

Updating Software Architectures : a style-based approach

Dalila Tamzalit*, Mourad Oussalah*, Olivier Le Goer* and Abdelhak-Djamel Seriai†

* LINA CNRS FRE 2729, Université de Nantes – 2, rue de la Houssinière, Nantes, F-44000 France

Tel : +33 2 51 12 58 47 Fax : +33 2 51 12 58 12

Email : {dalila.tamzalit, mourad.oussalah, olivier.le-goer}@univ-nantes.fr

†G.I.P, Ecole des Mines de Douai – 941, rue Charles Bourseul BP 10838 59508 Douai Cedex, France

Tel : +33 3 27 71 23 81 Fax : +33 3 27 71 29 17

Email : seriai@ensm-douai.fr

Abstract—Our paper deals with the update of component-based software architectures, i.e. all the modifications which can be performed on the architecture's elements to satisfy various requirements. Current researches on the component-based software engineering propose various mechanisms to adapt, evolve, customize, or reconfigure such architectures. We are convince that these mechanisms, even powerful, still require the architects' expertises. Moreover, it is noticed that these expertises conform to a style, i.e. a way of update which privileges certain concerns rather than others. It is what explains why a solution is selected instead of another, for the same main update issue. Our goal is to define updating styles, to represent and reuse update expertises. We wish to use this work to build an environment based on an updating style library in order to assist architects in modifying their component-based architectures.

Keywords—Style, Update, Component-Based Architecture

I. INTRODUCTION

An intrinsic characteristic of a software is to be necessarily updated in order to satisfy various new requirements. The update means all modifications made to a software system, at the various stages of software lifecycle, especially at the maintenance stage. The update include recurring concepts of the software engineering : evolution, adaptation, customization or reconfiguration. Updating component-based architectures is a major problematic of the component-based software engineering and various approaches were proposed to answer it. These approaches offer mechanisms to carry out what we call *update tasks*. When an update need appears, the execution of the update tasks to answer it requires an architect's expertise. To our knowledge, this update expertise was not captured, represented and re-used like was architectural design expertise by the means of patterns and styles [1], [2], [3], [4]. However, architects face recurring update problems : ie, to extend the functionalities, to improve the performances, to restructure the architecture, etc. Moreover, solutions chosen by the architects to achieve these goals have particularity and can be identified : there is a stylistic way to update. We propose in this paper a new concept, the *updating styles*, which we define as the combination of a proven competence and a stylized solution to an update need. Our style-centered approach aims at

capitalizing stylized expertises to reuse them afterward, in order to assist the architects in updating their architectures, by limiting the efforts, risks and costs.

In this paper, we initially expose the reports which justified this work. Then, we present the context of our work : the component-based architectures and the update of this type of architectures. Next, the updating styles are introduced by a definition and a specification before proposing a meta-model. The whole work presented in this paper is the first step for the build of an environment to assist updating of component-based architectures, named USE (Updating Style Environment).

II. MOTIVATIONS

The idea to capture expertises to build softwares is not new, process patterns[5] do it. We will see further in this paper in section V-A.2 that updating styles have similarities with process patterns but have some different characteristics.

A. Problematic

Our problematic results from several observations of software updating.

a) *Updating is a complex work*: in increasing complex architectures solutions are seldom commonplace : it is necessary to be able to identify what tasks must be carried out and in what order. Moreover, they must be carried out in the respect of description constraints (e.g a connector must connect at least two components) or architectural constraints (e.g pipe&filter architecture)[6]. Without dedicated tools, it is the architect's expertise who determines the success or the failure of updates.

b) *Updating is a recurring work*: some update issues are shared by a great number of architects : interoperability, reconfiguration, upgrade, adjustment to the available resources, etc. Others are specifics for particular projects. In all cases, the same tasks are regularly encountered and necessary.

c) *Updating is a stylized work*: consciously or unconsciously, an architect chooses a solution rather than another to solve its update problem. These different ways of update converge towards the same objective but privilege different concerns : quality, simplicity, speed, costs. Moreover, this

style is often identifiable through the architects' behavior : some privilege the quality by systematically versioning the elements before modifying them, others always choose the fastest solution or the cheapest one.

d) *Updating is a reusable work*: there are recurring needs and there are recurring proven and stylized solutions to answer them. Expertises must be captured and represented to be re-used by architects, by the same way that elements of designs are re-used. That is a means of reducing the risks, the efforts and, well on, the costs.

