Project no. FP6-028038

Palette

Pedagogically sustained Adaptive LEarning Through the exploitation of Tacit and Explicit knowledge

Instrument: Integrated Project

Thematic Priority: Technology-enhanced learning

D.INF.06 – Template-Driven Transformations

Due date of deliverable: 31 Mars 2008
Actual submission date: 28 April 2008

Start date of project: 1 February 2006 Duration: 36 months

Organisation name of lead contractor for this deliverable: UNIFR

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
R	Public	**PU**

Keyword List: document engineering, structured document, template evolution, transformation collaborative editing
Responsible Partner: Aida Boukottaya, UNIFR

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
1	22/02/2008	Draft	Aida Boukottaya
2	25/03/2008	Draft	Aida Boukottaya Micael Paquier
3	01/04/2008	Draft (sent to evaluators)	Aida Boukottaya
4	15/04/2008	Evaluation	Stéphane Sire, EPFL Annick Rossier, UNIFR , Vincent Quint, INRIA
5	18/04/2008	Final	Aida Boukottaya

Deliverable manager

- Aida Boukottaya, UNIFR

List of Contributors

- Aida Boukottaya, UNIFR
- Micael Paquier, EPFL

List of Evaluators

- Stéphane Sire, EPFL
- Annick Rossier, UNIFR
- Vincent Quint, INRIA

Summary:

In this deliverable we investigate the problem of XTiger templates evolution. We first discuss the different kinds of changes that may be needed to evolve a template based on motivating examples. Starting from a set of template modification primitives, we show how the new template is valid (according to the XTiger language), and how all existing instances could be transformed to conform to the modified template. Finally, we describe the current prototype user interface.

1 Introduction 4
1.1 Motivating Examples of Template evolution 4
1.2 System requirements 5
1.3 Related work 6
1.4 Outline 7

2 Taxonomy and Semantics of Evolution Primitives 7
2.1 XTiger template representation 7
2.2 Evolution primitives: 9
2.2.1 Types change primitives 9
2.2.2 Elements change primitives 11
2.2.3 Making the motivating template example evolve: 11
2.3 High level evolution primitives: 12

3 Impact on validity and transformation generation 13

4 Template evolution prototype user interface 14
4.1 The overview Window: 14
4.2 Evolve a template 15
4.3 Transform instances 16

5 Conclusions and future work 17

6 Bibliography: 17

7 Appendices 18
7.1 The XTiger code of the motivating example 18
7.2 Primitives application Results 20

1 Introduction

In the context of WP2, XTiger templates have been introduced in Amaya to allow CoP members to specify the type of documents they want to produce [D.INF.01]. A template defines for example which elements are permissible in a document, the order in which such elements must appear and how they are nested to form a hierarchical structure. In a community of practice context, templates continuously evolve to reflect a change in the practices, to adhere to new users' requirements, to correct initial design errors, to allow expansion of the template scope over time or to simply allow for incremental maintenance. However, templates updates have a major consequence: documents being valid for the original template are no more guaranteed to meet the constraints described by the evolved template. These documents should be *adapted*, *restructured* and *revalidated* against the new template.
The manual restructuring and validation is difficult, time consuming (especially when it concerns a large number of documents), and requires generally specific skills. Moreover, the observation of CoP members performing manual adaptation shows that this operation frequently results in introducing errors and inconsistencies. Thus, approaches for automatically adapting documents to the new template are needed to maintain documents validity. The validity is a crucial issue since the template is often relevantly exploited in several applications such as querying, transforming and document retrieval.

In this deliverable, we address the problem of XTiger templates evolution. We first present a set of atomic evolution primitives to be applied to the basic components of a template. We show that the proposed primitives preserve the consistency of the template (i.e. ensured to transform a consistent template into a consistent template according to the XTiger specifications[1]). These primitives are made available to the user through a graphical interface that we are currently finalizing. Moreover, we propose high level primitives in order to express more complex updates. A high level primitive is represented as a sequence of atomic primitives that can be executed as a single operation. These primitives have been introduced to facilitate the template evolution task for CoP members and to describe more common sequence of atomic primitives. Finally, we show how we produce automatically valid instances against the evolved template.

The major goal of the proposed approach is to make the task of template evolution easy for CoP members. This is done by (1) enable them to perform evolution operations without knowing the XTiger syntax and (2) ensure the validity of documents in an automatic manner.

1.1 Motivating Examples of Template evolution

Here we present motivating examples that show how changes in templates lead to various data management issues that must be addressed. The example has been the basis of the discussion with Did@ctic CoP members. Figure 1 depicts an example[2] of a template that describes the bibliography used by a CoP group in order to support their activities. Three categories of bibliographic references are described: Articles, Journals, and Books.

Example 1: Let us consider the template change where the definition of the element "Journal", which must have an "editor" subelement, is relaxed such that it is optional to have the editor subelement. For such template change, we would need to verify that (1) the suggested change leads to a new legal template conforming to the XTiger specifications and (2) the corresponding changes are propagated to the existing instances to conform the new template definition. A single occurrence of the "editor" subelement in the instances would still conform to the new template where this subelement is optional. Therefore this template modification requires no changes to the underlying instances.

