On-Demand Quality-Oriented Assistance in Component-Based Software Evolution

Chouki Tibermacine, Régis Fleurquin, and Salah Sadou

VALORIA, University of South Brittany, France
{Chouki.Tibermacine,Regis.Fleurquin,Salah.Sadou}@univ-ubs.fr

Abstract. During an architectural evolution of a component-based software, certain quality attributes may be weakened. This is due to the lack of an explicit definition of the links between these non-functional characteristics and the architectural decisions implementing them. In this paper, we present a solution that aims at assisting the software maintainer during an evolution activity on his demand. It requires the definition of a documentation during development, organized in the form of bindings between formal descriptions of architectural decisions and their targeted quality attributes. Through an assistance algorithm, the approach uses this documentation in order to notify the maintainer of the possible effects of architectural changes on quality requirements. We also present a prototype tool which automates our proposals. This tool and the overall approach has been experienced on a real-world software in order to validate them.

1 Introduction

An intrinsic characteristic of software, addressing a real world activity, is the need to evolve in order to satisfy new requirements. Maintenance is now, more than ever, an inescapable activity, the cost of which is ever increasing (between 80 % and 90 % of the software total cost [5, 19]). Among the maintenance activities, the checking of functional and non-functional non regression of a software, after its evolution, is one of the most expensive. It consists in checking the existence of the newly required service or property after modification, on the one hand, and in verifying that the other properties and/or services have not been altered on the other hand. This checking is done during the regression testing stage. When problems are found, a rework on the software architecture is required. This involves a sequence of iterations of the maintenance activities, which make undoubtedly its cost grow more and more.

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

In the next section, we show, through a system architecture, how some quality requirements can be mapped into architectural decisions and how problems can rise when evolving this architecture. We present, in section 3, the principles of our approach which helps to resolve the problem pointed in the section before. The proposed solution is based on a documentation, which is introduced in section 4. An algorithm which uses this documentation and assists the maintenance activity is discussed and illustrated by an example in section 5. A prototype tool for evolution assistance, and the validation of the approach are then presented in section 6. Before concluding and presenting the perspectives, we discuss some related works in section 7.

2 Illustrative Example

Along this paper, we use a simple imaginary example which represents a Museum Access Control System (MACS)[1]. Figure 1 provides an overview of its architecture. The system receives as input the necessary data for user authentication (`Authentication` component). After identification, the data is sent to the access control component (`AccessCtl`). The latter consults an authorization database (the component `AuthDataStore`) to check if the user[2] is authorized to enter the museum or not. Then, it adds other data elements (entrance hour, the visited gallery in the museum, etc.) to the received data flow and sends it to the `Logging` component. The component `AdminService` allows the administration of the database abstracted by the component `AuthDataStore`. The `Logging` component provides an interface for persistent local logging. This interface is used by the archiving data store component (`ArchivDataStore`) which provides an interface to a data retrieval service component (`DataRetrievalService`). This component implements an interface which allows local supervision of the museum and querying of logs. The two components `AdminService` and `DataRetrievalService` export their interfaces via the same component `DataAdminRetrieval`. After local archival storage, the data is transmitted (by the component `ServerTrans`) via the network to the central server of the organization responsible for the museum security for central archival storage.

2.1 Some Architectural Decisions and their Rationale

The architecture, described in Figure 1, was designed taking into account quality requirements defined in the NFRs (Non-Functional Requirements) specification. We present some of these requirements and their architectural mappings.

[1] This software system is not a real-world component-based one. We just defined a few years ago within a development project a formal specification of it. We think that it is simple to present and to use as an illustrative example.

[2] Users of the museum are visitors, exhibition organizing committee members, museum administrative and service employees