B. Goals

The preceding reports reveal that the re-use of stylized expertises can be beneficial, like is the re-use of design or code in the objects and components paradigms. Patterns and styles proved their reliability for the software design and we wish to exploit them for the software update. We propose *updating styles*, intended for people charged to update the whole or a part of a component-based software architecture. We indifferently nominate these persons under the term *architects*. For them, the updating styles offer the following advantage :

- 1) To capitalize a stylized expertise : to capture architect's stylized expertise and thus to accelerate future updates by reusing it.
- 2) To propose a stylized expertise : to assist an inexperienced architect by proposing preset stylized update expertises.

Our work consists in defining, specifying, formalizing, and modeling the updating styles and their relationships. This work will make possible to build a library of updating styles, exploited by an environment to guide the updates of component-based architectures.

III. CONTEXT OF WORK : COMPONENT-BASED ARCHITECTURES

We briefly point out, in this section, the bases of the components paradigm and present our context of work.

A. Component paradigm

Component-based approach is regarded as a paradigm which fills the lacks of the object paradigm[7]. Component-based proposals thus appeared to answer requirement engineering issues, specifications, coding and systems modeling. Nowadays, the turning point of the component-oriented development seems to be well accepted. We share this interest and base our work on a clear specified architecture, i.e. where used concepts are clearly defined.

B. Architectural Description Languages

Architectural description languages (ADL) emerge as notation support for the architectures models [8]. There is a large variety of ADLs emerging from various academic and industrial groups. The coverage of the ADLs is broad, each one has its characteristics and aims at precise concerns of architectural design[9] and there is no ADL which is best

appropriate for all the possible purposes, but they agree nevertheless on a common minimal set of component paradigm concepts.

C. Architectural concepts

To update a component-based architecture consists into update its constituent elements : the architectural elements. It is commonly admitted by the research community working on architectural description languages that there are at least four fundamental architectural elements [9] : Component, Connector, Interface, Configuration. For a clear definition of these elements, refer to [10].

D. Reification of architectural elements

We consider that any element which can be modified must be reified, to be able to manage its update. For example, it is the case of the four aforesaid fundamental architectural elements. Hence, they are considered as first-class entities. We currently consider only one configuration at the same time, before extending our work to a set of configurations. At first, that enables us to focus on a given configuration and its concepts.

E. A three-level architecture

In order to ensure this reification, we consider that a component-based software architecture must be described on three abstraction levels (Fig 1) :

- M2 : is the level where all the architectural elements are specified. It constitutes the highest abstraction level called *Meta-Architecture level*.
- M1 : is the *Architecture level* and allows the description of an architecture by using one or more architectural elements of the Meta-Architecture level.
- M0 : lastly, the *Application level* is an instantiation of the Architecture level. It represents an application at the run-time.

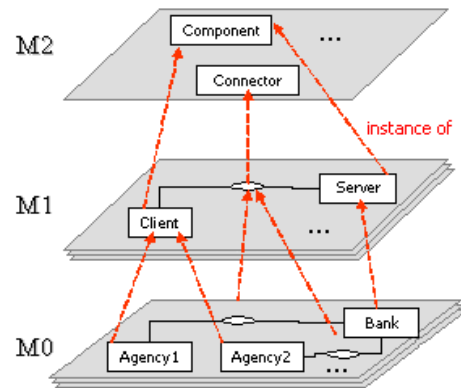


Fig. 1. Abstraction levels of component-based architectures

An architecture is an instance of the meta-level. An application is an instance of an architecture. This hierarchy is

significant because any update operated on a level impacts directly on the lower level¹.

IV. UPDATE OF COMPONENT-BASED ARCHITECTURES

We have just defined what the concept of software component-based architecture covers. It remains to clarify the concept of update. One finds various terms in the literature on the dynamics of software architectures, according to the context and objectives. It is necessary to standardize the terminology to be able to deal with these problems in a complete way : we include all the modifications made to an architecture under the generic term of *update*.

A. Update's meaning

Update thus refers to the changes brought to a component-based architecture. One can update the functionalities according to needs. Update is also to correct bugs and to replace defective or obsolete elements. One can update the configuration for a new environment of deployment, update what exists to be able to integrate most recent software elements or update elements' behavior according to the available resources. So, the updates adjust a software system in order to respect constraints fixed by the designer, the developer, the customer and the context : performances, reliability, variable resources, required functionalities, most recent software elements, etc.