[1] http://www.w3.org/Amaya/Templates/XTiger-spec.html
[2] The sample template is used as a running example in the remainder of this deliverable. Its XTiger description is described in the appendices.

Example 2: Let us now consider another template change where the definition of an author name is split into two subelements First Name and Last Name. Performing the suggested change leads to the invalidity of the related instances. To remain valid according to the new template, each author name instance (in each article and each book) should be removed and two subelements First Name and Last Name should be created, paying attention to keeping their content corresponding to the original author name. Manually modifying the instances will be time consuming. Moreover, if the user is not familiar with the XTiger language (which is the case of the majority of CoP members) the task will be infeasible.

Example 3: Let us consider another template change where the bibliographic references should be reordered. For examples, books should appear before articles. Such change seems very easy, however this alter the instances validity. The information should be reordered in all available instances. Manual execution is also in this case time consuming and error prone.

Example 4: After the use of the template and its related instances, the CoP group notices that it is easier and more practical for them to classify articles and books by authors rather than by categories. This change is complex to perform: (1) in term of modifications to the template code and (2) modifications of the available instances to adapt them (this requires the complete restructuring of each instance, the elimination of duplicates (each author must appear only once)) which is again a time consuming process. This kind of change is not elementary (involves a lot of operations and modifications). It is equivalent to the creation of another template (where publications are classified by author) and the use of the automatic matching module to perform the transformation (see the example called library available at http://docreuse.epfl.ch:8443/). However, this kind of high level[3] evolution primitives should also be introduced in the Template evolution prototype to answer specific CoP needs and avoid the execution of a lot of evolution primitives. More examples of high level evolution primitives will be given in section 2.3.

1.2 System requirements

As we can notice, the manual update of a template and the revalidation of the related instances require a lot of effort and thus is unacceptable for applications and environments where the information sources frequently change. Based on the previous examples as well as discussions with CoP members (Did@ctic), we deduce a set of requirements summarized in the following:

- The system should propose a taxonomy of atomic template evolution primitives that provides a system independent way to specify changes to XTiger templates and answer CoP members' need,
- The proposed evolution primitives should be easy to extend. High level primitives could be defined in order to answer specific CoP needs,
- Two forms of system integrity should be ensured during the evolution process: *legal* templates (respecting XTiger specifications) and *valid instances*.
- The proposed prototype should provide a user interface enabling CoP members performing template evolution. XTiger code changes as well as transformation scripts should be transparent and automatically done,
- An efficient memory of templates evolution should be built in order to keep track of the evolution process and to enable CoP members to perform transformation whenever they need,
- The developed prototype should be compatible with other DocReuse modules (same search facilities, same repository, etc.,) and other PALETTE services (according to the scenarios of WP5 task 4).

[3] A High level evolution primitive is the execution of a sequence of atomic primitives in a single update in order to express in a more compact way common evolution need of a CoP.

Since an XTiger template is a combination of structural information (expressed using the XTiger statements) and target language elements (XHTML), the evolution prototype essentially deals with structural evolution and not with rendering issues.

university_name
address
Articles
cover Title : title
Author
Name : name
City : city
State : state
Zip : 0000
Journals
cover Name : journal name
Editor : editor name
Books
cover Title : title
Price : 00.00 Frs
Publisher : publisher
Author
Name : name
City : city
State : state
Zip : 0000

Figure 1: A motivating example of a template

1.3 Related work

The need for schema evolution is not a new problem and much effort has been done toward automating such process. Many traditional database projects [Bretl 89], [Zicari 91], [Lerner 96], [Claypool 98] have focused on the schema evolution issues, where the main goal is to develop mechanisms to change not only the schema but also the underlying objects.
More recently, several works have been dedicated to XML structure/data evolution. XML schema evolution has been investigated for schemas expressed by DTDs in [Kramer 01], where a set of DTDs evolution operations have been proposed and their semantics have been discussed in detail. Issues related to the impact of such operations on existing instances have not been addressed; more focus has been given to prove the completeness and the soundness of the proposed change taxonomy. DTD evolution has also been investigated in [Bertino 02] where authors focus on the dynamic adaptation of

DTDs based on the structure of most existing instances. This is done using structure mining techniques.
More recent work focused on the evolution of XML schemas[4]. Authors in [Guerrini 06] proposed a set of evolution primitives dealing with more specific XML schema features (Typing, Type restriction/extension, etc.). Moreover, authors investigate how to minimize instances revalidation, that is, detecting the document parts potentially invalidated by the schema changes that should be revalidated. The idea to provide high level primitives is also exposed but no explicit high level primitives are given. The problem of instances revalidation has been investigated in [Raghavachari 04]. In our work, we adopt the same model as in [Guerrini 06] in order to represent XTiger statements and provide a theoretical way to describe evolution primitives. However, the latter model as well as the evolution primitives have been modified in order to fit the specification of XTiger. In addition to this, the proposed primitives have been negotiated with CoP members to answer their specific needs. Authors in [Guerrini 06] focus on detecting the document parts potentially invalidated by the schema changes. Revalidation and transformation issues are not addressed. In our work, automatic transformation generation is of major concern. The basic idea is to keep track of the updates made to the template in a *mapping* file, and to identify the portions of the template that require a revalidation because of these updates. From the mapping file, a transformation script is generated automatically and the document portions affected by those updates can then be identified and adapted in an automatic manner. For this, we reuse and adapt our work on schema matching and automatic transformation generation as described in D.INF.01 and demonstrated at http://docreuse.epfl.ch:8443/.