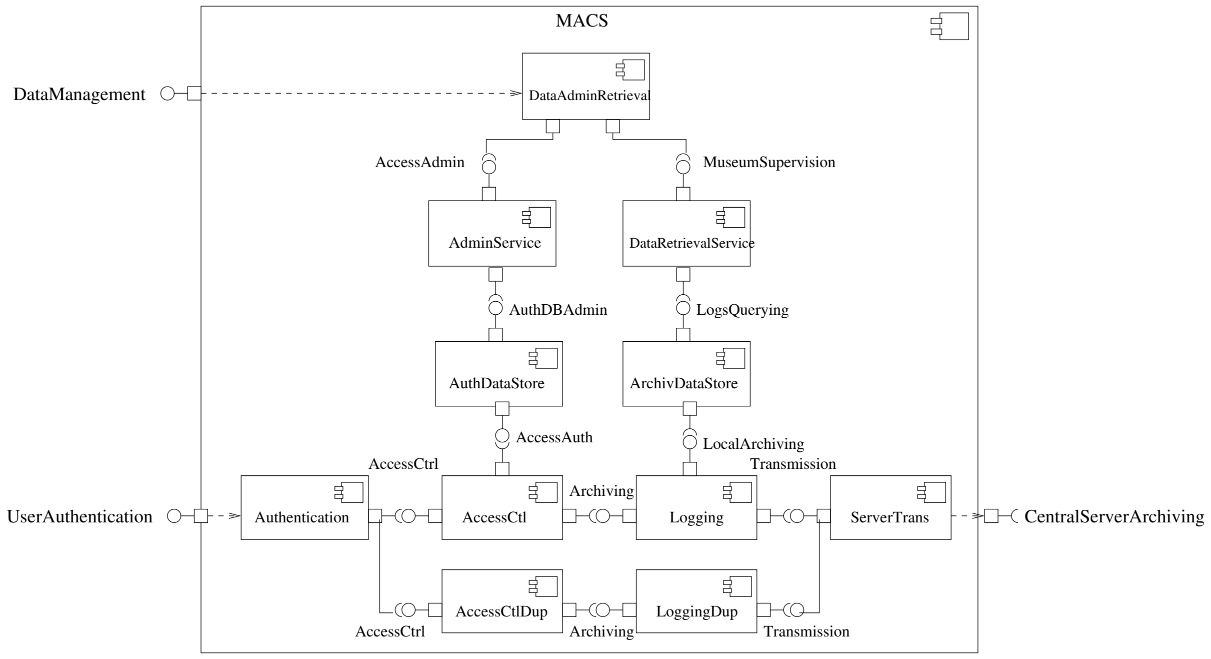

Fig. 1. A Simplified architecture of a Museum Access Control System

1. "*The system should be easily maintained.*" (We mark this maintainability quality attribute QA1.) This is ensured by the layered pattern [20], which can be seen if we decompose the system's architecture vertically. (We mark this architectural decision AD1.)
2. "*The software system should be portable over different environments. It can serve different applications for museum supervision or access control data administration.*" (This portability property is marked QA2.) To reach this *portability* level, a façade component -with analogy to façade objects [7]- was designed as front to MACS's internals. In Figure 1, this is performed by the component `DataAdminRetrieval`. All communications from client applications to access MACS's data management services transit by this component. (We mark this architectural decision AD2.)
3. "*The access control functionality should be more available for service employees.*" (We mark this availability property QA3) In the bottom of Figure 1, the sequence of components `AccessCtl` and `Logging` is duplicated. This redundancy scheme (marked AD3) is a mean to make the system fault tolerant and thus fulfills the availability requirement. If one of the two components (`AccessCtl` or `Logging`) fails, the sequence below (`AccessCtlDup` and `LoggingDup`) takes over the process. In this degraded mode of the system functioning, the component `AccessCtlDup` authorizes the access only to

service employees. Logs remain in the state of the component `LoggingDup` and are not persistent in the `ArchivDataStore` component. Then, the data is transmitted to the central server.
The sequence of these duplicates is organized as a pipeline [20]. (We mark this decision AD4.) In a pipeline each filter (component) has only one reference to the downstream component. This guarantees a certain level of maintainability (minimal coupling, QA1) required for such emergency solution. This pattern also guarantees a certain level of performance defined in the NFRs specification, but not detailed here. We mark this last quality attribute QA4.
The developers have introduced a data abstraction component (`DataRetrivalService`), which abstracts details of the underlying databases. This architectural decision is marked AD5. Indeed, this traditional practice fulfills the first attribute (QA1).

2.2 Some Evolution Scenarios and their Consequences

Let us assume that the maintenance team receives two change requests that must be tackled urgently. As a consequence, changes are made without taking into account the associated design documentation. The first request imposes that henceforth some data should be directly transmitted to the central server. That is, a part of information about service employees should be directly sent to the `ServerTrans` component after identification. This gives the system more (time and space) performance. Thus, the system maintainers decide to create a link between `AccessCtl` component and `ServerTrans` component; and between `AccessCtlDup` and `ServerTrans` components. (We mark this change ACG1.) In the last case, the `AccessCtlDup` component finds itself with two links. The first one with the `LoggingDup` component for the data flow that is not affected by the modification and the second one with the `ServerTrans` component for the data that is directly transmitted to the central server. This modification makes the system lose the benefits of the pipeline structure (breaks AD4). Therefore, the initial level of maintainability (QA1) of the system is now weakened.

While the first change request has a non-functional goal, the second change is of functional nature. It asks to add a new component representing a notification service: `DB_UpdateNotification`. This component notifies the client applications, subscribed to its services, when updates are done on the archiving database. This new component exports a *publish/subscribe* interface via the port that provides the interface `DataManagement`. It implements a *publish on-demand* interaction pattern and uses directly the component `ArchivDataStore`. This change (marked ACG2) makes the system lose the benefits of the façade pattern guaranteed by the component `DataAdminRetrieval` (AD2) and consequently weakens the availability quality attribute QA2.

