NFRs-Aware Architectural Evolution of Component-Based Software*

Chouki Tibermacine
VALORIA Lab.
Université de Bretagne Sud
F-56000 Vannes, France
tibermac@univ-ubs.fr

Régis Fleurquin
VALORIA Lab.
Université de Bretagne Sud
F-56000 Vannes, France
fleurqui@univ-ubs.fr

Salah Sadou
VALORIA Lab.
Université de Bretagne Sud
F-56000 Vannes, France
sadou@univ-ubs.fr

ABSTRACT

During software maintenance, some non-functional properties may be lost. This is due to the lack of an explicit definition of their links with the corresponding architectural choices. In this paper, we present a solution that automates the checking of non-functional properties after the evolution of a component-based software. Our approach emphasizes the interest of formally documenting the links binding non-functional requirements to architectural choices. The proposed formalism is based on the Object Constraint Language (OCL) applied to a software component metamodel. We also present a prototype tool which uses this documentation to warn the developer of possible effects of an architectural change on non-functional requirements.

1. INTRODUCTION

Among the maintenance activities, regression testing is one of the most expensive. When problems are found during this activity, corrections on the software architecture are required. This involves a sequence of iterations of these maintenance activities, which increases undoubtedly its cost more and more. Often, the problems are caused by the lack of knowledge, during maintenance, of the reasons which had led, the initial architects, to make some architectural choices. Indeed, without this knowledge, it is easy to lose some properties by a simple change on a specific architectural choice.

In this paper, we present an approach which helps reduce the number of the necessary iterations, in the context of component-based software. It consists of warning the software developer of the possible loss of certain non-functional properties during an evolution, well before starting regression tests. Under the assumption that the architecture of an application is determined by the non-functional requirements (NFRs) [1] –mainly the quality attributes, such as, maintainability, portability, availability–, we propose to formally document the links binding non-functional properties to their realizing architectural choices. Thus, we automate the checking of these non-functional properties after an architectural change has been made.

In the next section, we present the principles of our approach which resolves the problem pointed above. In section 3, we present an implementation of our approach, which is based on a constraint language and an association mechanism. Before concluding and presenting the perspectives, we discuss some related works in section 4.

*This material is based upon work supported by the Brittany Region Council under contract number 20046839.

2. PRINCIPLES OF OUR APPROACH

Our approach aims at solving the problems mentionned in the previous section, by expliciting, in a formal way, the reasons behind the architectural choices. Based on the assumption that architectural choices are determined by non-functional requirements, we propose to maintain the knowledge of the links binding non-functional properties to architectural choices. Thus, it becomes possible to automatically warn the developer, at each architectural evolution stage, of the potential deterioration of some non-functional properties.

In the remaining of this paper, we use the following definitions:

Non-Functional Property (NFP): a statement of the software non-functional specification;

Architectural Choice (AC): a part of the software architecture that targets one or several NFPs;

Non-Functional Tactic (NFT): a couple composed of an AC and an NFP defining the link binding one architectural choice to one non-functional property;

Non-Functional Strategy (NFS): a set of all the NFTs defined for a software;

The NFS is elaborated during the development of the first version of the architecture. NFTs appear in each development stage where a motivated AC is made. Thus, the NFS is built gradually and enriched as the project evolves. NFTs can even be inherited from a Software Quality Plan and, thus, can emerge even before the beginning of the software development. During the maintenance, the NFS may be modified along the following three rules:

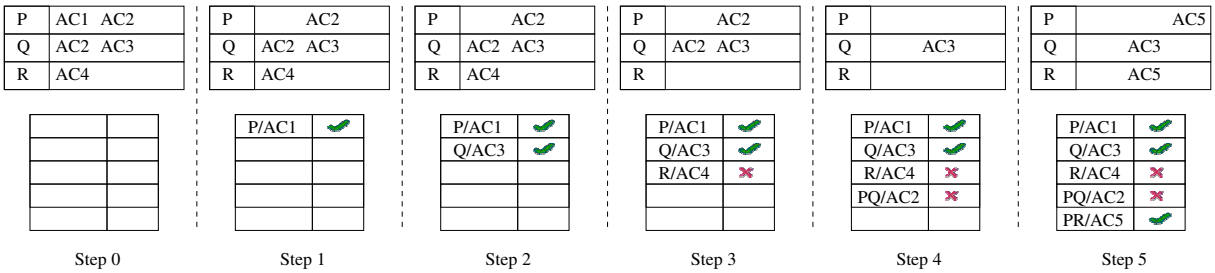

Figure 1: Assisting the evolution activity with NFS specifications

- **Rule 1:** "*a consistent system is a system where all its NFPs are involved in at least one NFT*". This condition ensures that, at the end of the maintenance process, there is no dangling NFP (i.e. with no associated AC). The breach of this condition implies *de facto* the obligation to modify the non-functional specification;
- **Rule 2:** "*we should not prohibit an evolution stage. We simply notify the attempt of breaking an AC and we specify the affected NFPs stated in the NFS*". It is of the developer's responsibility, fully aware of the consequences, to maintain or not the modification. If this modification is maintained, the corresponding NFTs are discarded. Indeed, The substitution of an architectural choice by another may be done without affecting the targeted non-functional properties. Moreover, we can be brought to invalidate, temporarily, a choice to perform a specific modification;
- **Rule 3:** "*we can add new NFTs to the NFS*". Thus, during an evolution, new architectural choices can complete, improve or replace old ones.

On the basis of these rules, we illustrate a maintenance scenario in figure 1. Let us suppose that in step 0 of a maintenance process, there are three NFPs: P, Q and R. During development, each one has been associated to a number of ACs. Choices AC1 and AC2 guarantee the property P; AC2 and AC3 target the property Q; and R is achieved by AC4. Thus, the NFS is composed of five NFTs. These are shown in the table at the top left of figure 1. In the first step, the maintainer of the system modifies the architecture. The result does not anymore hold the choice AC1. In this case, the maintainer is notified that the property P could be affected. He decides to continue his activity, aware of what he is doing and what the consequences are. The NFS is considered valid (the table on the bottom of the figure), while the first rule is satisfied (there is still a choice (AC2) associated to the property P). In step 2, the choice AC3 associated to the property Q is affected. After notification, the developer does not maintain his decision this time, and undoes the changes made on the architecture. Thus, choice AC3 is still present in the architecture. Later (in step 3), he modifies the system and affects choice AC4. He is warned that the property R will not hold anymore, because only AC4 guarantees it. Nevertheless, he decides to carry on. But this time, the NFS becomes invalid because there is a dangling property (R) with no associated choices. In step 4, the maintainer does the same with AC2. For one reason or another, he decides to continue and the NFS specification remains invalid. This time, it contains also another property (P) with no associated architectural choices. In step 5, rule 1 is satisfied again by adding a new AC (AC5) which guarantees the properties P and R. The NFS becomes valid since there is, at least, one architectural choice associated to each initial property.

We have shown through this scenario that our approach helps to obtain an architecture satisfying the initial non-functional requirements.

3. NFS IMPLEMENTATION

As stated previously, our approach requires the use of a formal language to describe an AC [9], an association mecanism to express an NFT, and a support tool for NFS evaluation.

3.1 AC Description Language

In this implementation, an AC is perceived as an architectural constraint. It is difficult to enumerate all types of constraints which the developers could be led to express. Nevertheless, it is obvious that constraints of the following kinds should be expressible: i) general rules of an organization's Software Quality Manual and Quality Plan; ii) constraints for respecting a particular architectural or design pattern.

These different types of constraints are seen as invariants and purely atemporal. From a temporal perspective, another type of constraints should be expressible. These rules have a temporal dimension and thus involve two consecutive versions of the architecture. For example, "we cannot add more than one provided interface to a component from one version to another". This type of constraints can be found in the general evolution rules of an organization, as stated in [10].

We chose a two-level solution to manage this diversity. The first level is based on OCL. Usually, OCL is used to constrain a model by navigating on it, whereas, we use it to constrain a model by navigating on a metamodel. We propose to slightly modify the syntax and semantics of the context part in OCL. At the syntactic level, we impose that every context introduces an identifier. Furthermore, this identifier must be the name of a particular instance of the metaclass cited in the context. At the semantic level, we interpret the constraint with the meaning it would have in the context of the metaclass but limiting its scope only to the instance cited in the context.

The second level takes the form of a generic MOF-compliant metamodel ArchMM (see figure 2). It contains abstractions present in analysis/design models, such as those provided by Architecture Description Languages (ADLs, like