1.4 Outline

The remainder of this paper proceeds as follows. Section 2 provides taxonomy and semantics of proposed evolution primitives. In Section 3, we study the impact of such primitives on the validity and we expose techniques used to generate transformation scripts. Section 4 reviews our prototype user interface. In Section 5 we present our conclusions and future directions.

2 Taxonomy and Semantics of Evolution Primitives

2.1 XTiger template representation

By analogy to XML schema language and programming language, we distinguish between *types* and *elements*. Elements represent *the content* (what the end-user sees in the instances). Types are used to define *the structure and the constraints* of these elements. In order to represent the XTiger templates, we adapt the XML Schema model proposed by authors in [Guerrini 06].
Like in a schema language, types in XTiger are used to define pieces of structure that may occur at several places in a template or in several templates. Several types are available: basic types, union types and components. XTiger offers three basic types (number, boolean, and string). XTiger's xt:component is a constructor that creates a new type (a component[5]) containing other constructors both from the XTiger language and from the target language. XTiger also offers the possibility to define a new type (a union type) as the union of other types using the xt:union constructor.
Moreover, a template contains the skeleton of a target language document and some XTiger statements that are used to generate instances. The latter (skeleton and statements) is called the *template body*. Elements belong to the template body and represent the content that may appear in a given instance. Each element (represented by XTiger statements) is associated to a type. Such type is either explicitly named (can be re-used in the definition of other types/elements by referencing to its name) or embedded in the element description (this notion is equivalent to the notion of anonymous types in XML schemas). Type declaration (component and union) in XTiger is global (meaning that once a type is declared, its name is unique and can be used wherever in the template). Elements are local in the sense that they are meaningful in their structural context (e.g. the element Title is local to the definition of a book or an article).

[4] W3C XML schema. Specification available at http://www.w3.org/XML/Schema
[5] This notion is similar to Complex Types in XML Schema

Table 1 presents the representation of the template described in section 1.1. An example[6] of element is *Article* which is the element that represents the structure of an article. Element Article is composed of several other elements; this composition is described within the type *Article-Type*.
The notion of typing in XTiger has been discussed with CoP members. They advocate the use of explicit typing to enhance their reuse (this is done in the XTiger syntax by the declaration of components in the Head element and their latter instantiation using the xt:use constructor). However, since CoP members are not familiar with XTiger language, the typing is completely transparent for them and presented in a graphical manner through the rectangle metaphor as in Amaya and DocReuse modules (see examples of templates at http://www.w3.org/Amaya/Templates/Overview.html).
The template evolution prototype processes the templates in order to make the notion of typing explicit to end-users. In case of inline declaration from the original template, the system will create types in the Head to enable their reuse when they evolve.

Elements	Types	Details
University-name	String	
Address	String	
Articles	repeat — Article (Anonymous type)	Article → Article-Type
Journals	repeat — Journal (Anonymous type)	Journal → Journal-Type
Books	repeat — Book (Anonymous type)	Book → Book-Type
Article	Article-Type: Title, Author	Title → String Author → Author-Type
Journal	Journal-Type: Name, Editor	Name → String Editor → String
Book	Book-Type: Title, Price, Publisher, Author	Title → String Price → Number Publisher → String Author → Author-Type
Author	Author-Type: Name, Address	Name → String Address → Address-Type
Address	Address-Type: City, State, Zip	City → String State → String Zip → Number

Table 1: XTiger template representation

[6] Elements and types in our motivating example have the same name for the clarity of the example.

2.2 Atomic evolution primitives:

As in [Guerrini 06], we consider three categories of atomic evolution primitives: *insertion*, *modification*, and *deletion*. These primitives are applied to types and elements (as defined in the previous section). Moreover, modifications can be further classified in three sub-categories: structural, re-labelling, and migration modifications. Structural modifications allow to modify the type of an element and its constraints (e.g,. cardinality constraints). Re-labelling modifications allow to change the name of an element/type. Migration modifications cover two cases, first moving a subelement from an element to another one and second transforming a local type/element to a global type/element (and viceversa). Since all types are global and elements are local, we limit ourselves to the first case. Table 2 reports the evolution primitives relying on the proposed classification. In what follows, we discuss in more details each primitive. In order to ensure that the modified template remains well-formed and valid according to the XTiger specifications, applicability conditions are enforced on each primitive.

	Insertion	**Modification**	**Deletion**
Types	Insert-Component Insert-UnionType Duplicate-Type	Insert-Subelement Rename-Subelement Change-SubelementType Change-SubelementCardinality Change-Subelement-position Remove-Subelement Modify-UnionType Rename-Type	Remove-Type
Element	Insert-Element Duplicate-Element	Rename-Element Change-Type Change-Cardinality	Remove-Element

Table 2: Classification of template evolution primitives

2.2.1 Types change primitives

The next three primitives involves types and do not require existing instances to be updated, because they just declare new types but do not change anything that was already in the template. These primitives are examples of validity preserving primitives as described in section 3.