The lack of knowledge during evolution about the reasons which have led the initial architects to make such decisions, may easily lead to break some architectural decisions and consequently affect the corresponding quality attributes. The two simple examples above illustrate how can we lose such properties. This is often noticed during regression testing. Thus, it is necessary to perform changes on the architecture another time, and to iterate, frequently, for many times. Several similar remarks have been noticed by our industrial partner when evolving one of its complex system (a cartographic converter from different binary files and spatial databases to SVG (Scalable Vector Graphics) format, for using them in a Geographic Information System -GIS-). We didn't present this system as an illustrative example in this paper for reasons of brevity and simplicity.

3 Principles of the Approach

Our approach aims at solving the problems quoted in the previous section. It consists in making explicit and formal the reasons behind architectural decisions. The choice of a formal language to specify this documentation guarantees not only the unambiguity of descriptions but also allows the automation of some operations, like the preservation of architectural choices throughout the development process of a component-based software [23].

Based on the assumption that architectural decisions are determined by the quality information stated in the requirements specification, we propose to maintain the knowledge of the links binding quality attributes to architectural decisions. This knowledge is of great interest for maintainers on two accounts:

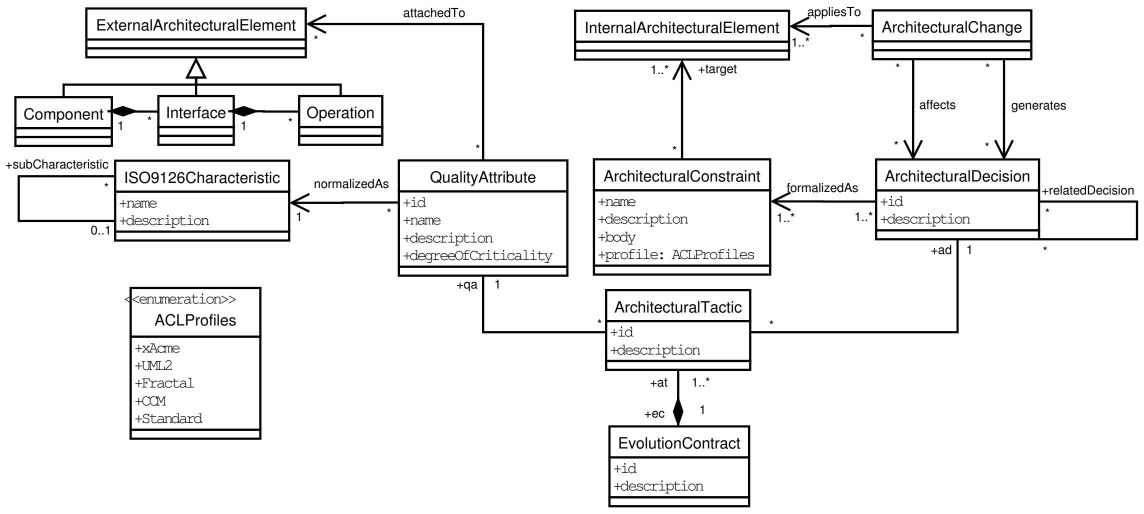

Fig. 2. Structure of Contracts

– **Preservation of Quality Attributes:** it will be possible to warn the developer, at each architectural maintenance stage, of the potential deterioration of some quality attributes;

- **Architecture Comprehension:** it is easier to understand a specific architecture when we already know its motivations. Starting from the specification of a targeted evolution, it becomes possible to identify the related architectural artifacts.

In the remaining of this paper, we use some terms which we define as following:

Quality Attribute (QA): a quality characteristic targeted by one of the statements in the non-functional requirements specification;
Architectural Decision (AD): a part of the software architecture that targets one or several QAs;
Architectural Tactic (AT): a couple composed of an AD and a QA defining the link between one architectural decision and one quality attribute;
Evolution Contract (EC): a set of ATs defined for a given software.

These definitions are illustrated in Figure 2. An architectural decision may have several related decisions, which are present in different ATs. For example, AD3 (replicated components) in section 2 has one related decision which is AD4 (pipeline). Indeed, if AD3 is removed from MACS architecture, AD4 will also disappear. As explained in section 4.2, an architectural decision is formalized as an architectural constraint. Each constraint is specified using an architectural predicate language, called ACL. This language has several profiles dedicated to existing architecture/component models, for example xAcme [26] or CORBA Component Model [14] (more details are given in section 4.2). A constraint targets an internal architectural element. Internal architectural elements are architectural abstractions present in the metamodel of the used architecture or component description language (e.g. a component, a connector or an interface). An architectural change is applied on one or many internal architectural elements. It can affect or generate one or several architectural decisions. A QA corresponds to an ISO/IEC 9126 [9] characteristic or sub-characteristic (e.g. maintainability, portability or usability). Each QA has a degree of criticality (inspired from Kazman's QA scores and Clements's QA priorities [3]). The value of this degree, specified by developers, stipulate the importance of this quality attribute in the architecture. It takes values between 1 and 100. The sum of values of all these degrees should be at most 100. Starting from non-functional requirements (NFRs), we may extract one or several QAs. Each one is attached to several external architectural elements. These elements represent the common **externally visible** architectural concepts present in existing component models (component, interface and operation). These external architectural elements are a subset of internal architectural elements.

