is lower then the size of the cache is smaller and the validity duration is shorter.

To enable the use of terms such as “high”, “low”, “small” or ”big”, we consider that the sensor and the actuator used to reify every extra-functional properties (such as the size of cache, or the validity duration) offer services that define these terms. For instance, the cache component can offer a port that enables the configuration of the size of its memory with two services such as setLowMemory and setHighMemory. The mapping between these services and the rules can be ensured thanks to naming rules for such services.

```
proxy.averageLoad is 'High' =>
   cache.size is big
   and cache.validityTime is '
       Long'

proxy.averageLoad is 'Low' =>
   cache.size is 'Small'
   cache.validityTime is 'Short
       '
```

```
mode WithCache is
   trigger is proxy.SimilarRequestNumber is
       'High'
   entry is addCache()
   exit is removeCache()

   proxy.averageLoad is 'High' =>
      cache.size is 'Large'
      and cache.validityTime is 'Long'

   proxy.averageLoad is Low =>
      cache.size is 'Small'
      cache.validityTime is 'Short'
end
```

Fig. 4. An reconfiguration-based adaptation described using rules

4.3 Mixing Architectural Adaptations and Reconfigurations

In order to mix these two kinds of requirements in a single notation, the whole adaptation policy has to be defined in terms of modes. Each mode reflects one state of the architecture and includes all the rules which manage the functional data configuration.

Each mode refers to the architectural reconfigurations needed to switch (and to switch back) to this particular architectural mode. It defines also the rules that manage the functional data configuration of the components involved in this architectural mode. The right side of Figure 4 shows the mode that is defined to control the addition and the removal of a cache component into the web server architecture.

In this example, to enter the *WithCache* mode, the web server component (the composite component that manage the whole collaboration) has to run the *addCache* operation. The shift from the *normal* mode to the *WithCache* mode is triggered by a rule which set that the shift is performed when the number of similar requests detected on the proxy is “High”. This information comes to the composite component thanks to the service which reifies the sensor. This service must be available on the port which connect the *web server* composite component to *proxy* component.

5 Related Work

Many tools have been developed in recent years to manage architectural adaptation or re-configuration at run time in component-based software [1, 2] . However, there is still a gap between the modelling level (where designers specify component, data, and behaviour) and the implementation level where component are running.

Various Architecture Description Languages (ADLs) have been developed in the past as shown in [3]. However, most of them only describe software architecture in a static way. Recently several works have shown the interest of describing the software architecture dynamic [4]. For example, AADL [5] allows designers to model different modes of the same system, where each mode represents a particular state of the architecture evolution but there is no way to describe how the system switches between two modes.

In [6], Allen and al. suggest a first way to manage architectural re-configuration in component-based model. They extend the Wright component model to design component assemblies and perform consistency and verifications. This extension reuses the behaviour notation of Wright to model the reconfiguration. It allows the architect to view the architecture in terms of a set of possible architectural snapshots, each with its own steady-state behaviour. Transitions between these snapshots are accounted by special reconfiguration-triggering events. To introduce the dynamism in an architecture description, the architect has to modify the component's alphabet, and allow new messages to occur in port descriptions. Through this approach, the interface of a component is extended to describe when reconfigurations are permitted in each protocol in which it participates. Thanks to these new events, a "reconfiguration view" consumes these events to trigger reconfigurations. Contrary to our approach, they mainly work to represent some events that trigger the reconfiguration. In our work, we consider that any event can trigger the reconfiguration and we specify the reconfiguration policy at the composite level with a set of predefined modes.

Some component models have also been developed to describe component architectures. As seen in [7] most of them only deal with the structure of the component architecture. For instance, SOFA [8, 9] use CSP to describe behaviour protocols on ports and enables static verification. But the behavioural description only deals with signal emissions and signal receptions and no syntax is related to the architecture reconfiguration.

Rainbow [10] provides another appraoch to talk with component-based self-adaptation. This approach is very closed to our but has not been designed to support architectural reconfiguration and data reconfiguration in the same time.

In Fractal [11] user can defined some specific *controllers* on composite components to manage the internal of the component. Thus, adaptation policies can be described thanks to these controllers. However, controllers are a very low-level mechanism and they are related to the implementation. Our approach allows designers to express adaptation policies in the early stages of the design process.

6 Conclusion

In this paper, we introduce a way to model adaptation policies into component-based architecture. The contribution of our approach is to take into account both the architectural adaptation and the functional data reconfiguration. To do this, the description of adaptation policies is based on the definition of several modes for the architecture. Each mode corresponds to a particular state of the architecture and can be related to some extra-functional properties measured in the architecture. The designer specifies a trigger condition for each mode which specifies when the system have to switch from one mode to this one. He specifies also the rules that manage the functional data adaptation inside each mode. This work is a first step in order to manage adaptation policies in early design steps in component-based architectures. However, it is still difficult to design the modes. From the same architecture state, a different order of the adaptation modes can lead to different architectures. We plan to express the whole behaviour of each component in order to simulate the architecture and to predict performance for self-adaptive architectures such as memory consumption.

References

1. Batista, T.V., de La Rocque Rodriguez, N.: Dynamic reconfiguration of component-based applications. In: PDSE. (2000) 32–39
2. Batista, T.V., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In Morrison, R., Oquendo, F., eds.: Software Architecture, 2nd European Workshop, EWSA 2005, Pisa, Italy, June 13-14, 2005, Proceedings. Volume 3527 of LNCS., Springer (2005) 1–17
3. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. In: IEEE Transactions on Software Engineering. Volume 26. (January 2000) 23
4. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: 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, ACM Press (2004) 28–33
5. SAE, A..E.C.S.C.: Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506 (November 2004)
6. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. Lecture Notes in Computer Science **1382** (1998)
7. Lau, K.K., Wang, Z.: A survey of software component models (April 2005) Preprint CSPP-30, School of Computer Science, The University of Manchester.
8. Plásil, F., Bálek, D., Janecek, R.: SOFA/DCUP: Architecture for component trading and dynamic updating. In: CDS '98: Proceedings of the International Conference on Configurable Distributed Systems, Washington, DC, USA (1998)
9. Plasil, F., Visnovsky, S.: Behavior protocols for software components. IEEE Trans. Softw. Eng. **28**(11) (2002) 1056–1076
10. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54
11. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B.: An open component model and its support in java. In Crnkovica, I., Stafford, J.A., Schmidt, H.W., Wallnau, K., eds.: Component-Based Software Engineering: 7th International Symposium, CBSE 2004. Volume 3054 of LNCS., Springer Verlag (Jan 2004)