Insert-Component

The primitive can be applied provided that type name uniqueness is not violated. Moreover, the primitive takes as a parameter a component type structure: a set of subelements and the relationship among subelements and their type (either a basic type or a type already defined in the template or a target language type).

Insert-UnionType

The primitive can be applied provided that type name uniqueness is not violated. Moreover, the primitive takes as a parameter the type definition of a union type i.e,. several types, each of which being a basic type, a target language type, or a constructed type (component or other union). All the member type names in the list should be distinct and already declared before.

Duplicate-Type

This primitive can be applied for the duplication of a given type (union or component). It takes as parameter the name of the type to duplicate and generates a new type having the same type definition but a new name (in order to respect the type name uniqueness condition). This primitive is used in the case where a type already used in the declaration of given elements should be slightly modified for

another element. The primitive is applied provided that the original type is already defined. This notion is useful since type inheritance is not considered in the XTiger language.

The following primitives concern the modification of a given type. Users must be aware that modifying a type will modify the content of all elements using this type. For a component, we could modify its structure by inserting a new subelement, changing the position of existing subelements, changing the cardinality of its sub-elements, rename its sub-elements, and changing their types. Modifying a union type meaning either inserting or removing the union members. Renaming a type and removing a type are applied to both components and unions. Specific primitives for manipulating a subelement in a given component are provided. These primitives share the same semantics as primitives for manipulating elements in the template body (section 2.2.2). However, we prefer to differentiate them since the end-user could define a type independently of the fact that elements using this type exist or not in the body. Moreover, this choice eliminates any ambiguity for the end-users.

Insert-Subelement

A subelement insertion allows extending the information of a given structure type definition. The primitive takes as parameters the name of the subelement to be inserted and its type (that must be a type already defined in the template, a basic type or a target language type), and the position at which the subelement must be inserted in the type structure. The type to which the subelement is added must be an XTiger component and must not contain subelements with the same name.

Change-Subelement-position

This primitive changes the position of a subelement in a component declaration. The component as well as the subelement must be already defined. The primitive takes as parameter the subelement and its new position.

Rename-Subelement

This primitive renames a given subelement in a component declaration. The element as well as the component must already be defined.

Change-Subelement-Type

This primitive changes the subelement type in a given component declaration unless the type is already defined or is a basic type or a type from the target language.

Change-SubelementCardinality

In the XTiger language, cardinality constraints are described in different manner:

- Using the repeat constructor both on elements and on components (to generate a repeated structure). In this case, this primitive takes as parameter the element or the structure that we want to make repeatable (these elements and structures must be already defined) as well as the minOccurs (minimum number of times the component must be repeated. If this attribute is absent, the minimum is 0) and the maxOccurs (maximum number of times the component may be repeated. If this is absent, it means that no upper bound is considered).
- Using the optional constructor. This is equivalent to an `xt:repeat` with `maxOccurs="1"` and `minOccurs="0"`.

This primitive is applied on subelements of a given component. It takes as parameter the subelement to make repeatable or optional as well as minOccurs and maxOccurs values. The subelement and the component must be already defined. Existing cardinality constraints could also be modified and/or removed using this primitive.

Remove-Subelement

The remove-subelement operation removes a sub-element from a component. The removal implies that the sub-element will be removed from all the other elements having this component as type.

Modify-UnionType
For union types there are two kinds of modification: insertion of a member type, removal of a member type. In the case of a removal, the position of the member type to be eliminated or its name must be specified.

Rename-Type
This primitive renames a given type (component or union). The type must already be defined and there must not be any other type with the new proposed name.

Remove-Type
This primitive is applied for the deletion of both union types and components. The primitive can be applied provided that no elements exist of the type to be deleted.

2.2.2 Elements change primitives

These primitives are applied to elements (in the template body) meaning to the content of instances.

Insert-Element
The primitive takes as parameters the name of the element to be inserted, its position and its type (that must be already defined or be a basic type or a target language type).

Change-Cardinality
This primitive takes as parameter the element to be made repeatable or optional. The element must be already defined. Existing cardinality constraints can also be modified and/or removed using this primitive.

Change-Element-position
This primitive changes the position of an element in the body of a template. The element must be already defined.

Rename-Element
This primitive renames a given element. The element must already be defined.

Change-Element-Type
changes the element type unless the type is already defined or is a basic type or a type from the target language.

Remove-Element
The remove element operation aremoves an element.

Duplicate-Element
This primitive creates another element which is the copy of the original element.

Insert-freeContentArea
The xt: use element puts strong constraints on the structure and/or content of a part of a document. It is sometimes useful to have more flexibility. That is the role of the `bag` element. It indicates that any number of elements may appear at that position in an instance document, and it specifies the allowed types for these elements. This primitive inserts a bag element, as well as bag content (if any) should be precised.