4 Capturing Architecture Decisions and their Rationale in Contracts

Based on the structure presented above, to document an architecture, we need a language for defining evolution contracts and a tool set to edit and interpret this documentation.

4.1 Evolution Contract Organization

In order to specify textually ECs, we use an XML representation which conforms to the structure presented in Figure 2. The following listing illustrates an example of such specification.

```
<evolution-contract id = "000001">
  <architecture-tactic id = "000100">
    <description>
      This tactic ensures the Portability quality requirement by
      using a Facade Design Pattern
    </description>
    <quality-attribute id = "001000" name = "Portability"
         characteristic = "Portability">
      <description>
        The software system should be portable
        over different environments. It can
        serve different applications for museum
        supervision or access control
        data administration
      </description>
    </quality-attribute>
    <architecture-decision id = "010000">
      <description>
        Facade design pattern
      </description>
      <formalization profile = "Fractal">
        <!--Here we edit the ACL constraint-->
      </formalization>
    </architecture-decision>
  </architecture-tactic>
</evolution-contract>
```

This simple example illustrates a simple EC composed of only one of the ATs presented in section 2. This AT concerns the façade design pattern AD associated to the portability QA. The `formalization` element of this EC contains the formal definition of the AD which is described in the next section.

4.2 AD Definition Language

In order to formalize architectural decisions, we proposed a predicate language called ACL (Architectural Constraint Language). This language is based on a slightly modified version of UML's Object Constraint Language [15], called CCL (Core Constraint Language), and on a set of MOF metamodels. Each metamodel represents the architectural abstractions used at a given stage in the component-based development process. A couple formed by CCL and a given metamodel represents an ACL profile. Each profile can be used at a stage in the development process. We defined ACL profiles for xAcme, UML 2 [16], CORBA components, Enterprise JavaBeans [21] and many others. CCL navigates in the architecture's metamodel in order to define constraints on its elements.

Instead of presenting the grammar of ACL, we preferred to illustrate it through the description of two AD examples from section 2.1. We describe this ADs using the standard profile of ACL, which is composed of CCL and of a generic architecture metamodel called ArchMM [23]. This metamodel is used as an intermediate representation when transforming ACL constraints from one profile to another.

The listing below describes the constraint enforcing the façade architectural pattern in the component MACS.

```
context MACS:CompositeComponent inv:
let boundToDataManagement:Bag=MACS.port.interface
->select(i:Interface|i.kind = 'Provided'
and i.name = 'DataManagement').port.binding
in
((boundToDataManagement->size() = 1)
and (boundToDataManagement.interface
->select(i:Interface|i.kind = 'Provided').port.component.name
->includes('DataAdminRetrieval')))
```

This constraint states that the `DataManagement` provided interface of MACS component must be bound internally to one and only one interface. The latter corresponds to the provided interface of `DataAdminRetrieval` component.

The following constraint concerns the replicated components stated by AD3.

```
context MACS:CompositeComponent inv:
let startingPort:Port=MACS.port->select(p:Port|
i.name='UserAuthentication') in
let startingComponent:Component=startingPort.getInternalComponent() in

let endingPort:Port=MACS.port->select(p:Port|
i.name='CentralServerArchiving') in
let endingComponent:Component=endingPort.getInternalComponent() in

let paths:OrderedSet=MACS.configuration
.getPaths(startingComponent,endingComponent) in

paths.size() = 2 and paths->first()->excludesAll(paths->last())
```

This constraint uses two operations from ArchMM. The first one (`getIntern-alComponent()`) returns the subcomponent attached to a given port of a composite component. The second operation (`getPaths(c1:Component,c2:Component)`) returns an ordered set of all the paths between the components given as parameters. The returned paths are also represented by ordered sets of components. A returned path excludes the parameter components. The constraint states that it must exist two distinct paths between the component attached to `UserAuthentication` port and the component attached to `CentralServerArch-iving` port.

The two-level expression nature of ACL guarantees the homogeneity of constraints defined in different stages of the development process. Indeed, only metamodels change from one stage to another; the core constraint language remains the same. This has been of great interest when we performed transformations of constraints from one stage to another in order to automatically preserve architectural decisions [24].