B. Architectural levels concerned with update

Update needs may exist on each abstraction level of a component-based architecture. Indeed, one can want to update the concepts (M2 level), architectural descriptions (M1 level) or applications (M0 level). Nevertheless, handled entities will be of different natures because of their different abstraction level position.

C. Update tasks and mechanisms

To update a software architecture consists to carrying out an organized set of tasks : the update tasks. For instance : add a component, remove a connector, or modify an interface. These tasks are performed according to mechanisms' related to the context : reified elements, abstraction level considered, programming language used, etc. As an example, inheritance and sub-typing mechanisms supported by a majority of ADL, the wrapping (use of composite components), the Meta Object Protocols, the binary component adaptation (BCA), or even the combination of several of these mechanisms. Thus, diversity of these mechanisms and their variable degree of complexity make their classification and identification for reuse purposes difficult. Nevertheless, after a study of [12], [13], [14], [15], [16], [17], [18], [19] carried out in another work witch is out of scope of this paper, we notice that they all have required and provided properties :

- Required properties : opening (black/gray/white box), time (static vs dynamic), representation's formalism (how are the elements reified, including the used language)

- Provided properties (optional) : versionning (modify elements beforehand versioned), impacts (manage the impacts to keep architecture in a coherent state)

We can establish their common features within table I.

| | |
|-------------------|---|
| Name | A mechanism has a name |
| Goal | A mechanism describes its followed approach, the supported and unsupported changes. The support (automated, semi-automated, manual) is mentioned here |
| Inputs | It is the initial situation to execute the mechanism. The mechanism's required properties must be respected |
| Outputs | It is the situation resulting from the execution of the mechanism. The mechanism's provided properties must have been respected |
| Constraints {I/O} | A mechanism requires architectural or/and description constraints and must respect some. |

TABLE I
UPDATE MECHANISMS COMMON FEATURES

D. Update expertise

The update mechanisms are significant and ensure the execution of the involved update tasks. Nevertheless, the success of updating is determined by the way of correctly applying update tasks. This expertise is declined at various abstractions levels. That goes from a very declarative expertise described in natural language, until the very algorithmic and formal one which consists in performing a succession of precise update tasks until the objective is achieved.

V. UPDATING : A STYLE-CENTERED APPROACH

An updating style must capture a declarative expertise to update component-based architectures, i.e. a general competence to use update tasks in a proven way and in a stylistic way. There are often several updating solutions to the same problem : it is as many updating styles that can be captured and reused by the architects.

A. Presentation

Updating styles make possible to the architect to re-use his peers' concentrated expertises which previously faced similar update issues. So, updating styles have significant advantages :

- A style make possible to re-use of the update expertises : current expertises can be applied to other problems.
- By proposing an update expertise, the styles limit the errors in the modifications of the architecture's elements because they profit from proven know-hows.

We give importance to the updating styles definition. We propose the specifications of the updating styles which make possible to draw up their identification, i.e. a uniform manner to describe them. Then, we propose a three-levels model of updating styles, following the example of component-base architecture abstractions levels.

¹The impacts of a level towards its higher level exist : it is the emergence.[11]

1) *Updating styles definition*: we define the updating styles as follows :

An updating style characterizes a family of update processes sharing a recognizable characteristic, all conforming to a particular manner to update architectural elements. It captures a proven and stylized expertise which meets a recurring update need within component-based architectures, according to particular concerns, and consists in a declarative description of the update stages to be followed, that must any person using the style conforms to.

Notice that this definition is close to the process pattern spirit [20] :

A process pattern is a pattern which describes a proven, successful approach and/or series of actions for developing software.

2) *Formal aspect of the updating styles*: in addition to the declarative aspect, the fundamental difference between a style and a pattern is the more formal aspect of the style. In fact we consider that an updating style must be able to apprehend in a complete way update problems while being sufficiently formal to be exploited by tools (to transform it, check it, query it, etc.). Thus, we switch from a «contemplative» use of patterns to a «productive» use of styles.

3) *Updating styles specification*: in order to describe an updating style, we must emphasize features that we consider as essential. We consider that a style is composed of two parts (Fig 2) : a *header* which gives its characteristics and a *competence* which formalizes its expertise.