2.2.3 Making the motivating template example evolve

Let us consider the template in figure 1. Suppose that we want to make the template evolve in the following way: Element “address” becomes optional. A new element “affiliation” of type string is added to “Author-Type”. The “price” of a book is deleted. Element “Author” is made repeatable for an

article to cover the case where a given article is written by more than one author. To make these changes, we use the following primitives:

- The Change-Cardinality primitive applied to the element address,

- In order to insert the affiliation information related to a given author, we need to modify the Author structure type definition described through “Author-Type” component. The Insert-Subelement primitive is thus applied to Author-Type. The modification will also affect the authors of books since they share the same type,

- Since “price” is declared in the definition of the component “Book-Type”, the Remove-Subelement primitive is applied to the subelement “price” of the component “Book-Type”,

- In order to make the authors repeatable in the article element, the Change-SubelementCardinality should be applied to the subelement “Author” in the “Article-Type”.

The code of the template is then modified as detailed in appendix B.
It is important to say that the application of these primitives and the code generated are completely transparent to the end-user who will manipulate the elements/types through a graphical user interface as explained in section 4. The XTiger code is generated automatically by the system. In fact, for each primitive change, an XTiger code is predefined and automatically written to generate the modified template. Since this is done by the system, it is done in such a way to always ensure the consistency of the produced code with the XTiger specifications.

2.3 High level evolution primitives

Primitives mentioned in the previous section can be composed in high level primitives in order to express more complex updates. Their applications allow to perform sequences of atomic primitives as a single update. These higher-level primitives have been introduced to facilitate the template evolution task for CoP members. In fact, they allow to group and reuse a set of common manipulations. From the discussions with Did@ctic CoP members and from an analysis of their templates, we define a set of high level primitives to be implemented. The set is not definitive and will be extended/restricted through the trials made within the CoPs in the context of scenarios implementation:

- *Aggregator primitives*: they mainly include primitives for inserting, moving, changing whole substructures rather than a single element at a time.

- *Component creation from existing elements* in the body template: this is useful when the user defines a set of elements and then decides to create a type and to reuse this structure definition. It will be possible to select these elements and to create a new component rather than defining components from scratch.

- *Merge/split primitives*: Elements values are not always represented at the same level of atomicity. For example the author name in the template of figure 1 is represented using an element of type string. However, for a given reason a new design choice could be to separate the author name into a First-Name and a Last-name. Another example could be the address description. While in one template the address is separated in Street, City and Zip values, they are concatenated together in an Address element in another template. For such cases, instead of removing the elements and inserting new ones without any guarantee that the content will be preserved, we define two operations Merge and Split to perform this task and ensure that the content will be preserved.

- *Group by primitive*: We noticed that sometimes the content of a template may need to be restructured according to a certain classification. A Group By primitive (similar to the Group

By statement in query languages) is then introduced. This primitive answers the example 4 from the motivating examples of section 1.1.

3 Impact on validity and transformation generation

When a given template evolves, documents are no longer ensured to be valid for the new template and, as explained in the introduction should be revalidated in order to meet the new constraints. However, taking into account that updates usually affect only few template elements/types and thus not necessarily compromise the whole document validity, the revalidation of the whole instance is not necessary. As explained in [Guerrini 06] validity checks can be restricted to the needed elements/types and thus validation process could be considerably simplified. For this, several techniques will be used:

- Detect the validity preserving primitives (sequence of primitives) in order to avoid the revalidation process. In fact, some primitive preserve the validity of the instances. Inserting a new type or new optional element cannot comprise the validity of the already existing instances. Type renaming or duplication does not affect the validity of the instances but just the structural organization of the template.

- Build a *mapping file* that records the elements affected by the changes. It describes the original elements/types and the modified or new ones. The mapping file is a structured document that conforms to a predefined structure definition. Structuring the mapping file is essential for two reasons. First, it is easier to manipulate a structured mapping file either to modify it or to automatically generate transformation scripts from it. Second, structuring the mapping file in a standardized form (i.e., system independent) greatly increases its reusability.

- The next step is concerned with data translation, i.e., implementing the specification given by the mapping file. The result of this step is a transformation script expressed in the XSLT language. For each mapping element, the generated transformation rule has two roles: retrieve and insert. First, it retrieves data instances from the source by performing the required operations. Second, it must correctly insert elements at their actual places in the new template, in order to generate a valid document. To ensure the validity of the produced document, the XSLT generator traverses both the target template and the mapping file and generates transformation rules for each target element. For this, we will adapt the already developed XSLT generator module (see D.INF.01) to be used with the new mapping structure.

- Since we do not assume that the original and evolved templates represent the same data with the same constraints, there may be data in the evolved templates that is not represented in the original ones. In some cases, user interaction is required to produce new values. Additional values (default values) can be also added automatically in order to ensure the consistency of the transformed documents.

This part (check validity impact, generate mapping file and transformation generation) is not finalized and is still under development. More results and details will be given in next internal reports and deliverables.

4 Template evolution prototype user interface

This section describes the current state of the proposed user interface to validate the issues discussed in this deliverable. This interface will also evolve according to CoP members' feedback as well as ergonomic analysis as it was planned in the PALETTE project. The prototype is developed as a Web application.
When the user launches the application, an overview window appears (the role of this window is to build a visual efficient memory of the template evolution as well as instances transformation). The window consists in two parts (Figure 2).