5 Using Contracts in Evolution Assistance

In the proposed approach, an AT is perceived as a constraint which has to be checked for validity during each evolution. An evolution contract may be seen as a contract, because it documents the rights and the duties of two parties: the developer of the previous version of the architecture who guarantees the quality attributes and the developer of the new version who should respect the evolution contract established by the former. The evolution contract is elaborated during the development of the first version of the architecture. ATs appear in each development stage where a motivated AD is made. Thus, the evolution contract is built gradually and enriched as the project evolves. ATs can even be inherited from a Software Quality Plan and thus, can emerge even before starting the software development. Thereafter, this evolution contract can be modified in respect of the rules below:

– **Rule 1:** "*a consistent system is a system where each QA is involved in at least one AT*". This condition ensures that, at the end of the maintenance process, there is no dangling QA (i.e. with no associated ADs). 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, at the demand by the developer of a change validation, the attempt of breaking one or more ADs and we specify the affected QAs stated in the evolution contract*". 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 ATs are discarded. Indeed, The substitution of an architectural decision by another may be done without affecting the targeted quality attributes. Moreover, we can be brought to invalidate, temporally, a decision to perform a specific modification;
– **Rule 3:** "*we can add new ATs to the evolution contract*". Thus, during an evolution, new architectural decisions can complete, improve or replace old ones.

The previous rules are illustrated by the simple architectural maintenance scenario in Figure 3. Consider the previous example presented in section 2. The evolution contract associated to the MACS component, which contain 6 ATs, is illustrated in the bottom part of the figure. Note that for reasons of simplicity we organized the evolution contract by factorizing QAs. Architectural changes are represented at the top of the figure. The minus symbol stipulates that the corresponding AD has been affected, and the plus symbol shows that the AD is preserved or enhanced. Forward arrows mean that the architectural maintainer decides to validate her/his change, however backward arrows mean she/he does not maintain her/his decision. At the middle of the figure, we illustrate the evolution of the assistance system, the different warnings that it triggers and the validation or not of the different intermediate evolution contracts.