Fig. 2. The two parts of an updating style

a) *Header*: we thus outlines in the header of a style the five following features (Table II), close to those listed in section IV-C concerning the update mechanisms.

The goal of the style determines its generics degree. The more the style is specialized, the less it is reusable but the more it proposes a precise update solution to the architect. In other words, the use degree of a style is inversely proportional to its re-use degree.

| | |
|-------------------|---|
| Name | endows a style with an identity |
| Goal | description of the update goal |
| Inputs | the initial situation for which the style is appropriate. They are the required architectural elements necessary to perform the expertise |
| Outputs | the situation resulting from the execution of the updating style. They are the modified architectural elements provided by the execution of the expertise |
| Constraints {I/O} | specify architectural or/and description constraints on the inputs and/or the outputs |

TABLE II
UPDATING STYLE'S HEADER

b) *Competence*: a style has a competence, that offers a way to formalize and solve the concerned update problem, in a stylistic way.

- A competence can depend or not on other styles : thus, the competence is respectively non-final or final.
- A competence can be generic or on the contrary, be specialized : competence is then respectively abstract or concrete.

So, a competence – and by misnomer, a style – satisfies one of the characteristics presented in the table III.

| Generics | Competence/Style | Semantic |
|----------|--------------------|--|
| + + | Abstract non-final | Competence is expressed as a sequence of other styles, of which at least one is abstract |
| + | Abstract final | Competence is not expressed |
| - | Concrete non-final | Competence is expressed as a sequence of other concrete styles |
| - - | Concrete final | Competence corresponds to an achievable update task |

TABLE III
THE FOUR UPDATING STYLES CHARACTERISTICS

B. Inter-style relationships

Isolated updating styles have little interest. Their force comes from their number and their relationships. The inter-style relationships are of two different nature :

- *Specialization* : an updating style specializes another style : its goal is more precise. It inherits the header and the competence of the super-style, this mechanism make possible to re-use styles and to create new ones.
- *Dependence* : a style uses a sequence of other styles to express its competence². It is what explains its declarative aspect.

We consider that the updating styles are both connected by specialization and dependence relationships as a graph, but each of the two relations addresses a different concern. This is why we separately analyze the specialization relationships as a specialization tree, and the dependence relationships as

²the style's inputs, outputs and constraints must be compatible with those of the styles used in its competence

a dependence tree.

1) *Styles specialization tree*: represents the specialization of headers. The key idea is to partition the update solutions space (Fig 3), according to a classification based on a specialization of the goals : it is mainly a semantic classification.

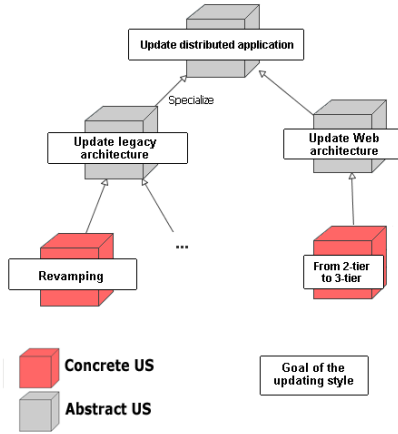


Fig. 3. Example : specialization tree to upgrade distributed applications

Hence, closer the updating styles are to the leafs of the tree, the more they are specialized and more they are close to the root, more they are generic.

2) *Styles dependency tree*: represents the dependence of competences. Non-final updating style provides an update service but requires other ones, and so on. Final updating styles stop these dependences. The figure 4 shows the dependences of a non-final concrete updating style. Notice that with each new dependence, competences get closer to the code level : they are successively refined.

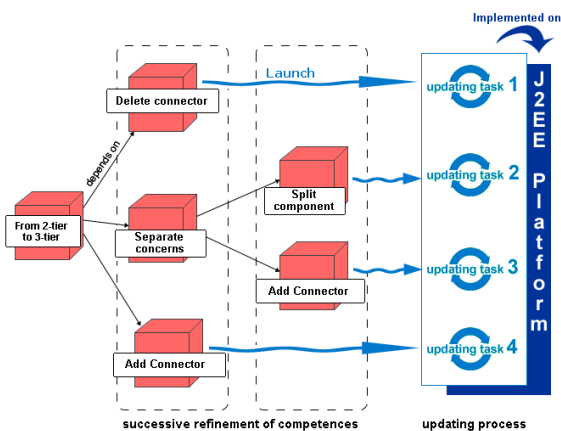


Fig. 4. Example : Dependence tree of an U.S to transform a 2-tier web architecture into a 3-tier one