4.1 The overview Window

a. The top section shows the templates that are available on the server and that may be evolved. The columns indicate the history of the evolution of the templates, i.e. the evolved templates of the selected template appear in another column on the right, like the column view in Mac OS X file browser. If the user selects an evolved template from this new column, a third column will show which templates (if any) have been evolved from this one and so on. The last column on the right is the inspector, giving a few more information on the currently selected template and displaying the button to click in order to proceed the evolution process.

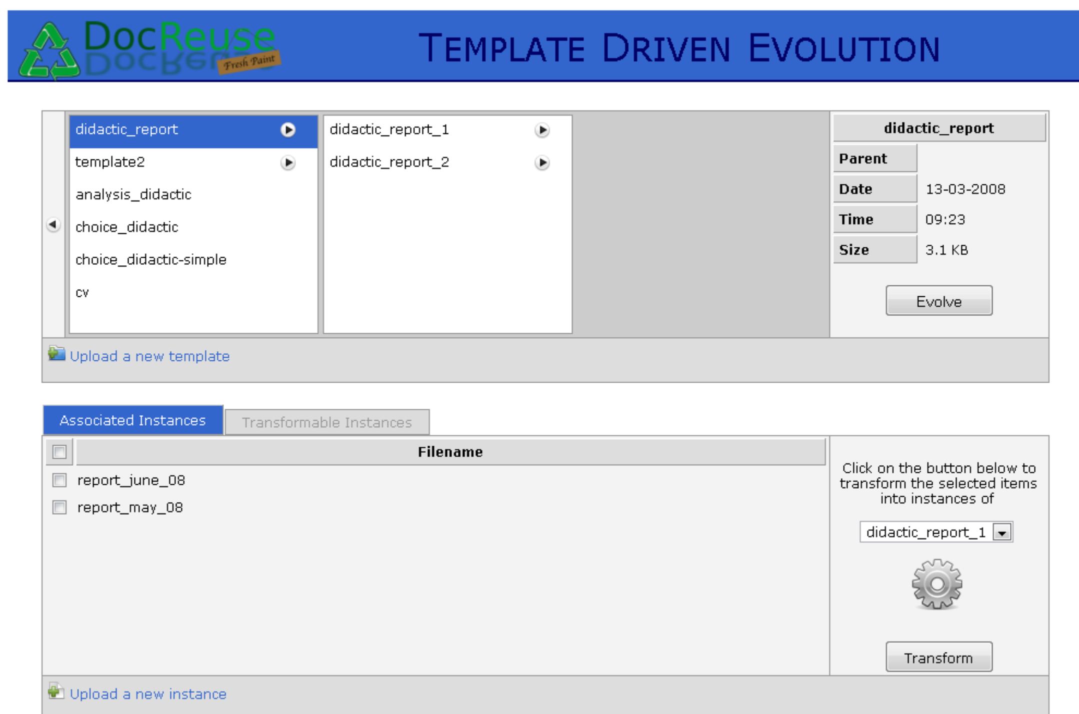

Figure 2 : Template evolution user interface

b. The bottom of the window shows the instance documents that are associated to the currently selected template. The user may select which ones he/she would like to transform, according to an evolved template that has been obtained from the currently selected one. In a second tab, the system also visualises the potentially transformable instances which are the instances related to parent templates, i.e. templates from which the currently selected template has been obtained.

If the user wants to add a new template, the action is simply performed through the upload screen that is displayed when the "Upload a new template" link is clicked (Figure 3). The user may then browse his/her files to select the template to upload. Once done, the new template is placed on the first column, i.e. it is considered as an initial template without any evolution.

Figure 3 : Upload template form

4.2 Evolve a template

a. Once the user has found the template he/she would like to modify, he/she has to click on the button "Evolve" that is displayed in the inspector column as already explained above. The evolution workspace is then displayed (Figure 4), showing two distinct parts. The left column shows the operations it is possible to perform on the selected elements/types of the template, while the rest of the space is used to display the template structure.

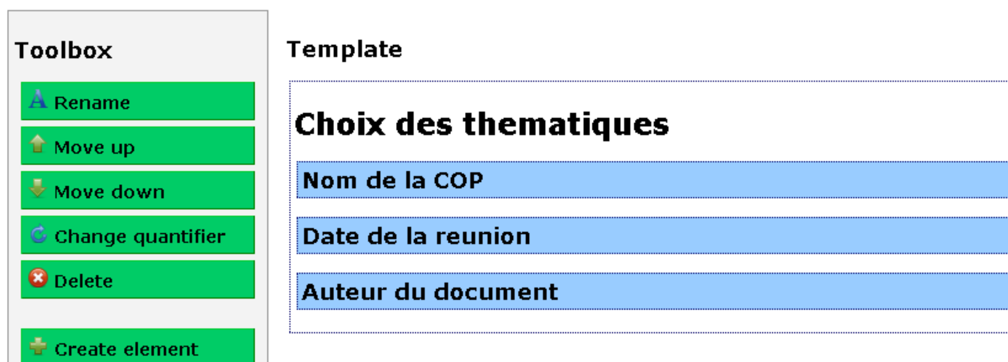

Figure 4 : Template evolution Workspace

b. For example, the user might wish to rename one of the three elements (colored in blue) of this simplified template. To perform this, the user has to select which element he/she would like to rename and then click on the green button corresponding to the renaming operation. An editing box appears where the user can modify the desired properties of the element.