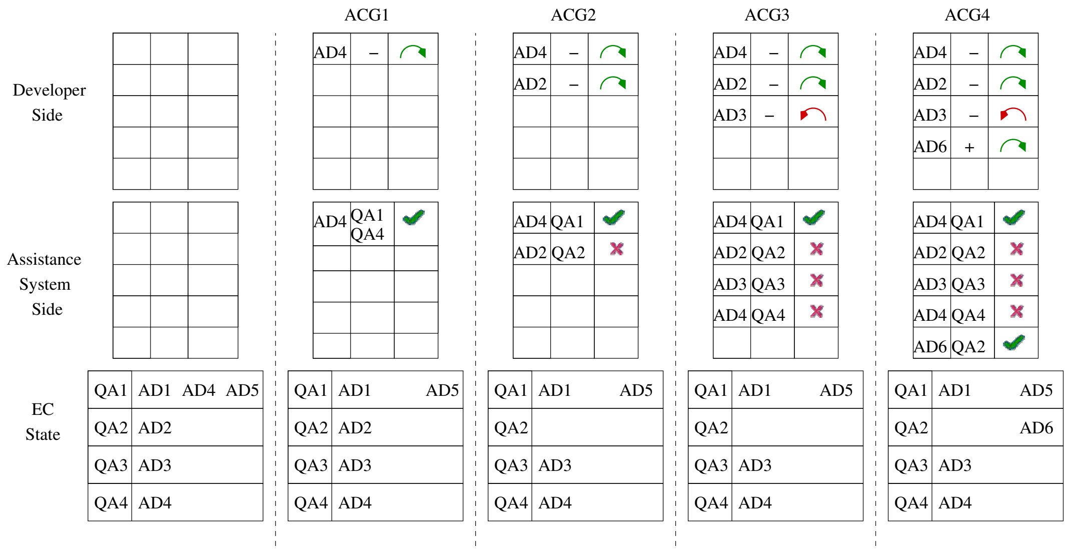

Fig. 3. Assisting the evolution activity with evolution contract

Let us suppose that a software maintainer applies ACG1, which was presented in section 2, to MACS architecture (second column of figure 3). As stated in that section, this change affects AD4. The assistance algorithm checks the ACL constraint present in MACS's EC which is associated to this decision, and consequently warn the maintainer that AD4 is altered and that QA1 and QA4 are also possibly affected. Knowing that ACG1 enhances QA4 -AT6(QA4,AD4) not affected-, the maintainer decides to validate his change. The EC is considered valid, while there is no QA without associated ADs (see the second table in the bottom of the figure).After that, the maintainer decides to apply ACG2, which was also presented in section 2. He is thus notified that AD2 and its associated QA, namely QA2, are potentially affected. He decides to continue, but the EC is in this case not valid, as soon as there is one QA (QA2) without associated AD (see the third table in the bottom of the figure). Later, the maintainer tries to apply ACG3. This change consists in removing the `LoggingDup` component. Concretely, the system maintainer discovered that data in this component is not consistent with data in the `ArchivDataStore` component. He is immediately notified that ACG3 alter AD3 (replicated components) and consequently QA3 (availability property) will be eventually affected. In addition, this AD has one related decision, namely AD4 which represents the pipeline pattern. The assistance system, by scanning the EC, warns the maintainer that AD4 and QA1 (Maintainability) and QA4 (Performance) are also affected. This time, the maintainer does not keep his changes and undoes the changes made on the architecture. Note that, the EC is still invalid. Finally, the maintainer decides to make a new architectural change (ACG4). It consists in removing the hierarchical connector between the newly added notification sub-component (`DB_UpdateNotification`) and MACS's `DataManagement` port, and adding a new connector between the former and the component `DataAdminRetrieval`. The motivation behind adding this new AD, marked **AD6**, is to re-ensure the portability quality characteristic and thus reintroduce QA2 in the EC. In this case, all QAs have corresponding ADs. The contract is thus considered valid.

6 Evolution Assistance Prototype Tool & Validation

In order to validate our approach we developed a prototype tool, called AURES (`ArchitURe Evolution aSsistant`), which allows the edition, the validation and the evaluation of ECs. In addition, it assists the software developer during an evolution operation. We experienced our approach with our industrial partner on real-world software system they developed the last year. We present in the following subsections the prototype tool, its structure and how it operates. After that, we introduce how we validated our approach.

6.1 AURES Architecture

This tool is organized as in Figure 4 and is composed of the following elements:

EC Editor: This component allows the edition of evolution contracts. It uses the XMI format of metamodels of the different ACL profiles to guide the developer in editing her/his architectural constraints. It asks the developer to introduce the necessary information discussed previously in order to complete and generate the EC.

EC Validator: This component validates an EC with an introduced architecture description. If the EC evaluation returns false, the developer is requested to correct either the EC or the architecture description; else this component produces an archive file composed of the architecture description and the EC files, saved by the `Version Handler` component.

Evolution Assistant: When a new version of the architecture is submitted, the EC of the latest version is then checked out from the `Version Handler` component. This EC is then reevaluated on the new architecture description. If the evaluation returns true, the new architecture description is associated to the EC and then it is saved in the `Version Handler` component; else the

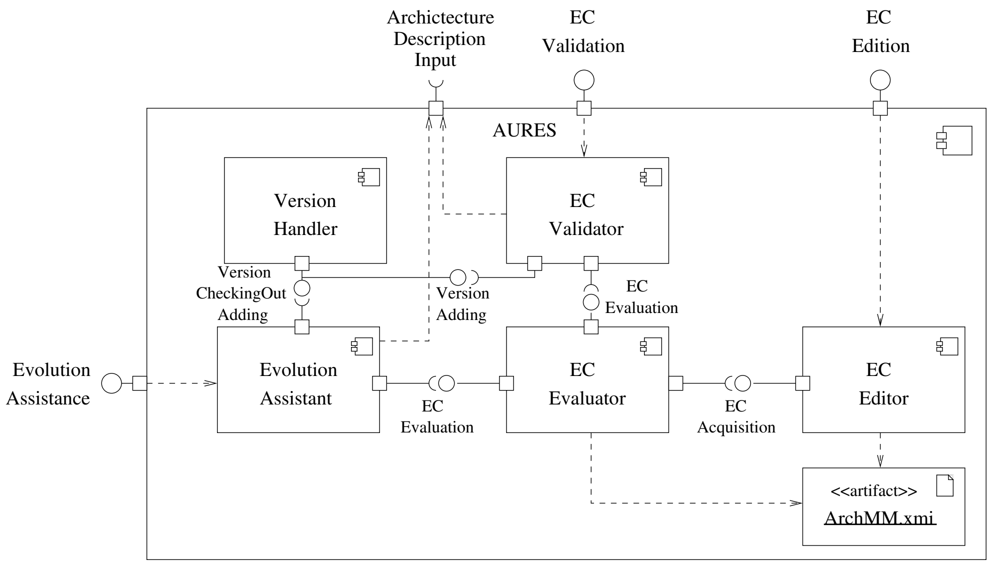

Fig. 4. A prototype tool for ECs edition and interpretation, and for evolution assistance

architecture evolver is warned that some ADs are altered and that consequently the associated QAs are affected.