C. Updating style model

We propose in this section a three abstraction levels model of updating styles as well as the reification of the elements

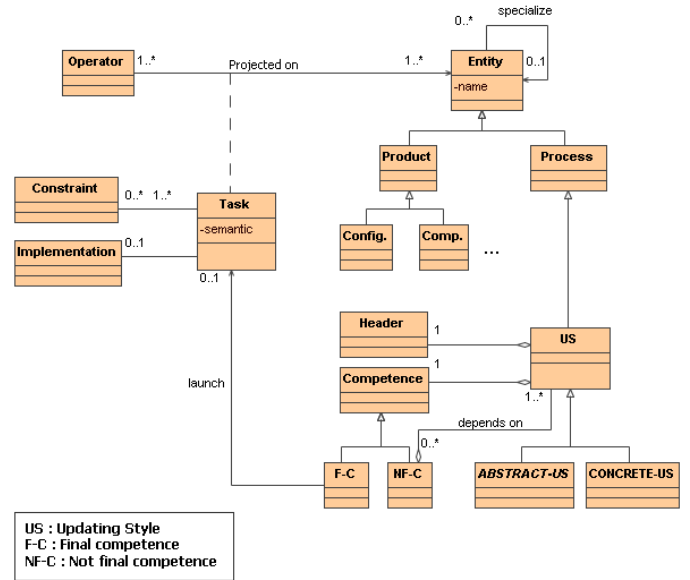


Fig. 5. Updating styles meta-model (UML Class Diagram)

of each level.

1) *A three-level model*: we define a three abstraction levels model for the updating styles, like illustrated within figure 6.

- M2 : meta level, one finds there the updating styles fundamental concepts.
- M1 : style level, contains styles description by using the concepts of the meta level.
- M0 : process level, the styles are instanced as a sequence of update task. Moreover, some tasks are linked to dedicated implementation units and can be executed.

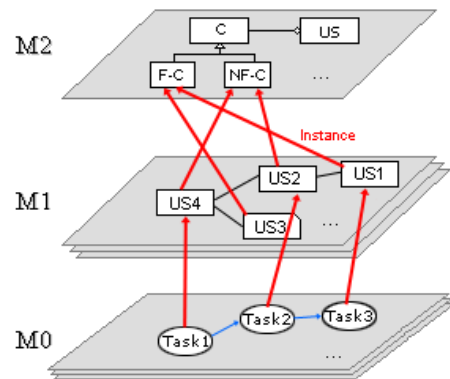


Fig. 6. The abstraction levels of the updating styles model

2) *M2 level*: the updating styles meta-model is currently describe by an UML class diagram (Fig 5). It tries to be as generic as possible. The updating styles, their properties and their characteristics are reified. Moreover, a final concrete updating style (classes CONCRETE-US and F-C), through its achievable characteristic, is associated to an update task

which we regard as the projection of an operator on an entity. Thus, even if in our context the architectural elements are product-typed entities, the meta-model does not exclude update tasks on process-typed entities like methods invocation or work flows. Then, it is considered a priori that any entity can be specialized. That allows the specialization of the updating styles and certain architectural elements like components for instance.

3) *M1 level*: by reifying the updating styles as components, the description of the M1 level corresponds to a component-based architectural description (Fig 7). Hence, our model becomes reflective, thus the updating styles can be updated by updating styles, as well as any other component-based architecture. Table IV specifies the correspondences between our updating styles model and a component-based architecture.

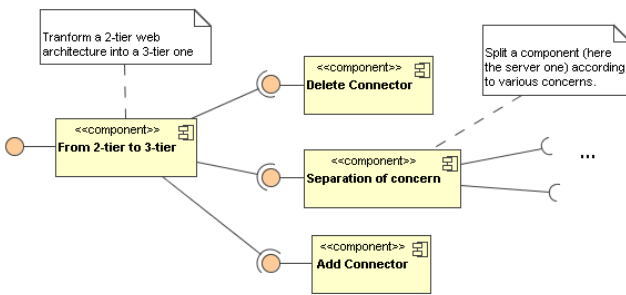


Fig. 7. Example : reification of the Updating styles as components

| UPDATING STYLE | COMPONENT PARADIGM |
|--------------------------|---------------------------------|
| Graph of styles | Configuration |
| Style | Component |
| Provided update service | Component's Provided Interface |
| Required update services | Component's Required Interfaces |
| Name, Goal, constraints | Component's Properties |
| Competence | Component's Behavior |
| Specialization | Inheritance link |
| Dependence | Connectors |