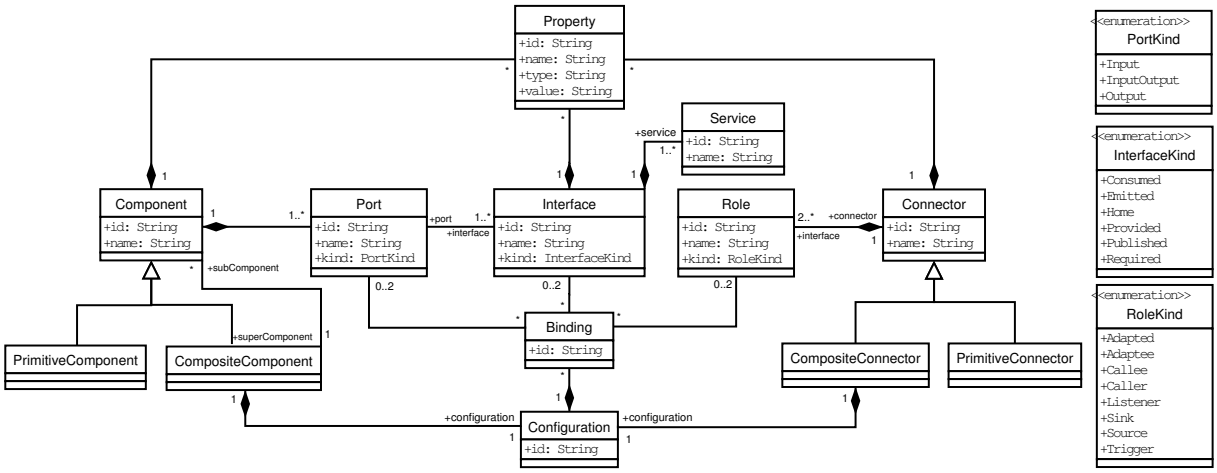

Figure 2: ArchMM: A generic metamodel representing architectural abstractions

Acme [4] and xADL 2.0 [2]) or UML 2, and implementation models, such as those provided by component technologies, like Enterprise JavaBeans and CORBA components. This metamodel enables us to adapt OCL, without making major changes on its syntax, to express constraints on architectures of component-based systems. This second level also allows, by a simple enrichment of the metamodel, to extend the list of the available operators without changing the syntax of the first-level language (i.e. OCL syntax). This two-level structure, called *ACL* –Architecture Constraint Language-, guarantees the extensibility of our formalism's expressiveness[1].

Let us consider a constraint which states that only one provided interface can be added to a primitive component, named `DataRetrievalService`, between two consecutive versions [10]. This is a typical evolution constraint because it involves two consecutive versions of the system. It fulfills the non-functional requirement stating that each component version should be evolved safely. It can be expressed as follows:

```
context DataRetrievalService:PrimitiveComponent inv:
DataRetrievalService.port.interface
->select(i:Interface|i.kind = 'Provided')@old
->comparedTo(DataRetrievalService.port.interface
->select(i:Interface|i.kind = 'Provided'))
->size() <= 1
```

Note that we have introduced a new syntactic construct to ACL so that we can reference old versions of architecture descriptions: the `@old` mark. The `@old` returns the descriptions of the former version of an architecture. In this example, it returns a collection of the provided interfaces descriptions of the `DataRetrievalService` component version before evolution. Note that we have also added a collection operation: `comparedTo(c2:collection(T)):Collection(T)`. It returns a collection which represents the difference between the collection to which it is applied and the parameter collection (c2). We introduced this new operation in order to not affect OCL semantics of the set difference operator (-). Indeed, traditionally in OCL, the set difference operator compares references of object collections, whereas `comparedTo(...)` uses the value of the `id` attribute. In our situation, we are confronted to a comparison of collections of architecture descriptions. The elements of these collections are objects which have different references from one version to another, but the same value of the `id` attribute.

[1]More details about ArchMM and ACL can be found in [9]

3.2 Association Mecanism

An association mechanism allows the binding of an architectural constraint (AC) described with ACL to a list of NFPs. Thus it merges several NFTs, sharing the same AC, in one structure. The mechanism we propose follows a structure inspired by the ISO 9126 quality standard [5]. A binding associates an AC to an external quality characteristic with its comment extracted from the specification document. These characteristics are organized in the form of a forest, which counts 6 trees. The roots of these trees are, at the moment, the 6 higher-level external characteristics stated by the standard: maintainability, portability, reliability, efficiency, functionality and usability. The second level of each tree contains the external sub-characteristics detailed by the standard. Concretely, an NFS specification starts by announcing the different NFPs associated to an AC; then the AC is described. It is illustrated in the form of the following XML structure:

```
<nfs id="000005">
  <nft id="000193">
    <nfp id="001043" characteristic="Portability">
      The software system should be portable
      over different environments. It can
      serve different applications for museum
      supervision or access control
      data administration
    </nfp>
    <ac id="010116" name="FacadePattern">
      <!--Here we edit the ACL constraint-->
    </ac>
  </nft>
</nfs>
```

Each `nfp` element has an attribute that specifies the external characteristic which represents the NFP. The property, as stated in the non-functional specification, is then introduced in this XML element. Each NFS specification is associated to the architecture description to which it applies.