EC Evaluator: To evaluate an EC, this component uses a temporary pivot model and the ArchMM metamodel. The pivot model is a direct instance of ArchMM, produced by the `Description Transformer` subcomponent, from a given architecture description. This subcomponent executes a set of XSL scripts to make XML transformations of architecture descriptions. For the moment, the prototype presented in this paper supports architectures described in the Fractal ADL [2] and in xAcme. However, the pivot model makes the tool easily extensible. The `EC Evaluator` is composed of an `ACL Compiler` which extends the OCL Compiler[11] and which requires the AD part of the EC. It also contains an `ACL Interpreter`, which evaluates ACL boolean expressions. Note that, this component is used by two other components: the `EC Validator` and the `Evolution Assistant`. When it is invoked by the first one, only constraints involving one version of the architecture are evaluated. However, if it is requested by the second one, all constraints are evaluated. The latest architecture description is checked out with the EC and transformed to the pivot model to be evaluated.

6.2 Example of Use

Let us consider one of the evolution scenarios presented in section 2.2. This concerns the addition of a component representing a notification service. Once this new architecture description introduced to the `Evolution Assistant` component, the latter checks out the EC corresponding to `MACS` from the component `Version Handler` and then displays a warning report. This report notifies the architecture evolver that the Façade pattern does not hold anymore and that the portability property has been affected. The architecture evolver should either modify his new architecture description, or the EC and consequently the NFRs specification.

6.3 Validation of the Approach

We experienced our approach on Alkanet, a GIS developed by our industrial partner. The project cost has been estimated at 2500 men-hours and its maintenance cost at 1400 men-hours. Starting from maintenance logs, we took the software components that have been the most affected by the maintenance. These components perform the conversion of different types of data format to SVG. A team composed of developers, who know the initial NFRs specification, has documented these components (their initial version) by the necessary evolution contracts. After that, starting from this documented version of the components, another group of developers, equivalent to that who has performed the maintenance initially, has performed the same set of evolution scenarios as in logs. They use AURES for validating changes on the components after applying each scenario. We noticed that the maintenance cost has been reduced by 35 %. Indeed, we passed from 600 men-hours estimated for the maintenance of the converter components to 390 men-hours. It is true that the chosen components are the most complex. For components of less complexity, the gain would be undoubtedly less. But, the most complex components have the highest maintenance costs (Lehman's 2nd law of system evolution [10]). This allowed us to extrapolate this result on the whole application without a lot of errors. According to the developers' declarations, evolution contracts helped them to better understand the architecture of the software to evolve. Furthermore, automatic checking of architectural constraints has been of great benefit.

7 Related Work

Capturing and documenting design rationale is a research challenge, with an increased interest in the software engineering community [22]. Within the context of software architecture, Tyree and Akerman in [25] discussed the importance of documenting architecture decisions and making them first-class entities in architecture description. They present a template to describe them at a high-level of abstraction during development. Their paper focuses on the methodological aspect of describing these templates and not on how they can be used when evolving an architecture as in our approach. In the architecture evolution field, Lindvall et al. presented a survey on techniques employed in diagnosing or researching degeneration in software architectures and treating it [8]. Architecture degeneration is seen as the deviation of the actual architecture from the planned one. Many of the technologies presented in this paper focus on recognition of architecture styles and design patterns, and their extraction from code. This helps to identify deviations by comparing architectures and their properties before and after an evolution. The authors also discuss visualization techniques of architectural changes to understand software evolution and thus deduce degenerated portions in the evolved architecture. For treating this degeneration, the authors present a survey of existing refactoring techniques. Our approach allows the degeneration identification by checking formal documents (evolution contracts) on architecture descriptions before and after evolution. It alerts software evolvers about degenerated components in the architecture and the consequences of this degeneration on quality requirements. It then assists them by using quality information to treat this degeneration.

As in software architecture evaluation methods like ATAM or SAAM [3], our approach forces the architect to create architecture documentation. However, while these evaluation methods are applied during design to detect if a given architectural decision is risky or not according to quality requirements; in our approach, we assume that a posteriori all decisions are non-risky and we care about their preservation during evolution. Our approach can be seen as a complementary technique to these methods and is used downstream. In this manner, we may use an evaluation method only during analysis/design stage and avoid its use after each evolution.

In the literature, non-functional properties (which include quality attributes) 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 NFR refinement to obtain a software product which complies to the initial NFRs [13, 4]. In the second approach, the non-functional information is embedded within the software product. It is the case in our approach, where evolution contracts, which embed statements of NFRs, are associated to architectural descriptions. In [6], Franch and Botella propose to formalize non-functional requirement specifications. The statements of these specifications are encapsulated in modules, which are associated to a component definition and to its implementations. They propose also 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. In our work, we focus on the architectural (structural) 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 or component configurations while changes in their approach are made at a fine-grained implementation level.