TABLE IV

CORRESPONDENCES BETWEEN UPDATING STYLES AND COMPONENTS

4) *M0 level*: here, the updating styles are instanced from M1 level in keeping with given software architectures, as update processes, i.e sequences of update tasks. Nevertheless, the abstract styles cannot be instanced due to their empty or incomplete competence ; only the concrete updating styles can be. In addition, the updating styles beforehand associated to dedicated implementation units can be executed on a given environment.

VI. USE : AN UPDATING STYLE BASED ENVIRONMENT

The updating styles and their meta-model have been defined, we also aim at proposing an environment to exploit them. This environment is not yet built but we expose here a survey of main features.

A. Presentation

USE is an environment to assist architects to modify their component-based architecture, exploiting a library of updating styles. To ensure an effective use of this library, it is necessary to have a selection mechanism of the styles according to update problems to solve. It is necessary to help the architect by directing his choice toward the most suitable updating style in the most automatic possible way.

The two-steps resolution of the architect's problem consists in :

- 1) The selection of the suitable updating styles : they are *candidates* updating styles. They are proposed to the architect who can select one of them.
- 2) Then the architect can : use the style as a guideline, update it, use it to build new ones, or execute it.

We come back on two mechanisms which appear significant to us : the selection and the execution of the updating styles.

B. Selection of updating styles

The specialization relationships make possible to navigate within the specialization tree. This is significant since we want to be able to select one or more updating styles in an automatic or semi-automatic way starting from the architect's needs. He expresses his need as indicated within table V.

| | |
|-----------------|---|
| Need | describe the architect's update need |
| Inputs | the current situation in which the need appeared. Indicate the architect's architectural elements |
| Constraints I/O | a set of constraints fixed on his architecture that the updating style have to respect |

TABLE V

DESCRIPTION OF AN UPDATE NEED BY THE ARCHITECT

The mechanism is quite the same that a depth-first search in a tree starting from the root. Indeed, the deepest updating styles are the most specialized, thus the most suitable for the architect's needs. The decision criterion is based on the mapping of the architect's needs and the goals of the current updating style, and thus, implicitly, the mapping of the stylistics characteristics. When this mapping is not enough any more to decide the path to borrow, the mapping of the inputs of the user and those of the style (and associated constraints) can make the navigation to continue. Thus, during the crossing of the tree, at each updating style found, a decision is taken (Fig 8) and a branch is chosen : the update solutions space decreases with each step in an automated way to lead to the selection of the best candidates updating styles.

C. Execution of an updating style

Once an updating style has been instanced as a sequence of update tasks according to a given architecture, the architect can execute this sequence on his environment (J2EE, CORBA, etc.) to automatically update his architecture. Such execution is only possible if all the involved updating styles have been

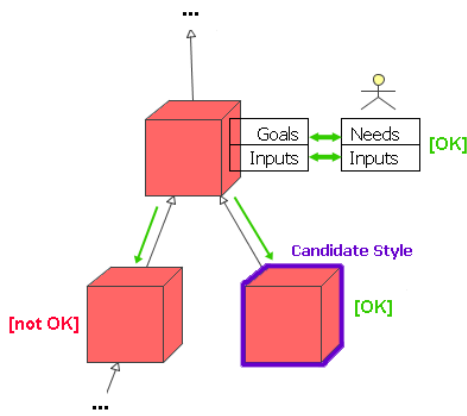


Fig. 8. Navigation to select the best candidates updating styles

associated to ad-hoc implementation units. Furthermore, even if we deal with an automated update process, some tasks will require the architect's intervention in order to fill the lacking informations. As an example, the task "Add a component" require a definition of an external component. After the last task, the architecture is supposed to be correctly updated.

VII. CONCLUSION AND PERSPECTIVES

First of all, the re-use by an architect of his peers' concentrated expertises facing update issues is a beneficial approach. It fully integrates today's software engineering spirit, on both the fields dealt with and the chosen approach. Secondly, we notice that, consciously or unconsciously, architects privilege certain concerns that others and thus update their architecture according to their own style. Hence, certain solution however valid, will be rejected by the architect. The addition of the stylistic notion makes possible to offer update expertises more suitable for architects' waiting.