Figure 5 : Renaming operation

c. Once the user is satisfied with the new template, he/she may save it and exit to go back to the initial screen, where the evolved template will appear as an evolution of the initial template.

4.3 Transform instances

Until now, we focused on the templates themselves. We will now review the simple mechanism that allows us to update the instances that are associated to an initial template. As previously mentioned, the first view also displays the instances that are related to the selected template. This view is constituted of two tabs, each one allowing us to transform these instances.

a. The first tab displays the instances that are associated to the initial template, which is currently selected. The user then selects the ones he/she would like to update (or transform), and the target template (which should be obtained by evolving the initial one) and finally clicks on the "Transform" button.

Figure 6 : Instances associated to selected template

b. The second tab shows which instances can be transformed to produce new ones that conform to the selected template. So in this case, the user has first to select the newly evolved template and then, the application displays the instances of the initial template (or/and its ancestors). The same selection process has to be done to finally transform them into instances of the selected template.

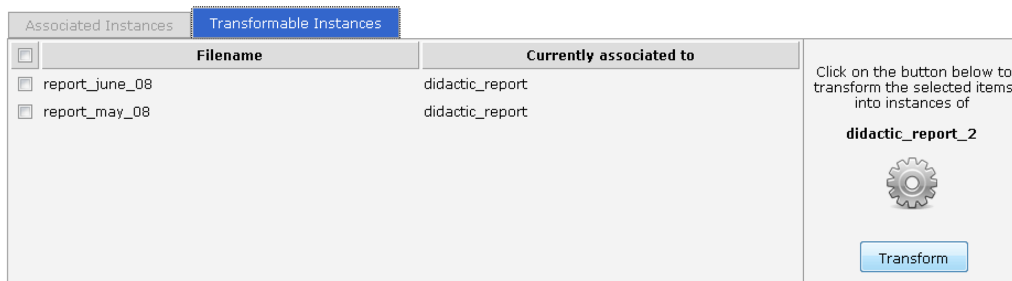

Figure 7 : Potentially transformable instances

5 Conclusions and future work

In this deliverable we have investigated the problem of XTiger templates evolution. This was done to answer an explicit need expressed by CoP members. For this, we proposed a set of atomic evolution primitives as well as some more high level evolution primitives used to make a sequence of atomic primitives usable and reusable in a more compact way.

We are extending this work in several directions. First of all, we are finalizing the user interface described in this deliverable according to CoP members' feedback. Second, the proposed set of evolution primitives will be tested through case studies and may be extended or modified. Third, the revalidation approach as well as the automatic generation of transformation scripts is being implemented and will be tested and evaluated over real templates collections. Potential user intervention in the revalidation process is also under study. Finally, we are also interested in making uniform the DocReuse modules (available at http://docreuse.epfl.ch/release.html) through several actions: (1) making uniform the layout and common functionalities of their user interfaces, (2) offering a common repository of templates, common search functionalities and common access policies. Issues regarding interoperability with other PALETTE services are also investigated according to WP5 scenarios.

6 Bibliography

[Bretl 89] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press, 1989.

[Zicari 91] R. Zicari. A Framework for O2 Schema Updates. In *7th IEEE Int. Conf. on Data Engineering*, pages 146–182, April 1991.

[Lerner 96] B.S. Lerner. A Model for Compound Type Changes Encountered in Schema Evolution. Technical Report UM-CS-96-044, University of Massachusetts, Amherst, Computer Science Department, 1996.

[Claypool 98] K.T. Claypool, J. Jin, and E.A. Rundensteiner. OQL SERF: An ODMG Implementation of the Template-Based Schema Evolution Framework. In *Centre for Advanced Studies Conference*, pages 108–122, November 1998.

[Kramer 01] D. K. Kramer and E. A. Rundensteiner. Xem: XML Evolution Management. *RIDE-DM*, 103–110, 2001.

[Bertino 02] E. Bertino, et al. Evolving a Set of DTDs according to a Dynamic Set of XML Documents. *EDBT Workshops*, LNCS 2490, 45–66, 2002.

[Guerrini 06] G. Guerrini, M. Mesiti, and D. Rossi. XML Schema Evolution, TR Universit'a di Genova, 2006.

[Raghavachari 04] M. Raghavachari and O. Shmueli. Efficient Schema-Based Revalidation of XML. *EDBT*, 639–657, 2004.