8 Conclusion & Future Work

In early 1990's, Perry and Wolf presented a model for software architectures. This model represents software architectures in the form of three basic abstractions: *Elements*, *Form* and *Rationale* [17]. *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 [12], there has been a little effort devoted to the last aspect. In this paper, we presented evolution contracts, as a contribution to the description of the last element in Perry's model. This evolution contract leads to make explicit and checkable architectural decisions (Elements and Form in Perry's model). Thus, it is possible to assist architectural evolution activity and prevent the loss of quality attributes (Rationale in Perry's model). It is, as we think best, a good practice for documenting software architectures and design rationale, and thus facilitating software comprehension in maintenance activities.

On the conceptual level, we plan studying: i) reusability, substitution and extension of ECs, and ii) quality quantification by associating metrics to QAs. This helps to better assist the maintenance activity. On the tool level, we project to stabilize the prototype before integrating it in a CASE tool. This would guide, in a continuous way, the system maintainer. We are also studying the feasibility of integrating the tool in a Configuration Management System. We limit our study to CMS dedicated to software architectures, like Mae [18]. Thus we take advantage from the enhanced control version capabilities of these systems.

References

1. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 2nd Edition*. Addison-Wesley, 2003.
2. E. Bruneton, C. Thierry, M. Leclercq, V. Quéma, and S. Jean-Bernard. An open component model and its support in java. In *Proceedings of CBSE'04. Held in conjunction with ICSE'04*, Edinburgh, Scotland, may 2004.
3. P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures, Methods and Case Studies*. Addison-Wesley, 2002.
4. L. M. Cysneiros and J. C. Sampaio do Prado Leite. Nonfunctional requirements: From elicitation to conceptual models. *IEEE TSE*, 30(5):328–350, 2004.
5. L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Professional*, 2(3), 2000.
6. X. Franch and P. Botella. Supporting software maintenance with non-functional information. In *Proceedings of the First IEEE Euromicro Conference on Software Maintenance and Reengineering (CSMR'97)*, pages 10–16, Berlin, Germany, March 1997. IEEE Computer Society.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Sofware*. Addison-Wesley Professional Computing Series. Addison Wesley Longman, Inc., 1995.

8. L. Hochstein and M. Lindvall. Combating architectural degenration: A survey. *Information and Software Technology*, 47(10):693–707, July 2005.
9. ISO. Software engineering - product quality - part 1: Quality model. International Organization for Standardization web site. ISO/IEC 9126-1. http://www.iso.org, 2001.
10. M. M. Lehman and J. F. Ramil. Software evolution in the age of component-based software engineering. *IEE Proceedings - Software*, 147(6):249–255, 2000.
11. S. Loecher and S. Ocke. A Metamodel-Based OCL-Compiler for UML and MOF. In *Proceedings of the workshop on OCL 2.0 - Industry standard or scientific playground?, 6th International Conference on the Unified Modelling Language and its Applications*, volume 154 of ENTCS, October 2003. Elsevier
12. N. Medvidovic and N. R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE TSE*, 26(1):70–93, 2000.
13. J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE TSE*, 18(6):483–497, June 1992.
14. OMG. Corba components, v3.0, adpoted specification, document formal/2002-06-65. Object Management Group Web Site: http://www.omg.org/docs/formal/02-06-65.pdf, June 2002.
15. OMG. Uml 2.0 ocl final adopted specification, document ptc/03-10-14. Object Management Group Web Site: http://www.omg.org/docs/ptc/03-10-14.pdf, 2003.
16. OMG. Uml 2.0 superstructure final adopted specification, document ptc/03-08-02. Object Management Group Web Site: http://www.omg.org/docs/ptc/03-08-02.pdf, 2003.
17. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
18. R. Roshandel, A. van der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae - a system model and environment for managing architectural evolution. *ACM TOSEM*, 11(2):240–276, April 2004.
19. R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices.* SEI Series in Software Engineering. Pearson Education, 2003.
20. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.
21. Sun-Microsystems. Enterprise javabeans(tm) specification, version 2.1. http://java.sun.com/products/ejb, November 2003.
22. A. Tang, M. A. Babar, I. Gorton, and J. Han. A survey of the use and documentation of architecture design rationale. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, pages 89–98, Pittsburgh, Pennsylvania, USA, November 2005, IEEE CS.
23. C. Tibermacine, R. Fleurquin, and S. Sadou. Preserving architectural choices throughout the component-based software development process. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, pages 121–130, Pittsburgh, Pennsylvania, USA, November 2005. IEEE CS.
24. C. Tibermacine, R. Fleurquin, and S. Sadou. Simplifying transformations of architectural constraints. In *Proceedings of ACM SAC (SAC'06), Track on Model Transformation*, Dijon, France, April 2006. ACM Press.
25. J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, March/April 2005.
26. xAcme: Acme Extensions to xArch. School of Computer Science Web Site: http://www-2.cs.cmu.edu/ acme/pub/xAcme/, 2001.