The updating styles, as defined in this paper, may be used in various fields, and are not limited to the update of component-based architectures. The meta-model is aiming at being as generic as possible to anticipate the future update needs by allowing the update of product-typed and process-typed entities. The updating styles, thanks to their standardized description which is more formal than the patterns one, can be handled by tools. The USE environment that we want to develop will be illustrating this by enabling to select and run automatically (or semi-automatically) updating styles from a library, therefore meeting the needs expressed by the architect. Such an approach enables non-skilled architects to update their architecture accurately and skilled architects to accelerate these updates. In both cases, the updates are made easily, with a lower risk and at the lowest cost.

Before starting to develop USE, the remaining of these work consists in deepening the meta-model's concepts so as to describe quietly the updating styles as components. Furthermore, our goal is to standardize the representation

of the architecture's levels of the updating styles. We have set correspondences in order to describe the M1 level of our model as a component-based architecture, to make it reflective. We now have to describe the M2 level as a component-based architecture. This will allow us to update the key concepts of the updating styles by updating styles. In this case, we will be dealing with updating meta-styles...

REFERENCES

- [1] R. J. J. V. Erich GAMMA, Richard HELM, *Design Patterns : Elements of Reusable Object-Oriented Softwares*. Addison Wesley, 1995.
- [2] H. R. P. S. F. Buschmann, R. Meunier and M. Stal, *Pattern-Oriented Software Architecture : A System of Patterns*. John Wiley and Sons, 1996.
- [3] R. T. Monroe and D. Garlan, "Style based reuse for software architecture," *Proceedings of the 1996 International Conference on Software Reuse*, April 1996.
- [4] D. Garlan, "What is style?" *Proceedings of Dagshtul Workshop on Software Architecture*, February 1995.
- [5] Ambler, *Process Patterns – Building Large-Scale Systems Using Object Technology*. Cambridge University Press/ SIGS Books, 1998.
- [6] 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)*, Pittsburgh, Pennsylvania, USA, November 2005.
- [7] C. Szyperski, *Component Software, Beyond Object-Oriented Programming (Second Edition)*. Addison-Wesley, ACM Press, 2002.
- [8] P. Kogut and P. Clements, "Features of architecture description languages," *Proceedings of the Software technologie Conference, Salt Lake City*, April 1995.
- [9] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on Software Engineering*, Vol. 26, 2000.
- [10] T. K. Adel Smeda and M. Oussalah, "Academic component models," in *Component engineering : Concepts, techniques and tools*. Vuibert Informatique, 2005, pp. 45–86.
- [11] D. Tamzalit and C. Oussalah, "From object evolution to object emergence," in *CIKM '99 : Proceedings of the eighth international conference on Information and knowledge management*. New York, NY, USA : ACM Press, 1999, pp. 514–521.
- [12] A. Ketfi, N. Belkhatir, and P.-Y. Cunin, "Automatic adaptation of component-based software," *ICSSEA*, 2002.
- [13] G. T. Heineman, "Adaptation of software components," *2nd Annual Workshop on Component-Based Software Engineering*, May 17-18 1999.
- [14] R. Keller and U. Holzle, "Binary component adaptation," *Lecture Notes in Computer Science, Vol. 1445*, p. 307-402, 1998.
- [15] J. Bosch, "Superimposition : A component adaptation technique," *Dept. of Computer Science and Business Administration/Blekinge Institute of Technology*, 1997.
- [16] G. Kiczales, J. Lamping, and G. Murphy, "Open implementation design guidelines," in *19th International Conference on Software Engineering*, May 1997.
- [17] Welch, I. S., Stroud, and R. J., "Using metaobject protocols to adapt third-party components," *Work in Progress Paper at Middleware'98*, 1998.
- [18] S. Göbel, "Encapsulation of structural adaptation by composite components," *WOSS'04 Newport Beach, CA10/31*, 11 january 2004.
- [19] R. Roshandel, A. van der Hoek, M. Mikic-Rakic, and N. Medvidovic, "Mae : A system model and environment for managing architectural evolution," *ACM Transactions on Software Engineering and Methodology*, 13(2), pp. 240-276, April 2004.
- [20] Coplien and Schmidt, *Pattern Languages of Program Design*. Addison-Wesley, 1995.