7 Appendices

7.1 The XTiger code of the motivating example

```
<?xml version="1.0" encoding="UTF-8"?>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml"
xmlns:xt="http://wam.inrialpes.fr/xtiger" xml:lang="en">
<head>
  <title>University</title>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
  <meta name="description" content="university librairy"/>
  <meta name="keywords" content="librairy university"/>
  <meta name="template_class" content="amaya-cv-en-0.1"/>

    <style type="text/css">...</style>
      <xt:head version="0.8" templateVersion="1.0">

            <xt:component name="address-type">
                  <div class="address">
                        <div class="city">
                              City : <xt:use types="string"
label="city">city</xt:use>
                        </div>
                        <div class="state">
                              State : <xt:use types="string"
label="state">state</xt:use>
                        </div>
                        <div class="zip">
                              Zip : <xt:use types="number"
label="zip">0000</xt:use>
                        </div>
                  </div>
            </xt:component>

            <xt:component name="author-type">
                  <h3>Author</h3>
                  <div class="name">
                        Name : <xt:use types="string"
label="author_name">name</xt:use>
                  </div>
                  <xt:use types="address-type" label="author_address"></xt:use>
            </xt:component>

            <xt:component name="book-type">
                  <div class="book">
                        <div class="title">
                              Title : <xt:use types="string"
label="title">title</xt:use>
                        </div>
                        <div class="price">
                              Price : <xt:use types="number"
label="price">00.00</xt:use> Frs
                        </div>
                        <div class="publisher">
                              Publisher : <xt:use types="string"
label="publisher">publisher</xt:use>
                        </div>
```

```
                          <div class="author">
                                <xt:use types="author-type"
label="author"></xt:use>
                          </div>
                    </div>
             </xt:component>

             <xt:component name="journal-type">
                    <div class="journal">
                          <div class="name">
                                Name : <xt:use types="string" label="name">journal
name</xt:use>
                          </div>
                          <div class="editor">
                                Editor : <xt:use types="string"
label="editor">editor name</xt:use>
                          </div>
                    </div>
                    <div class="end"></div>
             </xt:component>

             <xt:component name="article-type">
                    <div class="article">
                          <div class="title">
                                Title : <xt:use types="string"
label="title">title</xt:use>
                          </div>
                          <xt:use types="author-type" label="author"></xt:use>
                    </div>
             </xt:component>

      </xt:head>
</head>
<body>
      <div class="university">
             <h1>University</h1>
             <div class="name">
                    <h2><xt:use types="string" label="university-name">university-
name</xt:use></h2>
             </div>
             <div class="address">
                    <h2>Address</h2>
                    <xt:use types="string" label="address">address</xt:use>
             </div>
             <div class="library">
                    <h2>Articles</h2>
                    <xt:repeat label="articles">
                          <xt:use types="article-type" label="article"></xt:use>
                    </xt:repeat>
                    <h2>Journals</h2>
                    <xt:repeat label="journals">
                          <xt:use types="journal-type" label="journal"></xt:use>
                    </xt:repeat>
                    <h2>Books</h2>
                    <xt:repeat label="books">
                          <xt:use types="book-type" label="book"></xt:use>
                    </xt:repeat>
             </div>
      </div>
```

```
</body>
</html>
```

7.2 Primitives application Results

a. Change-Cardinality applied to element address

```
            <div class="address">
                  <xt:use types="string" label="address">address</xt:use>
            </div>
```

Becomes

```
            <xt:option label="option_address">
                  <div class="address">
                        <xt:use types="string" label="address">address</xt:use>
                  </div>
            </xt:option>
```

b. Insert-Subelement applied to Author-Type

```
            <xt:component name="author-type">
                  <h3>Author</h3>
                  <div class="name">
                        Name : <xt:use types="string"
label="author_name">name</xt:use>
                  </div>
                  <xt:use types="address-type" label="author_address"></xt:use>
            </xt:component>
```

Becomes

```
            <xt:component name="author-type">
                  <h3>Author</h3>
                  <div class="name">
                        Name : <xt:use types="string"
label="author_name">name</xt:use>
                  </div>
                  <xt:use types="address-type" label="author_address"></xt:use>
                  <xt:use types="string" label="affiliation">affiliation</xt:use>
            </xt:component>
```

c. Remove-Subelement applied to the subelement “Price” of the component “Book-Type”

```
            <xt:component name="book-type">
                  <div class="book">
                        <div class="title">
                              Title : <xt:use types="string"
label="title">title</xt:use>
                        </div>
                        <div class="price">
                              Price : <xt:use types="number"
label="price">00.00</xt:use> Frs
                        </div>
                        <div class="publisher">
                              Publisher : <xt:use types="string"
label="publisher">publisher</xt:use>
```

```
                    </div>
                    <div class="author">
                          <xt:use types="author-type"
label="author"></xt:use>
                    </div>
              </div>
        </xt:component>
```

Becomes

```
        <xt:component name="book-type">
              <div class="book">
                    <div class="title">
                          Title : <xt:use types="string"
label="title">title</xt:use>
                    </div>
                    <div class="publisher">
                          Publisher : <xt:use types="string"
label="publisher">publisher</xt:use>
                    </div>
                    <div class="author">
                          <xt:use types="author-type"
label="author"></xt:use>
                    </div>
              </div>
        </xt:component>
```

d. Change-SubelementCardinality applied to the subelement “Author” in the “Article-Type”.

```
        <xt:component name="article-type">
              <div class="article">
                    <div class="title">
                          Title : <xt:use types="string"
label="title">title</xt:use>
                    </div>
                    <xt:use types="author-type" label="author"></xt:use>
              </div>
        </xt:component>
```

Becomes

```
        <xt:component name="article-type">
              <div class="article">
                    <div class="title">
                          Title : <xt:use types="string"
label="title">title</xt:use>
                    </div>
                    <xt:repeat label="repeat-author" minOccurs="1">
                          <xt:use types="author-type"
label="author"></xt:use>
                    </xt:repeat>
              </div>
        </xt:component>
```