3.3 Support Tool

We developed a prototype tool, which allows the edition, the validation and the evaluation of NFS specifications, and the assistance of a software developer during an evolution operation. It is composed of an:

- **NFS Editor:** It uses the XMI format of the metamodel (ArchMM), to guide the developer in editing its architectural constraints (ACs) and their binds to NFPs.

- **NFS Validator:** This module validates the NFS of the initial version of the component by checking all its ACs. Then, it produces an archive file composed of the architecture description file and the NFS specification file.

- **Evolution Assistant:** During an evolution, the new version of the component's architecture is proposed to the tool. It evaluates the NFS using the old and the new architectural descriptions. If the NFS is valid, the new architecture description may be associated to the NFS specification and then it may be saved; else the architecture evolver is warned that some ACs may be altered and consequently that the associated NFPs may be affected.

The presented prototype takes into account only architecture descriptors defined in the Fractal model or xAcme ADL, but, was conceived to easily support other ADLs or component models.

4. RELATED WORK

In the literature, non-functional properties has been supported on the software development through two approaches. The first one is process-oriented, while the second is product-centric. In the first approach, methods for software development driven by NFRs are proposed. They support NFRs refinement to obtain a software product which is compliant with the initial NFRs [7]. In the second approach, the non-functional information is embedded within the software product. This is also the case in our approach, where NFS specifications are associated to architectural descriptions. Using the same approach, Franch and Botella [3] propose to formalize non-functional requirement specifications. These are encapsulated in modules, which are associated to a component definition and to its implementations. They also propose an algorithm, which allows the selection of the best implementation for a given component definition. This selection method can be used when a new implementation is proposed to ensure that the best one is used. The authors mean by "best" the implementation that better fits to its non-functional requirements. Compared to our work, components are seen differently. We focus on an architectural aspect of components, while they consider the abstract data type view of components. In addition, in our case, the maintenance is performed on architectural descriptions while changes in their approach are at an implementation level. Furthermore, in our work we deal at the moment only with static quality attributes, like maintainability. In their work, dynamic quality attributes (like, performance and reliability) are taken into account.

5. CONCLUSION AND FUTURE WORK

Perry and Wolf [8] modeled software architectures as a set of *Elements*, *Form* and *Rationale*. *Elements* are architectural entities responsible for processing and storing data or encapsulating interactions. *Form* consists of properties –constraints on the choice of elements, and relationships – constraints on the topology of the elements. The *rationale* captures the motivation for the choice of an architectural style, the choice of elements and the form. While the description of the two first aspects have received a lot of attention by the software architecture community [6], there has been a little effort devoted to the last aspect. In this paper, we presented NFS specifications, as a contribution to the description of the last element in Perry's model. This allows us to assist the architectural maintenance activity and, thus, prevent the loss of non-functional properties. In addition, it is, as we think best, a good practice for documenting software architectures and, thus, facilitating software comprehension in maintenance activities.

On the conceptual level, we plan: i) to associate metrics to the NFPs. This makes for a better assistance in the maintenance activity; ii) the definition of a library of architectural styles and design patterns, in order to facilitate the description of architectural constraints. On the tool level, we are working on its extension to support architectures described in UML 2 and in CORBA Component Model;

For more information about the project, on which depends this work, the reader is invited to follow the link: *http://www-valoria.univ-ubs.fr/Composants/se/current/Cell*

6. REFERENCES

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 2nd Edition*. Addison-Wesley, 2003.

[2] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM TOSEM*, 14(2):199-245, 2005.

[3] X. Franch and P. Botella. Supporting Software Maintenance with Non-Functional Information. In *Proceedings of CSMR'97*, pages 10-16, Berlin, Germany, March 1997. IEEE Computer Society.

[4] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47-68, Cambridge University Press, 2000.

[5] ISO. Software Engineering - Product quality - Part 1: Quality model. International Organization for Standardization web site. ISO/IEC 9126-1. http://www.iso.org, 2001.

[6] N. Medvidovic and N. R. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE TSE*, 26(1):70-93, 2000.

[7] J. Mylopoulos, L. Chung, and B. Nixon. Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. *IEEE TSE*, 18(6):483-497, 1992.

[8] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *Software Engineering Notes*, 17(4):40-52, 1992.

[9] C. Tibermacine, R. Fleurquin, and S. Sadou. Preserving Architectural Choices throughout the Component-based Software Development Process. In *Proceedings of WICSA'05*, Pittsburgh, Pennsylvania, USA, November 2005.

[10] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78-85, 2000.